

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ**

А.А. Дубаков

СЕТЕВОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие



Санкт-Петербург

2013

Дубаков А.А. Сетевое программирование: учебное пособие / А.А. Дубаков – СПб: НИУ ИТМО, 2013. – 248 с.

В пособии вводится понятие распределенных клиент-серверных приложений и рассматриваются способы их реализации с применением программирования сокетов, RMI, сервлетов и JSP на основе применения JavaSE (Java Standard Edition) и JavaEE (Java Enterprise Edition). Подробное рассмотрение технологий сопровождается практическими примерами по реализации клиентских и серверных компонент распределенной вычислительной архитектуры. Для демонстрации применения распределенных технологий используется популярная открытая среда разработки IDE Eclipse и сервер приложений GlassFish.

Пособие подготовлено на кафедре “Сервисов и услуг в инфокоммуникационных системах” НИУ ИТМО и предназначено для бакалавров по направлению 210700 «Инфокоммуникационные технологии и системы связи».

Рекомендовано к печати Ученым советом факультета инфокоммуникационных технологий. Протокол № 3 от 19 марта 2013 г.



В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена программа его развития на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики».

© Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики, 2013

© А.А. Дубаков, 2013

Оглавление

ВВЕДЕНИЕ.....	6
ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ СЕТЕВЫХ СОКЕТОВ.....	7
Основы сетевого взаимодействия	7
Архитектура клиент/сервер	8
Протоколы.....	9
IP адрес и порт.....	9
Сокеты	11
Классы Java для сетевого программирования	11
Создание приложения с использованием UDP протокола	14
Классы DatagramPacket и DatagramSocket	15
Создание сервера UDP.....	17
Создание клиента UDP	20
Пример разработки приложений UDP в IDE Eclipse	23
Вопросы для самопроверки.....	27
Создание сетевых приложений с использованием TCP ...	28
Идентификация методов классов Socket и ServerSocket.....	28
Создание сервера TCP/IP.....	30
Создание клиента TCP/IP.....	36
Разработка потокового взаимодействия в IDE Eclipse	40
Вопросы для самопроверки.....	44
ВВЕДЕНИЕ В RMI.....	44
Введение в распределенные приложения	44
Вызов удаленного метода	46
Компоненты приложения RMI.....	47
Архитектура RMI.....	48
Уровень удаленной ссылки	49
Транспортный уровень.....	49
Пакеты RMI	49
Этапы создания распределенного приложения.....	53
Создание удаленного интерфейса.....	53
Реализация удаленного интерфейса.....	54
Создание сервера RMI	55

Политика безопасности.....	57
Создание клиента RMI.....	60
Выполнение приложения RMI.....	61
Разработка RMI-приложения в среде IDE Eclipse.....	64
Вопросы для самопроверки.....	73
ТЕХНОЛОГИИ И АРХИТЕКТУРА JAVAEE.....	74
Введение в сервлеты Java.....	77
Понятие сервлета.....	77
Технология Java Servlet.....	77
Работа сервлетов.....	79
Иерархия классов сервлетов и методы жизненного цикла.....	80
Иерархия класса Servlet.....	80
Методы жизненного цикла сервлета.....	82
Создание сервлета.....	87
Программирование сервлета.....	87
Пример разработки сервлета.....	91
Servlet API и события жизненного цикла.....	103
Servlet API.....	103
Пакет javax.servlet.http.....	110
API жизненного цикла сервлета.....	117
Типы событий.....	118
Обработка событий жизненного цикла сервлета.....	119
Слушатели сеанса HTTP.....	122
Описание элементов дескриптора развертывания.....	129
Управление сессиями servlet.....	130
Приемы управления сессией.....	130
Использование Cookies.....	136
Обработка ошибок и исключений в сервлетах.....	149
Взаимодействие сервлетов.....	160
Вопросы для самопроверки.....	168
Введение в технологию JSP.....	169
Основные возможности JSP.....	169
Использование регулярных классов в JSP.....	175
Жизненный цикл JSP.....	179

Структура JSP-страницы.....	180
Кодирование элементов сценария JSP	186
Неявные объекты JSP	186
Действия JSP	188
Программирование JSP	191
Классы JSP API	191
Этапы создания приложения JSP.....	193
Понятие и работа с JavaBeans	198
Введение в JSP Expression Language (EL)	208
Введение в JSTL.....	218
Расширенный пример с использованием EL и JSTL.....	229
Вопросы для самопроверки	242
ЗАКЛЮЧЕНИЕ.....	244
ЛИТЕРАТУРА	245

Введение

В настоящее время мир переживает информационную революцию, вызванную широким внедрением в жизнь общества Интернета и Всемирной паутины, которая информационно связывает все сферы деятельности, все организации и конкретных людей. Суть этой революции заключается в интеграции в едином мировом информационном пространстве программно-аппаратных средств, средств связи и телекоммуникаций, информационных ресурсов и накопленных знаний в единую информационную и коммуникационную инфраструктуру. World Wide Web делает Интернет простым в использовании и обеспечивает ему мультимедийные возможности. Решающее значение для информационных стратегий организаций является присутствие в Интернете и создание распределенных информационных систем, обеспечивающих взаимодействие с заказчиками и поставщиками, маркетинг и многие другие виды деятельности. Все это вызывает повышенный интерес к технологиям создания инфокоммуникационных систем, основанных на распределенных вычислениях.

Одной из популярных технологий, созданных изначально для реализации систем распределенной обработки и сетевого программирования, является технология Java, поддерживающая удобные средства сетевого взаимодействия программных систем. Java предоставляет набор встроенных сетевых возможностей, которые обеспечивают легкость применения Java для разработки интернет и веб-приложений. Применение Java позволяет программам реализовать поиск информации в сети и взаимодействовать с программами, выполняющимися на других компьютерах на глобальном или национальном уровне, а также только в пределах организации и выполнять многие другие сетевые операции.

В настоящем пособии рассматриваются программные и технологические компоненты создания распределенных систем обработки данных на основе применения Java Platform, Standard Edition (Java SE) и Java Platform, Enterprise Edition (Java EE).

Введение в программирование сетевых сокетов

Основы сетевого взаимодействия

Java был разработан как язык сетевого программирования для обеспечения возможности создания клиент/серверных приложений, которые взаимодействуют друг с другом в сети. Java обеспечивает обширную библиотеку сетевых классов, которые позволяют быстро получать доступ к сетевым ресурсам.

Основные сетевые возможности Java реализуются в классах и интерфейсах пакета `java.net`, посредством которых Java обеспечивает потоковое взаимодействие, позволяющее приложениям рассматривать соединение, как поток данных. Классы и интерфейсы пакета `java.net` также предлагают коммуникации на основе передачи отдельных пакетов информации, которые обычно используется для передачи аудио и видео через Интернет. В этом разделе будут рассмотрено создание и управление сокетами, а также способы организации взаимодействия приложений на основе передачи пакетов и потоков данных.

Рассмотрение сети естественно будет касаться обеих сторон клиент/серверного взаимодействия. В общем случае клиент запрашивает выполнения некоторых действий, а сервер выполняет запрос и формирует ответ клиенту. Общая модель запрос-ответ реализуется между веб-браузерами и веб-серверами. Когда пользователь выбирает сайт для просмотра в браузере (клиентское приложение), запрос отправляется на соответствующий веб-сервер (серверное приложение). Сервер обычно отвечает клиенту, посылая соответствующие сформированные HTML-страницы.

Взаимодействие в сети рассматривается на основе понятия сокетов, которые позволяют приложениям рассматривать сетевые подключения как файлы, и программа может читать из сокета или писать в сокет, как она делает это с файлом. Слово `Socket` в переводе на русский язык означает "гнездо". Это название образовалось по аналогии с гнездами (разъемами) на аппаратуре, с которыми стыкуются разъемы. Сокет представляет собой программную конструкцию (объект), которая определяет конечную точку соединения.

Существуют два механизма, предназначенных для сетевого взаимодействия программ, - это сокеты датаграмм, которые используют пользовательский датаграммный протокол (`User Datagram Protocol`) (`UDP`) без установления соединения, и сокеты, использующие Прото-

кол управления передачей / Межсетевой протокол (Transmission Control Protocol/Internet Protocol) (TCP/IP), устанавливающий соединение.

Датаграмма - пакет данных, отправленный по сети, прибытие которого, время прибытия и содержание не гарантировано. Не гарантируется также и порядок доставки пакетов. При передаче пакета UDP по какому-либо адресу нет никакой гарантии того, что он будет принят, а также, что по этому адресу вообще существует потребитель пакетов. Аналогично, когда вы получаете датаграмму, у вас нет никаких гарантий, что она не была повреждена в пути следования или что отправитель ожидает подтверждения получения датаграммы. Использование UDP может привести к потере или к дублированию пакетов, что приводит к дополнительным проблемам, связанным с проверкой ошибок и обеспечением надежности передачи данных. Если вам необходимо добиться оптимальной производительности, и вы готовы сократить затраты на проверку целостности информации, пакеты UDP могут оказаться весьма полезными.

При использовании потоковых сокетов, программа устанавливает соединение с другим сокетом и, пока соединение установлено, поток данных протекает между программами, и говорят, что потоковые сокеты обеспечивают обслуживание на основе установления соединения. TCP/IP является потоковым протоколом на основе установления двусторонних соединений точка-точка между узлами Интернет, и взаимодействие между компьютерами по этому протоколу предназначено для реализации надежной передачи данных. Все данные, отправленные по каналу передачи, получаются в том же порядке, в котором они передавались. В отличие от датаграммных сокетов, сокеты TCP/IP реализуют высоконадежные устойчивые соединения между клиентом и сервером.

Архитектура клиент/сервер

Любой компьютер, сервер, подключенный к локальной или глобальной сети, называется хостом (host- хозяин, принимающий гостей). Клиент/серверная модель представляет собой архитектуру разработки приложений, предназначенную для отделения уровня представления данных от их обработки и хранения.

Клиентские хосты запрашивают обслуживание, и хост сервера обеспечивает обслуживание этих запросов. Запрос передается от клиента на сервер через сеть, обработка выполняется на сервере и скрыта от клиента. При этом один сервер может обслуживать несколько клиентов. На Рис. 1 представлена клиент/серверная архитектура, обслуживающая несколько клиентов.

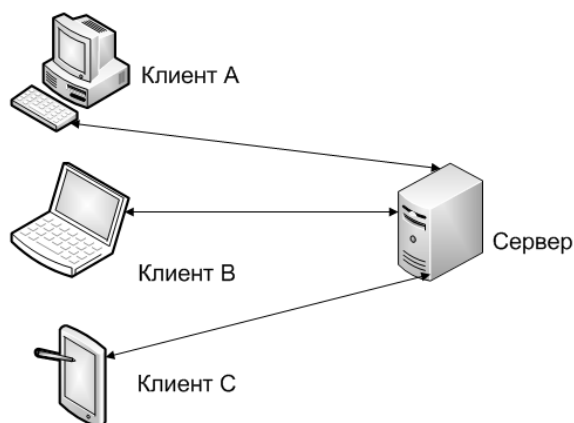


Рис. 1. Несколько клиентов, получающих доступ к серверу.

Сервер и клиент не обязательно являются отдельными компонентами оборудования, они могут быть реализованы программами, работающими на одной или различных машинах.

Серверная часть клиент/серверного приложения управляет ресурсами, распределенными среди нескольких пользователей, которые получают доступ к серверу вместе с другими клиентами.

Протоколы

При общении между людьми соблюдаются некоторые неписанные правила (или протоколы). Например, люди не разговаривают одновременно и непрерывно - никто не смог бы понять, что говорит другой человек, если бы они не следовали определенным правилам. Когда один говорит, собеседник слушает и наоборот. При этом люди говорят на языке и в темпе, которые понятны их собеседникам.

Когда общаются компьютеры, им также необходимо следовать определенным правилам. Данные посылаются от одной машины на другую в форме пакетов (packets). Правила определяют процедуру упаковки данных в пакеты, скорость передачи и разборки данных в их исходную форму. Эти правила называются сетевыми протоколами. Сетевой протокол является набором правил и соглашений, поддерживаемых системами, которые взаимодействуют в сети. Сетевое программное обеспечение обычно реализуется несколькими уровнями протоколов, располагающихся слоями, один над другим.

IP адрес и порт

Сервер Интернет может рассматриваться как набор сокетов, которые обеспечивают дополнительные возможности – обычно называемые

сервисами. Примерами сервисов являются электронная почта (email), сетевой удаленный доступ (Telnet) и протокол передачи файлов (FTP - File Transfer Protocol). Каждый сервис связан с портом (port), который является числовым адресом, через который обрабатывается запрошенный запрос, например, запрос Веб-страницы.

Протокол TCP запрашивает два элемента данных: IP адрес и номер порта. Каким образом происходит, что при вводе адреса URL - Uniform Resource Locator - <http://www.ifmo.ru>, браузер получает домашнюю страницу НИУ ИТМО? URL - это стандартизированный способ записи адреса ресурса в сети Интернет. Интернет протокол (Internet Protocol - IP) обеспечивает логический адрес, называемый IP-адресом сетевого устройства. Каждому доменному имени компьютера (www.ifmo.ru) в системе доменных имен (DNS - Domain Name System) соответствует IP-адрес. IP-адреса, используемые в Интернет, имеют определенный формат. Каждый адрес представляется 32-разрядным числом, состоящим из четырех 8-разрядных чисел (байт), каждое в диапазоне значений от 0 до 255. ИТМО имеет свое зарегистрированное имя, позволяющее www.ifmo.ru представляться IP адресом 194.85.160.58.

Если номер порта не указан, то используется номер порта по умолчанию для сервиса, примеры которых приведены в Таблицы 1.

Таблица 1.

Номер порта	Приложение
21	FTP - передает файлы
23	Telnet -обеспечивает удаленный доступ
25	SMTP - доставляет почтовые сообщения
67	BOOTP - обеспечивает конфигурацию во время загрузки
80	HTTP - передает Веб-страницы
109	POP - получает доступ к почтовому сервису на удаленной хосте

Еще раз рассмотрим структуру URL **<http://www.ifmo.ru>**.

Первый компонент URL (который является, http) обозначает, что используется протокол передачи гипертекстовых файлов (Hypertext Transmission Protocol - HTTP) для управления HTML-документами. Если файл не задан, большинство Веб серверов конфигурируются таким образом, что представляется файл с именем index.html. Таким образом,

IP адрес и порт являются определенными либо явной спецификацией всех частей URL, либо применением спецификации по умолчанию.

Сокеты

В клиент/серверной архитектуре приложений хост сервера обеспечивает сервисы, аналогичные обработке запросов базы данных. При этом взаимодействие, которое происходит между клиентом и сервером должно быть надежным, и данные не должны быть потеряны, а также должны быть доступны клиенту в той же последовательности, в которой сервер их отправлял.

Как уже отмечалось, TCP обеспечивает надежный канал соединения точка-точка (point-to-point) для клиент/серверных приложений, с помощью которого программы клиента и сервера устанавливают соединение и связывают сокеты. Сокеты - это название программного интерфейса хоста для обеспечения информационного обмена между процессами. Взаимодействующие процессы могут выполняться как на одном компьютере, так и на различных хостах, соединенных между собой через сеть. Сокеты используются для управления каналом связи между приложениями, установленным через сеть. Каждое TCP-соединение может быть однозначно идентифицировано своими двумя конечными точками. Таким образом, могут быть обеспечены множественные соединения клиента и сервера. После создания сокета, через него выполняется дальнейшее взаимодействие между клиентом и сервером.

Классы Java для сетевого программирования

Язык программирования Java содержит пакет `java.net`, классы и интерфейсы которого обеспечивают поддержку работы в сети и используют протоколы для сетевого программирования.

Классы пакета `java.net`

Сетевые классы содержат методы для выполнения таких задач как открытие и закрытие соединения с удаленным хостом, передача и получение пакетов данных, и доступ к ресурсам сети. Ниже представлены некоторые из классов, представленных в пакете `java.net`:

- `DatagramPacket`: Представляет объект датаграммного пакета (datagram packet), который содержит данные с информацией об адресе источника и адресе назначения, номера портов источника и назначения. Эта информация используется для передачи

датаграммного пакета от одного компьютера к другому по сети.

- **DatagramSocket**: Представляет объект датаграммного сокета, который может посылать и принимать пакеты датаграмм. Пакеты, посланные объектом **DatagramSocket**, могут приходить к получателю в любом порядке.
- **MulticastSocket**: Создает мультикастовый (multicast) объект датаграммного сокета, который используется для отправки и приема пакетов датаграмм для групп. IP адреса класса D (IP адреса между 224.0.0.0 и 239.255.255.255) используются для создания групп. Когда сообщение посылается на IP-адрес класса D, все клиенты, присоединенные к группе, получают отправленное сообщение. Этот класс содержит методы `joinGroup()` и `leaveGroup()`, которые дают возможность клиентам объединяться и покидать определенную группу.
- **InetAddress**: Создает объект, который содержит информацию, состоящую из IP-адреса и имени хоста (host name).
- **ServerSocket**: Создает объект сокета сервера, который слушает запросы клиента. Объекты **ServerSocket** используют номер порта для получения запросов клиента.
- **Socket**: Создает объект сокета клиента, который соединяется с объектом класса **ServerSocket** для посылки запросов на сервер.
- **URL**: Представляет объект, который может представлять файл или иной ресурс в Интернет.

Пакет `java.net` также содержит классы для обработки исключительных ситуаций, которые позволяют обрабатывать сетевые ошибки во время выполнения. Ниже представлены несколько классов обработки исключительных ситуаций, содержащихся в пакете `java.net`:

- **BindException**: Этот объект исключительных ситуаций генерируется, когда возникает ошибка при попытке связаться с сокетом локального адреса или порта. Например, когда не может быть доступен локальный адрес или запрашиваемый порт уже используется.
- **ConnectException**: Этот объект исключительных ситуаций генерируется, когда возникает ошибка во время соединения сокета с удаленным IP адресом и портом. Например, когда удаленное соединение не может быть установлено.
- **MalformedURLException**: Этот объект исключительных ситуаций генерируется, когда URL содержит недействительный про-

токол или если ссылка на URL не может быть успешно обработана.

- `UnknownHostException`: Этот объект исключительных ситуаций генерируется, когда IP-адрес хоста не может быть определен или не существует.

Класс `InetAddress`

Класс `InetAddress` позволяет определять имена хостов и связанные с ними IP-адреса, необходимые во время создания сетевых приложений. Объекты `InetAddress` инициализируются с использованием статических методов, объявленных в классе `InetAddress`. Некоторые из этих методов, используемые для инициализации объектов `InetAddress`, представлены ниже:

- `public static InetAddress getLocalHost()`: Возвращает объект `InetAddress`, который содержит IP адрес локального компьютера.
- `public static InetAddress getByName(String host)`: Возвращает объект `InetAddress`, который содержит IP адрес имени хоста, переданный методу как `String`.
- `public static InetAddress[] getAllByName(String host)`: Возвращает массив объектов `InetAddress`, который содержит IP адреса для имени хоста, переданный методу как `String`.

Статические методы `getLocalHost()`, `getByName()`, `getAllByName()` и `getByAddress()` вызывают исключительную ситуацию `UnknownHostException`.

Нестатические методы, определенные в классе `InetAddress`, могут быть доступны после инициализации объектов `InetAddress`. Нестатические методы определены в классе `InetAddress` и представлены ниже:

- `public boolean equals(Object obj)`: Возвращает `true`, если объект `InetAddress` имеет такой же IP адрес как объект `obj`, переданный как параметр.
- `public byte[] getAddress()`: Возвращает IP адрес объекта `InetAddress` в виде массива `byte`. Например, если IP адрес объекта `InetAddress ipAddress` равен `194.85.160.58`, то этот метод возвращает следующий массив `byte`:

```
ipaddress[0] = 194
ipaddress[1] = 85
ipaddress[2] = 160
ipaddress[3] = 58
```

- `public String getHostAddress()`: Возвращает IP адрес объекта `InetAddress` как `String`.

- `public String toString()`: Возвращает IP адрес хоста как `string` в формате имя хоста/IP адрес.

Рассмотрим пример инициализации и использования объекта `InetAddress`:

```
import java.net.*;

class InetAddressClass {
    public static void main(String arg[]) {
        try {
            /*
             * Создается объект InetAddress, используя
            getLocalHost()
             * статический метод класса InetAddress
             */
            InetAddress address = InetAddress.getLocalHost();
            /* Получение IP адреса хоста */
            String addressHost = address.getHostAddress();
            /* Вывод IP адреса хоста */
            System.out.println("Вывод IP адреса хоста
            "+addressHost);
            // Вывод имени хоста
            System.out.println("Вывод имени хоста
            "+address.getHostName());
        } catch (UnknownHostException e) {
            System.out.println("Error");
        }
    }
}
```

В этом примере определяется объект класса `InetAddress`. В методе `main()` класса `InetAddressClass` инициализируется объект `address`, используя статический метод `getLocalHost()`. Объект `InetAddress` содержит IP адрес компьютера, на котором выполняется программа. Метод `getHostAddress()` возвращает IP-адрес хоста как `string`. Метод `getHostName()` возвращает имя хоста как `string`. В результате выполнения класса `InetAddressClass` выводится IP-адрес и имя хоста.

Создание приложения с использованием UDP протокола

Как уже упоминалось, Java позволяет разрабатывать сетевые приложения с использованием датаграммных сокетов UDP и сокетов TCP/IP. Сокеты UDP используют протокол UDP для взаимодействия приложений через сеть. UDP является быстрым, без установления соединения и ненадежным протоколом. Пакет `java.net` содержит следующие два класса, позволяющие применять сокет UDP в приложении Java:

- Класс `DatagramPacket`
- Класс `DatagramSocket`

Классы DatagramPacket и DatagramSocket

Объект DatagramPacket является контейнером данных, состоящим из датаграммных пакетов, которые посылаются или принимаются через сеть. Следующие конструкторы используются для инициализации объектов DatagramPacket:

- `public DatagramPacket(byte[] buffer, int buffer_length)`: Создает объект DatagramPacket, который принимает и сохраняет данные в массиве byte. Длина буфера массива byte задается вторым параметром buffer_length.
- `public DatagramPacket(byte[] buffer, int buffer_length, InetAddress address, int port)`: Создает объект DatagramPacket, который посылает пакеты данных заданной длины. Пакеты данных посылаются на компьютер с заданным IP адресом и номером порта, передаваемыми как параметры.

Методы, определенные в классе DatagramPacket, могут быть использованы после инициализации объекта класса DatagramPacket. Таблица 2 представляет методы класса DatagramPacket.

Таблица 2.

Метод	Описание
<code>public InetAddress getAddress()</code>	Возвращает объект InetAddress, который содержит IP-адрес компьютера, на который посылается датаграммный пакет или от которого принимается датаграммный пакет
<code>public byte[] getData()</code>	Возвращает буферный массив byte, который содержит данные
<code>public int getLength()</code>	Возвращает длину буферного массива, который содержит данные
<code>public int getPort()</code>	Возвращает номер порта компьютера, на который посылается датаграммный пакет или откуда датаграммный пакет принимается.
<code>public void setAddress(InetAddress address)</code>	Устанавливает IP-адрес машины, на которую датаграммный пакет должен быть послан
<code>public void setData(byte[] buffer)</code>	Устанавливает массив byte в качестве данных для пакета.
<code>public void setPort(int port)</code>	Устанавливает номер порта на удаленном хосте
<code>public void setLength(int length)</code>	Устанавливает длину буфера обмена

Класс `DatagramSocket` содержит функциональность для управления объектами `DatagramPacket`. Объекты `DatagramPacket` отправляют и принимают сохраненные данные, используя объект `DatagramSocket`. Следующие конструкторы, используются для инициализации объекта `DatagramSocket`:

- `public DatagramSocket()`: Создает объект `DatagramSocket` и связывает его с любым доступным портом на локальном компьютере.
- `public DatagramSocket(int port)`: Создает объект и связывает его с портом на локальном хосте, заданным в параметре.
- `public DatagramSocket(int port, InetAddress address)`: Создает объект и связывает его с портом заданного хоста.

Конструктор класса `DatagramSocket` вызывает исключительную ситуацию `SocketException`.

Таблица 3 представляет методы класса `DatagramSocket`, которые используются для получения информации из объекта `DatagramSocket`:

Таблица 3.

Метод	Описание
<code>public InetAddress getInetAddress()</code>	Возвращает объект <code>InetAddress</code> , содержащий IP-адрес, с которым объект <code>DatagramSocket</code> связывается.
<code>public InetAddress getLocalAddress()</code>	Возвращает объект <code>InetAddress</code> , который содержит IP-адрес локального хоста, к которому объект <code>DatagramSocket</code> связывается.
<code>public int getLocalPort()</code>	Возвращает целое значение, которое представляет порт локального хоста, с которым объект <code>DatagramSocket</code> связывается.
<code>public void bind(SocketAddress address)</code>	Связывает объект <code>DatagramSocket</code> с объектом <code>SocketAddress</code> .
<code>public void close()</code>	Закрывает объект <code>DatagramSocket</code> .
<code>public void connect(InetAddress address, int port)</code>	Соединяет объект <code>DatagramSocket</code> с заданным IP-адресом и портом.
<code>public void disconnect()</code>	Разъединяет объект <code>DatagramSocket</code> .
<code>public boolean isBound()</code>	Возвращает <code>true</code> , если объект <code>DatagramSocket</code> связан с портом.
<code>public boolean isClosed()</code>	Возвращает <code>true</code> , если объект <code>DatagramSocket</code> закрывается.
<code>public boolean isConnected()</code>	Возвращает <code>true</code> , когда объект

Метод	Описание
	DatagramSocket соединяется с IP-адресом.
<code>public void receive(DatagramPacket packet)</code>	Получает датаграммный пакет от текущего объекта DatagramSocket.
<code>public void send(DatagramPacket packet)</code>	Передает датаграммный пакет от текущего объекта DatagramSocket.

Создание сервера UDP

Сервер UDP представляет собой сетевое приложение использующее протокол UDP для обслуживания запросов клиентских приложений. Для создания сервера UDP используется объект DatagramSocket, который принимает объекты DatagramPacket от клиентов. Для создания сервера UDP необходимо выполнить следующие шаги:

- Создать сокет, используя объект DatagramSocket.
- Создать объект класса DatagramPacket и использовать метод receive() для получения сообщения клиента.
- Создать объект класса DatagramPacket и использовать метод send() для отправки сообщения клиенту.
- Запустить сервер, вызывая конструктор класса сервера UDP в методе main().

Следующий фрагмент кода можно использовать для создания объекта DatagramSocket:

```
try
{
    DatagramSocket socket = new DatagramSocket(1501);
}
catch(SocketException se)
{
    System.out.println("Error");
}
```

В предыдущем фрагменте кода объект socket класса DatagramSocket связывается с портом номер 1501.

Объект DatagramPacket, который получает датаграммный пакет, содержит буфер для хранения датаграмм. Следующий фрагмент кода можно использовать для создания объекта DatagramPacket, который принимает датаграммные пакеты:

```
try
{
    DatagramPacket packet = new DatagramPacket(buffer,
buffer.length);
    socket.receive(packet);
}
```

```

catch(Exception e){
System.out.println("Error");
}

```

В предыдущем фрагменте кода создается объект `packet` класса `DatagramPacket`, который вызывает метод `receive()` для получения пакета от объекта `socket`.

Объект `DatagramPacket`, отправленный получателю, отличается от объекта принятых данных. Этот объект `DatagramPacket` содержит IP-адрес и номер порта хоста, с которого пакет отправлен. Следующий фрагмент кода можно использовать для отправки объекта `DatagramPacket` на заданный адрес:

```

try
{
DatagramPacket packet = new DatagramPacket(buffer, length,
address, port);
socket.send(packet);
}
catch(Exception e)
{
System.out.println("Error");
}

```

В предыдущем фрагменте кода создается новый объект `packet` класса `DatagramPacket`, который принимает четыре параметра:

- `buffer`: Задаёт буфер, который содержит данные.
- `length`: Задаёт длину буфера в байтах.
- `address`: Задаёт адрес, на который датаграмма отправлена.
- `port`: Задаёт номер порта, который удаленный компьютер использует для получения датаграммы.

Метод `send()` класса `DatagramSocket` отправляет адресату объект `DatagramPacket`.

Для запуска сервера UDP вызывается конструктор класса в методе `main()`. Следующий фрагмент кода можно использовать для запуска сервера UDP:

```

public static void main(String args[]) throws Exception
{
/* Запуск сервера */
new UDPServer();
}

```

В предыдущем фрагменте кода, создается объект класса `UDPServer`, который запускает серверное приложение UDP.

Следующий код позволяет создать `UDPServer`, который отображает полученные сообщения от клиента и отправляет клиенту ответные сообщения:

```

import java.io.*;
import java.net.*;
public class UDPServer

```

```

{
    /* Объявляются переменные */
    DatagramSocket socket = null;
    BufferedReader in = null;
    String str = null;
    byte[] buffer ;
    DatagramPacket packet;
    InetAddress address;
    int port;
    /* Конструктор класса UDPServer */
    public UDPServer() throws IOException
    {
        /* Создается объект DatagramSocket, который получает запросы клиента
        на номер порта 1501 */
        socket = new DatagramSocket(1501);

        /* Вызывается метод call()*/
        call();
    }
    public void call()
    {
        try
        {
            while (true)
            {
                buffer= new byte[256];
                /* Инициализируется объект DatagramPacket */
                packet = new DatagramPacket( buffer, buffer.length);
                /* Посылается пакет датаграмм, используя метод receive()
                класса DatagramSocket */
                socket.receive(packet);
                if(packet == null) break;
                System.out.println("Request string for sending to client ");
                try
                {
                    /*Создается входной поток, который считывает данные с консо-ли*/
                    in = new BufferedReader(new InputStreamReader(System.in));
                }
                catch(Exception e)
                {
                    System.out.println("Error : " + e);
                }
                str = in.readLine();
                buffer = str.getBytes();
                address = packet.getAddress();
                port = packet.getPort();
                packet = new DatagramPacket(buffer, buffer.length, address, port);
                /* Посылается датаграммный пакет */
                socket.send(packet);
            }
            /* Закрывается поток и сокет */
            in.close();
            socket.close();
        }
        catch(Exception e)
        {

```

```

        System.out.println("Error : " + e);
    }
}
public static void main(String args[]) throws Exception
{
    /* Запускается сервер */
    new UDPServer();
}
}

```

В рассмотренном коде создается объект socket класса DatagramSocket в конструкторе класса UDPServer. После инициализации сокета на порт 1501, вызывается метод call(), состоящий из методов receive() и send(), которые управляют клиент/серверным взаимодействием. Предыдущий код сохраняется как UDPServer.java и компилируется программой java.exe.

Создание клиента UDP

Клиент UDP представляет собой приложение, которое использует протокол UDP для отправки запросов на сервер и получения ответов от серверного приложения. В клиентском UDP-приложении, необходимо создать объект класса DatagramSocket, который принимает сообщения от сервера UDP, для чего необходимо выполнить следующие шаги:

1. Создать сокет, использующий объект класса DatagramSocket для установки соединения с сервером.
2. Создать объект класса DatagramPacket и использовать метод send() для отправки сообщения на сервер.
3. Создать объект класса DatagramPacket и использовать метод receive() для получения сообщений, отправленных сервером.

Следующий фрагмент кода можно использовать для создания объекта класса DatagramSocket для клиентского приложения:

```

try
{
    DatagramSocket socket = new DatagramSocket();
}
catch(Exception e)
{
    System.out.println("Error");
}
}

```

В предыдущем фрагменте кода конструктор связывает объект класса DatagramSocket с любым доступным локальным портом, поскольку не указан номер порта в параметре.

Этот объект DatagramPacket содержит IP-адрес и номер порта сервера, куда посылается запрос. Следующий фрагмент кода используется для отправки объекта DatagramPacket на заданный сервер:

```

try
{
DatagramPacket packet = new DatagramPacket(buffer, length,
address, port);
socket.send(packet);
}
catch(Exception e)
{
System.out.println("Error");
}

```

В предыдущем фрагменте кода создается объект `packet` с помощью конструктора класса `DatagramPacket`, который принимает четыре параметра. Метод `send()` класса `DatagramSocket` посылает объект класса `DatagramPacket` на сервер.

Следующий фрагмент кода можно использовать для создания объекта `DatagramPacket`, который принимает пакеты датаграмм от сервера:

```

try
{
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
socket.receive(packet);
str = new String(packet.getData());
System.out.println("Принятое сообщение : "+str);
}
catch(Exception e){
System.out.println("Ошибка");
}

```

В предыдущем фрагменте кода создается объект `packet` класса `DatagramPacket`, который вызывает метод `receive()`. Метод `getData()` принимает данные из объекта `packet` и сохраняет их в переменной типа `string`.

Следующий код используется для создания класса `UDPClient`, который посылает и принимает сообщения от `UDPServer`:

```

import java.io.*;
import java.net.*;
public class UDPClient
{
    /* Объявляются переменные */
    static DatagramSocket socket;
    static InetAddress address;
    static byte[] buffer;
    static DatagramPacket packet;
    static String str, str2;
    static BufferedReader br;
    public static void main(String arg[]) throws Exception{
    /* Создается входной поток, который читается с консоли */
        br = new BufferedReader(new InputStreamReader(System.in));
        while(true)
    {

```

```

/* Создается новый объект DatagramSocket и связывается с портом по умолчанию */
    socket = new DatagramSocket();
    address = InetAddress.getByName("127.0.0.1");
    buffer = new byte[256];
packet = new DatagramPacket(buffer, buffer.length, address, 1501);
/* Посылается DatagramPacket на сервер */
    socket.send(packet);
    System.out.println("Sending request ");
    packet = new DatagramPacket(buffer, buffer.length);
/* Принимается DatagramPacket от сервера */
    socket.receive(packet);
/* Принимаются данные от объекта пакета датаграмм и*/
    str = new String(packet.getData());
    System.out.println("Received message : "+str.trim());
    System.out.println("Do you want continue (Yes/No) : ");
    str2 = br.readLine();
/* Выход из цикла while */
    if(str2.equals("No")) break;
}
/* Закрывается объект сокет */
    socket.close();
}
}

```

В рассмотренном коде, цикл while в метод main() класса UDPClient отправляет и принимает запросы от UDPServer. Метод send() объекта socket отправляет запросы на сервер, а метод receive() принимает сообщения от UDPServer. Рис. 2 показывает вывод, полученный в результате взаимодействия клиента UDPClient и сервера UDPServer. (Для выполнения каждого приложения открывается отдельное командное окно).

```

C:\Windows\system32\cmd.exe - java UDPClient
F:\eclipse\workspace\ForBook\bin>java UDPClient
Sending request

C:\Windows\system32\cmd.exe
F:\eclipse\workspace\ForBook\bin>java UDPServer
Request string for sending to client
Hello, Client!
Request string for sending to client
Hello, Client!

```

Рис. 2. Вывод UDPClient.

На представленном рисунке сообщение «Sending request» показывает, что клиент отправил запрос на UDPServer. UDPServer выводит сообщение «Request string for sending to client» в окне консоли после при-

нения запроса от клиента. В окне сервера отображается вывод UDPServer, когда сервер получает сообщение от пользователя. UDPServer вводит сообщение в окне консоли «Hello, Client!» и отправляет его клиенту. Если пользователь вводит «No» в окне клиентской консоли, то процесс UDPClient завершается.

Пример разработки приложений UDP в IDE Eclipse

Постановка задачи

Необходимо разработать клиент/серверное приложение, в котором сервер может распространять сообщения всем клиентам, зарегистрированным в группе 233.0.0.1, порт 1502. Пользователь сервера должен иметь возможность ввода и отправки текстовых сообщений, а пользователь-клиент просматривает полученные сообщения.

Для решения поставленной задачи необходимо выполнить следующие шаги:

1. Создать новый проект.
2. Реализовать класс сервера для ввода и отправки сообщений.
3. Реализовать класс клиента для получения и просмотра сообщений.
4. Протестировать приложение – запустить сервер и клиент и отправить сообщение.

Подготовительный этап

Для реализации проекта необходимо установить и среду разработки IDE Eclipse.

Создание нового проекта

- 1) Выберите пункт меню File/New/Project, в окне выбора типа проекта укажите other/Java Project и нажмите Next.
- 2) Укажите имя проекта Socket и нажмите Finish.

Создание класса Server

Класс `Server` предназначен для отправки сообщений всем клиентам, зарегистрированным в группе 233.0.0.1. Создание класса `Server` включает в себя следующие основные шаги:

1. Создать сокет на основе класса `DatagramSocket`. Сокет сервера выполняет задачу отправки сообщений.
2. Создать объект класса `InetAddress`, представляющего адрес сервера. Адреса для групповой (multicast) передачи сообщений выбираются из диапазона 224.0.0.0 - 239.255.255.255. В нашем приложении будет использован адрес 233.0.0.1.
3. Организовать ввод сообщения с клавиатуры и создать объект `packet` класса `DatagramPacket`, который содержит введенные

данные и использует метод `send()` класса `DatagramSocket` для отправки пакета всем клиентам группы.

- 1) Для создания класса сервера щелкните правой кнопкой мыши на каталог `src` в окне `Package Explorer` и выберите `New/Class`
- 2) В появившемся окне в качестве имени пакета (`Package`) укажите `ru.ifmo.socket`, а в качестве имени класса (`Name`) задайте `Server`. Нажмите `Finish`.

Код класса `Server` приведен ниже:

```
package ru.ifmo.socket;
import java.io.*;
import java.net.*;
public class Server {
    private BufferedReader in = null;
    private String str = null;
    private byte[] buffer;
    private DatagramPacket packet;
    private InetAddress address;
    private DatagramSocket socket;
    public Server() throws IOException {
        System.out.println("Sending messages");
        // Создается объект DatagramSocket для
        // приема запросов клиента
        socket = new DatagramSocket();
        // Вызов метода transmit() для передачи сообщение всем
        // клиентам, зарегистрированным в группе
        transmit();
    }
    public void transmit() {
        try {
            // создается входной поток для приема
            // данных с консоли
            in = new BufferedReader(new
InputStreamReader(System.in));
            while (true) {
                System.out.println("Введите строку для передачи
клиентам: ");

                str = in.readLine();
                buffer = str.getBytes();
                address = InetAddress.getByName("233.0.0.1");
                // Посылка пакета датаграмм на порт номер 1502
                packet = new DatagramPacket(buffer, buff-
er.length, address,
                    1502);
                // Посылка сообщений всем клиентам в группе
                socket.send(packet);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                // Закрытие потока и сокета
                in.close();
            }
        }
    }
}
```

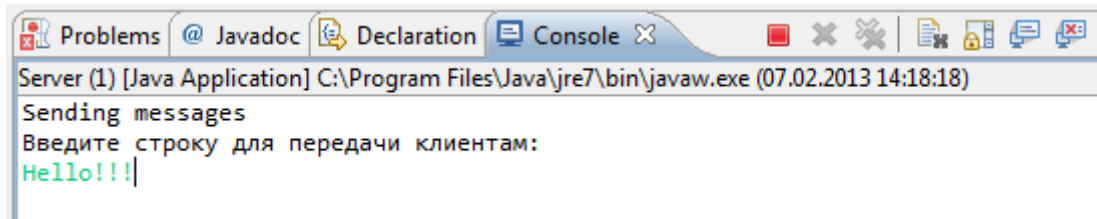



Рис. 4. Ввод сообщения в консоли сервера.

- 4) Переключитесь на консоль клиента и посмотрите, что клиент успешно принял сообщение, как показано на Рис. 5.

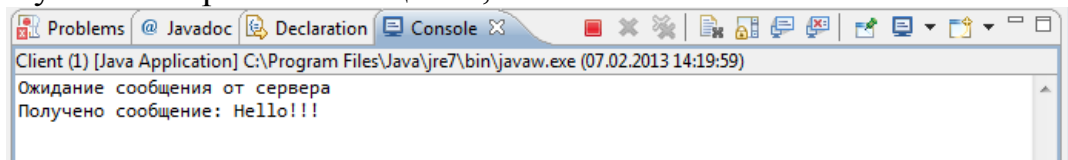


Рис. 5. Вывод сообщения в консоли клиента.

- 5) Остановка приложения осуществляется с помощью кнопки Terminate в представлении Console.

Вопросы для самопроверки

1. Какая модель представляет собой архитектуру разработки приложения, разработанную, чтобы отделять представление данных от их внутренней обработки и хранения?
2. _____ запрашивает сервисы, и сервер обслуживает эти запросы.
3. _____ представляют собой набор правил и соглашений, выполняемых системами, которые общаются через сеть.
4. Какая абстракция используется для управления линиями связи между приложениями через сеть?
5. Какой протокол обеспечивает надежный канал соединения точка-точка для клиент/серверных приложений для взаимодействия между приложениями?
6. Какой пакет языка программирования Java содержит классы и интерфейсы, которые обеспечивают сетевую поддержку?
7. Какие операции обеспечивают методы сетевых классов?
8. Перечислите обычно используемые классы, представленные в пакете java.net для организации взаимодействия .
9. Перечислите обычно используемые классы исключительных ситуаций, представленные в пакете java.net.
10. Какой класс дает Вам возможность определить имя хоста и связанные IP адреса, требуемые во время создания сетевых приложений?
11. Какой протокол является быстрым, ненадежным протоколом без

установления соединения.

12. Какой объект является контейнером данных, содержащим пакеты датаграмм, которые посылаются или принимаются через сеть?
13. Перечислите обычно используемые методы класса DatagramPacket.
14. Какой класс инкапсулирует функциональность для управления объектами DatagramPacket?
15. Перечислите обычно используемые методы класса, представленные в DatagramSocket.

Создание сетевых приложений с использованием TCP

Java поддерживает классы и методы, которые позволяют устанавливать соединение с удаленным компьютером, используя протокол TCP. В отличие от UDP, TCP является протоколом, ориентированным на установление соединения, которое гарантирует надежную связь между приложениями клиента и сервера. Взаимодействие с использованием протокола TCP, начинается после установления соединения между сокетами клиента и сервера. Сокет сервера "слушает" запросы на установление соединения, отправленные сокетами клиентов, и устанавливает соединение. После установления соединения между приложениями клиента и сервера, они могут взаимодействовать друг с другом.

Java упрощает сетевое программирование, путем инкапсуляции функциональности соединения сокета TCP в классы сокета, в которых класс Socket предназначен для создания сокета клиента, а класс ServerSocket для создания сокета сервера.

Идентификация методов классов Socket и ServerSocket

Socket является базовым классом, поддерживающим протокол TCP. Класс Socket обеспечивает методы для потокового ввода/вывода, облегчает выполнение операций чтения и записи в сокет и является обязательным для программ, выполняющих сетевое взаимодействие.

Для создания объектов класса Socket используются следующие конструкторы, определенные в классе Socket:

- `public Socket (InetAddress IP_address, int port):` Создает объект Socket, который соединяется хостом, заданным в параметрах IP_address и port.
- `public Socket (String hostname, int port):` Создает объект Socket, который соединяется с хостом, заданным параметрами имя хоста или IP адрес и port, который сервер "слушает".

Некоторые полезные методы класса Socket представлены в Таблице 4.

Таблица 4.

Метод	Описание
<code>public InetAddress getInetAddress()</code>	Возвращает объект <code>InetAddress</code> , который содержит IP-адрес, с которым соединяется объект <code>Socket</code> .
<code>public InputStream getInputStream()</code>	Возвращает входной поток для объекта <code>Socket</code> .
<code>public InetAddress getLocalAddress()</code>	Возвращает объект <code>InetAddress</code> , содержащий локальный адрес, с которым соединяется объект <code>Socket</code> .
<code>public int getPort()</code>	Возвращает удаленный порт, с которым соединяется объект <code>Socket</code> .
<code>public int getLocalPort()</code>	Возвращает локальный порт, с которым соединяется объект <code>Socket</code> .
<code>public OutputStream getOutputStream()</code>	Возвращает выходной поток объекта <code>Socket</code> .
<code>void close()</code>	Закрывает объект <code>Socket</code> .
<code>public String toString()</code>	Возвращает IP-адрес и номер порта сокета клиента как <code>String</code> .

Конструкторы и метод `close()` класса `Socket` в случае ошибки генерируют `IOException`, которая должна быть перехвачена и обработана.

`ServerSocket` представляет собой класс, используемый программой сервера для прослушивания запросов клиентов. `ServerSocket` реально не выполняет сервис, но создает объект `Socket` от имени клиента, через который выполняется взаимодействие с сокетом клиента.

Для создания и инициализации объектов `ServerSocket` используются следующие конструкторы, определенные в классе `ServerSocket`:

- `public ServerSocket(int port_number)`: Создает сокет сервера на заданный порт на локальной машине. Клиентам следует использовать этот порт, чтобы общаться с сервером. Если номер порта 0, то сокет сервера создается на любой свободный порт локальной машины.
- `public ServerSocket(int port, int backlog)`: Создает сокет сервера на заданный порт на локальной машине. Второй параметр задает максимальное количество соединений клиентов, которые сокет сервера поддерживает на заданном порту.

- `public ServerSocket(int port, int backlog, InetAddress bindAddr):` Создает сокет сервера на заданный порт. Третий параметр используется для создания сокета сервера хоста, подключенного к нескольким физическим линиям (multi-homed host). Сокет сервера принимает запросы клиента только с заданных IP адресов.

Некоторые полезные методы класса `ServerSocket` представлены в Таблице 5.

Таблица 5.

Метод	Описание
<code>public InetAddress getInetAddress()</code>	Возвращает объект <code>InetAddress</code> , который содержит адрес объекта <code>ServerSocket</code> .
<code>public int getLocalPort()</code>	Возвращает номер порта, с которого объект <code>ServerSocket</code> слушает запросы клиента.
<code>public Socket accept() throws IOException</code>	Заставляет сокет сервера слушать соединение клиента и принимать его. После установления соединения клиента с сервером метод возвращает сокет клиента.
<code>public void bind(SocketAddress address) throws IOException</code>	Связывает объект <code>ServerSocket</code> с заданным адресом (IP адрес и порт). Этот метод вызывает исключительную ситуацию <code>IOException</code> , когда происходит ошибка.
<code>public void close() throws IOException</code>	Закрывает объект <code>ServerSocket</code> . Этот метод вызывает исключительную ситуацию <code>IOException</code> , когда происходит ошибка.
<code>public String toString()</code>	Возвращает IP адрес и номер порта сокета сервера как <code>String</code> .

Создание сервера TCP/IP

Процедура разработки сервера состоит в создании объекта класса `ServerSocket`, который "слушает" клиентские запросы на установление соединения с определенного порта. Когда сервер распознает допустимый запрос, объект `ServerSocket` получает объект `Socket`, созданный клиентом. Взаимодействие между сервером и клиентом происходит с использованием этого сокета.

Класс `ServerSocket` пакета `java.net` используется для создания объекта, с помощью которого сервер слушает запросы удаленного входа. Класс `BufferedInputStream` управляет передачей данных от клиента к

серверу, а класс `PrintStream` управляет передачей данных от сервера к клиенту.

Метод `accept()` ожидает соединения клиента, прослушивая порт, с которым он связан. Когда клиент пытается соединиться с сокетом сервера, метод принимает соединение и возвращает сокет клиента, после чего этот сокет используется клиентом для общения с сервером. Выходной поток этого сокета является входным потоком для связанного клиента и наоборот, входной поток сокета является выходным для сервера. Исключение `IOException` генерируется, если происходит какая-либо ошибка во время установления соединения. Java заставляет обрабатывать возникающие исключительные ситуации.

Для создания серверного приложения сокета TCP, необходимо выполнить следующие шаги:

- Создать объект сокета сервера `ServerSocket`.
- Прослушивать запросы клиента на соединение.
- Запустить сервер.
- Создать поток соединения для запросов клиентов.

1. Создание Server

Класс `Server` должен расширять класс `Thread` и, таким образом, поддерживать многопоточное выполнение. Объект `ServerSocket` "слушает" запросы клиентов и конструктор класса `Server` создает объект `ServerSocket`. Сообщение об ошибке отображается, если возникает исключительная ситуация при запуске сервера.

Фрагмент кода для конструктора сервера выглядит следующим образом:

```
public Server()
{
    try
    {
        serverSocket = new ServerSocket(1001);
    }
    catch(IOException e)
    {
        fail(e, "Не могу запустить сервер.");
    }
    System.out.println("Сервер запущен. . .");
    this.start();    // Запускается поток
}
```

В приведенном фрагменте кода используется общий метод обработки ошибок `fail()`, который обеспечивает обработку всех исключительных ситуаций. Метод принимает два аргумента (объект

Exception и объект String) и выводит сообщение об ошибке. Фрагмент кода для метода fail() выглядит следующим образом:

```
public static void fail(Exception e, String str)
{
    System.out.println(str + "." + e);
}
```

2. Прослушивание запросов клиентов

Метод run() сервера, как любой поток, который реализует интерфейс Runnable, содержит инструкции для потока. В этом случае сервер переходит в бесконечный цикл и прослушивает запросы клиентов. Когда сервер обнаруживает подключение клиента, метод accept() класса ServerSocket выполняет соединение. При этом сервер создает объект класса Connection для клиента. Объект класса Socket передается конструктору класса Connection и взаимодействие между клиентом и сервером выполняется через этот сокет. Фрагмент кода для метода run() выглядит следующим образом:

```
public void run()
{
    try
    {
        while(true)
        {
            Socket client = serverSocket.accept();
            Connection con = new Connection(client);
        }
    }
    catch(IOException e)
    {
        fail(e, "Не прослушивается");
    }
}
```

3. Запуск сервера

Фрагмент кода для метода main() приведен ниже.

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

В этом фрагменте кода создается объект класса Server, который запускает поток.

4. Создание потокового соединения

Следующий фрагмент кода описывает класс Connection:

```
class Connection extends Thread
{
```



```

protected Socket netClient;
protected BufferedReader fromClient;
protected PrintStream toClient;
public Connection(Socket client)
{
    netClient = client;
    try
    {
        fromClient = new BufferedReader(new
InputStreamReader(netClient.getInputStream()));
        toClient = new PrintStream(netClient.getOutputStream());
    }
    catch(IOException e)
    {
        try
        {
            netClient.close();
        }
        catch(IOException e1)
        {
            System.err.println("Unable to set up streams"
+ e1);
        }
        return;
    }
    this.start();
}
public void run()
{
    String clientMessage;
    try
    {
        for(;;)
        {
            clientMessage = fromClient.readLine();
            if(clientMessage == null)
                break;
            // Посылает подтверждение клиенту
            toClient.println("Received");
        }
    }
    catch(IOException e)
    {}
    finally
    {
        try
        {
            netClient.close();
        }
        catch(IOException e)
        {}
    }
}
}
}

```

В представленном фрагменте кода класс Connection создает объект fromClient класса BufferedReader, который получает ввод от клиента, используя метод getInputStream(). Объект класса PrintStream (toClient) дает возможность серверу отправлять

клиенту данные, используя метод `getOutputStream()`. Таким образом, возникают необходимые (прием и передача) возможности взаимодействия.

Когда клиент соединяется с сервером, сервер использует метод `readLine()` объекта `fromClient`, чтобы запомнить сообщение, посланное клиентом в переменной `clientMessage` типа `String`. Метод `println()` используется для вывода сообщения "Received" сокету.

Для выхода из системы сервер завершает цикл. При этом выполняется блок `finally` для закрытия сокета клиента. Закрытие сокета является важным действием, поскольку сохранение активного соединения неизбежно приводит к потере памяти сервера. Блок `finally` обеспечивает закрытие ранее установленного соединения. Следует отметить, что сервер является многопоточным, и каждый клиент получает свой собственный поток на сервере.

Следующий код используется для создания класса `Server`, который принимает запросы соединения клиента и посылает строку `Login`, как ответное сообщение клиенту:

```
/* Программа для реализации простого клиентского сервера */
import java.io.*;
import java.net.*;

public class Server extends Thread {
    ServerSocket serverSocket; // Определяется переменная serverSocket
    public Server() {
        try {
            /*
             * Создание объекта ServerSocket, который принимает
запросы
             * соединения от клиентов от порта 1001
             */
            serverSocket = new ServerSocket(1001);
            System.out.println(serverSocket.toString());
        } catch (IOException e) {
            fail(e, "Could not start server.");
        }
        System.out.println("Server is running . . .");
        /* Стартует поток */
        this.start();
    }
    public static void fail(Exception e, String str) {
        System.out.println(str + "." + e);
    }
    public void run() {
        try {
            while (true) {
                /* Принимаются запросы от клиентов */
                Socket client = serverSocket.accept();
                /*
```



```

}
catch(UnknownHostException e)
{
System.err.println("Неопределенное имя хоста ");
System.exit(1);
}

```

В предыдущем фрагменте кода IP адрес равный '127.0.0.1' и порт равный '1001' определяют сокет, на котором сервер прослушивает запросы клиента.

2. Чтение и запись в сокет

После установления соединения между клиентом и сервером, клиент посылает запрос на сервер через сокет. Чтение и запись в сокет аналогичны чтению из файла и записи в файл. Чтобы обеспечить возможность клиенту общаться с сервером, необходимо выполнить следующие действия:

- Объявляются два объекта по одному для классов `PrintStream` и `BufferedReader`. Эти объекты будут использоваться для чтения и записи в сокет `socket`.

```

PrintStream out = null;      // Объект для записи в сокет
BufferedReader in = null;   // Объект для чтения из сокета

```

- Объекты `PrintStream` и `BufferedReader` связываются с сокетом.

```

out = new PrintStream(clientSocket.getOutputStream());
in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

```

Методы `getInputStream()` и `getOutputStream()` класса `Socket` позволяют клиенту взаимодействовать с сервером. Метод `getInputStream()` позволяет объекту `BufferedReader` читать из сокета, а метод `getOutputStream()` позволяет объекту `PrintStream` писать в сокет.

Объявляется еще один объект класса `BufferedReader` для связи со стандартным входом, чтобы данные, введенные в приложении клиенте, могли передаваться на сервер. Следующий фрагмент кода используется для чтения данных из окна консоли:

```

BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
String str;
while((str = stdin.readLine()).length() != 0)
{
out.println(str);
}

```

Представленный фрагмент кода позволяет пользователю вводить данные с клавиатуры. Цикл `while` продолжается до тех пор, пока пользователь не введет символ завершения ввода (`Ctrl-Z`).

3. Заккрытие соединения

Операторы, представленные ниже, закрывают потоки и соединения с сервером.

```
out.close();
in.close();
stdin.close();
```

Сокет клиента принимает имя пользователя и пароль, и обеспечивает связь. Для завершения соединения, пользователь должен ввести 'Bye'. Можно использовать следующий код для создания класса Client:

```
/* Программа для реализации простого клиентского сокета */
import java.net.*;
import java.io.*;

public class Client {
    public static void main(String[] args) throws IOException {
        Socket clientSocket;
        PrintStream out = null;
        BufferedReader in = null;
        try {
            /* Создается объект сокета, чтобы соединиться с сервером */
            clientSocket = new Socket("127.0.0.1", 1001);
            /* Создается выходной поток, чтобы посылать данные на сервер */
            out = new
PrintStream(clientSocket.getOutputStream());
            /* Создается входной поток, чтобы принимать данные с сервера */
            in = new BufferedReader(new InputStreamReader(
                clientSocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("Unidentified hostname ");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O ");
            System.exit(1);
        }
        /* Создается входной поток, чтобы читать данные из окна консоли */
        BufferedReader stdin = new BufferedReader(new
InputStreamReader(
                (System.in)));
        /* Чтение из сокета */
        String login = in.readLine();
        System.out.println(login);
        /* Прием login */
        String logName = stdin.readLine();
        out.println(logName);
        /* Чтение из сокета */
        String password = in.readLine();
        System.out.println(password);
        /* Прием password */
        String pass = stdin.readLine();
```

```

        out.println(pass);
        String str = in.readLine();
        System.out.println(str);
        while ((str = stdin.readLine()) != null) {
            out.println(str);
            if (str.equals("Bye"))
                break;
        }
        out.close();
        in.close();
        stdin.close();
    }
}

```

Представленный выше код сохраняется как Client.java. При выполнении класса Client, отображается сообщение для ввода Login. После приема Login на консоли класса Client появляется сообщение для ввода password. Рис. 7 показывает вывод класса Client, который посылает login Anatoly и password для входа на Server.

```

C:\Windows\system32\cmd.exe - java Client
F:\eclipse\workspace\ForBook\bin>java Client
Login:
Anatoly
Password:
password
Login:

```

Рис. 7. Вывод из приложения Client.

Введенный login в окне консоли класса Client передается классу Server. Рис. 8 показывает вывод класса Server при получении login Anatoly от класса Client.

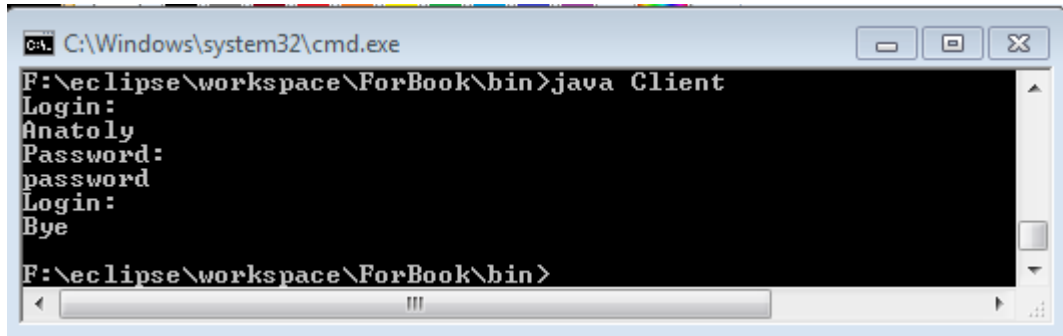
```

C:\Windows\system32\cmd.exe - java Server
F:\eclipse\workspace\ForBook\bin>java Server
ServerSocket [addr=0.0.0.0/0.0.0.0, port=0, localport=1001]
Server is running . . .
Anatoly logged in

```

Рис. 8. Получение Login от Client.

Когда на сервер передается строка «Bye» в качестве login от клиента, соединение между клиентом и сервером прекращается. Рис. 9 показывает вывод класса Server при передаче строки «Bye» от класса Client.



```
C:\Windows\system32\cmd.exe
F:\eclipse\workspace\ForBook\bin>java Client
Login:
Anatoly
Password:
password
Login:
Anatoly
Bye
F:\eclipse\workspace\ForBook\bin>
```

Рис. 9. Client завершает соединение с Server.

Разработка потокового взаимодействия в IDE Eclipse

Постановка задачи

Необходимо разработать клиент/серверное приложение, в котором сервер слушает запросы клиентов на порт 1500 и отправляет объект-сообщение, содержащий текущую дату/время сервера и строку сообщения. Пользователь-клиент должен иметь возможность просмотра полученного сообщения.

Для решения поставленной задачи необходимо выполнить следующие шаги:

1. Создать класс `DateMessage` с двумя полями: `Date` и `String` – для хранения и передачи сообщения клиенту.
2. Реализовать класс сервера для прослушивания соединений на порту 1500 и отправки сообщений. Задача класса сервера должна выполняться в отдельном потоке.
3. Реализовать класс клиента для получения и просмотра сообщений/
4. Протестировать приложение – запустить сервер и клиент, и проверить передачу и получение сообщения.

Создание класса `DateMessage`

1) Создайте новый Java-класс `DateMessage` в пакете `ru.ifmo.time`.

2) Скопируйте следующее содержимое класса:

```
package ru.ifmo.time;
```

```
import java.io.Serializable;
import java.util.Date;
public class DateMessage implements Serializable {
    private Date date;
    private String message;
    public DateMessage(Date date, String message) {
        this.date = date;
        this.message = message;
    }
}
```



```

    public Date getDate() {
        return date;
    }
    public void setDate(Date date) {
        this.date = date;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
}

```

Создание класса ServerTCP

Класс ServerTCP должен выполнять следующие задачи:

1. Создать серверный сокет на основе класса ServerSocket.
2. Ожидать запрос от клиента с помощью метода accept() серверного сокета.
3. Сформировать объект-сообщение и отправить его с помощью выходного потока клиентского сокета.

1) Создайте новый Java-класс ServerTCP в пакете ru.ifmo.time.

Код класса Server приведен ниже:

```

package ru.ifmo.time;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Calendar;
/**
 * Класс сервера (выполняется в отдельном процессе)
 */
public class ServerTCP extends Thread {
    // Объявляется ссылка
    // на объект - сокет сервера
    ServerSocket serverSocket = null;
    /**
     * Конструктор по умолчанию
     */
    public ServerTCP() {
        try {
            // Создается объект ServerSocket, который получает
            // запросы клиента на порт 1500
            serverSocket = new ServerSocket(1500);
            System.out.println("Starting the server ");
            // Запускаем процесс
            start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
/**

```

```

    * Запуск процесса
    */
    public void run() {
        try {
            while (true) {
                // Ожидание запросов соединения от клиентов
                Socket clientSocket = serverSocket.accept();

                System.out.println("Connection accepted from "
+ clientSocket.getInetAddress().getHostAddress());

                // Получение выходного потока,
                // связанного с объектом Socket
                ObjectOutputStream out =
new ObjectOutputStream(
clientSocket.getOutputStream());

                // Создание объекта для передачи клиентам
                DateMessage dateMessage = new DateMessage(
                    Calendar.getInstance().getTime(),
                    "Текущая дата/время на сервере");
                // Запись объекта в выходной поток
                out.writeObject(dateMessage);
                out.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public static void main(String args[]) {
        // Запуск сервера
        new ServerTCP();
    }
}

```

Создание класса ClientTCP

Класс ClientTCP позволяет клиенту присоединиться к серверу, используя его IP-адрес (в нашем случае localhost), и получить от него сообщение. Классу ClientTCP должен выполнять следующие основные задачи:

1. Создать сокет для доступа к серверу localhost на порт 1500.
 2. Получить входной поток сокета.
 3. Получить объект-сообщение из потока и отобразить полученные данные.
- Создайте новый Java-класс ClientTCP в пакете ru.itmo.time.

Код класса Client приведен ниже:

```

package ru.ifmo.time;

import java.io.ObjectInputStream;
import java.net.Socket;

public class ClientTCP {

```

```

public static void main(String args[]) {
    try {
        // Создается объект Socket
        // для соединения с сервером
        Socket clientSocket = new Socket("localhost", 1500);
        // Получаем ссылку на поток, связанный с сокетом
        ObjectInputStream in = new
ObjectInputStream(clientSocket.getInputStream());
        // Извлекаем объект из входного потока
        DateMessage dateMessage =
(DateMessage) in.readObject();
        // Выводим полученные данные на консоль
        System.out.println(dateMessage.getMessage());
        System.out.println(dateMessage.getDate());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Запуск и тестирование

Щелкните правой кнопкой мыши на класс ServerTCP в окне Package Explorer и выберите команду Run As/Java Application. На консоли отображается сообщение Starting the server.

Проделайте то же самое с классом ClientTCP. При запуске клиент пытается соединиться с сервером и обрабатывает полученное сообщение. В результате в консоли клиента выводится сообщение, как показано на Рис. 10.

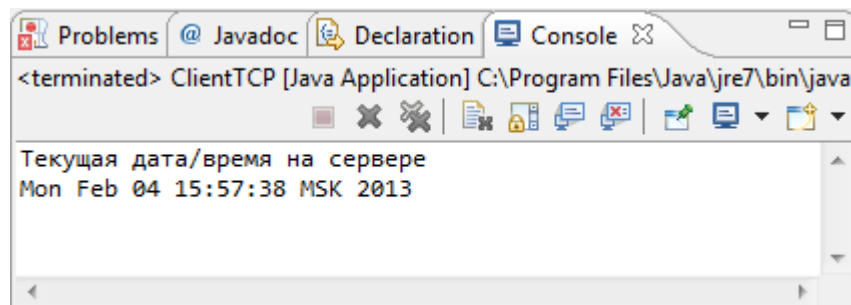


Рис. 10. Вывод сообщения в консоли клиента.

Выберите консоль сервера и посмотрите сообщение о приеме соединения от клиента, как показано на Рис. 11.

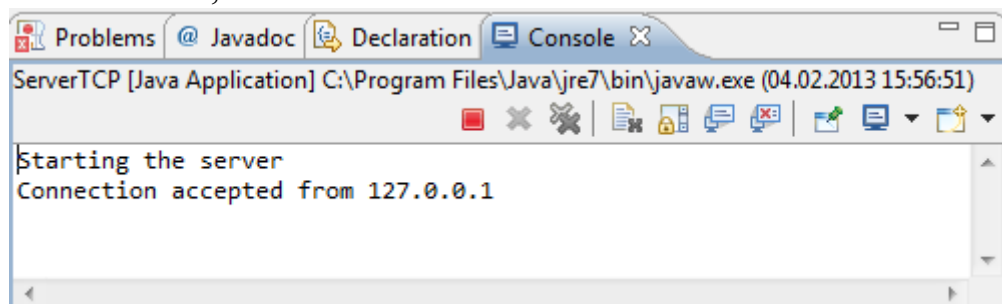


Рис. 11. Вывод сообщения о приеме соединения от клиента.

Вопросы для самопроверки

1. Какой пакет содержит классы для создания сетевых приложений TCP/IP?
2. Какой класс обеспечивает функциональность для создания сокетов для клиента?
3. Какой класс обеспечивает функциональность для создания сокетов для сервера?
4. Перечислите широко используемые методы класса Socket.
5. Перечислите широко используемые методы класса ServerSocket.

Введение в RMI

Введение в распределенные приложения

В настоящее время существуют три различных подхода к разработке приложений: традиционный подход, клиент/серверный подход и компонентный подход.

При использовании традиционного подхода единственное приложение управляет логикой представления, логикой обработки и взаимодействием с базой данных (обычно в виде набора файлов). База данных также расположена на локальном компьютере. Такие приложения называются также монолитными и характеризуются тем, что даже при незначительной модификации, расширении и развитии приложения, приложение должно перекомпилироваться и собираться вновь. Например, незначительные изменения в структуре базы данных может потребовать изменения всех функций и методов приложения. Такой подход делает очень трудоемкой процедуру изменения, распространения и сопровождения приложения.

С целью устранения недостатков традиционного подхода была внедрена клиент/серверная архитектура (также называемая двухслойная (two-tier) архитектура). В такой архитектуре данные отделяются от клиентской части, хранятся централизованно на сервере и к ним предоставляется доступ по технологии клиент/сервер. При этом логика обработки объединяется со слоем представления либо на стороне клиента, либо на стороне сервера, которая содержит код для связи с базой данных. Если логика обработки объединяется со слоем представления, то клиент называется "толстым", а если логика обработки объединяется с сервером базы данных, то сервер называется "толстым".

Однако и клиент/серверная архитектура также имеет определенные недостатки, а именно:

- Любое изменение бизнес-логики требует изменений в алгоритмах обработки. При изменении алгоритмов обработки либо слой представления, либо код доступа к базе данных нуждается в изменении, в зависимости от места расположения бизнес-логики.
- Реализованные приложения, использующие двухслойную архитектуру, могут трудно масштабироваться из-за ограниченного количества доступных для клиента соединителей с базой данных. Запросы соединения, превышающие определенное количество, просто отбрасываются сервером.

Недостатки клиент/серверной архитектуры устраняются в трехслойной архитектуре, в которой логика представления располагается на стороне клиента, доступ к базе данных контролируется на стороне сервера, а логика обработки находится между этими двумя слоями. Слой логики обработки относится к серверу приложений (также называемый средним слоем трехслойной компонентной архитектуры). Этот тип архитектуры называется серверо-центрическим, поскольку он дает возможность компонентам приложения выполняться на среднем слое сервера приложений, применяющего правила обработки независимо от интерфейса представления и реализации базы данных. Эти компоненты могут быть разработаны, используя любой язык программирования, который позволяет создание компонентов. Компоненты могут быть централизованы для упрощения разработки, поддержки и развертывания. Так как средний слой управляет логикой обработки, нагрузка распределяется между клиентом, сервером базы данных и сервером, управляющим логикой обработки. Эта архитектура также обеспечивает эффективный доступ к данным. Проблема с ограничением соединений базы данных минимизируется, поскольку база данных видит только слой логики обработки и не всех ее клиентов. При этом в случае двухслойного приложения соединение базы данных устанавливается заблаговременно и поддерживается, в то время как в трехслойном приложении (Рис. 12) соединение устанавливается только когда требуется доступ к данным и закрывается, как только данные получены или переданы на сервер.



Рис. 12. Архитектура распределенного приложения.

Приложения, в которых логика представления, логика обработки и база данных размещаются на нескольких компьютерах, называется распределенными (distributed) приложениями.

Вызов удаленного метода

Кроме рассмотренного взаимодействия процессов (программ) с использованием сокетов существует еще один способ взаимодействия - вызов удаленного метода (Remote Method Invocation - RMI). Вызов удаленных методов является реализацией идей (Remote Procedure Call - RPC) для языка программирования Java. Идея вызова удаленных процедур состоит в расширении механизма передачи управления и данных внутри программы, выполняющейся на одной машине, на передачу управления и данных через сеть. То есть, клиентское приложение обращается к процедурам, хранящимся на сервере. Средства удаленного вызова процедур предназначены для облегчения организации распределенных вычислений. Наибольшая эффективность использования RPC достигается в тех приложениях, в которых существует интерактивная связь между удаленными компонентами с небольшим временем ответов и относительно малым количеством передаваемых данных. Такие приложения называются RPC-ориентированными.

RMI представляет собой спецификацию, которая дает возможность одной виртуальной Java-машине - Java Virtual Machine (JVM), вызывать методы в объектах, расположенных в другой JVM. Эти две JVM могут быть запущены на различных компьютерах или выполняться в отдельных процессах одного компьютера. RMI применяется на среднем слое трехслойной архитектуры, облегчая возможность программистам вызывать распределенные компоненты в сетевой среде.

Sun Microsystems разработала поддержку RMI как упрощенную альтернативу сложному кодированию, связанному с программированием сокета сервера. Для использования RMI, программисту нет необхо-

димости знать программирование сокета или организовывать многопоточное выполнение, что позволяет ему концентрироваться на реализации логики обработки.

Компоненты приложения RMI

Распределенное RMI приложение содержит два компонента:

- RMI-сервер
- RMI-клиент

RMI-сервер включает объекты, методы которых должны вызываться клиентами удаленно. Сервер создает несколько удаленных объектов и делает ссылку на эти объекты в регистре RMI. Регистр RMI является выполняемым сервисом, который выполняется на сервере RMI. Удаленные объекты, созданные сервером, регистрируются в регистре с помощью уникального имени. Клиент получает удаленную ссылку на один или несколько удаленных объектов из регистра, просматривая имена объектов, и затем клиент вызывает методы на удаленном объекте(ах), чтобы получить доступ к сервисам удаленного объекта(ов). Таким образом, RMI обеспечивает механизм, при котором сервер и клиент связываются и передают информацию в обоих направлениях. Рис. 13 иллюстрирует функциональность приложений в RMI.



Рис. 13. Функциональность приложений в RMI.

Однажды получив ссылку на удаленный объект, в дальнейшем методы в удаленном объекте вызываются точно так же, как методы локального объекта.

Одной из центральных достоинств RMI является возможность загрузки определения класса объекта в ситуации, когда загружаемый класс не представлен в виртуальной машине клиента. Вся функциональность объекта, доступная только на одной JVM, может быть передана в другую JVM, расположенную удаленно. RMI доставляет объекты в соответствии с шаблоном, определенным классом, следовательно, не

меняется их поведение при пересылке. Эта особенность позволяет динамически определять и расширять функциональность клиентского приложения путем использования поведенческих возможностей серверного приложения.

Архитектура RMI

Архитектура RMI состоит из трех уровней (Рис. 14.):

- Уровень стаб/скелетон (Stub/Skeleton)
- Уровень удаленной ссылки (Remote Reference Layer)
- Транспортный уровень

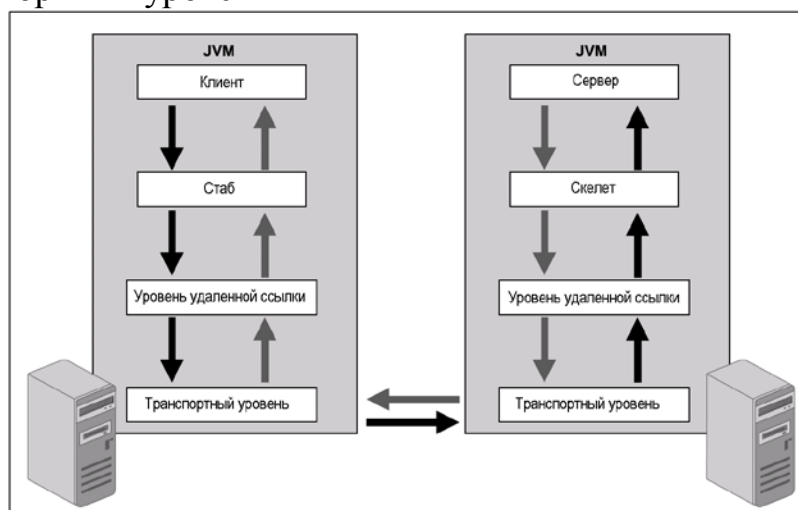


Рис. 14. Архитектура RMI.

Уровень стаб/скелетон прослушивает вызовы удаленных методов, сделанные клиентом, и направляет эти вызовы на удаленные RMI сервисы на сервере. Этот уровень состоит из стаб и скелетон.

Запрос метода удаленного объекта на клиентской стороне начинается со стаб. Стаб выступает как прокси (проху) на стороне клиента, представляющего удаленный объект. (Прокси, по определению, является сетевой службой (процессом), позволяющей клиентам выполнять косвенные запросы к другим сетевым службам). К нему обращаются программы, как к любому другому локальному объекту, выполняющемуся на клиенте, и он (стаб) обеспечивает доступ к методам удаленного объекта. Стаб связывается с удаленным методом объекта через скелетон, который реализуется на сервере.

Скелетон выступает как прокси на стороне сервера, который продолжает общение со стаб посредством:

- чтения параметров для вызова метода
- выполнения вызова удаленного сервиса объекта реализации
- получения возвращаемого значения

- записи возвращенного значения обратно на стаб.

Уровень удаленной ссылки

Уровень удаленной ссылки интерпретирует и управляет ссылками, сделанными клиентами к удаленному объекту на сервере. Этот уровень представлен как на клиенте, так и на сервере. RRL на стороне клиента принимает запросы о методах от стаб, которые передаются, как упакованный поток данных на RRL на стороне сервера. Упаковка (marshalling) представляет собой процесс, в котором параметры, переданные клиентом, преобразуются в формат, который может быть передан по сети. RRL на стороне сервера выполняет распаковку (unmarshaling) параметров, которые посланы удаленному методу через скелетон. Распаковка представляет собой процесс, в котором ранее упакованные параметры, переданные клиентом RRL клиентской стороне, преобразуются в формат, который скелетон понимает. Во время возвращения значений от скелетона, данные вновь обрабатываются маршallingом и передаются клиенту через RRL со стороны сервера.

Транспортный уровень

Транспортный уровень является связующим звеном между RRL на стороне сервера и RRL на стороне клиента. Транспортный уровень отвечает за создание новых соединений и управление существующими соединениями. Он также отвечает за управление удаленными объектами, которые находятся в адресном пространстве транспортного уровня. Следующие операции поясняют процедуру соединения клиента с сервером:

- При получении запроса от RRL на стороне клиента, транспортный уровень устанавливает соединение сокета с сервером через RRL на стороне сервера.
- Затем, транспортный уровень передает установленное соединение с RRL на стороне клиента и добавляет ссылку на соединение в свою таблицу.

Пакеты RMI

Ниже представлены пакеты RMI, состоящие из различных классов и интерфейсов, предназначенных для создания и выполнения различных распределенных приложений, использующих RMI:

- Пакет `java.rmi` обеспечивает удаленный интерфейс, класс для доступа к удаленным именам, зарегистрированным на сервере и менеджер безопасности для RMI.
- Пакет `java.rmi.registry` обеспечивает классы и интерфейсы, которые используются удаленным регистром.
- Пакет `java.rmi.server` обеспечивает классы и интерфейсы, которые используются для реализации удаленных объектов, стабы и скелетоны, а также поддержку RMI связи.
- Пакет `java.rmi.dgc` обеспечивает классы и интерфейсы, которые используются распределенным сборщиком мусора RMI (RMI-distributed garbage collector).

Пакет `java.rmi`

Пакет `java.rmi` объявляет интерфейс `Remote` и классы `Naming` и `RMISecurityManager`. Он также содержит ряд классов исключительных ситуаций, которые используются с RMI.

Все удаленные объекты должны реализовать интерфейс `Remote`. Этот интерфейс не содержит методов и используется для идентификации удаленных объектов. Пакет `java.rmi` состоит из следующих классов:

- Класс `Naming`: Содержит статические методы для доступа удаленных объектов через URL. Этот класс поддерживает следующие методы:
 - `rebind()`: Связывает имя удаленного объекта с заданным URL и обычно используется объектом сервера.
 - `unbind()`: Удаляет связь между именем объекта и URL.
 - `lookup()`: Возвращает удаленный объект, заданный URL и обычно используется объектом клиента.
 - `list()`: Возвращает список URL, которые известны регистру RMI.
- Класс `RMISecurityManager`: Определяет политику безопасности по умолчанию для удаленного объекта стаб. Политика применяется только для приложений. Апплеты используют класс `AppletSecurityManager` для RMI. Метод `setSecurityManager()` класса `System` используется для установки объекта `RMISecurityManager` в качестве менеджера безопасности, который должен быть использован для стаб RMI.

Пакет `java.rmi` определяет набор исключительных ситуаций. Класс `RemoteException` является родительским для всех исключительных ситуаций, которые генерируются во время RMI.

Удаленным объектам, которые доступны локально, нет необходимости вызывать исключительную ситуацию `RemoteException`.

Пакет `java.rmi.registry`

Пакет `java.rmi.registry` содержит интерфейсы `Registry` и `RegistryHandler` и размещается в регистре. Эти интерфейсы используются для регистрации и доступа к удаленным объектам по имени. Объекты `Remote` регистрируются, когда они связываются с процессом регистрации хоста. Процесс регистрации запускается, когда выполняется команда `start rmiregistry`. Команда определяет методы `rebind()`, `unbind()`, `list()` и `lookup()`, которые используются классом `Naming` для связи имен и URL ссылок RMI.

Пакет `java.rmi.server`

Пакет `java.rmi.server` реализует интерфейсы и классы, которые поддерживают клиентскую и серверную части RMI. Пакет `java.rmi.server` состоит из следующих классов и интерфейсов:

- Класс `RemoteObject`: Реализует интерфейс `Remote` и обеспечивает удаленную реализацию класса `Object`. Все объекты, которые реализуют удаленные объекты, расширяют класс `RemoteObject`.
- Класс `RemoteServer`: Расширяет класс `RemoteObject` и является общим классом, который является подклассом конкретных типов реализации удаленного объекта.
- Класс `UnicastRemoteObject`: Расширяет класс `RemoteServer` и обеспечивает реализацию `RemoteObject` по умолчанию. Классы, которые реализуют `RemoteObject` обычно подклассы объекта `UnicastRemoteObject`. Путем расширения `UnicastRemoteObject`, подкласс может использовать по умолчанию RMI для общения на основе транспортного сокета. Класс выполняется все время.
- Класс `RemoteStub`: Обеспечивает абстрактную реализацию стаба на стороне клиента. Статический метод `setRef()` используется, чтобы связать стаб на стороне клиента с соответствующими удаленными объектами.

- Интерфейс `RemoteCall`: Обеспечивает методы, которые используются всеми стаб и скелетонами в RMI.
- Интерфейс `Skeleton`: Обеспечивает метод для получения доступа к методам удаленного объекта и этот интерфейс реализуется удаленными скелетонами.
- Интерфейс `Unreferenced`: Дает возможность определить, когда клиент больше не ссылается на удаленный объект. Этот интерфейс реализуется классом `Remote`.

Пакет `java.rmi.dgc`

Пакет `java.rmi.dgc` содержит классы и интерфейсы, которые используются распределенным сборщиком мусора. Серверная сторона распределенного сборщика мусора реализует этот интерфейс. Этот пакет содержит:

- Класс `Lease`, который создает объекты, используемые для отслеживания ссылок объектов.
- Метод `dirty()`, который используется, чтобы отмечать, что клиент ссылается на удаленный объект.
- Метод `clean()`, который используется, чтобы отмечать, что удаленная ссылка завершилась.

Распределенная сборка мусора

Java обеспечивает распределенный механизм сборки мусора, который автоматически удаляет объекты, не используемые клиентами RMI. RMI использует алгоритм сборки мусора методом подсчета числа ссылок для управления ссылками на удаленные объекты. Уровни стаб и скелет используют интерфейс `java.rmi.dgc` для реализации механизма распределенной сборки мусора. Удаленный объект, который реализует интерфейс `java.rmi.server.Unreferenced` получает извещение, когда ссылки клиента больше не существует. Когда не существует локальной или удаленной ссылки на удаленный объект, выполняется сборка мусора.

Когда какой-либо новый удаленный объект обращается в JVM, выполняющийся объект RMI класса `Lease` (аренды) присваивает единицу счетчику ссылок на объект и отмечает удаленный объект как измененный. Аналогично, когда существующий удаленный объект удаляется из JVM, объект класса `Lease` уменьшает счетчик на единицу и отмечает объект как "чистый". В механизме подсчета ссылок удаленная

ссылка может быть не активной в JVM в течение выделенного времени. Системное свойство `java.rmi.dgc.leaseValue` устанавливает выделенное время. Если ссылка не возобновляется соединением к удаленному объекту в течение завершения выделенного времени, RMI перераспределяет память, выделенную удаленному объекту, используя механизм распределенной сборки мусора. Выделенное время устанавливается в миллисекундах и по умолчанию равно 10 минутам.

Этапы создания распределенного приложения

Распределенное приложение, использующее RMI, содержит несколько компонентов, в том числе, файл интерфейса, файл сервера и файл клиента. В соответствии со спецификацией RMI, необходимо выполнять определенную последовательность шагов для создания распределенного приложения RMI, а именно:

- создать удаленный интерфейс;
- реализовать удаленный интерфейс;
- создать сервер RMI;
- создать клиент RMI;
- выполнить приложение RMI.

Создание удаленного интерфейса

Удаленный интерфейс определяет конкретные методы, которые могут быть удаленно вызваны клиентом. Удаленный интерфейс должен объявить каждый из методов, которые должны быть вызваны из других JVM. Удаленные интерфейсы имеют следующие характеристики:

- Удаленный интерфейс должен быть объявлен `public`. Это делается потому, что в большинстве приложений, клиент не является частью того же самого пакета, что и удаленный интерфейс.
- Удаленный интерфейс расширяет интерфейс `java.rmi.Remote`.
- Каждый метод должен объявлять `java.rmi.RemoteException` в свои собственные утверждения возбуждения для обработки сетевых проблем соединения сервера.

Следующий фрагмент кода может быть использован для определения удаленного интерфейса, который представляет удаленный объект:

```
import java.rmi.*;  
public interface Hello extends Remote
```

```

{
    /* Объявляется удаленный метод */
    public String sayHello() throws RemoteException;
}

```

Здесь интерфейс Hello объявляет метод sayHello(), который активирует RemoteException.

Реализация удаленного интерфейса

Далее необходимо реализовывать удаленный интерфейс, чтобы создать удаленный класс сервера, который обеспечивает информацию об объектах сервера. Удаленный сервисный класс определяет все методы, которые объявлены в удаленном интерфейсе. Необходимо импортировать пакеты java.rmi и java.rmi.server. Можно использовать следующий фрагмент кода, чтобы импортировать пакеты Java:

```

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

```

Удаленный сервисный класс расширяет класс UnicastRemoteObject для реализации метода удаленного интерфейса. Класс UnicastRemoteObject расширяет класс RemoteServer в java.rmi и определяет методы класса RemoteServer. Рис. 15 показывает иерархию класса UnicastRemoteObject:

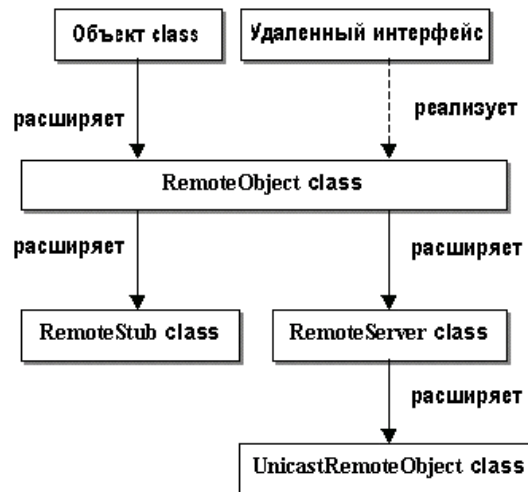


Рис. 15. Иерархия класса UnicastRemoteObject

Можно создать класс HelloImpl, реализующий удаленный интерфейс Hello после импортирования пакетов Java. Следующий фрагмент кода можно использовать для создания удаленного класса сервиса:

```

public class HelloImpl implements Hello extends UnicastRemoteObject
{

```

```

    ..

```

```
}
```

Удаленный экземпляр объекта должен быть экспортирован, что дает возможность удаленному объекту получать доступ к запросам удаленного метода, прослушивая определенный порт. Поскольку класс `HelloImpl` расширяет класс `UnicastRemoteObject`, то он экспортируется автоматически.

Если реализуемый класс уже расширяет класс, отличный от `UnicastRemoteObject`, необходимо явно экспортировать удаленный объект, вызывая метод `UnicastRemoteObject.exportObject()` из конструктора реализации класса

Метод `super()` вызывает конструктор класса `UnicastRemoteObject`, который экспортирует удаленный объект. Следующий фрагмент кода может быть использован для определения конструктора по умолчанию для удаленного сервисного класса:

```
public HelloImpl() throws RemoteException
{
    super();
}
```

Необходимо определить все методы удаленного интерфейса в удаленном сервисном классе после определения конструктора по умолчанию. Следующий фрагмент кода можно использовать для определения удаленного метода в удаленном сервисном классе:

```
public String sayHello() throws RemoteException
{
    return "Hello! Student.";
}
```

Следующий фрагмент кода можно использовать для реализации удаленного интерфейса в удаленном сервисном классе:

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloImpl extends UnicastRemoteObject implements Hello
{
    /* Определяется конструктор по умолчанию */
    public HelloImpl() throws RemoteException
    {
        super();
    }
    /* Определяется удаленный метод */
    public String sayHello() throws RemoteException
    {
        return "Hello! Student.";
    }
}
```

В представленном коде сервисный класс `HelloImpl` содержит реализацию метода `sayHello()`, объявленного в интерфейсе `Hello`.

Создание сервера RMI

Серверный класс RMI должен содержать объекты, которые удаленно вызываются клиентом. Для создания удаленного серверного объекта необходимо создать объект в методе `main()` серверного класса RMI. Так, например, для создания удаленного объекта класса `Hello` можно использовать следующий оператор:

```
Hello h = new HelloImpl();
```

Затем, необходимо выполнить регистрацию серверного объекта в регистре до того, как он должен будет принимать запросы от клиента. При этом имя и ссылка на серверный объект передаются в RMI регистр для регистрации объектов сервера. Имя объекта сервера используется, чтобы получить доступ к объекту стаб, использующему механизм просмотра (`lookup`). Следующий оператор можно использовать для выполнения регистрации объекта сервера в регистре:

```
Naming.rebind("server",h);
```

Метод `rebind()` ожидает два параметра:

- Первый параметр представляет собой строку URL, которая содержит местоположение и имя удаленного объекта. Если порт не специфицирован, регистр RMI использует по умолчанию порт 1099. Если пользователь определяет порт, то строка URL должна иметь следующий вид `"rmi://ip-address:1234/server"`.
- Второй параметр является ссылкой на реализацию объекта.

Метод `setSecurityManager()` класса `SecurityManager` используется в серверном классе для установки менеджера безопасности для приложения RMI, чтобы неавторизованный клиент не смог вызвать серверный объект. Для установки подлинности клиента RMI, необходимо создать файл политики безопасности, который содержит все требуемые разрешения. Следующий код можно использовать для создания сервера RMI для распределенного приложения:

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloServer
{
    public static void main(String args[])
    {
        try
        {
            /* Устанавливается менеджер безопасности */
            System.setSecurityManager(new RMISecurityManager());
            /* Создается удаленный объект */
            Hello h = new HelloImpl();
            /* Удаленный объект связывается с регистром RMI */
            Naming.rebind("server",h);
            System.out.println("Object is registered.");
            System.out.println("Now server is waiting for client request...");
        }
        catch(Exception e)
    }
}
```



```

        {
            System.out.println("Error : "+e);
        }
    }
}

```

В представленном коде, метод `rebind()` класса `Naming` может вызывать удаленную исключительную ситуацию, поэтому метод `rebind()` определяется внутри блока `try-catch`.

Политика безопасности

Если приложение RMI взаимодействует с базой данных или другими ресурсами, ограничивающими доступ, то необходимо создавать политику разрешением полного доступа. Модель безопасности Java обеспечивать разрешенные права для приложений Java, что позволять только разрешенным приложениям получать доступ к ресурсам системы, таким как файлы или память. Следующие разрешения прав поддерживаются моделью безопасности Java, в частности:

`AllPermission`: Обеспечивает все разрешения для приложения.

`AudioPermission`: Позволяет приложению доступ к аудио-ресурсам системы.

`AuthPermission`: Дает возможность конечному пользователю аутентифицировать доступ для приложения.

`AWTPermission`: Позволяет приложению доступ и отображение `Abstract Window Toolkit (AWT)` в приложении.

`FilePermission`: Позволяет получить доступ к файлам или папкам. Можно перемещать, выгружать файлы с определенного места, если имеется `FilePermission` для файла.

`NetPermission`: Обеспечивает различные сетевые разрешения, что позволяет выполнять такие операции как пересылка файлов по сети, используя программирование сокета или RMI.

`PropertyPermission`: Обеспечивает свойства разрешений, такие как чтение и запись.

`ReflectPermission`: Позволяет выполнять т.н. отражающие операции, такие как проверка идентификаторов доступа для класса или объекта. Отражающие операции дают возможность получать информацию о методах приложений.

`RuntimePermission`: Обеспечивает разрешения периода выполнения для приложения Java. `RuntimePermissions` дает возможность приложению выполнять операции загрузки библиотеки класса во время выполнения.

`SecurityPermission`: Обеспечивает различные разрешения прав, такие как доступ к политике безопасности и идентификация объектов в приложении.

`SerializablePermission`: Обеспечивает разрешения сериализации для создания и использования последовательных (`serializable`) объектов в приложении.

`SocketPermission`: Позволяет получать доступ приложениям в сети, используя сокет. `SecurityPermission` также дает приложениям право взаимодействовать с клиентом, имеющим определенный номер порта и имя хоста.

`SQLPermission`: Дает право выполнять SQL-запросы в приложении.

Файл политики безопасности является простым текстом и может быть создан как текстовым редактором так и графической утилитой **Policy Tool**. Файл политики с именем создается с именем *.java.policy* в HOME каталоге, и содержит утверждения, определяющие разрешения для требуемых объектов. Для создания файла политики безопасности необходимо выполнить следующие шаги:

1. Для выполнения утилиты Policy Tool Java в командной строке введите **policytool**. В результате отображается окно Policy Tool представленное на Рис. 16.

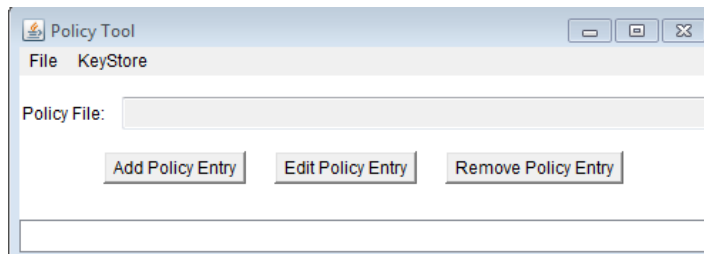


Рис. 16. Инструментальное окно политики.

2. Нажмите кнопку Add Policy Entry (Добавить политику) для добавления нового файла политики безопасности. Рис. 17 показывает диалоговое окно Policy Entry.

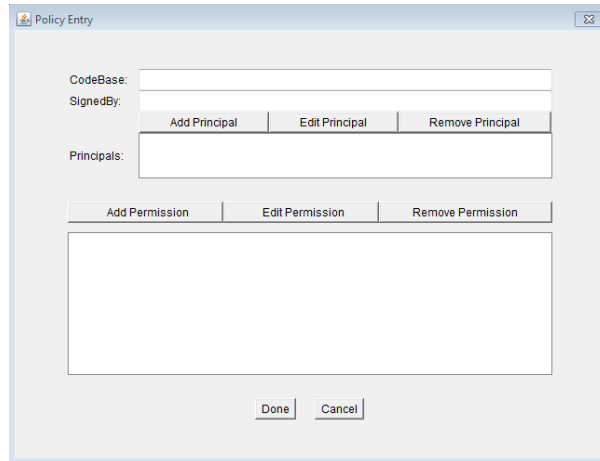


Рис. 17. Диалоговое окно ввода политики.

3. Нажмите кнопку Add Permission (Добавить разрешение), чтобы выдать требуемые разрешения клиентам на получение доступа к ресурсам сервера. Рис. 18 показывает пример диалогового окна Permissions (Разрешения).

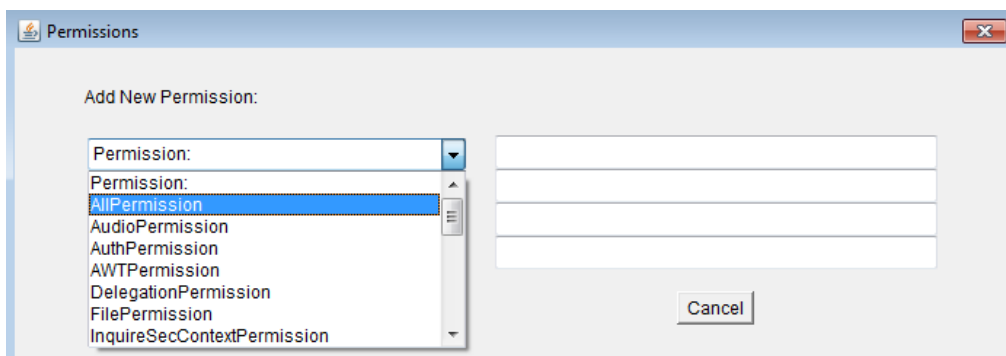


Рис. 18. Диалоговое окно разрешений

4. Выберите AllPermission из списка Permission диалогового окна Permissions.
5. Нажмите кнопку ОК, чтобы закрыть диалоговое окно Permissions.
6. Нажмите кнопку Done диалогового окна Policy Entry.
7. Выберите команду File->Save As в окне Policy Tool.
8. Сохраните файл политики безопасности как **.java.policy** в подкаталоге HOME операционной системы. Сообщение подтверждения отображается в окне сообщения Status. Рис. 19 показывает диалоговое окно сообщения Status.

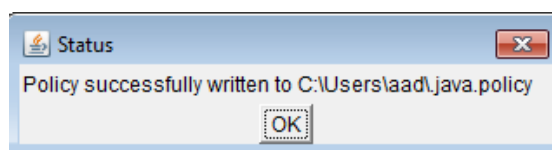


Рис. 19. Диалоговое окно сообщения Status.

9. Нажмите кнопку ОК в окне сообщения Status.

10. Выберите команду File->Exit в окне Policy Tool, чтобы закрыть утилиту Java Policy Tool.

Дополнительная информация по управлению политикой безопасности может быть получена на сайте <http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/policytool.html>.

Создание клиента RMI

Как уже отмечалось, клиент использует объект стаб для получения доступа к удаленному объекту, который находится на сервере. Имя объекта сервера определяется в методе lookup() класса java.rmi.Naming для поиска объекта стаб. Следующий оператор можно использовать для получения доступа к объекту стаб, используя метод lookup():

```
Hello h = (Hello)Naming.lookup("rmi://172.16.28.8/server");
```

В этом операторе метод lookup() может генерировать удаленную исключительную ситуацию, поэтому его следует поместить внутри блока try-catch. Сервер представляет имя серверного объекта. Когда приложение запускается на локальном компьютере, то можно использовать имя localhost вместо IP адреса сервера RMI. Следующий код можно использовать для создания клиента RMI для распределенного приложения:

```
import java.rmi.*;
public class HelloClient
{
    public static void main(String args[])
    {
        try
        {
            /* Поиск удаленного объекта в регистре RMI */
            Hello h = (Hello)Naming.lookup("rmi://127.0.0.1/server");
            System.out.println("Client: Hello!");
            System.out.println("Server: " + h.sayHello());
        }
        catch(Exception e)
        {
            System.out.println("Error : "+e);
        }
    }
}
```

Все исходные файлы Java распределенного приложения Hello компилируются стандартным компилятором Java javac для генерации файлов с расширением class. После компиляции исходных файлов Java, необходимо сгенерировать стаб и скелетон, с помощью которых осуществляется взаимодействие между клиентом и сервером.

Выполнение приложения RMI

Перед выполнением приложения необходимо зарегистрировать объекты сервера в регистре RMI. Клиент может вызывать объекты, зарегистрированные на сервере, через сеть. Для выполнения приложения RMI необходимо выполнить следующие шаги:

1. Сгенерировать стаб и скелетон удаленного сервисного класса.
2. Стартовать регистр RMI.
3. Запустить RMI сервер распределенного приложения.
4. Запустить RMI клиент распределенного приложения.

1. Генерация стаб и скелетон

Компилятор RMI **rmic** компилирует класс удаленного сервиса, который реализуют удаленный интерфейс, и генерирует стаб и скелетон. Стаб позволяет клиенту общаться с определенным удаленным объектом. Скелетон представляет клиенту объект, который размещается на удаленном хосте. Следующая команда позволяет сгенерировать стаб и скелетон:

```
rmic [опции] <ClassFile>
```

ClassFile в команде определяет имя удаленного сервисного класса. Опции представляют параметры, обеспечивающие дополнительные возможности RMI компилятора, представлены в Таблице 6.

Таблица 6.

Опции	Описание
-bootclasspath <path>	Переопределяет размещение файла класса начальной загрузки.
-classpath <path>	Переопределяет путь к классам переменной среды по умолчанию.
-d <directory>	Задаёт имя подкаталога, где генерируются стаб и скелетон. По умолчанию, стаб и скелетон генерируются на текущем устройстве.
-depend	Компилирует все файлы, которые связаны с удаленным сервисным классом.
-extdirs <path>	Переопределяет размещение установленных расширений.
-g	Генерирует номера строк и локальные переменные в форме таблицы.
-keep	Сохраняет файл '.java', который генерирует стаб и скелетон.

Опции	Описание
<code>-nowarn</code>	Не отображает предупреждения, когда генерируется стаб и скелетон.
<code>-vcompat</code>	Создает стаб и скелетон, который совместим с ранними версиями протоколов RMI.
<code>-verbose</code>	Отображает сообщение, когда удаленный файл сервера скомпилирован <code>rmic</code> .
<code>-v <version></code>	Создает стаб и скелетон для заданной версии JDK.

Команда генерации стаб и скелет для приложения RMI:

```
rmic HelloImpl
```

Файлы `HelloImpl_Stub.class` и `HelloImpl_Skel.class` генерируются в том же подкаталоге, в котором содержится удаленный сервисный класс `HelloImpl`. Тестировать распределенное приложение можно на локальном компьютере, сохраняя все файлы в одном подкаталоге перед развертыванием его в сети. Для выполнения приложения в сети необходимо создать два подкаталога - серверный и клиентский. В серверном подкаталоге сохраняются файлы:

- `Hello.class`
- `HelloImpl.class`
- `HelloServer.class`
- `HelloImpl_Stub.class`

В клиентском подкаталоге сохраняются файлы:

- `Hello.class`
- `HelloImpl_Stub.class`
- `HelloClient.class`

2. Запуск регистра RMI

Для запуска регистра RMI на сервере выполняется команда `start rmiregistry` в командной строке. По умолчанию регистр запускается на порт 1099. Для запуска регистра RMI на другой порт, необходимо указать номер порта в командной строке следующим образом:

```
start rmiregistry 1234
```

Аналогично, если регистр запускается на отличный порт от 1099, необходимо указать номер порта в строке URL, заданной в методах `rebind()` и `lookup()` класса `Naming`. Для запуска регистра RMI на порт по умолчанию может использоваться следующая команда:

```
start rmiregistry
```

При выполнении команда открывает новое окно `Command Prompt`, в котором выполняется `rmiregistry`.

Необходимо остановить и перезапустить сервис `rmiregistry`,

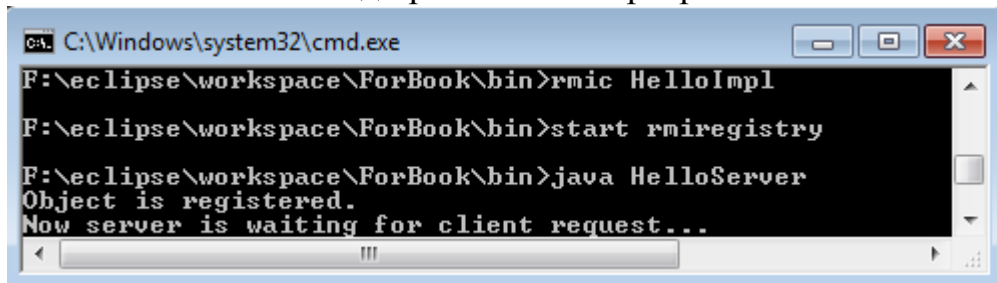
если Вы модифицируете удаленный интерфейс.

3. Выполнение сервера RMI

Необходимо запустить сервер, чтобы обслуживать запросы клиента. Следующая команда, чтобы запустить RMI сервер HelloServer:

```
java HelloServer
```

Рис. 20 показывает вывод приложения сервера RMI:



```
C:\Windows\system32\cmd.exe
F:\eclipse\workspace\ForBook\bin>rmic HelloImpl
F:\eclipse\workspace\ForBook\bin>start rmiregistry
F:\eclipse\workspace\ForBook\bin>java HelloServer
Object is registered.
Now server is waiting for client request...
```

Рис. 20. Вывод приложения сервера RMI.

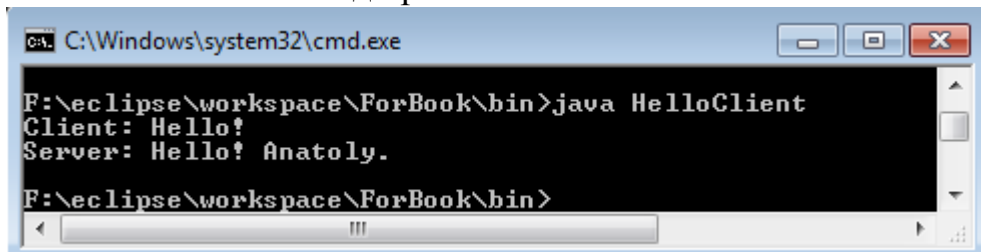
4. Выполнение клиента RMI

Следующая команда выполняет программу RMI клиента:

```
java HelloClient
```

Если Вы запускаете клиента с того же самого компьютера, то команда задается в отдельном командном окне.

Рис. 21 показывает вывод приложения клиента RMI.



```
C:\Windows\system32\cmd.exe
F:\eclipse\workspace\ForBook\bin>java HelloClient
Client: Hello!
Server: Hello! Anatoly.
F:\eclipse\workspace\ForBook\bin>
```

Рис. 21. Вывод приложения клиента RMI.

Клиент RMI вызывает метод sayHello() для вывода сообщения, полученного от сервера RMI. Рис. 22 представляет общение между RMI-клиентом и RMI-сервером:

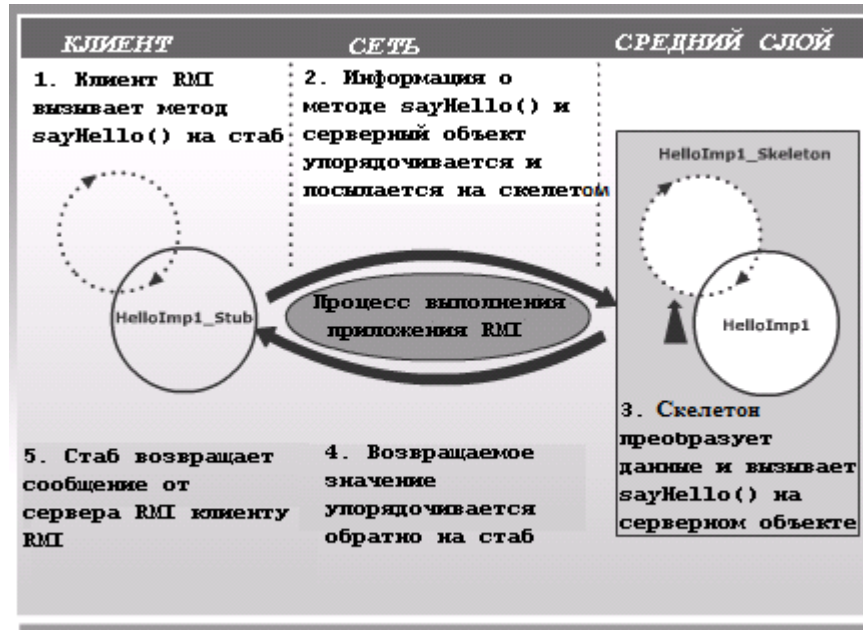


Рис. 22. Работа распределенного RMI приложения.

Разработка RMI-приложения в среде IDE Eclipse

Постановка задачи

Необходимо разработать клиент/серверное приложение для удаленной регистрации участников научной конференции. Сервер организаторов конференции должен содержать базу данных (БД) и RMI-сервис для приема и записи регистрационных сведений в БД. Участникам конференции необходимо предоставить приложение с графическим интерфейсом для ввода и отправки данных на сервер с помощью вызова удаленного метода RMI.

Для решения поставленной задачи необходимо выполнить следующие шаги:

1. Создать новый проект.
2. Создать в БД таблицу `registration_info` для хранения данных регистрации участников конференции.
3. Создать сериализуемый Java-класс `RegistrationInfo` для представления и передачи данных регистрации.
4. Реализовать интерфейс `ConfServer` и класс реализации `ConfServerImpl` удаленных методов сервера.
5. Создать клиентское приложение – разработать графический интерфейс (Swing) и обеспечить вызов удаленного метода сервера.
6. Выполнить приложение.

Подготовительный этап

Для реализации проекта необходимо установить и настроить среду разработки Eclipse, Apache Derby и Derby Plugins.

Создание нового проекта

- 1) Выберите пункт меню File/New/Project, в окне выбора типа проекта укажите other/Java Project и нажмите Next.
- 2) Укажите имя проекта Rmi и нажмите Finish.

Создание таблицы registration_info

- 1) Подключитесь к БД Derby и запустите сервер БД.
- 2) Для хранения SQL-скриптов создадим новый файл registration_info.sql. В окне Package Explorer щелкните правой кнопкой мыши на значок проекта и выберите New/File, укажите имя файла registration_info.sql и нажмите Finish.
- 3) Скопируйте в файл следующие команды:

```
-- подключение
connect
'jdbc:derby://localhost:1527/myDB;create=true;user=me;password=mine';

-- создание таблицы
create table registration_info(first_name varchar(20), last_name
varchar(20), organization varchar(100), report_theme varchar(300), email
varchar(20));

-- отключение и выход
disconnect;
exit;
```

- 4) Сохраните файл нажатием клавишей Ctrl-S
- 5) Щелкните правой кнопкой мыши на файл registration_info.sql в окне Package Explorer и выберите Apache Derby/Run SQL Script using 'ij'
- 6) В случае успешного выполнения скрипта в консоли выводится следующее:

```
ij version 10.7
ij> -- подключение
connect
'jdbc:derby://localhost:1527/myDB;create=true;user=me;password=mine';
ij> -- создание таблицы
create table registration_info(first_name varchar(20), last_name
varchar(20), organization varchar(100), report_theme varchar(300), email
varchar(20));
0 rows inserted/updated/deleted
```

Создание класса RegistrationInfo

Перед созданием интерфейса и класса реализации, создадим обычный сериализуемый Java-класс RegistrationInfo, который будет использоваться для представления и передачи данных об участнике конференции.

- 1) Создайте новый Java-класс, нажав правой кнопкой мыши на каталог src и выбрав пункт меню New/Class. Назовите класс RegistrationInfo и разместите его в пакете ru.ifmo.rmi.
- 2) В классе Employee создайте пять полей, соответствующих столбцам таблицы registration_info, добавьте конструкторы и набор get/set методов. Полный код класса RegistrationInfo приведен ниже:

```
package ru.ifmo.rmi;
import java.io.Serializable;
public class RegistrationInfo implements Serializable {
    private String firstName;
    private String lastName;
    private String organization;
    private String reportTheme;
    private String email;
    public RegistrationInfo() {}
    public RegistrationInfo(String firstName, String lastName,
        String organization, String reportTheme, String email) {
        super();
        this.firstName = firstName;
        this.lastName = lastName;
        this.organization = organization;
        this.reportTheme = reportTheme;
        this.email = email;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getOrganization() {
        return organization;
    }
    public void setOrganization(String organization) {
        this.organization = organization;
    }
    public String getReportTheme() {
        return reportTheme;
    }
    public void setReportTheme(String reportTheme) {
        this.reportTheme = reportTheme;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
```

```

        this.email = email;
    }
}

```

Создание интерфейса `ConfServer` и класса реализации `ConfServerImpl`

Интерфейс `ConfServer` объявляет удаленные методы, которые могут быть вызваны клиентом RMI. В нашем случае интерфейс будет содержать один метод `registerConfParticipant`, принимающий характеристики участника конференции, и сохраняющий полученные данные в БД.

- 1) Для создания нового интерфейса щелкните правой кнопкой мыши на пакет `ru.ifmo.rmi` в окне `Package Explorer` и выберите `New/Interface/`
- 2) В появившемся окне в качестве имени класса (Name) задайте `ConfServer` и убедитесь что в качестве имени пакета (Package) указано `ru.ifmo.rmi`. Нажмите `Finish`.

Код интерфейса `ConfServer` приведен ниже:

```

package ru.ifmo.rmi;
import java.rmi.*;
public interface ConfServer extends Remote {
    int registerConfParticipant(RegistrationInfo registrationInfo)
        throws RemoteException;
}

```

Создание класса реализации `ConfServerImpl`

Класс `ConfServerImpl` содержит реализацию удаленного метода регистрации участников конференции. Объект класса `ConfServerImpl` представляет собой удаленный сервис и должен быть зарегистрирован командой `rmiregistry` под определенным именем в регистре RMI, входящей в состав JDK. Регистр RMI, обеспечивает хранение, поиск и выполнение методов объекта удаленными клиентами.

Перед регистрацией объекта в регистре RMI необходимо, во-первых, указать путь к откомпилированному классу реализации `ConfServerImpl` (каталог `bin` в каталоге проекта). Во-вторых, необходимо настроить параметры менеджера безопасности (`security manager`), таким образом, чтобы виртуальная машина сервера могла запускать код объектов, пришедших (например, по сети), в качестве аргументов вызова удаленных методов. Эти настройки могут быть указаны с помощью файлов конфигурации, параметров запуска приложения, либо в самом коде метода. В приведенном ниже примере используется последний способ.

Создание класса `ConfServerImpl` включает в себя следующие основные задачи:

1. Реализацию интерфейса `ConfServerImpl`.

2. Создание конструктора.
3. Обеспечение реализации удаленного метода registerConfParticipant.
4. Создание метода main(), выполняемого при запуске сервера, где выполняется:
 - указание регистру RMI пути к файлу класса реализации сервера путем установки значения системного свойства java.rmi.server.codebase;
 - создание и настройка менеджера безопасности RMISecurityManager;
 - создание и регистрация в регистре RMI удаленного объекта ConfServer.

- 1) Для создания класса щелкните правой кнопкой мыши на пакет ru.ifmo.rmi в каталоге src окна Package Explorer и выберите New/Class.
- 2) В появившемся окне в качестве имени класса (Name) задайте ConfServerImpl. Нажмите Finish.

Код класса ConfServerImpl приведен ниже:

```

package ru.ifmo.rmi;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.security.Permission;
import java.sql.*;
public class ConfServerImpl extends UnicastRemoteObject implements
ConfServer {
    /* Определяется конструктор по умолчанию */
    public ConfServerImpl() throws RemoteException {
        super();
    }
    /* Определение удаленного метода */
    public int registerConfParticipant(RegistrationInfo
registrationInfo)
        throws RemoteException {
        try {
            // Регистрация драйвера БД Derby

            Class.forName("org.apache.derby.jdbc.ClientDriver").newInstance();
            // Получение соединения с БД
            Connection con = DriverManager

            .getConnection("jdbc:derby://localhost:1527/myDB;create=true;user=
me;password=mine");
            // Запись полученных данных в БД
            PreparedStatement st = con
                .prepareStatement("insert into registra-
tion_info "
                                + "(first_name, last_name,
organization, "
                                + "report_theme, email) ")

```

```

        + "values (?, ?, ?, ?,
?)"");
        st.setString(1, registrationInfo.getFirstName());
        st.setString(2, registrationInfo.getLastName());
        st.setString(3, registrationInfo.getOrganization());
        st.setString(4, registrationInfo.getReportTheme());
        st.setString(5, registrationInfo.getEmail());
        st.executeUpdate();
        st.close();
        // Получение количества зарегистрированных участников
        Statement st1 = con.createStatement();
        int count = 0;
        ResultSet rs = st1
            .executeQuery("Select count(*) from reg-
istration_info");
        if (rs.next()) {
            count = rs.getInt(1);
        }
        st1.close();
        return count;
    } catch (Exception e) {
        e.printStackTrace();
        throw new RemoteException(e.getMessage(), e);
    }
}
/* Метод main() */
public static void main(String args[]) {
    try {
        // Указание расположения классов RMI
        System.setProperty("java.rmi.server.codebase",
            "file:///F:/Eclipse/workspace/Rmi/bin/");
        // Установка менеджера безопасности (если не установлен):
        // Создается новый объект анонимного
        // класса RMISecurityManager
        // и переопределяется метод checkPermission.
        // Метод не содержит кода, следовательно, не определяет
        // никаких ограничений
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new
RMISecurityManager() {
                public void checkConnect(String host, int port,
                    Object context) {
                }
                public void checkConnect(String host,
int port) {
                }
                public void checkPermission(Permission
perm) {
                }
            });
        }
        // Создание экземпляра класса ConfServerImpl
        ConfServerImpl instance = new ConfServerImpl();
        // Регистрация объекта RMI под именем ConfServer
        Naming.rebind("ConfServer", instance);
    }
}

```

```

        System.out.println("Сервис зарегистрирован");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Создание клиента

Класс `ConfClient` обращается к удаленному хосту (в нашем примере `localhost`) и получает ссылку на удаленный объект из регистра RMI. После этого клиент получает возможность вызова удаленных методов.

- 1) Щелкните правой кнопкой мыши на пакет `ru.ifmo.rmi` в каталоге `src` окна `Package Explorer` и выберите `New/Class`.
- 2) В появившемся окне в качестве имени класса (Name) задайте `ConfClient`. Нажмите `Finish`.

Код класса `ConfClient` приведен ниже:

```

package ru.ifmo.rmi;
import javax.swing.*;
import java.rmi.*;
import java.awt.event.*;
import java.awt.*;

public class ConfClient {
    /* Объявляются переменные */
    static JFrame frame;
    static JPanel panel;
    JLabel lbLastName;
    JLabel lbFirstName;
    JLabel lbOrganization;
    JLabel lbReportTheme;
    JLabel lbEmail;
    JTextField txtLastName;
    JTextField txtFirstName;
    JTextField txtOrganization;
    JTextField txtReportTheme;
    JTextField txtEmail;
    JButton submit;
    /* Определяется конструктор по умолчанию */
    public ConfClient() {
        /* Создается JFrame */
        frame = new JFrame("Регистрация участника конференции");
        panel = new JPanel();
        /* Набор менеджеров разметки */
        panel.setLayout(new GridLayout(5, 2));
        frame.setBounds(100, 100, 400, 200);
        frame.getContentPane().setLayout(new BorderLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        /* Define the swing components on the JFrame */
        lbLastName = new JLabel("Фамилия");
        lbFirstName = new JLabel("Имя");
        lbReportTheme = new JLabel("Тема доклада");
        lbOrganization = new JLabel("Организация");
    }
}

```

```

        lbEmail = new JLabel("Email");
        txtLastName = new JTextField(15);
        txtFirstName = new JTextField(15);
        txtOrganization = new JTextField(70);
        txtReportTheme = new JTextField(100);
        txtEmail = new JTextField(15);
        submit = new JButton("Отправить");
        /* Добавление в панель компонентов swing */
        panel.add(lbLastName);
        panel.add(txtLastName);
        panel.add(lbFirstName);
        panel.add(txtFirstName);
        panel.add(lbOrganization);
        panel.add(txtOrganization);
        panel.add(lbReportTheme);
        panel.add(txtReportTheme);
        panel.add(lbEmail);
        panel.add(txtEmail);
        frame.getContentPane().add(panel, BorderLayout.CENTER);
        frame.getContentPane().add(submit, BorderLayout.SOUTH);
        frame.setVisible(true);
        submit.addActionListener(new ButtonListener());
    }
    /* Создание класса ButtonListener */
    class ButtonListener implements ActionListener {
        /* Определение метода actionPerformed() */
        public void actionPerformed(ActionEvent evt) {
            try {
                // Получение удаленного объекта
                // Если сервер размещен на удаленном компьютере,
                // то вместо localhost указывается имя
                // хоста сервера
                ConfServer server = (ConfServer) Naming
                .lookup("rmi://localhost/ConfServer");
                // Формирование сведений о регистрации для
                // отправки на сервер
                RegistrationInfo registrationInfo = new
                RegistrationInfo(
                    txtFirstName.getText(),
                    txtLastName.getText(),
                    txtOrganization.getText(),
                    txtReportTheme.getText(),
                    txtEmail.getText());
                // Вызов удаленного метода
                int count = serv-
                er.registerConfParticipant(registrationInfo);
                JOptionPane.showMessageDialog(
                    frame,
                    "Регистрация выполне-
                    на успешно"
                    +
                    "\nКоличество зарегистрированных участников - "
                    + count
                    + "\nСпасибо за участие");
            } catch (Exception e) {
                JOptionPane.showMessageDialog(frame, "Ошибка");
            }
        }
    }

```

```

        System.out.println(e);
    }
}
// Определение метода main()
public static void main(String args[]) {
    // Создание объекта класса Client
    new ConfClient();
}
}

```

Запуск и тестирование

Каждый из классов `ConfServerImpl` и `ConfClient` содержит метод `main()` и является самостоятельным приложением, которое может быть запущено на отдельном компьютере. В нашем случае роль клиента и сервера будет выполнять один и тот же компьютер.

- 1) Запустите службу регистра RMI с помощью команды `rmiregistry`. В Windows это действие может быть выполнено с помощью команды Пуск/Выполнить. Служба регистра RMI обеспечивает хранение удаленных объектов и доступ к ним клиентов и должна быть запущена на протяжении всего времени работы приложений с удаленными объектами.
- 2) Щелкните правой кнопкой мыши на класс `ConfServerImpl` в окне Package Explorer и выберите команду Run As/Java Application. В результате выполнения в службе RMI регистрируется объект `ConfServer`. В случае успешной регистрации выводится сообщение, показанное на Рис. 22.

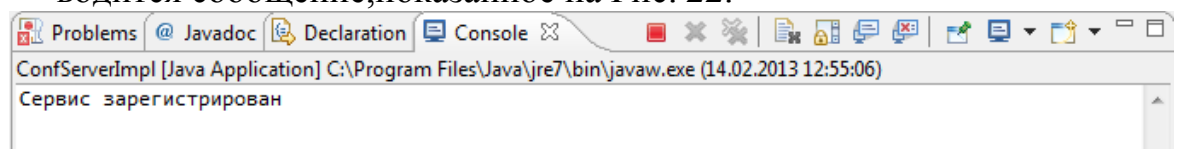


Рис. 22. Вывод сообщения сервера RMI.

- 3) Аналогичным образом запустите класс `ConfClient`. В появившемся окне укажите данные регистрации нового участника и нажмите кнопку Отправить. Результаты успешного выполнения программы приведены на Рис. 23.

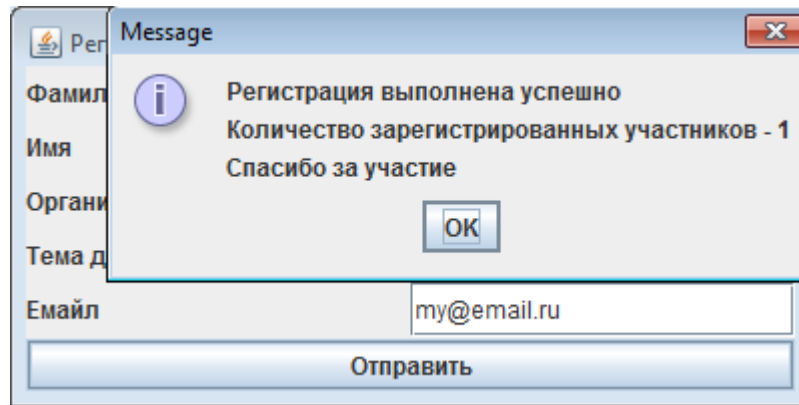


Рис. 23. Вывод диалогового окна клиента RMI.

- 4) Можно проверить появилась ли запись о новом участнике в таблице БД `registration_info`. Для этого откройте файл `registration_info.sql`, закомментируйте строку `create table registration_info`, добавьте следующий запрос и сохраните файл:


```
select * from registration_info
```
- 5) Выполните скрипт, который выводит данные о зарегистрированном участнике конференции в виде, аналогичном показанному на Рис. 24.

```
<terminated> [Rmi] - D "F:\eclipse\workspace\Rmi\registration_info.sql"
ij> -- создание таблицы
-- create table registration_info(first_name varchar(20), last_name varchar
select * from registration_info;
FIRST_NAME      |LAST_NAME      |ORGANIZATION
-----
Anatoly         |Dubakov        |NRU ITMO
```

Рис. 24. Результат работы запроса.

Вопросы для самопроверки

1. Дайте определение распределенным приложениям.
2. На каком слое реализуется RMI в трехслойной модели.
3. Из каких 3-х уровней состоит архитектура RMI?
4. Из каких четырех пакетов состоит RMI? Дайте характеристику этих пакетов.
5. Что представляет собой и где размещается регистр RMI?
6. Где объявляются методы, которые могут быть вызваны удаленно клиентом.
7. Какие операции должен выполнить удаленный сервисный класс для сервера RMI?
8. Какой компонент обеспечивает для Java-приложения менеджер безопасности.
9. Какие шаги необходимо выполнить для создания и выполнения

приложения RMI?

Технологии и архитектура JavaEE

В предыдущих разделах мы рассмотрели базовые средства Java создания серверных приложений, обеспечивающих клиент/серверное взаимодействие - современную архитектуру распределенных приложений. Для сокращения стоимости и эффективности проектирования и разработки распределенных систем фирма Sun Microsystems разработала компонентную архитектуру Java Enterprise Edition (JavaEE). Последняя спецификация платформы Sun Microsystems на основе JDK 6.0 носит название JavaEE и предлагает многоуровневую распределенную модель приложений, многократно используемые компоненты, унифицированную модель безопасности, гибкое управление транзакциями, поддержку WEB-сервисов посредством интегрированного обмена данными на основе языка Extensible Markup Language (XML), открытых стандартов и протоколов.

Это платформеннонезависимое решение не связано с каким-либо продуктом и Application Program Interface (API) одного поставщика, а является открытой спецификацией, которую могут реализовать различные разработчики, обеспечивая наилучший уровень удовлетворения технологических и бизнес требований. Более того, JavaEE представляет архитектуру для разработки приложений, которые могут выполняться в различных операционных системах, таких как Windows, UNIX и MacOS. Таким образом, разработанное JavaEE приложение может быть выполнено под управлением любого совместимого со спецификацией JavaEE сервера в любой операционной системе, практически, без каких-либо изменений.

Приложения JavaEE состоят из компонентов, которые представляют собой отдельные функциональные единицы программного обеспечения, и собираются в приложение JavaEE вместе с классами и файлами, взаимодействующими с другими компонентами. Спецификация определяет следующие JavaEE компоненты:

- Клиентские приложения и апплеты, которые выполняются на клиентской части приложения.
- Java Servlet и Java Server Pages (JSP) выполняются на сервере.
- Компоненты Enterprise JavaBeans (EJB) являются спецификацией технологии серверных компонентов, содержащих бизнес-логику, которые выполняются на сервере.

Компоненты JavaEE разрабатываются на языке Java и компилируются также как и любые программы на языке Java. Отличием является

ся то, что компоненты JavaEE объединяются в архитектуру стандартного пакета (архивы EAR, WAR, JAR) приложения JavaEE, проверяются на соответствие спецификации JavaEE и разворачиваются на сервере JavaEE, под управлением которого они и выполняется.

Сервер JavaEE обеспечивает определенные сервисы в форме контейнеров (Container) для каждого типа компонентов. Поскольку нет необходимости разрабатывать эти сервисы самостоятельно, разработчик концентрируется исключительно на решении бизнес проблем. Контейнер представляет собой интерфейс между компонентом и определенной функциональностью низкого уровня, поддерживающей компоненты. Перед выполнением любой компонент распределенного приложения должен быть собран в модуль JavaEE и размещен в соответствующий контейнер.

JavaEE включает компоненты, которые могут использоваться для разработки приложений на уровне презентации, а также для реализации бизнес-логики. Стандартная спецификация JavaEE определяет API для управления бизнес транзакциями, использования элементов безопасности, инфраструктуру инструментальных средств для поддержки среды выполнения приложения, а также средства для внутренней и внешней интеграции. Процесс сборки приложений предполагает задание установок контейнеров как для каждого из компонентов приложения JavaEE, так и для всего приложения. Установки контейнера определяют поддержку, обеспечиваемую сервером JavaEE, включающую такие сервисы, как безопасность, управление транзакциями, просмотр Java Naming and Directory Interface (JNDI) и удаленную связность. На Рис. 25 представлена архитектура многоуровневого приложения, на котором выделены уровни презентации, реализации бизнес логики и база данных.

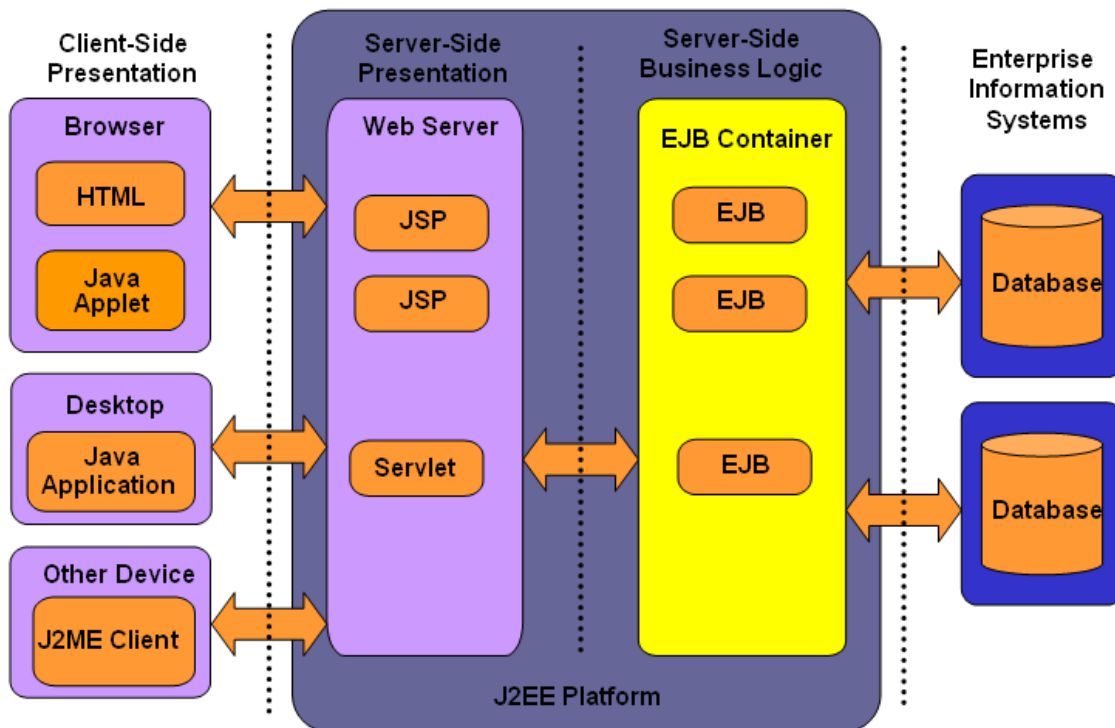


Рис. 25. Архитектура многоуровневого приложения JavaEE.

JavaEE является открытой спецификацией, которая реализуется как в составе коммерческих разработок Oracle, Web Sphere (IBM), Web Logic, так и большим количеством открытых систем, таких как Apache, Jboss, Jetty и ряда других. В пособии используется свободно распространяемая серверная реализация спецификации JavaEE фирмы Oracle, которая называется GlassFish (glassfish.java.net) и свободно распространяемая интегрированная среда разработки (Integrated Development Environment - IDE) Eclipse (www.eclipse.org).

Вообще говоря, JavaEE поддерживает большое число технологий, обеспечивающих широкие возможности для расширения функциональности распределенных вычислений. Очевидно, что в рамках предлагаемого пособия не представляется возможным рассмотреть детально, а тем более применить все технологии архитектуры JavaEE. Полную информацию можно получить из списка литературы, приведенного в конце пособия, а также на сайте <http://www.oracle.com/us/technologies/java/overview/index.html>, однако, основные технологии JavaEE, необходимые при создании распределенных приложений технологии JSP и Servlet, представлены в последующем материале.

Введение в сервлеты Java

Понятие сервлета

Рассмотрим ситуацию, в которой пользователю надо зайти на веб-сайт электронной почты. При этом необходимо представить идентификационные сведения (имя и пароль) для аутентификации. Веб-сайт принимает введенные данные и проверяет их, используя серверные программы. Эти серверные программы могут быть написаны с использованием различных серверных технологий, таких как Common Gateway Interface (CGI – Общий шлюзовой интерфейс), Active Server Pages (ASP – Активные серверные страницы) и сервлеты.

CGI-скрипты разрабатываются на языках программирования C, C++ или Perl. Если серверное приложение использует CGI-скрипты для обработки клиентских запросов, то сервер создает отдельную виртуальную машину для выполнения каждого экземпляра CGI-скрипта, в результате чего производительность сервера уменьшается из-за большого количества одновременных запросов.

ASP – это серверная технология, которая позволяет разработчику совмещать HTML и скриптовый язык на одной странице. Для объектов ASP поддерживается многопоточность, и, следовательно, они могут одновременно обслуживать несколько запросов. Ограничение ASP состоит в том, что эта технология не совместима со многими коммерческими веб-серверами.

Сервлеты – это серверные программы, написанные на Java. В отличие от CGI-скриптов, код инициализации сервлета выполняется только один раз. Кроме того, обработка каждого клиентского запроса выполняется в отдельном потоке на сервере, что предотвращает создание лишних процессов, увеличивая, таким образом, производительность сервера. При этом сервлеты наследуют все свойства языка программирования Java. Например, подобно всем стандартным классам Java, сервлет не зависит от платформы и может использоваться в различных операционных системах. Также сервлет может пользоваться богатыми возможностями Java, такими как работа в сети, многопоточность, JDBC, локализация, RMI и многими другими.

Технология Java Servlet

Сервлеты позволяют расширять функциональность серверного приложения для формирования динамического содержимого в веб-

приложении. Кроме наследуемых достоинств языка Java, сервлеты обладают следующими характеристиками:

- *Безопасность*: Веб-контейнер обеспечивает среду выполнения для сервлета. Сервлеты наследуют параметры безопасности, предоставляемые веб-контейнером, что позволяет разработчикам сосредоточиться на функциональности сервлета и оставить обеспечение проблем безопасности веб-контейнеру.
- *Управление сессией*: Это механизм отслеживания состояния пользователя при выполнении нескольких запросов. Сессия сохраняет идентичность и состояние клиента при выполнении нескольких запросов.
- *Сохранение данных экземпляра объекта*: Сервлеты позволяют увеличить производительность сервера, сокращая количество обращений к диску. Например, если клиент заходит на сайт банка для проверки баланса или получения ссуды, номер его счета должен проверяться в базе данных при выполнении каждого действия. Вместо многократной проверки номера счета в базе данных, сервлеты сохраняют номер счета в памяти объекта до тех пор, пока пользователь находится на веб-сайте.

Чтобы получить доступ к сервлету, сначала необходимо откомпилировать сервлет, а затем развернуть файл класса сервлета на Java-совместимом сервере приложений, таком, например, GlassFish Application Server. Сервер приложений JavaEE предоставляет веб-контейнер для управления различными компонентами веб-приложения, в том числе, HTML-страницами и сервлетами. Веб-контейнер обеспечивает среду исполнения для сервлета. Веб-контейнеры предоставляют следующие сервисы, требуемые веб-приложениям:

- Управление различными стадиями жизненного цикла сервлета, в том числе, инициализация экземпляра сервлета, обработка клиентского запроса и удаление экземпляра сервлета.
- Определение ограничений безопасности, которые разрешают доступ к развернутым сервлетам только авторизованным пользователям.
- Управление транзакциями во время записи и чтения данных сервлетом из базы данных.
- Создание и удаление экземпляров сервлетов из пула экземпляров в процессе обслуживания многократных запросов.

Работа сервлетов

Когда клиент отправляет запрос конкретного сервлета серверу приложений JavaEE, последний передает запрос веб-контейнеру, который проверяет, существует ли экземпляр требуемого сервлета и, если экземпляр сервлета существует, то веб-контейнер передает запрос сервлету, который обрабатывает запрос клиента и отправляет ответ обратно клиенту. Взаимодействие между сервером приложений и веб-контейнером при обработке клиентского запроса показано на Рис. 26.

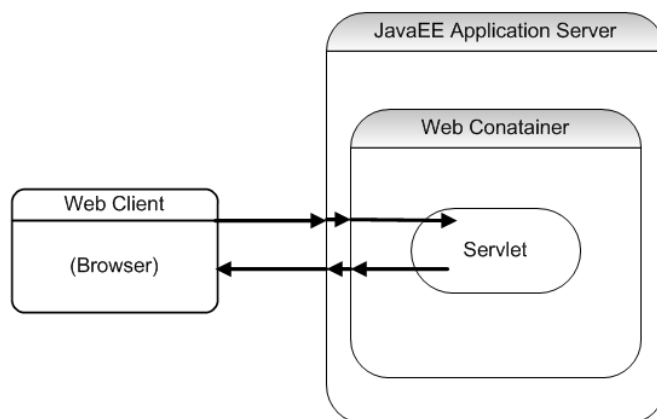


Рис. 26. Взаимодействие между сервером приложений и веб-контейнером при обработке клиентского запроса.

Если экземпляр сервлета не существует, то веб-контейнер находит и загружает класс сервлета. Затем веб-контейнер создает экземпляр сервлета и инициализирует его. После этого экземпляр сервлета начинает обработку запроса и Веб-контейнер передает ответ клиенту, сгенерированный сервлетом. Рис. 27 показывает этапы, выполняемые веб-контейнером при первом получении запроса от клиента.

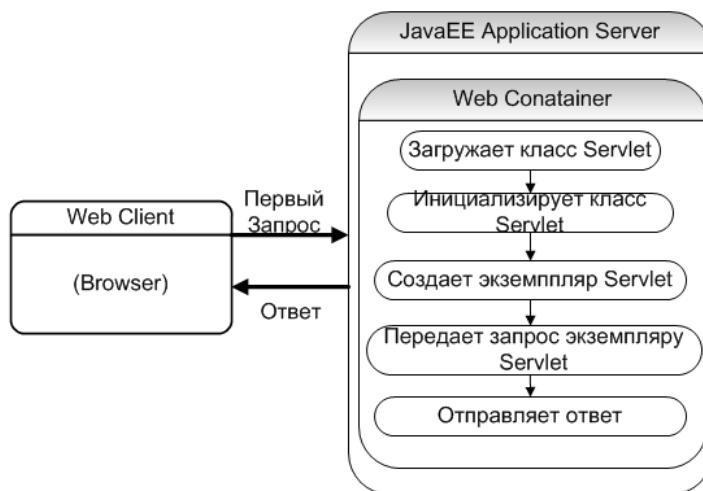


Рис. 27. Этапы, выполняемые веб-контейнером.

Иерархия классов сервлетов и методы жизненного цикла

Веб-контейнер управляет сервлетом, вызывая различные методы жизненного цикла. Эти методы определены в Java Servlet API и сосредоточены в наборе классов и интерфейсов, которые можно использовать для разработки сервлета. Эти классы и интерфейсы находятся в пакетах `javax.servlet` и `javax.servlet.http`.

Иерархия класса Servlet

Интерфейс `Servlet` является корневым интерфейсом иерархии сервлетов и всем сервлетам необходимо явно или неявно реализовывать этот интерфейс. Класс `GenericServlet` из Servlet API реализует интерфейс `Servlet` и кроме того он реализует интерфейс `ServletConfig` из Servlet API и интерфейс `Serializable` из стандартного пакета `java.io`. Объект интерфейса `ServletConfig` используется веб-контейнером для передачи конфигурационной информации сервлету при его инициализации.

Для разработки сервлета, который взаимодействует по протоколу HTTP, необходимо расширить класс `HttpServlet` в сервлете. Класс `HttpServlet` расширяет класс `GenericServlet` и таким образом предоставляет встроенную функциональность HTTP. Например, `HttpServlet` предоставляет методы, которые позволяют сервлету обрабатывать запрос клиента, приходящий с помощью определенного метода HTTP. Рис. 28 показывает общий схему иерархии интерфейсов и классов в пакетах `javax.servlet` и `javax.servlet.http`:

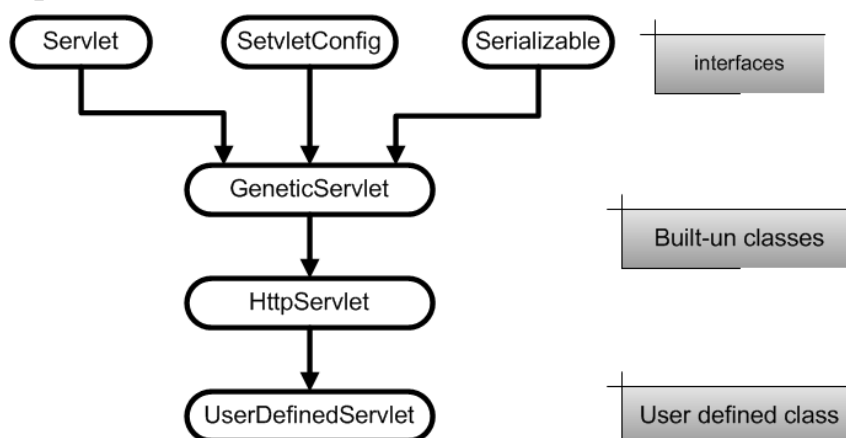


Рис. 28. Общая схема иерархии сервлетов.

Интерфейс `javax.servlet.Servlet`

Рис. 29 представляет иерархию объектов класса `Servlet` с соответствующими доступными методами.

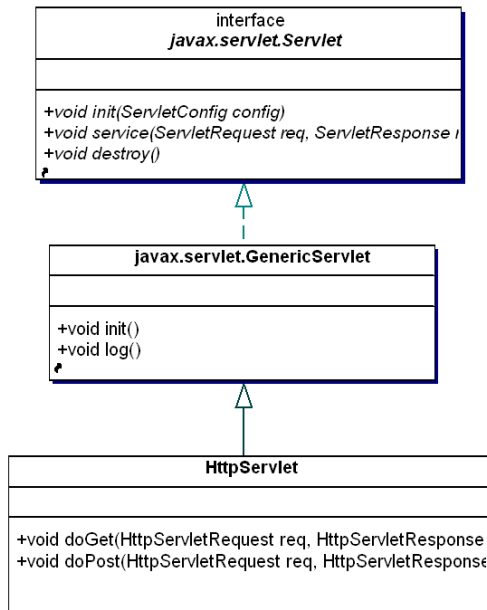


Рис. 29. Иерархия объектов класса servlet.

Интерфейс Servlet из пакета javax.servlet определяет методы, которые веб-контейнер вызывает в процессе управления жизненным циклом сервлета. В таблице 7 представлены методы интерфейса javax.servlet.Servlet.

Таблица 7.

Метод	Описание
public void destroy()	Веб-контейнер вызывает метод destroy() перед уничтожением экземпляра сервлета.
public ServletConfig getServletConfig()	Возвращает объект ServletConfig, который содержит данные о параметрах инициализации сервлета.
public String getServletInfo()	Возвращает строку, которая содержит информацию о сервлете, такую как автор, версия и авторское право.
public void init(ServletConfig config) throws ServletException	Веб-контейнер вызывает этот метод после создания экземпляра сервлета.

Интерфейс javax.servlet.ServletConfig

Интерфейс javax.servlet.ServletConfig реализуется веб-контейнером, чтобы иметь возможность передавать данные о конфигурации сервлету во время инициализации сервлета. Сервлет инициализируется передачей веб-контейнером объекта ServletConfig методу init().

Объект `ServletConfig` содержит данные для инициализации, а также предоставляет доступ к объекту `ServletContext`.

Параметры инициализации – это пары имен и их значений, которые можно использовать для передачи информации сервлету. Например, можно указать URL JDBC как параметр инициализации сервлета. При инициализации сервлет может использовать значение URL, чтобы получить соединение с базой данных. Контекст, известный `ServletContext` или `Web context` создается Веб-контейнер `container` как объект интерфейса `ServletContex` и позволяет сервлету взаимодействовать с веб-контейнером, который содержит этот сервлет. Для каждого web-приложения существует единственный `ServletContext`, и он доступен всем ресурсам приложения. В таблице 8 представлены несколько методов интерфейса `javax.servlet.ServletConfig`:

Таблица 8.

Метод	Описание
<code>public String getInitParameter(String param)</code>	Возвращает строку, содержащую значение указанных параметров инициализации или null, если параметр не существует.
<code>public Enumeration getInitParameterNames()</code>	Возвращает имена всех параметров инициализации как объект <code>Enumeration</code> , содержащий объекты <code>String</code> . Если параметры инициализации не определены, то возвращается пустой <code>Enumeration</code> .
<code>public ServletContext getServletContext()</code>	Возвращает объект <code>ServletContext</code> для сервлета, который позволяет взаимодействие с веб-контейнером.

Методы жизненного цикла сервлета

Интерфейс `javax.servlet.Servlet` определяет методы жизненного цикла сервлета: `init()`, `service()` и `destroy()`. Веб-контейнер вызывает методы сервлета `init()`, `service()` и `destroy()` во время его жизни. Ниже представлена последовательность, в которой веб-контейнер вызывает методы жизненного цикла сервлета:

1. Веб-контейнер загружает класс сервлета и создает один или более экземпляров класса сервлета.
2. Веб-контейнер вызывает метод `init()` экземпляра сервлета во время инициализации сервлета. Метод `init()` вызывается только один раз в жизненном цикле сервлета.

3. Веб-контейнер вызывает метод `service()`, разрешая сервлету выполнить обработку запроса клиента.
4. Метод `service()` обрабатывает запрос и возвращает ответ Веб-контейнеру.
5. После этого сервлет ожидает получения и обработки последующих запросов, как это происходило на этапах 3 и 4.
6. Веб-контейнер вызывает метод `destroy()` перед удалением экземпляра сервлета. Метод `destroy()` также вызывается только один раз во время жизни сервлета.

Метод `init()`

Метод `init()` вызывается только однажды во время инициализации жизненного цикла сервлета, что дает возможность сервлету выполнить любую работу по установке, например, открытие файлов или установку соединений с их серверами. Если сервлет установлен на сервере постоянно, он загружается при запуске сервера. В противном случае сервер активизирует сервлет при получении первого запроса от клиента на выполнение услуги данным сервлетом. Сначала веб-контейнер преобразует запрашиваемый URL в соответствующий сервлет, находящийся в веб-контейнере, а затем назначает этот сервлет для выполнения. После этого веб-контейнер создает объект интерфейса `ServletConfig`, который содержит данные конфигурации при запуске - параметры инициализации сервлета, вызывает метод `init()` сервлета и передает ему объект `ServletConfig`.

Метод `init()` генерирует исключение `ServletException`, если веб-контейнер не может инициализировать ресурсы сервлета. Гарантируется, что метод `init()` закончится перед любым другим обращением к сервлету, например, вызов метода `service()`. Следующий фрагмент кода показывает сигнатуру метода `init()`:

```
public void init (ServletConfig config) throws ServletException
```

Следующий фрагмент кода показывает метод `init()` сервлета Java для его инициализации:

```
public class ServletLifeCycle extends HttpServlet
{
    int count;
    public void init (ServletConfig config) throws ServletException
    { count=0; }
}
```

Метод `service()`

Метод `service()` является ядром сервлета и выполняет обработку запросов клиента. Всякий раз, когда веб-контейнер получает клиентский запрос, он вызывает метод `service()`. Метод `service()` вызывается только после завершения инициализации сервлета. При вызове метода `service()` веб-контейнер передает объект интерфейса `ServletRequest` и объект интерфейса `ServletResponse`. Объект `ServletRequest` содержит данные о запросе клиента, а объект `ServletResponse` содержит информацию, возвращаемую сервлетом клиенту. Следующий фрагмент кода показывает сигнатуру метода `service()`:

```
public void service(ServletRequest request, ServletResponse response)  
throws ServletException, IOException
```

Метод `service()` генерирует исключение `ServletException`, когда возникает исключительная ситуация, которая прерывает нормальную работу сервлета. Также метод `service()` генерирует исключение `IOException`, когда возникает исключительная ситуация ввода или вывода.

Метод `service()` отправляет клиентский запрос одному из методов обработки запроса интерфейса `HttpServlet`: `doGet()`, `doPost()`, `doHead()` или `doPut()`. Методы обработки запросов принимают объекты `HttpServletRequest` и `HttpServletResponse` как параметры от метода `service()`.

Метод `service()` не переопределяется в `HttpServlet`, так как веб-контейнер автоматически вызывает метод `service()`. Функциональность HTTP сервлетов содержится в методах `doGet()` или `doPost()`.

Метод `doGet()`

Метод `doGet()` обрабатывает клиентский запрос, использующий GET-метод протокола HTTP. GET является одним из методов запроса HTTP, который, в основном, используется для извлечения статических ресурсов. Когда в строке браузера вводите адрес URL для просмотра статической веб-страницы, браузер использует метод GET для выполнения запроса. Аналогично, когда выбирается гиперссылка на веб-странице для получения доступа к некоторому ресурсу, то запрос отправляется с применением метода GET. Также, используя метод GET, можно отправлять данные, введенные пользователем в теге `Form` HTML. Данные, отправленные с применением GET-метода, добавляются как строка запроса к URL. Тип метода HTTP указывается в форме HTML, используя атрибут `METHOD` тега `FORM`, например, следующим образом:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Пример Form, использующей GET</title>
</head>
<body>
    <H2 ALIGN="CENTER">Пример Form, использующей GET</H2>
    <FORM ACTION="http://localhost:8080/SomeProgram" METHOD="GET">
        <CENTER>
            First name: <INPUT TYPE="TEXT" NAME="firstName" VAL-
UE="Tom"><BR>
            Last name: <INPUT TYPE="TEXT" NAME="LastName" VALUE="Cruse">
                <P> <INPUT TYPE="SUBMIT"> </P>
        </CENTER>
    </FORM>
</body>
</html>

```

Для обработки клиентских запросов, которые получены с использованием метода GET, необходимо переопределить метод `doGet()` в классе сервлета. В методе `doGet()` можно извлекать данные от клиента из объекта `HttpServletRequest`, а также использовать объект `HttpServletResponse` для формирования ответа клиенту. Следующий фрагмент программы демонстрирует сигнатуру метода `doGet()`:

```

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException

```

Метод `doPost()`

Метод `doPost()` обрабатывает запросы в сервлете, которые отправлены клиентом, используя метод POST протокола HTTP. Например, если клиент вводит регистрационные данные в форму HTML, данные могут быть отправлены с использованием метода POST. В отличие от метода GET, запрос POST отправляет данные как часть тела запроса HTTP и в результате отправленные данные не видны как часть URL.

Для обработки запросов в сервлете, которые отправляются с использованием метода POST, необходимо переопределить `doPost()`. В методе `doPost()` можно обработать запрос и отправить ответ клиенту. Следующий фрагмент кода демонстрирует сигнатуру метода `doPost()`:

```

protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException

```

Метод `doHead()`

Метод `doHead()` обрабатывает запросы, отправленные с использованием метода HEAD протокола HTTP. Метод HEAD отправляет запро-

сы серверу аналогично методу GET. Единственное отличие между методами GET и HEAD заключается в том, что метод HEAD возвращает заголовок ответа, который содержит, такие элементы, как Content-Type (Тип содержимого), Content-Length (Длина содержимого) и Last-Modified (Дата последнего изменения). Метод HEAD используется для определения, существует ли запрашиваемый ресурс. Он также используется для получения информации о ресурсе, например, о типе ресурса. Эта информация требуется перед повторным запросом содержимого ресурса. Также можно использовать метод HEAD для определения времени, когда ресурс был последний раз изменен. Это помогает определить, можно ли использовать ресурс из кэша, или необходимо обновить кэш.

Чтобы обработать запрос, отправленный с помощью метода HEAD, следует переопределить метод doHead() в сервлете. Следующий фрагмент кода демонстрирует сигнатуру метода doHead():

```
protected void doHead(HttpServletRequest request, HttpServletResponse response)  
throws ServletException, IOException
```

Метод doPut()

Метод doPut() обрабатывает запросы, отправленные с использованием метода PUT протокола HTTP. Метод PUT позволяет клиенту сохранить информацию на сервере. Например, можно использовать метод PUT для отправки файла с изображением на сервер. Следующий фрагмент кода демонстрирует сигнатуру метода doPut():

```
protected void doPut(HttpServletRequest request, HttpServletResponse response)  
throws ServletException, IOException
```

Метод destroy()

Метод destroy() вызывается для освобождения всех ресурсов, занимаемых сервлетом, например, открытых файлов и соединений с базой данных, перед выгрузкой сервлета. Этот метод может быть пустым, если нет необходимости выполнения каких-либо завершающих операций, или в нем можно выполнить действия по освобождению удерживаемых ресурсов. Веб-контейнер вызывает метод destroy() перед удалением экземпляра сервлета в следующих случаях:

- Период времени, указанный для жизни сервлета, истек. Период времени сервлета – это период, в течение которого сервлет сохраняется в активном состоянии веб-контейнером для обслуживания клиентских запросов.
- Веб-контейнеру нужно выгрузить экземпляры сервлетов для экономии памяти.

- Веб-контейнер заканчивает работу.

В методе `destroy()` можно написать код для освобождения ресурсов, занимаемых сервлетом. Метод `destroy()` также используется для сохранения любой предназначенной для длительного использования информации перед тем, как экземпляр сервлета будет удален. Необходимо создать метода `destroy()` таким образом, чтобы избежать закрытия необходимых ресурсов до тех пор, пока все вызовы `service()` не завершатся. Следующий фрагмент кода демонстрирует метод `destroy()`:

```
public void destroy();
```

Создание сервлета

Для создания сервлета, необходимо выполнить следующие шаги:

1. Написать код сервлета.
2. Откомпилировать и упаковать сервлет.
3. Развернуть сервлет как приложение JavaEE.
4. Вызвать сервлет из браузера.

Программирование сервлета

Как уже отмечалось, пакеты сервлетов определяют два абстрактных класса, которые реализуют интерфейс `Servlet`: класс `GenericServlet` (из пакета `javax.servlet`) и класс `HttpServlet` (из пакета `javax.servlet.http`). Эти классы предоставляют реализацию по умолчанию для всех методов интерфейса `Servlet`. Большинство разработчиков используют либо класс `GenericServlet`, либо класс `HttpServlet`, и замещают некоторые или все методы. Чтобы написать класс сервлета мы будем использовать расширение интерфейса `HttpServlet`. Двумя наиболее распространенными типами запросов HTTP (их также называют методами запросов) являются `get` и `post`, которые соответственно получают (или извлекают) информацию от клиента и помещают (или отправляют) данные клиенту. В классе `HttpServlet` определены методы `doGet` и `doPost` для реакции на запросы типа `get` и `post` клиента. Эти методы вызываются методом `service` класса `HttpServlet`, который, в свою очередь, вызывается при поступлении запроса на сервер. Метод `service()` сначала определяет тип запроса, а затем вызывает соответствующий метод. Необходимую функциональность и необходимо реализовать в сервлете.

Чтение и обработка клиентского запроса

После объявления и инициализации различных переменных сервлета, необходимо описать функциональность сервлета. Для этого пере-

определяются методы `doGet()` и `doPost()` класса `HttpServlet`. Методы `doGet()` и `doPost()` получают запросы HTTP, используя интерфейс `HttpServletRequest`. Объект `HttpServletRequest` содержит информацию о запросе, отправленном клиентом с помощью запроса HTTP. Можно получить значения параметра запроса, вызывая метод `getParameter()` интерфейса `HttpServletRequest`. Метод `getParameter()` интерфейса `HttpServletRequest` принимает имя параметра запроса как строковое значение и возвращает строку, которая содержит соответствующее значение параметра. Следующий фрагмент программы показывает сигнатуру метода `getParameter()`:

```
String getParameter(String arg);
```

Ниже приведен фрагмент кода показывающий получение параметра `firstName`, отправленного браузером:

```
String user=request.getParameter("firstName");
```

Отправка ответа клиенту

Для отправки ответа клиенту HTTP-сервлеты используют объект интерфейса `HttpServletResponse`. Чтобы отправить символьные данные клиенту, нужно получить объект `java.io.PrintWriter`. Можно использовать метод `getWriter()` интерфейса `HttpServletResponse`, чтобы получить объект `PrintWriter`. Следующий фрагмент кода показывает пример использования метода `getWriter()` для получения объекта `PrintWriter`:

```
PrintWriter out = response.getWriter();/* Здесь response является объектом  
HttpServletResponse. */
```

Можно указать тип содержимого ответа, используя метод `setContentType()` интерфейса `HttpServletResponse`, как это делается в следующем фрагменте кода сервлета:

```
response.setContentType("text/html");
```

Отправить данные клиенту можно, используя метод `println()` класса `PrintWriter`, как показывает следующий фрагмент кода:

```
out.println("Welcome "+ user+"!");
```

Компиляция и упаковка сервлета

Для формирования файла класса сервлета необходимо откомпилировать подготовленный код сервлета. Перед компиляцией сервлета нужно подключить необходимые библиотеки используемого сервера приложений к классам. В этой связи следует отметить, что процесс компиляции, упаковки и развертывания приложения вручную достаточно трудоемок и существенным образом определяется типом и версией используемого сервера приложений. Мы будем использовать `GlassFish` в качестве сервера приложений, и при использовании другого сервера

приложений, например Jboss от RedHat или WebLogic от компании BEA Systems, следует указывать подключаемые классы соответствующим образом. Естественно, эти трудности возникают, когда не используется среда разработки и все действия выполняются из командной строки. В пособии используется среда IDE Eclipse и правильная настройка сервера приложений гарантирует видимость и подключение всех необходимых библиотек.

Большинство операций, выполняемых в процессе разработки компонентов веб-приложения, автоматизируется используемой средой разработки, и суть этих операций, в принципе, может не интересовать разработчика до поры до времени. Однако, для полного понимания архитектуры веб-приложения и сервлета, в частности, ниже представлены сведения по описанию структур архивов развертывания.

После компиляции сервлета его необходимо упаковать с помощью создания дескриптора развертывания. Дескриптор развертывания – это XML-файл с именем `web.xml`, содержащий конфигурационную информацию о веб-приложении, в котором он размещается. Файл дескриптора развертывания `web.xml` может управлять регистрацией сервлета, отображением URL, файлами `welcome`, MIME-типами, ограничениями доступа к страницам и работой сервлетов в распределенных средах. Файл дескриптора развертывания не зависит от сервера и упрощает процесс развертывания.

Дескриптор развертывания можно создать с помощью XML-редактора. Код ниже приведен пример простого дескриптора развертывания:

```
<?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/j2ee/web-app_3_0.xsd">
<display-name>SampleApp</display-name>
<servlet>
    <display-name>SampleServlet</display-name>
    <servlet-name>SampleServlet</servlet-name>
    <servlet-class>SampleServlet</servlet-class>
</servlet>
</web-app>
```

В представленном коде первая строка дескриптора развертывания является *прологом*. Пролог устанавливает версию XML и кодировку, которая используется для кодирования данных. В данном примере используется кодировка UTF-8.

В файле `web.xml` используются следующие атрибуты:

`version`: Атрибут задает версию схемы. Схемы XML используются для подтверждения действительности XML-документа.

`xmlns`: Атрибут задает пространство имен схемы дескриптора развертывания. Все схемы дескрипторов развертывания JavaEE используют пространство имен `http://java.sun.com/xml/ns/javaee`. Пространство имен позволяет схеме или множеству схем быть однозначно идентифицируемым.

`xsi:schemaLocation`: Этот атрибут задает местоположение схемы. После указания данных о схеме, необходимо внести данные о развертываемом сервлете.

Тег `<displayname>` внутри тега `<web-app>` задает имя веб-компонента, который необходимо развернуть. Тег `<servlet>` содержит информацию о сервлете. Он содержит следующие теги:

`<display-name>`: Отображаемое имя сервлета

`<servlet-name>`: Предназначенное для доступа имя сервлета

`<servlet-class>`: Имя класса сервлета

После создания дескриптора развертывания можно упаковать сервлет. JavaEE определяет стандартную структуру упаковки сервлета в приложение JavaEE, для того чтобы сделать его переносимым между различными серверами приложений, что позволяет серверам приложений легко размещать и загружать файлы приложений из стандартной структуры каталогов. Чтобы создать упакованную структуру веб-приложения, необходимо создать следующие каталоги:

Корневой каталог: Корневой каталог содержит статические ресурсы, такие как, файлы HTML, файлы JSP и файлы изображений. Например, в корневом каталоге можно разместить файл `LoginPage.html`.

Подкаталог WEB-INF внутри корневого каталога: WEB-INF содержит файл дескриптора развертывания приложения `web.xml`, который описывает различные данные о конфигурации веб-приложения.

Подкаталог classes: classes размещается внутри каталога WEB-INF и содержит файлы классов приложения. Например, можно разместить файл `SampleServlet.class` сервлета `SampleServlet` в каталоге `classes`.

Подкаталог lib: lib размещается также внутри каталога WEB-INF и содержит Java Archive files (JAR – Архивные файлы Java) библиотек, которые требуются компонентам приложения.

На Рис. 30 представлена стандартная структура упаковки веб-приложения Java.

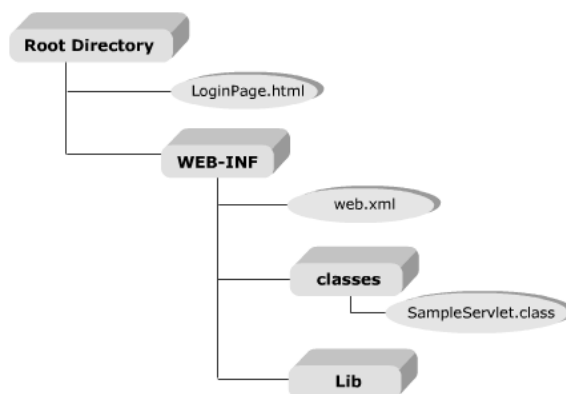


Рис. 30. Стандартная структура упаковки.

После размещения специальных файлов приложения в стандартной структуре каталогов, необходимо упаковать приложение в файл веб-архива (Web Archive (WAR)). WAR- файл – это заархивированное JavaEE веб-приложение. Для создания WAR-файла с помощью базовых компонентов Java, в командной строке вводится следующая команда:

```
jar cvf <имя war-файла>
```

В результате выполнения этой команды создается файл с расширением war, который может быть развернут на сервере приложений.

Большинство серверов приложений предоставляют утилиты, которые автоматически генерируют дескриптор развертывания и выполняют упаковку. Кроме того, все современные IDE, к которым относятся и предлагаемый вашему вниманию Eclipse, содержит все необходимые средства для создания JavaEE приложения и его развертывания на сервере.

Пример разработки сервлета

Постановка задачи

Необходимо разработать веб-приложение, использующее сервлет, для поиска информации о сотрудниках организации. Данные о сотрудниках хранятся в таблице Employee. Для выполнения поиска пользователь указывает фамилию сотрудника и просматривает информацию о найденных сотрудниках (допускается существование нескольких сотрудников с одинаковыми фамилиями).

Для решения поставленной задачи необходимо выполнить следующие шаги:

1. Создать новый проект
2. Создать таблицу employee и заполнить ее данными
3. Разработать сервлет, который выбирает из БД записи, соответствующие запросу пользователя и отображает результат.
4. Упаковать приложение и развернуть на сервере.

5. Протестировать работу приложения в браузере.

Подготовительный этап

Для реализации проекта необходимо установить и настроить Eclipse, GlassFish, Apache Derby и Derby Plugins.

Создание нового проекта

1) Выберите пункт меню File/New/Project, в окне выбора типа проекта укажите Web/Dynamic Web Project и нажмите Next (Рис. 30).

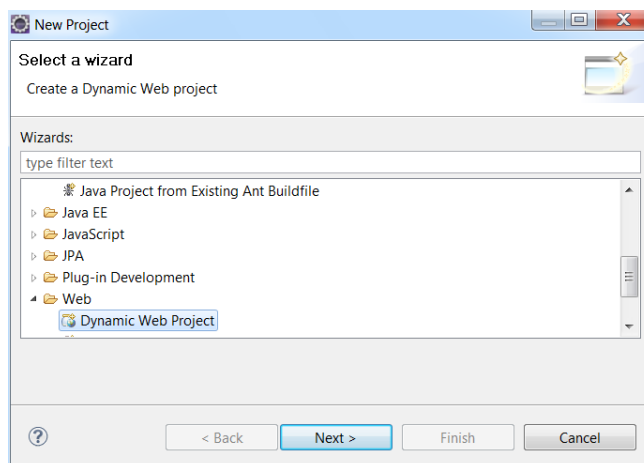


Рис. 30. Создание нового проекта.

2) Укажите имя проекта Servlet. Остальные настройки, связанные с сервером приложений, на котором будет разворачиваться веб-приложение, должны отобразиться автоматически, в том числе и сервер, на котором планируется разворачивать (deploy) приложение. Если этого не произошло, то выберите сервер из поля со списком, как показано на Рис. 31 и нажмите Next.

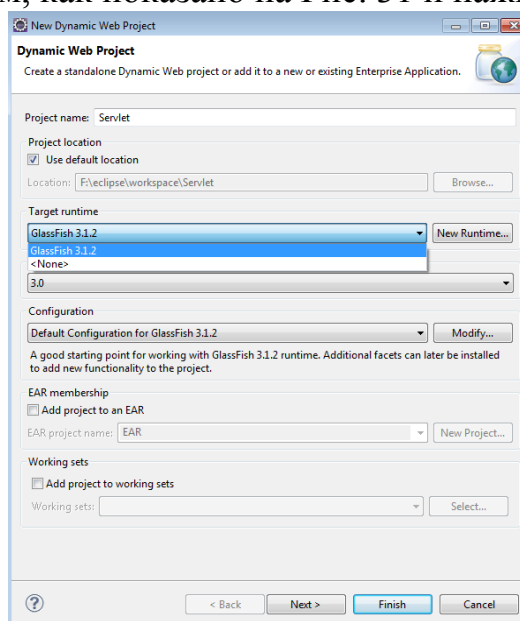


Рис. 31. Выбор сервера для развертывания сервлета.

- 3) Откроется окно показанное ниже, в котором следует принять установки по умолчанию и нажать кнопку Next (Рис. 32).

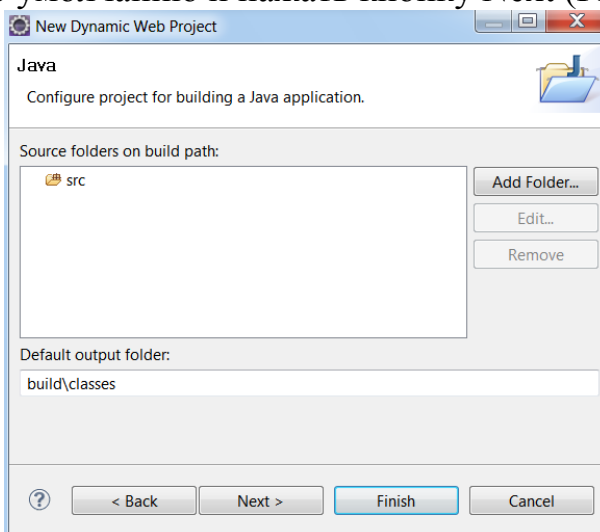


Рис. 32. Очередной шаг создания проекта.

Обратите внимание на значение поля Default output folder, которое указывается, но в дальнейшем его содержимое в проекте не раскрывается.

- 4) В появившемся окне (Рис. 33) можно отметить поле Generate web.xml deployment descriptor и нажать кнопку Finish. Эту установку следует сделать исключительно для знакомства с содержанием этого сгенерированного файла. Версия 6 JavaEE позволяет выполнять развертывание приложений без данного файла, а руководствуясь аннотациями в исходных текстах Java.

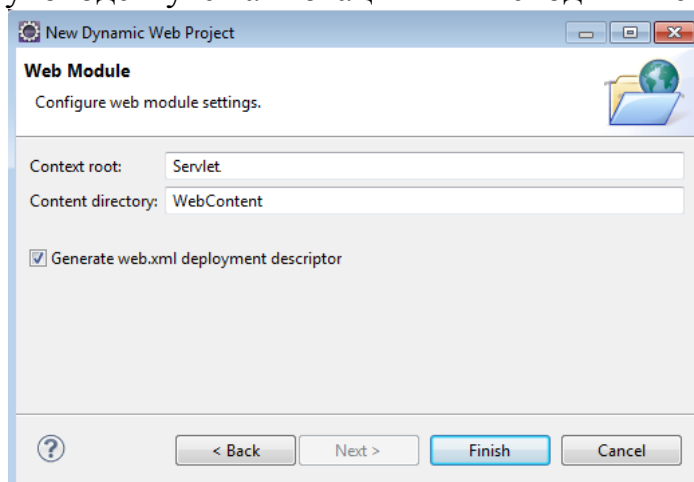


Рис. 33. Установка флага создания web.xml.

- 5) В результате будет построен проект структуры приложения, показанный на Рис. 34.

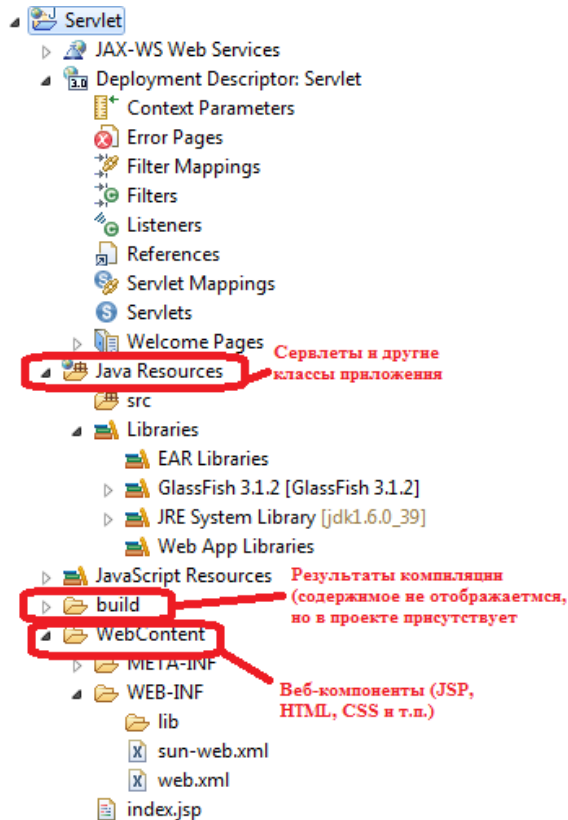


Рис. 34. Структура Web-приложения в окне Project Explorer.

Создание таблицы `employee` и вставка тестовых данных

Запустите сервер БД Derby, а именно, в командном окне (Start/All program/Accessories/Command Prompt необходимо ввести команду `asadmin.bat` из `C:\glassfish3\bin\asadmin.bat` и в появившейся командной строке `asadmin>` ввести `start-database`. Команда отработает правильно, если предварительно запущен сервер приложений Glassfish3. (Естественно, что в команду Path надо добавить путь `C:\glassfish3\bin` с тем, чтобы иметь возможность выполнить команду `asadmin` без указания пути к месту ее размещения (`C:\glassfish3\bin`).

При успешном выполнении в командном окне будет выведена информация, показанная на Рис. 35.

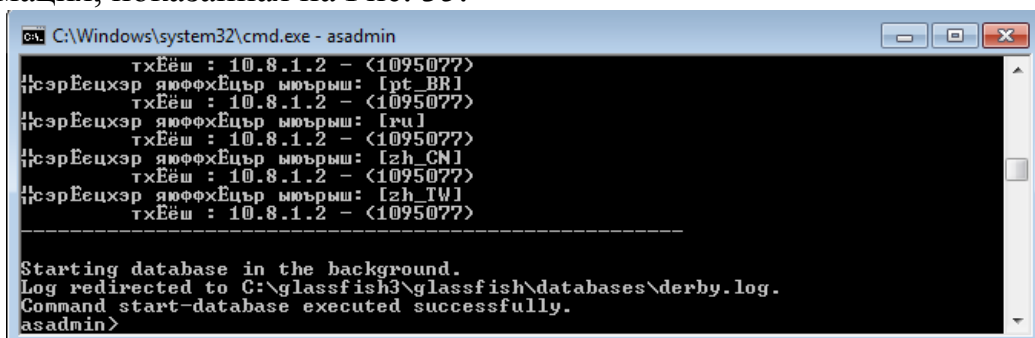


Рис. 35. Вывод сообщений при запуске сервера БД Derby.

- 1) Выберите проект Servlet и нажмите правую кнопку для открытия контекстного меню, из которого последовательно выберите Apache Derby/Add Apache Derby nature, что позволит выполнять некоторые действия с сервером баз данных Derby из среды IDE Eclipse.
- 2) Для хранения SQL-скриптов создадим новый файл employee.sql. В окне Project Explorer щелкните правой кнопкой мыши на значок проекта Servlet и выберите New/File, укажите имя файла employee.sql и нажмите Finish, как показано на Рис. 36.

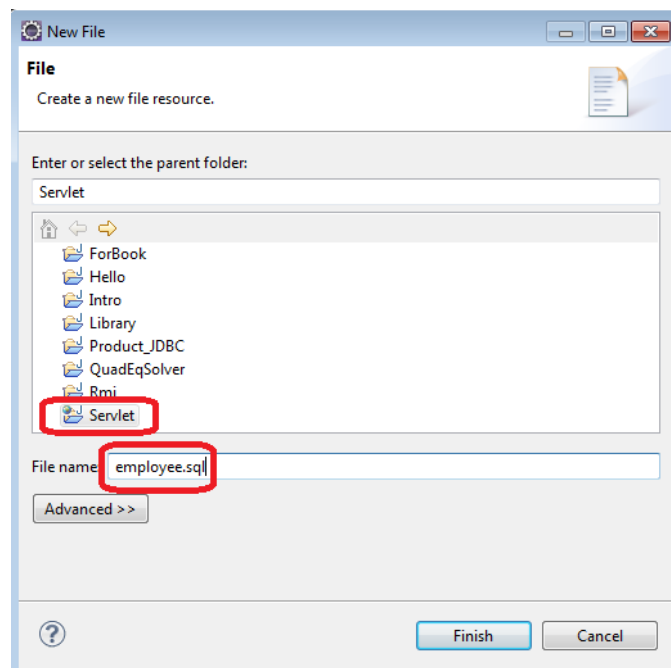


Рис. 36. Диалоговое окно создания файла.

- 3) Созданный файл автоматически открывается для редактирования. Скопируйте в файл следующие команды:

```
-- подключение
connect
'jdbc:derby://localhost:1527/myDB;create=true;user=me;password=mine';
-- прокомментируйте следующую строку, если требуется пересоздать
таблицу
-- drop table employee;
-- создание таблицы
create table employee(id integer, first_name varchar(20),
last_name varchar(20), designation varchar(20), phone
varchar(20));
--вставка тестовых данных
insert into employee values (1, 'Ivan', 'Ivanov', 'Manager', '11-
22-33');
insert into employee values (2, 'Nikolay', 'Ivanov', 'Programmer',
'33-44-55');
insert into employee values (3, 'Sergey', 'Petrov', 'System
administrator', '12-34-56');
```

```

insert into employee values (4, 'Alexey', 'Petrov', 'Manager',
'56-78-90');
insert into employee values (5, 'Vitaliy', 'Kuznetsov',
'Technician', '55-66-77');
-- выбрать все из таблицы для проверки
select * from employee;
-- отключение и выход
disconnect;
exit;

```

4) Сохраните файл нажатием на Ctrl-S.

5) Щелкните правой кнопкой мыши на файл employee.sql в окне Project Explorer и выберите Apache Derby/Run SQL Script using 'ij'.

6) В случае успешного выполнения скрипта в консоли выводится следующая информация:

```

ij version 10.7
ij> -- подключение
connect
'jdbc:derby://localhost:1527/myDB;create=true;user=me;password=mine';
ij> -- раскомментируйте следующую строку, если требуется пересоздать
таблицу
-- drop table employee;
-- создание таблицы
create table employee(id integer, first_name varchar(20), last_name
varchar(20), designation varchar(20), phone varchar(20));
0 rows inserted/updated/deleted
ij> --вставка тестовых данных
insert into employee values (1, 'Ivan', 'Ivanov', 'Manager', '11-22-
33');
1 row inserted/updated/deleted
ij> insert into employee values (2, 'Nikolay', 'Ivanov', 'Programmer',
'33-44-55');
1 row inserted/updated/deleted
ij> insert into employee values (3, 'Sergey', 'Petrov', 'System
administrator', '12-34-56');
1 row inserted/updated/deleted
ij> insert into employee values (4, 'Alexey', 'Petrov', 'Manager', '56-
78-90');
1 row inserted/updated/deleted
ij> insert into employee values (5, 'Vitaliy', 'Kuznetsov',
'Technician', '55-66-77');
1 row inserted/updated/deleted
ij> -- выбрать все из таблицы для проверки
select * from employee;

```

ID	FIRST_NAME	LAST_NAME	DESIGNATION	PHONE
1	Ivan	Ivanov	Manager	11-22-33
2	Nikolay	Ivanov	Programmer	33-44-55
3	Sergey	Petrov	System administrator	12-34-56


```

4          |Alexey          |Petrov          |Manager
|56-78-90
5          |Vitaliy          |Kuznetsov       |Technician
|55-66-77

```

```

5 rows selected
ij> -- отключение и выход
disconnect;
ij> exit;

```

После выполнения этого скрипта база данных MySQL будет создана в подкаталоге `c:\glassfish3\glassfish\databases\`

Разработка сервлета

Перед началом программирования сервлета, создадим обычный Java-класс `Employee`, который будет использоваться для представления и передачи данных о сотрудниках.

- Создайте новый Java-класс, нажав правой кнопкой мыши на проект `Servlet` и выбрав пункт меню `New/Class`. Назовите класс `Employee`, укажите для него пакет `ru.ifmo.javaee` и нажмите кнопку `Finish`, как показано на Рис. 37.

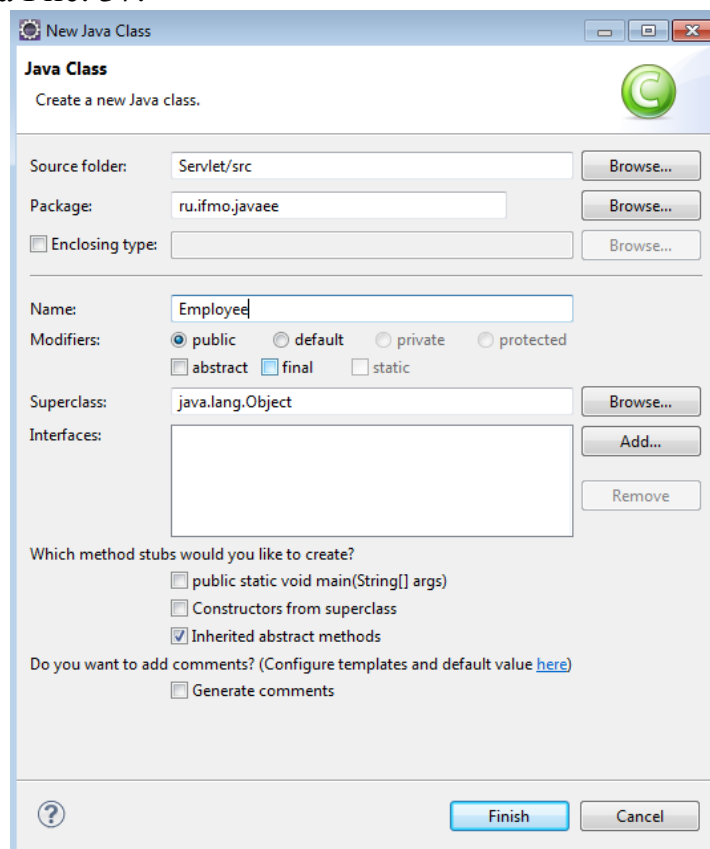


Рис. 37. Диалоговое окно создания нового класса.

- В классе `Employee` создайте пять полей, соответствующих столбцам таблицы `employee`, добавьте конструкторы и набор `get/set` методов. Полный код класса `Employee` приведен ниже:

```

package ru.ifmo.javaee;
import java.io.Serializable;
public class Employee implements Serializable {
    private Long id;
    private String firstName;
    private String lastName;
    private String designation;
    private String phone;
    public Employee() {
    }
    public Employee(Long id, String firstName, String lastName,
        String designation, String phone) {
        super();
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.designation = designation;
        this.phone = phone;
    }
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getDesignation() {
        return designation;
    }
    public void setDesignation(String designation) {
        this.designation = designation;
    }
    public String getPhone() {
        return phone;
    }
    public void setPhone(String phone) {
        this.phone = phone;
    }
}

```

Теперь приступим к разработке сервлета.

- Для создания нового сервлета щелкните правой кнопкой мыши на проект Servlet, выберите пункт меню New/Other, укажите Web/Servlet и нажмите Next.

- Укажите пакет (Java package), в котором будет размещен сервлет, например, ru.ifmo.javaee.
- Укажите имя класса сервлета (Class name) EmployeeServlet и нажмите Finish, как показано на Рис. 38.

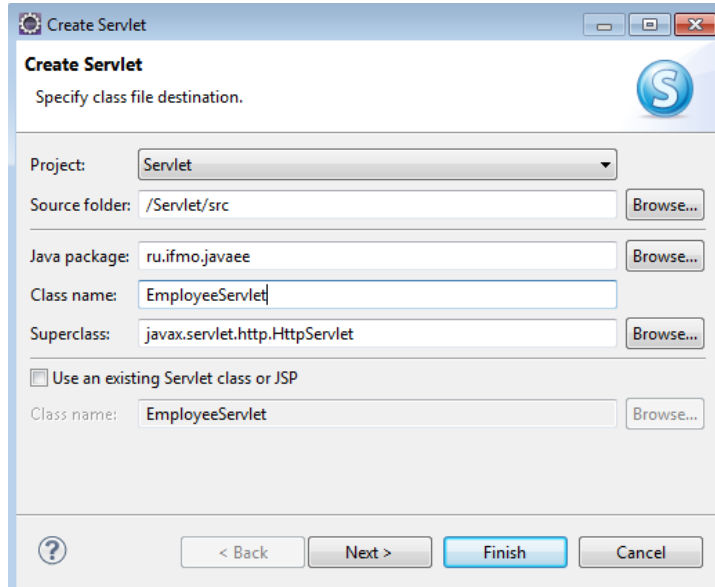


Рис. 38. Диалоговое окно создания сервлета.

- Программный код сервлета приведен ниже. Основная бизнес логика сосредоточена в методе doGet(), который выполняется на сервере после того, как пользователь запустил сервлет на выполнение. Скопируйте код и сохраните сервлет, нажав Ctrl-S.

```

package ru.ifmo.javaee;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.util.ArrayList;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/EmployeeServlet")
public class EmployeeServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public EmployeeServlet() {
        super();
    }
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        try {
            response.setContentType("text/html;charset=UTF-8");

```

```

// Получение из http-запроса значения параметра lastname
String lastname = request.getParameter("lastname");
// Коллекция для хранения найденных сотрудников
ArrayList<Employee> employees = new ArrayList<Employee>();
// Загрузка драйвера БД Derby

Class.forName("org.apache.derby.jdbc.ClientDriver").newInstance();
// Получение соединения с БД
Connection con = DriverManager.getConnection(

"jdbc:derby://localhost:1527/myDB;create=true;user=me;password=min
e");
// Выполнение SQL-запроса
ResultSet rs = con.createStatement().executeQuery(
"Select id, first_name, last_name, designation, phone "
+ "From employee " + "Where last_name like '"
+ lastname + "'");
// Перечисление результатов запроса
while (rs.next()) {
// По каждой записи выборки формируется
// объект класса Employee.
// Значения свойств заполняются из полей записи
Employee emp = new Employee(
rs.getLong(1),
rs.getString(2),
rs.getString(3),
rs.getString(4),
rs.getString(5));
// Добавление созданного объекта в коллекцию
employees.add(emp);
}
// Закрываем выборку и соединение с БД
rs.close();
con.close();
// Выводим информацию о найденных сотрудниках
PrintWriter out = response.getWriter();
out.println("Найденные сотрудники<br>");
for (Employee emp: employees) {
out.print(emp.getFirstName() + " " +
emp.getLastName() + " " +
emp.getDesignation() + " " +
emp.getPhone() + "<br>");
}
} catch (Exception ex) {
ex.printStackTrace();
throw new ServletException(ex);
}
}
protected void doPost(HttpServletRequest request,
HttpServletRequest response) throws ServletException, IOException
{
}
}

```

- При создании сервлета с помощью мастера (как было описано выше), конфигурационный файл web.xml генерируется автоматически. Для просмотра/редактирования этого файла в окне Project Explorer дваж-

ды щелкните на элемент Servlet/Web Content/WEB-INF/web.xml и появится окно редактирования, как показано на Рис. 39.

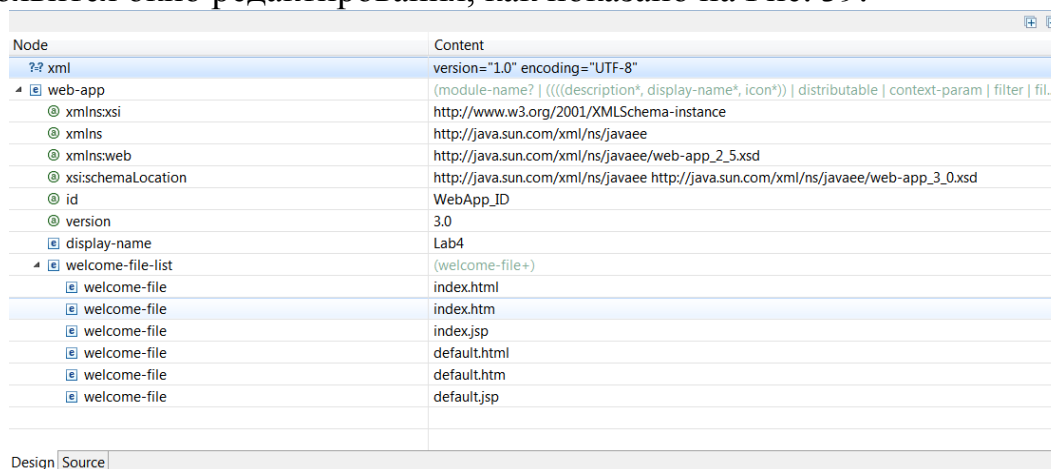


Рис. 39. Окно редактирования web.xml.

В редакторе Web XML Editor не представлена конфигурация Servlets, поскольку используется аннотация для описания сервлета, в частности, утверждение `@WebServlet("/EmployeeServlet")` конфигурирует сервлет, к которому следует обращаться по ссылке `/EmployeeServlet`.

Содержимое файла web.xml также можно просмотреть в виде исходного XML-документа. Для этого следует выбрать закладку Source в нижней части окна редактора, как показано на Рис. 40.



Рис. 40. Окно редактирования web.xml в исходном коде.

Сборка, развертывание и тестирование приложения

- 1) Проверьте, чтобы в пункте меню Project|Build Automatically был установлен «флажок» (Рис. 41), означающий, что автоматически выполняется компиляция исходных файлов и упаковка приложения. Eclipse в случае успешной сборки проекта автоматически выполняет и развертывание приложения.

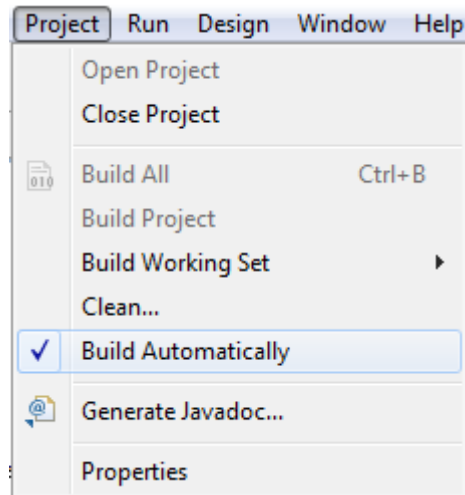


Рис. 41. Пункт меню Project

- 2) Также необходимо иметь в виду, что сервер баз данных Derby должен быть запущен.
- 3) Для проверки работоспособности выберите в проекте Servlet файл сервлета, и, нажав правую кнопку из контекстного меню последовательно выберите Run As, и затем Run on Server. После этого откроется диалоговое окно выбора сервера для развертывания и проверки работы сервлета, в котором следует выбрать сервер GlassFish и нажать кнопку Finish, как показано на Рис. 42.

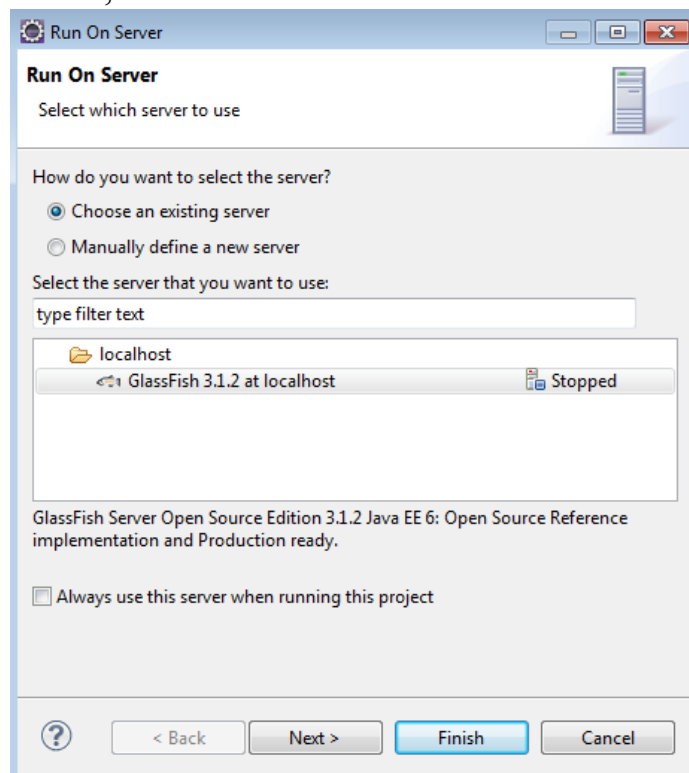


Рис. 42. Диалоговое окно выбора сервера приложений.

Если сервер еще не запущен, то он стартует, и в процессе запуска может вывести диалоговое окно для ввода Login/password, как показано на Рис. 43.

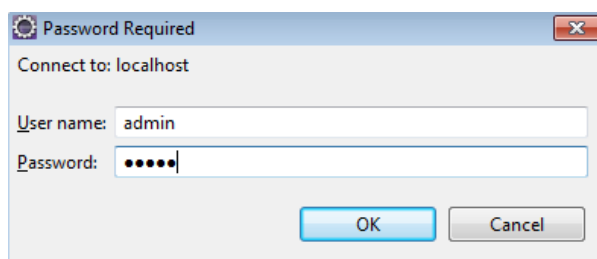


Рис. 43. Диалоговое окно при запуске GlassFish AS.

- 4) В зависимости от установок IDE Eclipse в меню Windows|Preferences|General|Web Browser, откроется браузер, в котором произойдет обращение к сервлету EmployeeServlet, и в результате выполнения сервлета будет выведена единственная строка заголовка «Найденные сотрудники», либо выведено приветствие "Hello, World!" из файла Index.jsp по умолчанию.

Скорректируйте ссылку на `http://localhost:8080/Server/EmployeeServlet?lastname=Ivanov`. Обратите внимание, что в запросе выполняется обращение к сервлету и передается параметр `lastname` со значением `Ivanov`. Результат работы серверной программы можно наблюдать на Рис. 44.

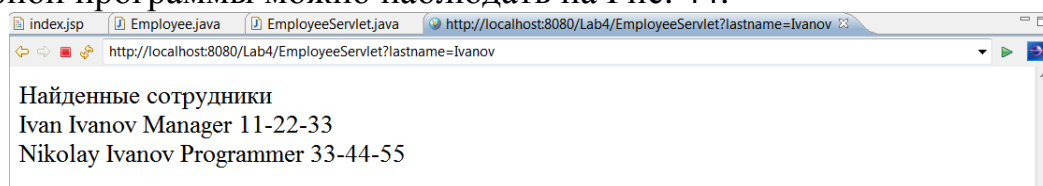


Рис. 44. Результат работы сервлета EmployeeServlet.

Servlet API и события жизненного цикла

Servlet API

Как уже упоминалось ранее, Servlet API – это коллекция классов и интерфейсов Java, которые позволяют создавать сервлеты и располагаются в пакетах `javax.servlet` и `javax.servlet.http`.

Классы и интерфейсы пакета `javax.servlet` обеспечивают взаимодействие между сервлетом и клиентом. При создании сервлета необходимо реализовать интерфейс `Servlet` непосредственно или расширением класса, который реализует этот интерфейс. Интерфейс `Servlet` определяет различные методы жизненного цикла. Другими широко используемыми интерфейсами пакета `javax.servlet` являются:

- Интерфейс `ServletRequest`
- Интерфейс `ServletResponse`
- Интерфейс `ServletContext`

Рис. 45 показывает обобщенную объектную модель, представляющую иерархию классов пакета `javax.servlet`:

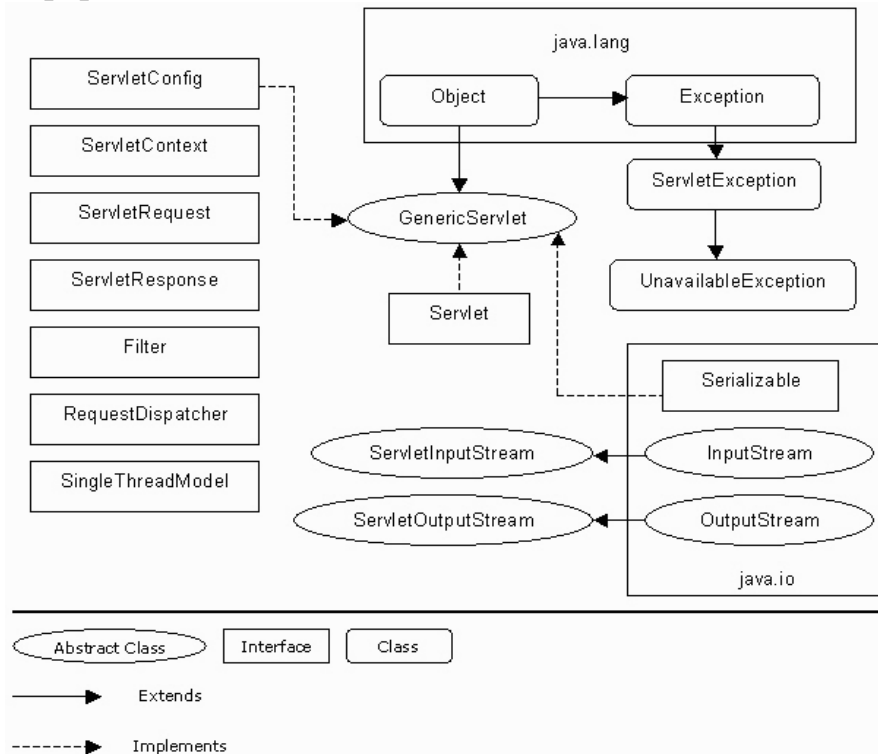


Рис. 45. Обобщенная объектная модель пакета `javax.servlet`.

Интерфейс `ServletRequest` определяет несколько важных методов в дополнение к методам, которые рассматриваются в этом разделе:

- `public String getProtocol()`, который возвращает название протокола и версии протокола, использованного для отправки запроса.
- `public BufferedReader getReader()`, который возвращает тело объекта запроса в символьной форме.
- `public String getScheme()`, который возвращает тип протокола, используемого для выполнения запроса: HTTP, FTP или HTTPS.

Интерфейс `ServletRequest`

Интерфейс `ServletRequest` содержит методы для обработки клиентских запросов к сервлету. При вызове сервлета, веб-контейнер передает объекты, которые реализуют интерфейсы `ServletRequest` и `ServletResponse`, методу `service()` сервлета. Таблица 9 показывает различные методы интерфейса `ServletRequest`.

Таблица 9.

Метод	Описание
<code>public String getParameter(String paramName)</code>	Возвращает объект <code>String</code> , содержащий значение заданного параметра запроса.
<code>public String[] getParameterValues(String paramName)</code>	Возвращает массив объектов <code>String</code> , содержащий все значения параметра запроса.
<code>public Enumeration getParameterNames()</code>	Возвращает <code>Enumeration</code> , содержащий все имена параметров в виде объектов <code>String</code> , которые содержит запрос к сервлету.
<code>public String getRemoteHost()</code>	Возвращает <code>String</code> , содержащий точное имя компьютера, с которого был отправлен запрос.
<code>public String getRemoteAddr()</code>	Возвращает <code>String</code> , содержащий IP-адрес компьютера, с которого был отправлен запрос.

Интерфейс `ServletResponse`

Интерфейс `ServletResponse` содержит методы, которые позволяют сервлету реагировать на клиентские запросы. Сервлет может отправить ответ в виде символьных или двоичных данных. Поток `PrintWriter` может быть использован для передачи символьных данных в ответе сервлета, а поток `ServletOutputStream` – для передачи двоичных данных. Таблица 10 показывает различные методы интерфейса `ServletResponse`.

Таблица 10.

Метод	Описание
<code>public ServletOutputStream getOutputStream() throws IOException</code>	Возвращает объект класса <code>ServletOutputStream</code> , который представляет выходной поток для передачи двоичных данных в ответе.
<code>public PrintWriter getWriter() throws IOException</code>	Возвращает объект класса <code>PrintWriter</code> , который сервлет использует для передачи символьных данных в ответе.

Метод	Описание
<code>public void setContentType(String type)</code>	Устанавливает тип Multipurpose Internet Mail Extensions (MIME) для ответа сервлета. Некоторые из MIME-типов: text/plain, image/jpeg и text/html.

Интерфейс `ServletContext`

Интерфейс `ServletContext` предоставляет информацию сервлетам о среде, в которой они выполняются. Контекст также называют контекстом сервлета или веб-контекстом, и он создается веб-контейнером как объект интерфейса `ServletContext`. Этот объект представляет контекст, внутри которого выполняется веб-приложение. Веб-контейнер создает объект `ServletContext` для каждого развернутого веб-приложения. Объект `ServletContext` можно использовать для определения пути к другим файлам веб-приложения, для доступа к другим сервлетам веб-приложения и записи сообщений в журнал сервера приложений. Также объект `ServletContext` можно использовать для установки атрибутов, к которым другие сервлеты приложения могут получать доступ. Таблица 11 показывает методы интерфейса `ServletContext`:

Таблица 11.

Метод	Описание
<code>public void setAttribute(String attrname, Object value)</code>	Связывает объект с именем и сохраняет пару имя/значение как атрибут объекта <code>ServletContext</code> . Если атрибут уже существует, то этот метод замещает существующий атрибут.
<code>public Object getAttribute(String attrname)</code>	Возвращает объект, хранимый в объекте <code>ServletContext</code> с именем, переданным как параметр.
<code>public Enumeration getAttributeNames()</code>	Возвращает Enumeration объектов <code>String</code> , который содержит имена всех атрибутов контекста.
<code>public String getInitParameter(String pname)</code>	Возвращает значение параметра инициализации с именем, переданным как параметр.
<code>public Enumeration getInitParameterNames()</code>	Возвращает Enumeration объектов, который содержит имена всех атрибутов, доступных в этом запросе.

Метод	Описание
<code>public int getMajorVersion()</code>	Возвращает целое значение, определяющее номер версии Servlet API, которую поддерживает веб-контейнер. Если ваш веб-контейнер поддерживает Servlet API версии 2.4, этот метод вернет 2.
<code>public int getMinorVersion()</code>	Возвращает целое значение, определяющее номер подверсии Servlet API, которую поддерживает веб-контейнер. Если веб-контейнер поддерживает Servlet API версии 2.4, этот метод возвращает 4.

Для использования объекта `ServletContext` необходимо получить объект `ServletContext` в методе `init()` сервлета. Метод `getServletContext()` интерфейса `ServletConfig` позволяет получить объект `ServletContext`. Следующий фрагмент кода можно использовать для получения объекта `ServletContext`:

```
ServletContext ctx;
public void init(ServletConfig cfig)
{
    ctx = cfig.getServletContext();
}
```

После получения объекта `ServletContext` можно установить атрибуты объекта `ServletContext` с помощью метода `setAttribute()`. Поскольку объект `ServletContext` доступен всем сервлетам веб-приложения, другие сервлеты могут извлекать атрибуты из объекта `ServletContext`, используя метод `getAttribute()`. Например, рассмотрим веб-приложение, в котором сервлет сохраняет URL JDBC как атрибут объекта `ServletContext` для доступа к базе данных, который другие сервлеты приложения могут извлекать.

В следующем примере используются методы `setAttribute()` и `getAttribute()` интерфейса `ServletContext` для создания сервлета `SettingCntx`, который устанавливает URL JDBC как атрибут контекста:

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.GenericServlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
```

```

import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
@WebServlet("/SettingCntx")
public class SettingCntx extends HttpServlet {
    private static final long serialVersionUID = 1L;
    ServletContext ctx;
    public SettingCntx() {
        super();
        // TODO Auto-generated constructor stub
    }
    public void init(ServletConfig cfig)
    {
        /* Получение объекта ServletContext */
        ctx = cfig.getServletContext();
    }
    public void service(ServletRequest request, ServletResponse re-
sponse) throws ServletException, IOException
    {
        /* Установка атрибута контекста */
        ctx.setAttribute("URL", "jdbc:odbc:EmployeesDB");
        /* Получение объекта PrintWriter */
        PrintWriter pw = response.getWriter();
        /* Отправка ответа, что атрибут URL установлен */
        response.setContentType("text/html");
        pw.println("<B>The JDBC URL has been set as a context at-
tribute</B>");
    }
}

```

В приведенной выше программе, сервлет SettingCntx получает объект ServletContext в методе init(). В методе service() сервлет использует метод setAttribute() для присвоения значения jdbc:odbc:EmployeesDB атрибуту URL. В конце сервлет использует объект PrintWriter для вывода сообщения, что атрибут установлен.

Сервлет RetrievingCntx извлекает атрибут URL и выводит значение атрибута. Можно использовать следующий код для создания сервлета RetrievingCntx для получения и вывода атрибута URL:

```

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/RetrievingCntx")
public class RetrievingCntx extends HttpServlet {
    private static final long serialVersionUID = 1L;
    ServletContext ctx;
    String url;
    public void init(ServletConfig cfig) {
        /* Получение объекта ServletContext */

```

```

        ctx = cfig.getServletContext();
    }
    public RetrievingCntx() {
        super();
    }
    protected void service(HttpServletRequest request,
        HttpServletResponse response) throws
ServletException, IOException {
        /* Извлечение атрибута URL */
        url = (String) ctx.getAttribute("URL");
        /* Получение объекта PrintWriter */
        PrintWriter pw = response.getWriter();
        /* Отправка ответа для вывода значения атрибута URL */
        response.setContentType("text/html");
        pw.println("<B>The URL value is </B>:    " + url +
"<BR>");}
}

```

В приведенной выше программе, сервлет RetrievingCntx получает объект ServletContext в методе init(). В методе service() сервлет использует метод getAttribute() для получения значения атрибута URL. В конце сервлет использует объект PrintWriter для вывода значения атрибута.

При выполнении сервлета SettingCntx, сервлет устанавливает URL как атрибут объекта ServletContext. Сервлет отправляет ответ, что атрибут установлен, как показано на Рис. 46.

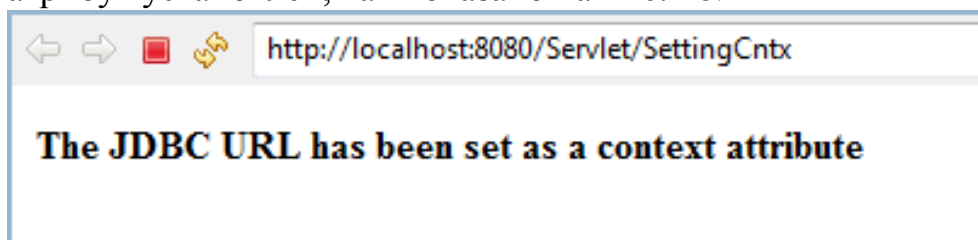


Рис. 46. Результат работы сервлета SettingCntx.

В процессе выполнения сервлет RetrievingCntx извлекает URL, который был установлен сервлетом SettingCntx и отправляет ответ для вывода значения URL, как показано на Рис. 47.

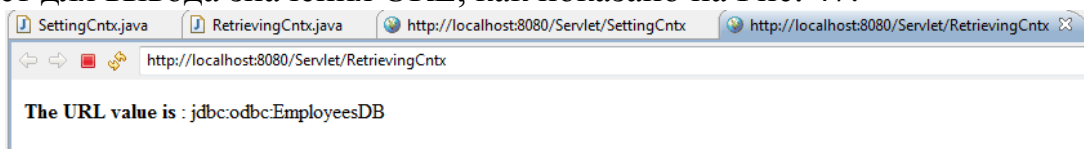


Рис. 47. Результат работы сервлета RetrievingCntx.

Параметры инициализации контекста

Параметры инициализации контекста - это пары имя/значение, которые можно задавать во время развертывания веб-приложения. Можно использовать параметры инициализации контекста для определения

информации, которая доступна для всех сервлетов веб-приложения, например, можно задать адрес электронной почты как параметр. Затем можно извлечь это значение в других сервлетах приложения, которым необходимо вывести ваш адрес.

При развертывании веб-приложения веб-контейнер считывает параметры инициализации из дескриптора развертывания и инициализирует ими объект `ServletContext`. Методы `getInitParameter()` и `getInitParameterNames()` объекта `ServletContext` можно использовать для получения значений параметров в сервлете.

Элемент `context-param` дескриптора развертывания определяет пару имя/значение для параметра инициализации контекста. Следующий фрагмент кода показывает элемент `context-param` дескриптора развертывания, который задает параметр с именем `email_id` и значение `aad@hotmail.com`:

```
<context-param>
  <param-name> email_id</param-name>
  <param-value> customer@hotmail.com</param-value>
</context-param>
```

Пакет `javax.servlet.http`

Пакет `javax.servlet.http` является расширением пакета `javax.servlet` и классы и интерфейсы этого пакета обрабатывают сервлеты, которые работают, используя протокол HTTP. Эти сервлеты также называются HTTP-сервлетами и для их создания необходимо расширить класс `HttpServlet`. Ниже представлены наиболее часто используемые интерфейсы пакета `javax.servlet.http`:

- Интерфейс `HttpServletRequest`
- Интерфейс `HttpServletResponse`
- Интерфейс `HttpSession`

Рис. 48 показывает обобщенную объектную модель, представляющую иерархию классов пакета `javax.servlet.http`:

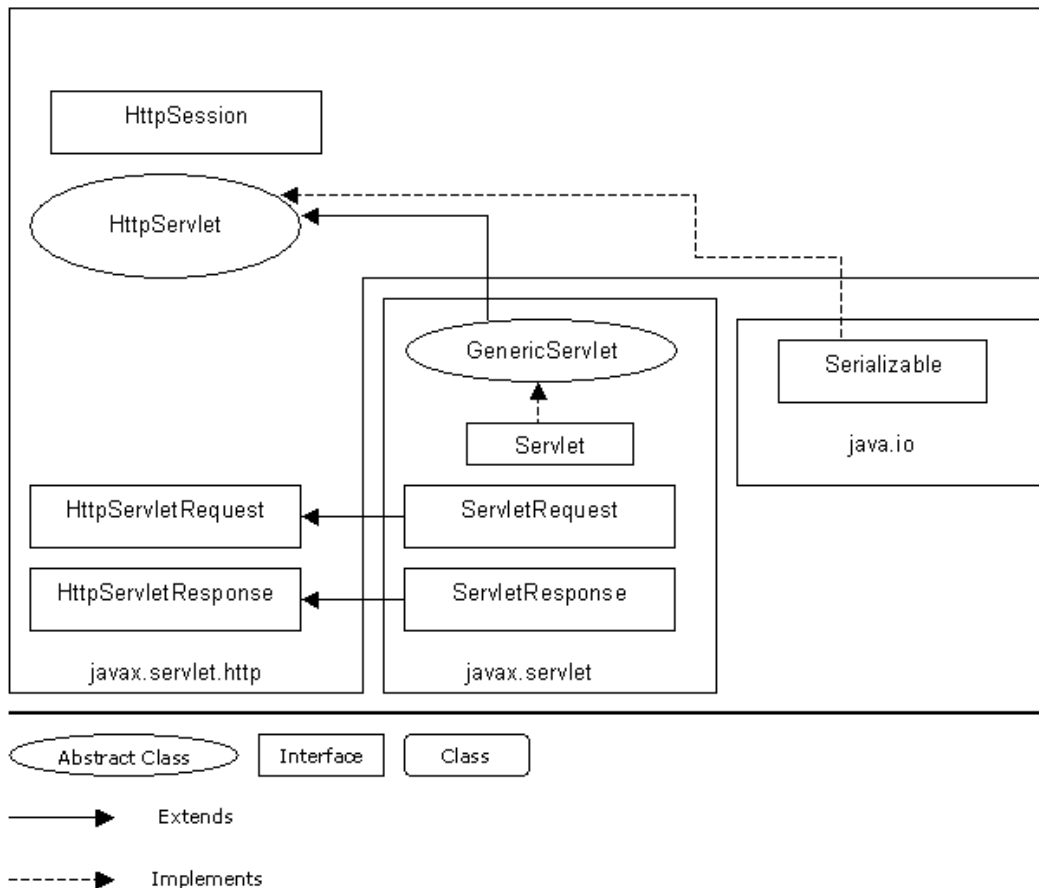


Рис. 48. Обобщенная объектная модель пакета javax.servlet.http
Интерфейс HttpServletRequest определяет несколько полезных методов в дополнение к тем, которые мы рассмотрим в этом разделе:

- public String getRequestedSessionId(), который возвращает id сеанса, определяющий клиентом.
- public boolean isRequestedSessionIdValid(), который проверяет, является ли id сеанса действующим, или нет.
- public String getAuthType(), который возвращает тип схемы, используемой для защиты сервлета.

Интерфейс HttpServletRequest

Интерфейс HttpServletRequest расширяет интерфейс ServletRequest для представления информации из запроса, отправленного клиентом HTTP. Он включает поддержку получения параметров запроса и доступ к информации из заголовка HTTP-запроса.

HTTP-запросы имеют несколько сопутствующих заголовков, предоставляющих дополнительную информацию о клиенте, например, на-

звание и версия браузера, отправляющего запрос и другие. Ниже представлены некоторые полезные заголовки HTTP-запроса:

Accept: Определяет MIME-тип, который клиент предпочитает использовать.

Accept-Language: Определяет язык, на котором клиент предпочитает получать запрос.

User-Agent: Определяет название и версию браузера, отправившего запрос.

В таблице 12 описаны некоторые методы интерфейса `HttpServletRequest`.

Таблица 12.

Метод	Описание
<code>public String getHeader(String fieldname)</code>	Возвращает значение поля заголовка запроса, такие как <code>Cache-Control</code> и <code>Accept-Language</code> , указанные в параметре.
<code>public Enumeration getHeaders(String sname)</code>	Возвращает все значения, связанные с конкретным заголовком запроса в виде объекта <code>Enumeration</code>
<code>public Enumeration getHeaderNames()</code>	Возвращает имена всех заголовков запроса, к которым сервлет может получить доступ, в виде объекта <code>Enumeration</code>

Сервлеты используют методы `getHeader()`, `getHeaderNames()` и `getHeaders()` для получения значений из заголовков HTTP-запроса. Следующий код можно использовать для получения информации из заголовка запроса:

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/HttpRequestHeaderDemo")
public class HttpRequestHeaderDemo extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public HttpRequestHeaderDemo() {
        super();
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        /* Получение всех имен заголовков в Enumeration */
```


Интерфейс HttpServletResponse

Интерфейс HttpServletResponse расширяет интерфейс ServletResponse и предоставляет методы для обработки ответов, кодов статуса и заголовков ответов для сервлетов, которые взаимодействуют с помощью HTTP. Таблица 13 описывает некоторые методы интерфейса HttpServletResponse.

Таблица 13.

Метод	Описание
<code>void addHeader(String hname, String hvalue)</code>	Добавляет заголовок hname со значением hvalue. Этот метод добавляет новый заголовок, если заголовок уже существует.
<code>void addIntHeader(String hname, int hvalue)</code>	Добавляет заголовок hname со значением hvalue.
<code>void addDateHeader(String hname, long datev)</code>	Добавляет заголовок hname со значением, равным datev. Значение datev должно быть в миллисекундах, которые прошли с полуночи 1 января 1970.
<code>boolean containsHeader(String hname)</code>	Возвращает true, если заголовок hname уже установлен с конкретным значением, и false, в противном случае.
<code>void sendRedirect(String url)</code>	Перенаправляет запрос указанному URL.

Установка заголовков ответа

Сервлет может предоставить дополнительную информацию о содержании ответа, который отправляется клиенту сервером приложений, добавляя новые HTTP-заголовки. Ниже представлены некоторые дополнительные заголовки HTTP-ответа, которые может устанавливать сервлет:

Content-Type: Определяет MIME-тип данных, отправленных сервлетом, такой как `text/html` и `text/plain`.

Cache-control: Определяет информацию для кэширования сервлета. Если значение `Cache-control` установлено в `no-store`, то сервлет не кэшируется браузером или прокси-сервером. Значение `max-age` задает время в секундах, в течение которого сервлет должен находиться в кэше.

Expires: Определяет время, когда содержимое сервлета может измениться, или когда его данные станут неверными, и что требуется перезагрузка сервлета браузером для вывода обновленных данных.

Сервлеты могут использовать методы `setHeader()` и `setDateHeader()` для установки значений HTTP-заголовков в объекте ответа. Следующий фрагмент кода можно использовать для установки данных HTTP-заголовка:

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    PrintWriter out = res.getWriter();
    /* Задание значения типа long для хранения времени до обновления
*/
    long mseconds;
    java.util.Date date = new java.util.Date();
    mseconds = date.getTime() + 10000;
    /* Установка заголовка Content-Type со значением text/html */
    res.setHeader("Content-Type", "text/html");
    /* Установка заголовка Expires со значением mseconds */
    res.setDateHeader("Expires", mseconds);
    /* Отправка текущих даты и времени в ответе */
    out.println("<HTML><BODY>");
    out.println(new java.util.Date());
    out.println("</BODY></HTML>");
}
```

В приведенном фрагменте кода первый вызов метода `setHeader()` устанавливает значение поля заголовка `Content-Type` в `text/html`. Затем значение поля заголовка `Expires` устанавливается в `mseconds`. Переменная `mseconds` является значением типа `long`, полученным добавлением 10000 миллисекунд к количеству миллисекунд, возвращенным методом `getTime()`. Метод `getTime()` класса `java.util.Date` возвращает количество миллисекунд, прошедшее с полночи 1 января 1970. Значение заголовка `Expires` гарантирует, что HTML-данные, сгенерированные сервлетом, которые показывают дату конечному пользователю, устареет через десять секунд. Это значит, что если конечный пользователь не запросит страницу через десять секунд, браузер отправит новый запрос серверу и выведет обновленные данные.

С помощью установки заголовка при помощи метода `setContentType` можно предоставить возможность клиенту вызвать для работы с полученными данными соответствующее приложение, как это делается в методе `doGet` сервлета, представленном ниже:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("application/vnd.ms-excel");
    PrintWriter out = response.getWriter();
    out.println("\tQ1\tQ2\tQ3\tQ4\tSUM");
    out.println("Apple\t718\t287\t932\t429\t=СУММ(B2:E2)");
}
```

```

        out.println("Pineapple\t757\t686\t973\t830\t=CYMM(B3:E3)");
    }

```

В результате выполнения этого сервлета полученные данные открываются в приложении Microsoft Excel, как показано на Рис. 50.

A	B	C	D	E	F
	Q1	Q2	Q3	Q4	Total
Apple	718	287	932	429	2366
Pineapple	757	686	973	830	3246

Рис. 50 Страница, открытая в MS Excel

Перенаправление запросов клиентов

Метод `sendRedirect()` интерфейса `HttpServletResponse` применяется для перенаправления запроса другому URL. Рассмотрим ситуацию, когда на веб-сайте университета предлагается онлайн-регистрация на технические и экономические дисциплины. На начальной странице веб-сайта находятся две альтернативные кнопки, для выбора технического или экономического направления. Если выбирается кнопка с техническим направлением и затем кнопка `Submit`, откроется онлайн-регистрационная форма для технических дисциплин, в противном случае открывается онлайн-регистрационная форма для экономических дисциплин. Следующий фрагмент кода показывает использование метода `sendRedirect()`:

```

public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    /*
     * Получение имени выбранной пользователем кнопки, отправ-
    ленной
     * как параметр запроса
     */
    String yourchoice = req.getParameter("chooseoption");
    /*
     * Если выбрана радиокнопка "economic", перенаправление на
    страницу
     * medical.html
     */
    if (yourchoice.equals("economic"))
        res.sendRedirect("http://localhost:8080/ctx/servlet/economic.html"
        );
    /*
     * Если выбрана радиокнопка "engineering", перенаправление
    на страницу
     * engineering.html
     */

```

```

        if (yourchoice.equals("engineering"))
            res.sendRedirect("http://localhost:8080/ctx/servlet/engineering.ht
ml");
    }

```

Интерфейс HttpSession

Протокол HTTP - это протокол без сохранения состояния, поскольку в протоколе нет ничего, что требует от браузера идентифицировать себя при каждом запросе, а также отсутствует постоянно установленное соединение между браузером и веб-сервером, которое сохранялось бы от страницы к странице. Когда пользователь посещает веб-страницу, браузер посылает HTTP-запрос серверу, который в свою очередь возвращает HTTP-ответ. Этим и ограничивается взаимодействие, и это представляет собой завершённую HTTP-транзакцию.

Интерфейс HttpSession обеспечивает методы для сохранения состояния конечного пользователя в веб-приложении на время сессии. Объект интерфейса HttpSession предоставляет средства для отслеживания и управления сессией конечного пользователя. Таблица 14 представляет несколько методов интерфейса HttpSession:

Таблица 14.

Метод	Описание
<code>public void setAttribute(String name, Object value)</code>	Связывает объект с именем и сохраняет пару name/value как атрибут объекта HttpSession. Если атрибут уже существует, то замещается существующий атрибут.
<code>public Object getAttribute(String name)</code>	Извлекает объект String, указанный в параметре, из объекта сеанса. Если не найден объект для указанного атрибута, то метод <code>getAttribute()</code> возвращает null.
<code>public Enumeration getAttributeNames()</code>	Возвращает Enumeration, содержащий имена всех объектов, которые связаны с атрибутами объекта сеанса.

API жизненного цикла сервлета

Во время жизненного цикла сервлета возникают различные события, связанные с контекстом сервлета, созданием сеанса и добавлением атрибутов в контекст сервлета. Веб-контейнер уведомляет класс-слушатель, когда возникает событие в течение жизненного цикла серв-

лета, и для того, чтобы получать уведомление о событии, классу-слушателю необходимо расширить интерфейс-слушатель Servlet API.

Типы событий

Ниже представлены различные события, которые генерируются во время жизненного цикла сервлета:

- События запроса к сервлету
- События контекста сервлета
- События сессии HTTP

События запроса к сервлету

Следующие два интерфейса представляют события запроса сервлета, которые имеют отношение к изменениям в объектах запроса, связанных с веб-приложением:

- `javax.servlet.ServletException`
- `javax.servlet.ServletRequestAttributeEvent`

Веб-контейнер создает объект `ServletRequestEvent` при:

- инициализации объекта запроса, когда запрос появляется;
- удалении объекта запроса, когда запрос не требуется.

Веб-контейнер создает объект `ServletRequestAttributeEvent`, когда происходит какое-либо изменение в атрибутах запроса к сервлету. Веб-контейнер создает объект `ServletRequestEvent` во время:

- добавления атрибута к объекту запроса к сервлету;
- удаления атрибута из объекта запроса к сервлету;
- замены атрибута в объекте запроса к сервлету другим атрибутом с тем же самым именем.

События контекста сервлета

События, которые связаны с изменениями в контексте веб-приложения, известны как события контекста сервлета. Следующие два интерфейса представляют события контекста сервлета:

- `javax.servlet.ServletContextEvent`
- `javax.servlet.ServletContextAttributeEvent`

Веб-контейнер создает объект `ServletContextEvent` во время:

- создания объекта `ServletContext` при фазе инициализации жизненного цикла сервлета;
- удаления объекта `ServletContext`.

Веб-контейнер генерирует объект `ServletContextAttributeEvent`, когда происходит какое-либо изменение в атрибуте контекста сервлета веб-приложения, в частности, при:

- добавлении атрибута к объекту контекста сервлета;
- удалении атрибута в объекте контекста сервлета;
- замене атрибута в объекте контекста сервлета другим атрибутом с таким же именем.

События сессии HTTP

События сессии HTTP связаны с изменениями в объекте сессии сервлета. Следующие интерфейсы представляют события сессии сервлета:

- `javax.servlet.http.HttpSessionEvent`;
- `javax.servlet.http.HttpSessionAttributeEvent`;
- `javax.servlet.http.HttpSessionActivationEvent`;
- `javax.servlet.http.HttpSessionBindingEvent`.

Веб-контейнер создает объект `HttpSessionEvent` при:

- создании новой сессии;
- недействительности сессии;
- завершении срока сессии.

Веб-контейнер генерирует объект `HttpSessionAttributeEvent`, когда происходит какое-либо изменение в объекте атрибута сессии, в частности, при:

- добавлении атрибута к объекту сессии;
- удалении атрибута из объекта сессии;
- замене атрибута в объекте сессии.

Веб-контейнер генерирует объект `HttpSessionBindingEvent`, когда объект сервлета связывается с сессией или разрывает связь сессии. Связывание объекта с сессией означает, что объект соотносится с сессией, а разрыв связи объекта сервлета означает, что объект не соотносится с сессией. Веб-контейнер генерирует объект `HttpSessionActivationEvent`, когда сеанс активизируется или деактивируется.

Обработка событий жизненного цикла сервлета

Интерфейс `ServletRequestListener` добавлен в версию Servlet API 2.4 в J2EE 1.4 (текущая версия 3.0 в JavaEE 6). Для успешной работы управления событиями жизненного цикла сервлета необходимо реализовать интерфейс `ServletContextListener` в классе, и после этого сервер

приложений JavaEE идентифицирует класс как класс-слушатель. Классы, которые получают уведомления о событиях жизненного цикла сервлета, известны как слушатели событий. Эти классы-слушатели реализуют один или более интерфейсов слушателей событий сервлета, которые определены в Servlet API. Классы-слушатели могут быть логически разделены на следующие категории:

- слушатели запросов к сервлету;
- слушатели контекста сервлета;
- слушатели сеанса http.

Слушатели запросов к сервлету

Слушатели запросов к сервлету – это классы, которые слушают и обрабатывают события запросов к сервлету. Слушатели запросов к сервлету могут реализовывать следующие интерфейсы для получения уведомления о событиях запроса:

- `javax.servlet.ServletRequestListener`;
- `javax.servlet.ServletRequestAttributeListener`.

Интерфейс `ServletRequestListener`

Интерфейс `ServletRequestListener` позволяет классу-слушателю получать уведомления от веб-контейнера об изменениях в объекте запроса сервлета. Необходимо реализовать интерфейс `ServletRequestListener` в классе-слушателе для получения уведомлений о событиях запроса. Ниже представлены методы интерфейса `ServletRequestListener`:

```
void requestInitialized(ServletRequestEvent e):
```

уведомляет класс-слушатель об инициализации запроса к сервлету, связанного с веб-приложением.

```
void requestDestroyed(ServletRequestEvent e):
```

уведомляет сервлет об удалении запроса к сервлету, связанному с веб-приложением.

Интерфейс `ServletRequestAttributeListener`

Интерфейс `ServletRequestAttributeListener` позволяет классу-слушателю получать уведомления от веб-контейнера об изменениях в атрибутах запроса. Ниже представлены методы интерфейса `ServletRequestAttributeListener`, которые можно использовать в классе-слушателе:


```
void attributeAdded
(ServletRequestAttributeEvent srae): информирует класс-
слушатель о добавлении атрибута запроса.
void attributeRemoved
(ServletRequestAttributeEvent srae): информирует класс-
слушатель об удалении атрибута запроса.
void attributeReplaced
(ServletRequestAttributeEvent srae): информирует класс-
слушатель о замене существующего атрибута запроса новым атрибутом
запроса.
```

Слушатели контекста сервлета

Слушатели контекста сервлета – это классы-слушатели, которые обрабатывают события контекста сервлета. Слушатели контекста сервлета могут реализовывать следующие интерфейсы для получения уведомлений об изменениях в контексте сервлета:

- `javax.servlet.ServletContextListener`;
- `javax.servlet.ServletContextAttributeListener`.

Интерфейс `javax.servlet.ServletContextListener`

Интерфейс `ServletContextListener` позволяет классу-слушателю получать уведомления от веб-контейнера об изменениях в контексте сервлета веб-приложения. Ниже представлены методы интерфейса `ServletContextListener`:

```
void contextInitialized(ServletContextEvent ce):
уведомляет класс-слушатель об инициализации контекста сервлета веб-
приложения.
```

```
void contextDestroyed(ServletContextEvent ce):
уведомляет класс-слушатель об удалении контекста сервлета веб-
приложения.
```

Интерфейс `javax.servlet.ServletContextAttributeListener`

Интерфейс `ServletContextAttributeListener` позволяет классу-слушателю получать уведомления от веб-контейнера об изменениях, происходящих в атрибутах контекста сервлета. Ниже представлены методы интерфейса `ServletContextAttributeListener`:

```

void attributeAdded
(ServletContextAttributeEvent scae): уведомляет класс-
слушатель о добавлении атрибута в контекст сервлета.
public void attributeRemoved
(ServletContextAttributeEvent scae): уведомляет класс-
слушатель об удалении атрибута из контекста сервлета.
public void attributeReplaced
(ServletContextAttributeEvent scae): уведомляет класс-
слушатель о замене существующего атрибута контекста сервлета новым
атрибутом.

```

Следующий фрагмент кода демонстрирует использование методов `attributeAdded()`, `attributeReplaced()` и `attributeRemoved()` интерфейса

`ServletContextAttributeListener`:

```

public void attributeAdded(ServletContextAttributeEvent scae)
{
    ServletContext sc=scae.getServletContext();
    /*Имя атрибута добавляется в журнал сервера*/
    sc.log("The Attribute Name:"+scae.getName());
    /*Значение атрибута добавляется в журнал сервера*/
    sc.log("The Attribute Value:"+scae.getValue());
}
/*Вызывается при замене имени атрибута в контексте сервлета*/
public void attributeReplaced(ServletContextAttributeEvent scae)
{
    /*Запись сообщения, что атрибут заменен*/
    ServletContext sc=scae.getServletContext();
    sc.log("The Attribute is replaced in Servlet Context ");
}
/*Вызывается при удалении атрибута из контекста сервлета*/
public void attributeRemoved(ServletContextAttributeEvent scae)
{
    /*Запись сообщения, что атрибут удален*/
    ServletContext sc=scae.getServletContext();
    sc.log("The Attribute is removed from Servlet Context ");
}

```

В распределенных приложениях слушатели атрибута контекста сервлета получают уведомления об изменениях в атрибутах контекста сервлета только от локальной JVM.

Слушатели сеанса HTTP

Классы-слушатели, которые получают уведомления о событиях сеанса HTTP, известны как слушатели сеанса HTTP и могут реализовывать следующие интерфейсы для получения уведомлений о событиях сеанса:

- `javax.servlet.http.HttpSessionListener`;

- javax.servlet.http.HttpSessionAttributeListener;
- javax.servlet.http.HttpSessionActivationListener.

Интерфейс javax.servlet.http.HttpSessionListener

Интерфейс HttpSessionListener позволяет классу-слушателю получать уведомления от веб-контейнера об изменениях, имеющих место в объекте сеанса, связанного с веб-приложением. Ниже представлены методы интерфейса HttpSessionListener:

void sessionCreated (HttpSessionEvent hse): уведомляет класс-слушатель о создании объекта сеанса.

void sessionDestroyed (HttpSessionEvent hse): Уведомляет класс-слушатель об удалении существующего объекта сеанса.

Интерфейс javax.servlet.http.HttpSessionAttributeListener

Интерфейс HttpSessionAttributeListener позволяет классу-слушателю получать уведомления от веб-контейнера об изменении в атрибутах объекта сеанса. Ниже представлены методы интерфейса HttpSessionAttributeListener:

void attributeAdded (HttpSessionBindingEvent sbe): уведомляет класс-слушатель о добавлении атрибута в объект сеанса.

void attributeRemoved (HttpSessionBindingEvent sbe): уведомляет класс-слушатель об удалении атрибута из объекта сеанса.

void attributeReplaced (HttpSessionBindingEvent sbe): уведомляет класс-слушатель о замене атрибута сеанса новым атрибутом.

В распределенных приложениях слушатели атрибута контекста сервлета получают уведомления об изменениях в атрибутах сеанса только от локальной JVM.

Интерфейс javax.servlet.http.HttpSessionActivationListener

Интерфейс HttpSessionActivationListener позволяет классу-слушателю получать уведомления от веб-контейнера об изменениях в состоянии объекта сессии, соответствующего сервлету. Ниже представлены методы интерфейса HttpSessionActivationListener:

```
void sessionDidActivate(HttpSessionEvent se):
```

уведомляет класс-слушатель об активизации новой сессии в веб-приложении.

```
void sessionWillPassivate(HttpSessionEvent se):
```

уведомляет класс-слушатель о завершении существующей сессии в веб-приложении.

Рассмотрим пример применения средств управления событиями.

Постановка задачи:

Разработать приложение, которое будет записывать в журнал время, когда объекты запроса и контекста инициализируются, и когда добавляется атрибут к объекту контекста. Кроме того, приложение должно также записывать время, когда атрибут удаляется из объекта контекста и объекты запроса и контекста уничтожаются.

Решение

Чтобы решить поставленную задачу, необходимо выполнить следующие шаги:

1. Создать сервлет, который добавляет атрибут в объект контекста.
2. Создать сервлет, который записывает в журнал время события.
3. Развернуть оба сервлета.
4. Запустите программу.

1. Создание сервлета, который добавляет атрибут в объект контекста

Сервлет ServletEvents добавляет атрибут к экземпляру интерфейса ServletContext. Событие иницируется, когда атрибут добавляется в объект контекста, который обрабатывается другим сервлетом. Сервлет ServletEvents затем удаляет атрибут из объекта контекста. Следующий код можно использовать для создания сервлета ServletEvents:

```
package ru.ifmo.javaee;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;

import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```

@WebServlet("/ServletEvents")
public class ServletEvents extends HttpServlet {
    private static final long serialVersionUID = 1L;
    ServletContext ctx;
    PrintWriter pw;
    public void init(ServletConfig cfg)
    {
        /*Получение объекта ServletContext*/
        ctx = cfg.getServletContext();
    }

    public ServletEvents() {
        super();
    }
    public void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
    {
        /*Установка атрибута контекста*/
        ctx.setAttribute("URL", "jdbc:odbc:EmployeesDB");
        /*Получение объекта PrintWriter*/
        pw = response.getWriter();
        /*Отправка ответа, что атрибут URL установлен*/
        response.setContentType("text/html");
        /*Печать времени добавления атрибута к объекту контекста. */
        pw.println("<B>The JDBC URL has been set as a context at-
tribute at "
                    + new Date() + "</B></BR>");
        /*Удаление атрибута из объекта контекста. */
        ctx.removeAttribute("URL");
        /*Печать времени удаления атрибута из объекта контекста. */
        pw.println("<B>The JDBC URL has been removed from context at " +
new Date() + "</B>");
    }
}

```

В предложенном коде извлекается объект интерфейса ServletContext. После получения объекта контекста атрибут добавляется к объекту контекста и значение атрибута выводится со временем, когда произошло добавление к контексту. После вывода значения атрибута и времени, атрибут удаляется из объекта контекста. Также выводится время удаления объекта.

Создание этого сервлета в среде Eclipse ничем не отличается от того, что мы уже делали. Полученный сервлет после сохранения в проекте автоматически развернется на сервере.

2. Создание сервлета, который записывает время событий

Теперь создадим сервлет ServletEventListener, который обрабатывает все события, генерируемые сервлетом ServletEvents. Сервлет ServletEventListener реализует интерфейсы ServletContextListener,

ServletContextAttributeListener и ServletRequestListener. При создании этого специального типа класса необходимо выбрать в процессе создания тип создаваемого класса listener. Для этого выберите указателем мыши пакет `ru.ifmo.javaee` в Project Explorer и, нажав правую кнопку в контекстном меню, выберите New/Listener. Откроется диалоговое окно, в которое введите имя создаваемого класса ServletEventListener и нажмите кнопку Next, как показано на Рис. 51.

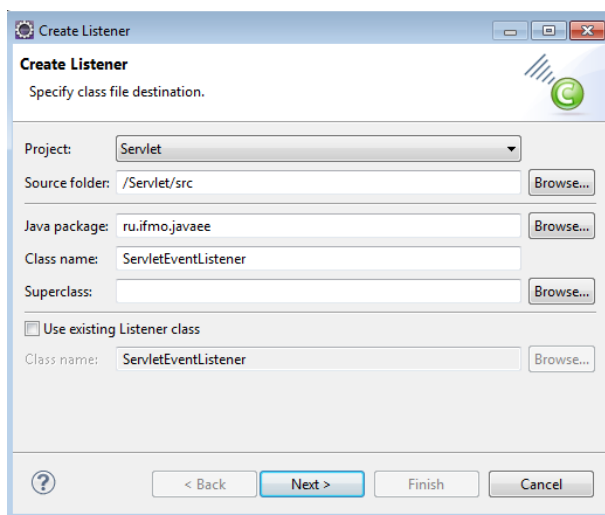


Рис. 51. Диалоговое окно определения имени класса

Откроется диалоговое окно, в котором необходимо указать те события и интерфейсы, которые будет реализовывать создаваемый класс-слушатель. Установите флажки как показано на Рис. 52 и нажмите Finish.

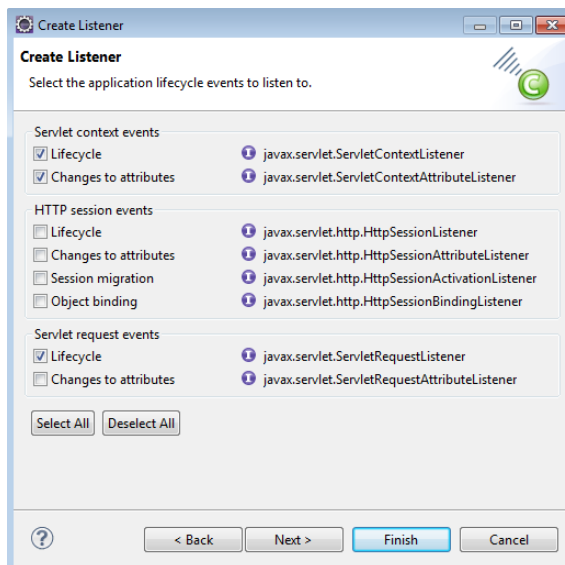


Рис. 52. Диалоговое окно содержания класса-слушателя.

Можно использовать следующий код для создания класса ServletEventListener:

```
package ru.ifmo.javaee;
import java.util.Date;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextAttributeEvent;
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;
import javax.servlet.annotation.WebListener;
/**
 * Application Lifecycle Listener implementation class
 ServletEventListener
 *
 */
@WebListener
public class ServletEventListener implements ServletContextListener,
    ServletContextAttributeListener, ServletRequestListener {
    ServletContext ctx;
    String name;
    String value;
    /**
     * Конструктор по умолчанию.
     */
    public ServletEventListener() {
    }
    public void requestDestroyed(ServletRequestEvent sre) {
        /* Запись в журнал времени удаления запроса. */
        ctx.log("A request has been destroyed at: " + new Date());
    }
    public void contextInitialized(ServletContextEvent sce) {
        ctx = sce.getServletContext();
        /* Запись в журнал времени инициализации контекста. */
        ctx.log("Context has been initialized at " + new Date());
    }
    public void attributeAdded(ServletContextAttributeEvent event) {
        /* Получение имени и значения атрибута. */
        name = event.getName();
        value = (String) event.getValue();
        /* Запись в журнал времени добавления атрибута в объект контекста.
*/
        ctx.log("An attribute with name: " + name + " and value: "
+ value
                + " has been added to the context at: " + new
Date());
    }
    public void attributeReplaced(ServletContextAttributeEvent event)
{
        /* Запись в журнал времени замены атрибута. */
        ctx.log("Attribute with name: " + name + " and value: " +
value
                + " has been replaced context at: " + new
Date());
    }
}
```

```

    }
    public void attributeRemoved(ServletContextAttributeEvent event) {
        /* Запись в журнал времени удаления атрибута из объекта кон-
        текста. */
        ctx.log("Attribute with name: " + name + " and value: " +
        value
            + " has been removed from the context at: " + new
        Date());
    }
    public void contextDestroyed(ServletContextEvent sce) {
        /* Запись в журнал времени удаления контекста. */
        ctx.log("Context has been destroyed at " + new Date());
    }
    public void requestInitialized(ServletRequestEvent sre) {
        /* Запись в журнал времени инициализации запроса. */
        ctx.log("A request has been initialized at: " + new
        Date());
    }
}

```

В предложенном коде определяются все методы трех интерфейсов `ServletContextListener`, `ServletContextAttributeListener` и `ServletRequestListener`. Внутри каждого метода записывается время появления события в файл `server.log`. О том, что данное приложение представляет собой слушатель свидетельствует аннотация `@WebListener` перед заголовком класса, которая значительно упрощает развертывание класса.

Запустите сервлет, который выведет информацию, как представлено на Рис. 53.



Рис. 53. Результат работы сервера `ServletEvents`

Отслеживая события время жизни, наш класс-слушатель `ServletRequestListener` запишет в файл `log` сервера информацию о событиях как показано на Рис. 54. (Файл расположен в `Glassfish3\glassfish\domains\domain1\log`)


```

server - Блокнот
Файл Правка Формат Вид Справка
Loading application [Servlet] at [/Servlet]#
[#|2013-02-28T15:34:52.350+0400|INFO|glassfish3.1.2|javax.enterprise.system.tools.admin.org.glassfish.deployment.admin|_threadID=67;_threadName=Thread-2;
[Servlet was successfully deployed in 207 milliseconds. [#]
[#|2013-02-28T15:39:43.281+0400|INFO|glassfish3.1.2|javax.enterprise.system.container.web.com.sun.enterprise.web|_threadID=36;_threadName=Thread-2;|PwC1412:
webModule[null] servletcontext.log():A request has been initialized at: Thu Feb 28 15:39:43 MSK 2013|#]
[#|2013-02-28T15:39:43.283+0400|INFO|glassfish3.1.2|javax.enterprise.system.container.web.com.sun.enterprise.web|_threadID=36;_threadName=Thread-2;|PwC1412:
webModule[null] servletcontext.log():A request has been destroyed at: Thu Feb 28 15:39:43 MSK 2013|#]
[#|2013-02-28T15:40:09.595+0400|INFO|glassfish3.1.2|javax.enterprise.system.container.web.com.sun.enterprise.web|_threadID=37;_threadName=Thread-2;|PwC1412:
webModule[null] servletcontext.log():A request has been initialized at: Thu Feb 28 15:40:09 MSK 2013|#]
[#|2013-02-28T15:40:09.601+0400|INFO|glassfish3.1.2|javax.enterprise.system.container.web.com.sun.enterprise.web|_threadID=37;_threadName=Thread-2;|PwC1412:
webModule[null] servletcontext.log():An attribute with name: URL and value: jdbc:odbc:employeesDB has been added to the context at: Thu Feb 28 15:40:09 MSK 2013|#]
[#|2013-02-28T15:40:09.601+0400|INFO|glassfish3.1.2|javax.enterprise.system.container.web.com.sun.enterprise.web|_threadID=37;_threadName=Thread-2;|PwC1412:
webModule[null] servletcontext.log():Attribute with name: URL and value: jdbc:odbc:employeesDB has been removed from the context at: Thu Feb 28 15:40:09 MSK 2013|#]
[#|2013-02-28T15:40:09.602+0400|INFO|glassfish3.1.2|javax.enterprise.system.container.web.com.sun.enterprise.web|_threadID=37;_threadName=Thread-2;|PwC1412:
webModule[null] servletcontext.log():A request has been destroyed at: Thu Feb 28 15:40:09 MSK 2013|#]

```

Рис. 54. Содержимое файла server.log

Описание элементов дескриптора развертывания

Дескриптор развертывания – это файл XML с именем `web.xml`, который содержит конфигурационную информацию о веб-приложении. Существует только один файл `web.xml` для каждого веб-приложения. Элементы файла `web.xml` позволяют задавать параметры инициализации для сервлета, MIME-тип для различных файлов, указывать классы-слушатели и устанавливать шаблон URL для сервлета. Несмотря на то, что, как правило, содержание этого файла формируется автоматически, ниже описаны некоторые из часто используемых элементов дескриптора развертывания вместе с примерами их использованием:

<context-param>: Задает параметры инициализации контекста сервлета веб-приложения, как демонстрируется в следующем фрагменте кода:

```

<context-param>
  <param-name> rmihost </param-name>
  <param-value> 192.162.100.4 </param-value>
</context-param>

```

<init-param>: Задает параметр инициализации для сервлета. В отличие от параметра инициализации контекста, который доступен всем сервлетам веб-приложения, этот параметр доступен только сервлету, для которого он объявлен. Следующий фрагмент кода демонстрирует использование элемента `init-param`:

```

<init-param>
  <param-name> title <param-name>
  <param-value> This is the First Servlet </param-value>
</init-param>

```

<mime-mapping>: Задает соответствие между расширением файла и MIME-типом, как показано в следующем фрагменте кода:

```

<mime-mapping>
  <extension>html</extension>
  <mime-type> text/html </mime-type>
</mime-mapping>

```

<servlet-mapping>: Задает соответствие между сервлетом и шаблоном URL, как показано в следующем фрагменте кода:

```

<servlet-mapping>
  <servlet-name>MyServlet</servlet-name>

```

```
<url-pattern>/test</url-pattern>  
</servlet-mapping>
```

После определения данного соответствия в дескрипторе развертывания сервлета, веб-контейнер будет сопоставлять следующий URL сервлету `MyServlet`, как показано ниже:

```
http://localhost:8080/servletctx/test
```

`<session-config>`: Задает сеансовую информацию для сервлета, такую как значение тайм-аута для сеанса, как показано в следующем фрагменте кода:

```
<session-config>  
    <session-timeout>30</session-timeout>  
</session-config>
```

Данный выше элемент дескриптора развертывания задает, что сессия сервлета заканчивается через 30 минут.

`<listener>`: Задает имя класса-слушателя, который реагирует на события времени жизни сервлета, как показано в следующем фрагменте кода:

```
<listener>  
    <listener-class>ContextListenerHandler</listener-class>  
</listener>
```

Управление сессиями servlet

Управление сессией – это процесс отслеживания действий пользователя на веб-страницах. Например, в онлайн-магазине, пользователь может выбрать продукт и добавить его в магазинную тележку. Когда пользователь переходит на другую страницу, товары в магазинной тележке должны сохраняться, чтобы пользователь мог проверить содержимое тележки и затем заказать их.

Отслеживание сессии также может быть использовано для учета предпочтений пользователя, поэтому управление сеансом является неотъемлемой частью веб-приложения.

Приемы управления сессией

Как уже отмечалось, HTTP является протоколом без сохранения состояния и, поэтому, не может хранить информацию о действиях пользователя на веб-страницах. Однако существуют определенные приемы, которые помогают сохранять информацию о пользователе веб-страниц по протоколу HTTP, представленные ниже:

- скрытое поле формы;
- перезапись URL;
- cookies;
- Servlet Session API.

Скрытое поле формы

Скрытые поля формы могут использоваться для сохранения информации о сессии пользователя при взаимодействии с веб-приложением. Скрытое поле формы определяется на HTML-странице, и оно является невидимым при просмотре из браузера, как показано в следующем фрагменте кода в форме HTML-страницы:

```
<HTML>
<FORM METHOD="GET" Action="/servlet/TestServlet">
<input type="hidden" name="id" value="Labc">
- - - - -
- - - - -
</HTML>
```

Например, в онлайн-магазине скрытые поля формы могут использоваться для сохранения информации о пользователе. Пользователь записывает имя в регистрационную форму HTML. Эта информация считывается сервлетом, который затем генерирует форму HTML, содержащую скрытое поле формы (hidden) и кнопку **Submit**. Когда пользователь нажимает кнопку **Submit**, другой сервлет получает имя пользователя, сохраняет в скрытом поле формы и выводит сообщение с приветствием.

Следующий код может использоваться при создании страницы для входа в приложение, которая вводит имя пользователя и вызывает сервлет `HiddenServlet`, когда пользователь нажимает кнопку **Login**:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<FORM ACTION = "http://localhost:8080/Servlet/HiddenServlet" METHOD =
POST >
    Username: <INPUT TYPE = TEXT NAME = "user" ><BR>
             <INPUT TYPE = SUBMIT VALUE = "Login" >
</FORM>
</body>
</html>
```

Рис. 55 показывает результат работы файла `Login.html` с именем пользователя **Customer**, отображаемым в текстовом поле **Username**:

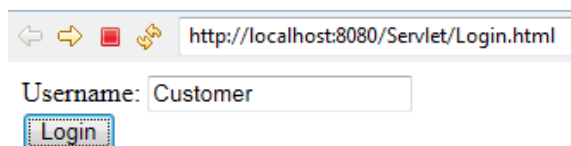


Рис. 55. Результат вывода `Login.html`.

При нажатии пользователем кнопки **Login**, введенная информация отправляется сервлету `HiddenServlet`. Затем сервлет получает имя пользователя и сгенерирует форму HTML со скрытым полем формы и кнопкой **Submit**. Скрытое поле формы содержит имя пользователя. Следующий код используется для создания сервлета `HiddenServlet`:

```

package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/HiddenServlet")
public class HiddenServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public HiddenServlet() {
        super();
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        /* Вызов метода doPost() класса HttpServlet. */
        doPost(req, res);
    }
    /*
    * Переопределение метода doPost() класса HttpServlet для реализа-
    ции
    * функциональности сервлета.
    */
    public void doPost(HttpServletRequest req, HttpServletResponse
res)
        throws ServletException, IOException {
        /*
        * Получение параметров, связанных с пользователем (пароль и
        * регистрационное имя) из объекта запроса
        */
        String username = req.getParameter("user");
        PrintWriter pw = res.getWriter();
        pw.println("Hello! click Submit to proceed");
        pw.println("<Form name=\"login\" ac-
tion=\"http://localhost:8080/javaee/SecondServlet\">");
        /* Добавление скрытого поля */
        pw.println("<input type=\"hidden\" name=\"user\" value=" +
username
                + ">");
        pw.println("<input type=\"Submit\" val-
ue=\"Submit\"></form>");
    }
}

```

Рис. 56 показывает форму HTML, сгенерированную сервлетом `HiddenServlet`.

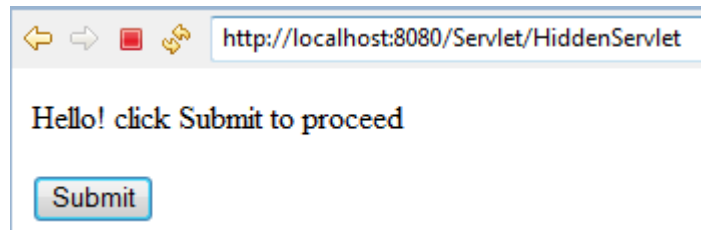


Рис. 56. Результат работы HiddenServlet.

При нажатии пользователем кнопки **Submit**, значение, сохраненное в скрытом поле формы, передается сервлету SecondServlet и сервлет SecondServlet получает значение в скрытом поле формы, после чего выводит сообщение с приветствием в браузер клиента. Следующий код используется для создания сервлета SecondServlet:

```
package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/SecondServlet")
public class SecondServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public SecondServlet() {
        super();
        // TODO Auto-generated constructor stub
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
        doPost(req, res);
    }
    public void doPost(HttpServletRequest req, HttpServletResponse
    res) throws ServletException, IOException
    {
        /* Получение параметров, связанных с именем пользователя в
        скрытом поле формы из объекта запроса. */
        String uname = req.getParameter("user");
        PrintWriter pw = res.getWriter();
        pw.println("Hello! "+uname);
    }
}
```

Рис. 57 показывает результат работы SecondServlet, в котором пользователя приветствуется по имени.

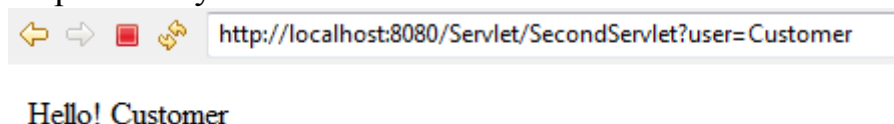


Рис. 57. Результат работы сервлета SecondServlet.

Перезапись URL

Перезапись URL – это прием управления пользовательских сессий на основе модификации URL. Обычно этот прием используется, когда передаваемая информация не является конфиденциальной, поскольку URL может быть легко перехвачен в процессе передачи. Например, в онлайн-магазине сервлет может модифицировать URL, чтобы включить дополнительную информацию о пользователе. Затем сервлет может отобразить URL, и, когда пользователь щелкает по гиперссылке с этим URL, информация отправляется другому сервлету, который получает информацию о пользователе и выводит сообщение с приветствием. Следующий код используется для создания сервлета RewriteServletURL, который модифицирует и отображает URL.

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/RewriteServletURL")
public class RewriteServletURL extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public RewriteServletURL() {
        super();
    }
    public void doGet(HttpServletRequest req, HttpServletResponse
res) throws ServletException, IOException
    {
        doPost(req, res);
    }
    public void doPost(HttpServletRequest req,
HttpServletResponse res) throws ServletException, IOException
    {
        /* Получение параметров, связанных с пользователем (па-
роль и регистрационной имя) из объекта запроса. */
        String username = req.getParameter("user");
        PrintWriter pw = res.getWriter();
        res.setContentType("text/html");
        pw.println("Hello! <a
href=\"http://localhost:8080/Servlet/SecondServlet?uname=" +
username + "\"> click here </a>to proceed");
    }
}
```

В вышеприведенной программе сервлет RewriteServletURL читает имя пользователя из клиентского запроса, добавляет его к URL и отображает ссылку на другой сервлет с именем SecondServlet. Рис. 58 показывает результат работы сервлета RewriteServletURL.

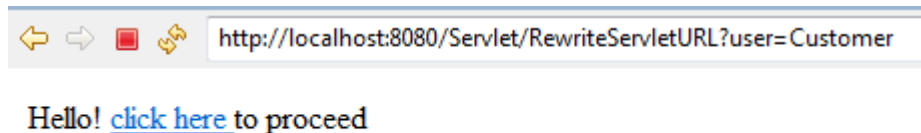


Рис. 58. Результат работы RewriteServletURL.

Когда пользователь щелкает по гиперссылке, сервлет SecondServlet считывает имя пользователя, добавленное к URL, и выводит сообщение с приветствием. Следующий код используется для создания сервлета SecondServlet:

```
package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/SecServlet")
public class SecServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public SecServlet() {
        super();
        // TODO Auto-generated constructor stub
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        doPost(req, res);
    }
    public void doPost(HttpServletRequest req, HttpServletResponse
res)
        throws ServletException, IOException {
        /*
        * Получение параметров, связанных с именем пользователя в
скрытом поле
        * формы из объекта запроса.
        */
        String uname = req.getParameter("uname");
        PrintWriter pw = res.getWriter();
        pw.println("Hello! " + uname);
    }
}
```

В приведенном коде сервлет получает имя пользователя из запроса и выводит сообщение с приветствием клиенту. Рис. 59 показывает результат работы сервлета SecondServlet:

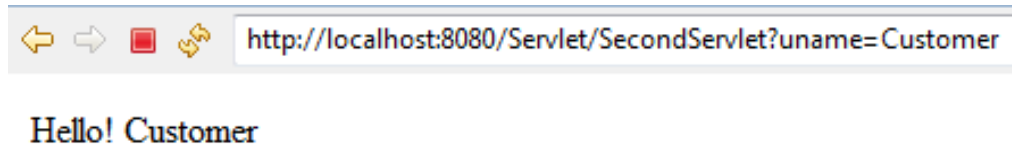


Рис. 59. Результат работы SecondServlet.

Использование Cookies

Cookies – это маленькие текстовые файлы, которые сохраняются сервером приложений в браузере клиента для отслеживания всех пользователей. Cookie хранит значения в форме пары имя/значение, например, cookie может иметь имя `user` со значением `Michael`. Cookies создаются сервером и отправляются клиенту в заголовках ответа HTTP. Клиент сохраняет cookies на локальном жестком диске и отправляет их вместе с заголовками запросов HTTP серверу. Веб-браузер обычно поддерживает 20 cookies для одного пользователя, а размер каждого cookie не может превышать 4 Кбайта. Ниже представлены некоторые характеристики cookies.

- Cookies могут быть доступны только тому серверу приложений, который записал их на браузер клиента. Например, если cookie создается сервером приложений `www.macromedia.com` и отправляется клиенту или браузеру, то cookie может быть отправлен обратно только этому же серверу.
- Cookies могут быть использованы сервером для нахождения имени компьютера, IP-адреса или любых других характеристик компьютера клиента, после получения удаленного адреса компьютера клиента, где хранятся cookies.

Servlet API обеспечивает поддержку cookies на основе класса `Cookie` пакета `javax.servlet.http`, представляющего cookie. Класс `Cookie` предоставляет конструктор, который получает имя и значение для создания cookie. Следующий фрагмент кода показывает сигнатуру конструктора для создания cookie с именем `name` и значением `value`:

```
public Cookie(String name, String value);
```

Интерфейс `HttpServletResponse` предоставляет метод `addCookie()` для добавления cookie к объекту ответа, а затем отправки его клиенту. Следующий фрагмент кода показывает, как добавить cookie:

```
response.addCookie(cookie);
```

Интерфейс `HttpServletRequest` предоставляет метод `getCookies()`, который возвращает массив cookies, который содержится в объекте запроса. Следующий фрагмент кода показывает, как получить cookies:

```
Cookie cookie[] = request.getCookies();
```


Класс `Cookie` содержит методы для установки и получения различных свойств `cookie`. В таблица 15 описываются методы класса `Cookie`:

Таблица 15.

Метод	Описание
<code>public String getName()</code>	Возвращает имя <code>cookie</code> .
<code>public void setMaxAge(int expiry)</code>	Устанавливает максимальное время, которое браузер клиента сохраняет значение <code>cookie</code> .
<code>public int getMaxAge()</code>	Возвращает максимальный возраст <code>cookie</code> в секундах.
<code>public void setValue(String value)</code>	Устанавливает новое значение для <code>cookie</code> .
<code>public String getValue()</code>	Возвращает значение <code>cookie</code> .

Рассмотрим приложение Интернет-магазина, которое отслеживает пользователя с помощью `cookies`. Когда пользователь использует приложение первый раз, сервлет приложения получает имя пользователя, сохраняет его в `cookie` и записывает его в браузере клиента. При следующем запросе от клиента другие сервлеты могут получить имя пользователя, хранимое в этом `cookie`, чтобы идентифицировать клиента.

Код ниже может использоваться для создания сервлета `CookieServlet`, который получает имя пользователя и сохраняет его в `cookie`, и затем отправляет `cookie` клиенту, используя объект `response`:

```

package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/CookieServlet")
public class CookieServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public CookieServlet() {
        super();
        // TODO Auto-generated constructor stub
    }
    PrintWriter pw = null;
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        /* Вызов метода doPost() класса HttpServlet. */
        doPost(req, res);
    }
}

```

```

    }
    /*
     * Переопределение метода doPost() класса HttpServlet, который
реализует
     * функциональность сервлета.
     */
    public void doPost(HttpServletRequest req, HttpServletResponse
res)
        throws ServletException, IOException {
        pw = res.getWriter();
        /* Получение информации от пользователя из запроса
HttpServlet. */
        String username = req.getParameter("user");
        String password = req.getParameter("password");
        /* Создание cookie, которое содержит имя пользователя. */
        Cookie ck = new Cookie("user", username);
        /* Добавление cookie к браузеру клиента. */
        res.addCookie(ck);
        pw.println("Cookie containing user name is stored in your
browser.");
    }
}

```

В приведенном коде CookieServlet создает экземпляр класса Cookie и сохраняет информацию о пользователе, полученную от интерфейса HttpServletRequest. Рис. 60 показывает результат работы сервлета CookieServlet при вызове по URL `http://localhost:8080/Servlet/CookieServlet?user=Customer,password=123`.

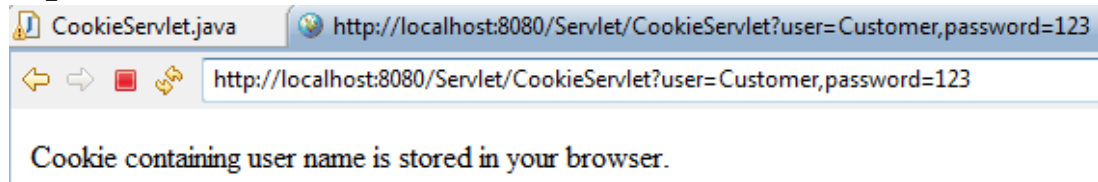


Рис. 60. Результат работы сервера CookieServlet.

Когда клиент отправляет запрос приложению, cookie отправляется вместе с запросом. Следующий код используется для создания сервлета RetrieveCookie, который получает cookie из объекта запроса и извлекает из него имя пользователя и пароль:

```

package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/RetrieveCookie")
public class RetrieveCookie extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public RetrieveCookie() {

```

```

        super();
    }
    PrintWriter pw=null;
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
        /* Вызов метода doPost() класса HttpServlet. */
        doPost(req,res);
    }
    /* Переопределение метода doPost() класса HttpServlet, который
    реализует функциональность сервлета. */
    public void doPost(HttpServletRequest req, HttpServletResponse
    res) throws ServletException, IOException
    {
        /* Получение экземпляра класса PrintWriter. */
        pw=res.getWriter();
        String username=null;
        /* Получение cookies, если таковые есть, хранимых в браузере кли-
        ента. */
        Cookie ck[] = req.getCookies();
        if (ck!=null)
        {
            for (int i=0; i<ck.length;i++)
            {
                if (ck[i].getName().equals("user"))
                /* Получение имени пользователя из cookie. */
                username = ck[i].getValue();
            }
            pw.println(" Hello! &nbsp;&nbsp;&nbsp;"+username);
        }
        else
        {
            pw.println("No cookies found");
        }
    }
}

```

В приведенном коде сервлет RetrieveCookie получает cookies, хранимые у клиента, используя метод `getCookies()` интерфейса `HttpServletRequest`, и выводит полученное имя пользователя в браузере. Рис. 61 показывает информацию о пользователе, полученную из cookie:

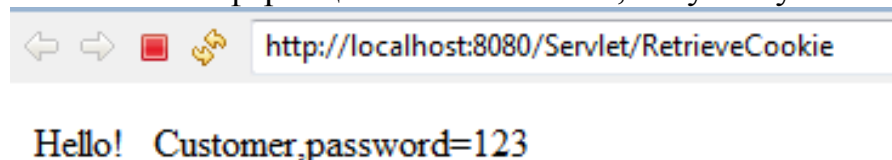


Рис. 61 Результат работы RetrieveCookie.

Сервер приложений может сохранять cookies на компьютере клиента, только если cookies разрешены в браузере клиента.

Servlet Session API

Следующие классы и интерфейсы, определенные в Servlet Session API, можно использовать для создания и управления пользовательскими сессиями:

`javax.servlet.http.HttpSession`,
`javax.servlet.http.HttpSessionListener` и
`javax.servlet.http.HttpSessionBindingListener`.

Интерфейс `javax.servlet.http.HttpSession` предоставляет методы для управления сессией пользователя, позволяющие создавать объект интерфейса `HttpSession` для сохранения информации сессии в виде пары имя/значение, которые позже можно извлечь для обработки сессии пользователей. Таблица 16 описывает методы интерфейса `HttpSession`.

Таблица 16.

Метод	Описание
<code>public void setAttribute(String name, Object value)</code>	Добавляет атрибут с уникальным именем в объект сессии и сохраняет пару <code>name/value</code> в текущей сессии. Если существует объект с таким атрибутом, то новый объект замещает существующий.
<code>public Object getAttribute(String name)</code>	Извлекает объект, связанный с именем атрибута, указанным в методе, из объекта сессии. Если объект с указанным атрибутом не найден, то метод возвращает <code>null</code> .
<code>public Enumeration getAttributeNames()</code>	Возвращает имена всех объектов, которые находятся в объекте сессии.
<code>public void removeAttribute(String name)</code>	Удаляет атрибут с именем, указанным в методе, из объекта сессии.
<code>public void setMaxInactiveInterval(int interval)</code>	Устанавливает максимальное время, на которое сессия будет оставаться активной. Время задается в секундах. Если в течение этого времени нет запросов от клиента, то сервер уничтожает сессию. Отрицательное значение в этом методе означает, что сессия должна всегда оставаться активной.
<code>public int getMaxInactiveInterval()</code>	Возвращает максимальное время в секундах, в течение которого сервер не уничтожит сессию, даже если нет клиентского запроса.
<code>public String getId()</code>	Возвращает строку, которая содержит идентификатор, связанный с сессией.

Метод	Описание
<code>public void invalidate()</code>	Уничтожает сессию. Все объекты, связанные с сессией, автоматически освобождаются.

Следующий код показывает использование методов интерфейса `HttpSession` для создания сессии пользователя, записи информации пользователя в объект сессии и получение информации из объекта сессии:

```

package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
@WebServlet("/SessionServlet")
public class SessionServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public SessionServlet() {
        super();
        // TODO Auto-generated constructor stub
    }
    /* Переопределение метода doGet() в HttpServlet. */
    public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
        /* Вызов метода doPost() в HttpServlet. */
        doPost(req, res);
    }
    /* Переопределение метода doPost() в HttpServlet. */
    public void doPost(HttpServletRequest req, HttpServletResponse
    res) throws ServletException, IOException
    {
        /* Получение имени и пароля пользователя из HttpServletRequest.*/
        String username = req.getParameter("user");
        String password = req.getParameter("password");
        /* Получение экземпляра PrintWriter. */
        PrintWriter pw = res.getWriter();
        /* Создание объекта HttpSession и проверка, существует ли уже сеанс для
        этого пользователя, вызовом метода getSession() в HttpServletRequest.
        Если сеанс не существует, метод getSession() создает новый сеанс.*/
        HttpSession session = req.getSession(true);
        /* Установка username и password как атрибутов в сессии пользователя. */
        session.setAttribute("user", username);
        session.setAttribute("password", password);
        pw.println("Your user name has been set as an attribute to
        the session<br>");
        /* Получение атрибутов username и password из сеанса пользователя. */
        String user=(String) ses-
        sion.getAttribute("user");
        String pass=(String) ses-
        sion.getAttribute("password");
    }
}

```

```

        pw.println("Your user name retrieved from the session is
"+user+" "+pass);
    }
}

```

В коде сервлет получает информацию пользователя и проверяет, существует ли сессия для этого пользователя. Если сессия пользователя не существует, сервлет вызывает метод `getSession()` интерфейса `HttpServletRequest` для создания объекта интерфейса `HttpSession` и устанавливает `username` и `password` как атрибуты в объекте сессии. Затем сервлет извлекает имя и пароль пользователя, сохраненные в сессии, и выводит их значения клиенту.

Результат работы сервлета `SessionServlet` представлен на Рис. 62.

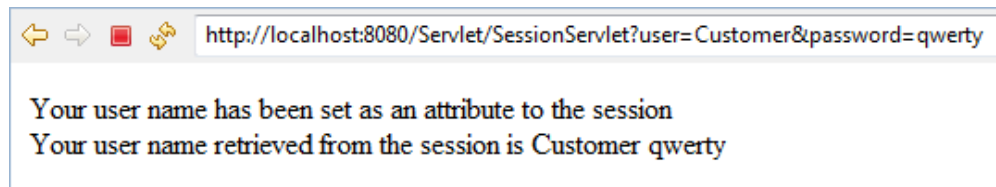


Рис. 62. Результат работы сервлета `SessionServlet`.

Сессия пользователя завершается с помощью ее удаления. Можно потребовать удаление сессии в различных ситуациях, например, когда пользователь выходит из приложения, или, когда пользователь бездействует определенный период времени. Завершить сессию можно, используя различные методы интерфейса `HttpSession`:

- Устанавливая максимальное время простоя с помощью метода `setMaxInactiveInterval(int interval)`. Время простоя – это время, в течение которого клиент не производит никаких действий.
- Явно вызывая метод `invalidate()` объекта сеанса. Можно использовать следующий код для уничтожения сеанса:

```

HttpSession ss=req.getSession(true);
session.invalidate();

```
- Указывая значение таймаута сессии в минутах в элементе `session-timeout` дескриптора развертывания.

Следующий фрагмент кода демонстрирует элемент `<session-timeout>`, который можно включить в дескриптор развертывания для установки опции уничтожения сессии:

```

<session-config>
    <session-timeout>
30
    </session-timeout>
</session-config>

```

В качестве законченного примера применения управления сессией рассмотрим следующую задачу.

Постановка задачи

Разработать веб-приложение для продажи вещей, с возможностью отслеживания покупки пользователя и выполнения обновления суммы покупки.

Решение

Для выполнения поставленной задачи необходимо выполнить следующие шаги.

1. Создать пользовательский интерфейс.
2. Создать сервлет, который устанавливает подлинность информации пользователя и выводит информацию о продуктах.
3. Создать сервлет для сохранения информации о выбранных вещах в покупке.
4. Создать сервлет для расчета и вывода суммы покупки.

1. Создание пользовательского интерфейса

Пользовательский интерфейс создается для ввода имени и пароля от пользователя. Можно использовать следующий код для создания страницы Login.html:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<BODY>
  <FORM ACTION="http://localhost:8080/Servlet/FirstServlet" METHOD=POST >
    Username: <INPUT TYPE=TEXT NAME="user" ><BR>
    <INPUT TYPE=SUBMIT VALUE="Login" >
  </FORM>
</body>
</html>
```

Рис. 63 показывает результат работы страницы Login.html.

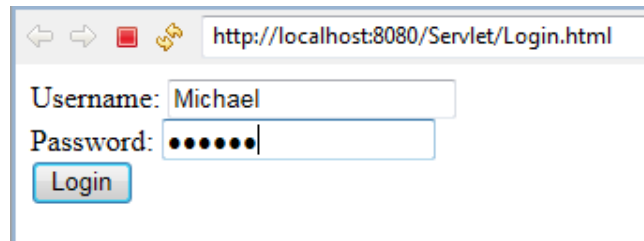


Рисунок 63. Результат работы Login.html

Замечание

Рекомендуется всегда использовать идентификатор (id) сессии конечного пользователя для последующей проверки пользовательской информации на каждой странице, которую он посещает. Получайте id сеанса на первой странице и сохраняйте его в переменной. Извлекайте id сеанса пользователя на каждой странице и сравнивайте его с id сеансом, сохраненным в переменной. Можно передать эту переменную другой странице, добавив ее к объекту запроса.

2. Создание сервлета, который устанавливает подлинность информации пользователя и выводит информацию о продуктах

Введенная пользователем информация должна быть проверена, что и выполняется сервлетом FirstServlet. Если информация пользователя верна (введено имя Michael и пароль qwerty), сервлет FirstServlet сохраняет информацию в объекте сессии и выводит данные о продаваемых товарах. Следующий код можно использовать для создания сервлета FirstServlet:

```
package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
@WebServlet("/FirstServlet")
public class FirstServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public FirstServlet() {
        super();
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        /* Вызов метода doPost() класса HttpServlet. */
        doPost(req, res);
    }
    /*
```



```

    * Переопределение метода doPost() в HttpServlet, который реализу-
ет
    * функциональность сервлета.
    */
    public void doPost(HttpServletRequest req, HttpServletResponse
res)
        throws ServletException, IOException {
        String username = req.getParameter("user");
        String password = req.getParameter("password");
        int counter = 0;
        PrintWriter pw = res.getWriter();
        /* Аутентификация пользователя. */
        if (username.equals("Michael") && password.equals("qwerty"))
        {
            pw.println(username + "! Welcome to Online Shopping."
+ "<BR>");
            counter = 0;
            /* Создание сеанса для пользователя и сохранение име-
ни пользователя. */
            HttpSession session = req.getSession(true);
            session.setAttribute("user", username);
        } else {
            pw.println("Sorry! Invalid username and password");
            counter = 1;
        }
        if (counter == 0) {
            /* Вывод данных пользователю. */
            pw.println("<HTML><BODY>");
            pw.println("<HR>");
            pw.println("<FORM ACTION =
http://localhost:8080/Servlet/SecondSessionServlet METHOD=POST>");
            pw.println("<TABLE WIDTH=500>");
            pw.println("<TR><TH>ITEM NO</TH> <TH>SHIRT TYPE </TH>
<TH>BUY</TH> </TR> ");
            pw.println("<TR><TD> 1 </TD><TD> PeterEngland </TD>
<TD> <INPUT NAME = c1 TYPE = CHECKBOX VALUE = PeterEngland ></TD> </TR>
");
            pw.println("<TR><TD> 2 </TD><TD> Excaliber </TD>
<TD> <INPUT NAME = c2 TYPE = CHECKBOX VALUE = Excaliber ></TD> </TR> ");
            pw.println("<TR><TD> 3 </TD><TD> Vaun Newman </TD>
<TD> <INPUT NAME = c3 TYPE = CHECKBOX VALUE = VaunNewman></TD> </TR> ");
            pw.println("<TR><TD> 4 </TD><TD> Wills Classic </TD>
<TD> <INPUT NAME = c4 TYPE = CHECKBOX VALUE = WillsClassic></TD> </TR>
");
            pw.println("</TABLE>");
            pw.println("<INPUT TYPE = SUBMIT VALUE = SUBMIT>");
            pw.println("</FORM>");
            pw.println("</BODY></HTML>");
            pw.close();
        }
    }
}

```

Рис. 64 показывает результат работы сервлета FirstServlet с двумя выбранными в нем предметами.

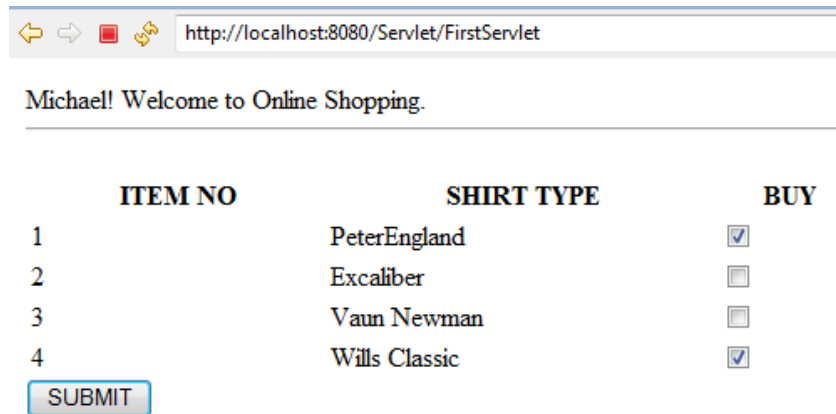


Рис. 64. Результат работы сервлета FirstServlet

3. Создание сервлета для сохранения информации о выбранных предметах

После выбора предметов из списка, необходимо их объекте сессии и передать эту информацию сервлету FinalServlet. Следующий код можно использовать для создания сервлета SecondSessionServlet:

```
package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
@WebServlet("/SecondSessionServlet")
public class SecondSessionServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public SecondSessionServlet() {
        super();
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        /* Вызов метода doPost() класса HttpServlet. */
        doPost(req, res);
    }
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        /* Объявление локальных переменных для хранения информации сеанса.*/
        String user = null;
        String item = null;
        int i = 1;
        Enumeration en = null;
        /* Получение экземпляра класса PrintWriter. */
        PrintWriter pw = response.getWriter();
```

```

        /* Получение объекта сеанса. */
        HttpSession session = request.getSession(true);
        /* Получение значения, связанного с "user". */
        user = (String) session.getAttribute("user");
    /* Получение имен всех параметров, переданных конечным поль-зователем.*/
    en = request.getParameterNames();
    while (en.hasMoreElements()) {
        String sname = (String) en.nextElement();
        /* Сохранение имен в объекте сеанса. */
        session.setAttribute("c" + i, sname);
        i++;
    }
    /* Сохранение значения счетчика в объекте сеанса. */
    session.setAttribute("counter", i + "");
    /* Отправка запроса сервлету FinalServlet. */
    RequestDispatcher disp =
    request.getRequestDispatcher("FinalServlet");
    disp.forward(request, response);
}
}
}

```

В представленном коде сервлет SecondSessionServlet получает информацию о выбранных товарах из сеанса пользователя. Сервлет SecondSessionServlet вызывается каждый раз, когда пользователь обновляет набор товаров в магазинной тележке. Затем сервлет SecondSessionServlet передает эту информацию сервлету FinalServlet, используя объект RequestDispatcher.

4. Создание сервлета расчета и вывода суммы покупки

Необходимо получить итоговые данные о выбранных товарах из сеанса пользователя и рассчитать итоговую сумму покупки, которую необходимо оплатить пользователю. Следующий код можно использовать для создания сервлета FinalServlet, который получает последние данные сеанса и выводит их в окне браузера:

```

package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
@WebServlet("/FinalServlet")
public class FinalServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public FinalServlet() {
        super();
        // TODO Auto-generated constructor stub
    }
    protected void doGet(HttpServletRequest request,

```

```

        HttpServletResponse response) throws
ServletException, IOException {
    // TODO Auto-generated method stub
    doPost(request, response);
}
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        String shcode1 = null;
        String shcode2 = null;
        String shcode3 = null;
        String shcode4 = null;
        int price1 = 0;
        int price2 = 0;
        int price3 = 0;
        int price4 = 0;
        int totalPrice = 0;
        String username = "";
        PrintWriter pw = response.getWriter();
        /* Получение объекта сессии. */
        HttpSession session = request.getSession(true);
        /* Получение значения, связанного с "user". */
        username = (String) session.getAttribute("user");
        /* Получение значения, связанного с "counter". */
        String counter = (String) session.getAttribute("counter");
        int count = Integer.parseInt(counter);
        for (int i = 1; i <= count - 1; i++) {
            /* Создание массива строк. */
            String shname[] = new String[count - 1];
            shname[i - 1] = (String) session.getAttribute("c" +
i);

            /* Вычисление итоговой цены выбранных товаров. */
            if ((shname[i - 1]).equals("c1")) {
                price1 = 65;
            }
            if ((shname[i - 1]).equals("c2")) {
                price2 = 70;
            }
            if ((shname[i - 1]).equals("c3")) {
                price3 = 85;
            }
            if ((shname[i - 1]).equals("c4")) {
                price4 = 75;
            }
        }
        /* Вычисление итогового счета. */
        totalPrice = price1 + price2 + price3 + price4;
        /* Вывод итогового счета, который пользователь должен опла-
        тить. */
        pw.println(username + ", your bill is $" + totalPrice);
        pw.println("</BODY></HTML>");
        pw.close();
    }
}

```

В предложенном коде сервлет `FinalServlet` получает данные о товарах из сеанса пользователя, рассчитывает общую сумму и выводит ее в окне браузера. Рис. 65 показывает результат работы `FinalServlet`, который выводит сумму покупки пользователя.

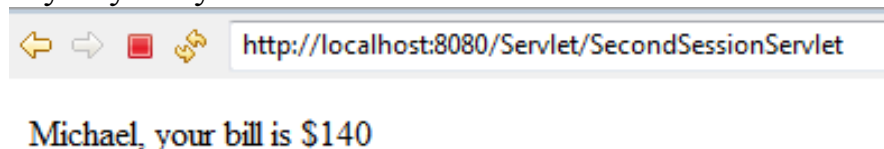


Рис. 65. Результат работы сервлета `FinalServlet`

Обработка ошибок и исключений в сервлетах

Сервер приложений при обработке запросов клиентов может столкнуться с проблемами, связанными, например, с тем, что запрашиваемый ресурс не найден, или работающий сервлет генерирует исключение. Предположим, что в онлайн-магазине возникает проблема, если пользователь выбирает товар, который временно не доступен, и пытается добавить его в магазинную тележку. В этом случае должно отображаться сообщение об ошибке в окне браузера, которое должно генерироваться сервлетом с помощью обработки исключений в сервлете. Также можно предусмотреть настраиваемые страницы ошибок для отображения исключений и сообщений об ошибках, а также записи этих сообщений в файл журнала сервера приложений.

Исключения в сервлетах

Servlet API определяет два класса исключений, которые определены в пакете `javax.servlet`: `ServletException` и `UnavailableException`. Класс `UnavailableException` является подклассом класса `ServletException`.

Класс `javax.servlet.ServletException`

Класс `javax.servlet.ServletException` определен в пакете `javax.servlet`, является подклассом класса `java.lang.Exception` и определяет исключение в сервлете, которое сервлет генерирует во время обработки запроса клиента. Этот класс содержит метод `getRootCause()`, который возвращает объект типа `java.lang.Throwable`, который представляет основную причину исключения.

Класс `javax.servlet.UnavailableException`

Класс исключения `javax.servlet.UnavailableException` генерируется сервлетом, когда сервлет временно или постоянно недоступен. Ниже представлены методы, предоставляемые классом `UnavailableException`:

- `public boolean isPermanent()`: Возвращает Boolean значение, которое возвращает постоянно недоступен сервлет или нет. Если сервлет недоступен постоянно, то этот метод возвращает `true`.
- `public int getUnavailableSeconds()`: Возвращает время в секундах, в течение которого сервлет будет оставаться недоступным.

Обработка ошибок

Как уже отмечалось, сервлеты в веб-приложениях могут генерировать ошибки и исключения, предоставляя пользователям информацию об ошибках и исключениях и отправляя соответствующие сообщения об ошибках и коды состояния, представляющие различные типы ошибок. Кроме того, можно создавать настраиваемые страницы ошибок для форматирования и вывода информации о типе ошибки и возникающих исключениях и/или записывать сообщения в файл журнала сервера.

Сообщения об ошибках и коды состояния

Код состояния (status code) – это информация, которую сервер приложений отправляет клиенту о состоянии выполнения запроса. Код состояния возвращается браузеру клиента в объекте ответа. Класс `HttpServletResponse` пакета `javax.servlet.http` определяет поля, которые представляют коды состояния, используемые для отправки кодов из сервлета. Коды состояния логически сгруппированы в следующие пять категорий:

`Information` (Информация): Группа информационных кодов представляет сообщения о приеме и обработке запроса. Ниже представлены некоторые из полей `HttpServletResponse`, которые представляют информационные коды:

- `SC_SWITCHING_PROTOCOLS`: Представляет код состояния 101 и указывает на переключение протоколов сервером приложений. (Обычно это предложение о переходе на более новую версию HTTP).

- `SC_CONTINUE`: Представляет код статуса 100 и указывает, что переданная клиентом часть запроса получена и не была отвергнута сервером, и клиент может продолжать взаимодействие с сервером приложений.

`Success` (Успешность): Группа кодов успеха представляет сообщение об успехе, которое указывает на то, что запрос удачно выполнен. Ниже представлены некоторые из полей `HttpServletResponse`, которые представляют коды успеха:

- `SC_OK`: Представляет код состояния 200 и указывает, что запрос успешно принят.
- `SC_ACCEPTED`: Представляет код состояния 202 и указывает, что сервер принял запрос и обрабатывает его.

`Redirection` (Перенаправление): Группа кодов указывает, что запрос перенаправлен на другую страницу для обработки. Ниже представлены некоторые из полей `HttpServletResponse`, которые представляют коды перенаправления:

- `SC_MOVED_PERMANENTLY`: Представляет код состояния 301 и указывает, что ресурс удален на длительное время.
- `SC_MOVED_TEMPORARILY`: Представляет код состояния 302 и указывает, что ресурс временно удален.

`Client Error` (Ошибка клиента): Группа кодов указывает, что запрос имеет некоторую ошибку и не может быть выполнен. Ниже представлены некоторые из полей `HttpServletResponse`, которые представляют коды ошибки клиента:

- `SC_BAD_REQUEST`: Представляет код состояния 400 и указывает, что синтаксис клиентского запроса некорректен.
- `SC_NOT_FOUND`: Представляет код состояния 404 и указывает, что требуемый клиентом, недоступен.
- `SC_GONE`: Представляет код состояния 410 и указывает, что ресурс больше недоступен.

`Server Error` (Ошибка сервера): Группа ошибок сервера указывает, что сервер не в состоянии выполнить запрос клиента. Ниже представлены некоторые из полей `HttpServletResponse`, которые представляют коды ошибки сервера:

- `SC_INTERNAL_SERVER_ERROR`: Представляет код состояния 500 и указывает, что на сервере имеет место ошибка, которая препятствует выполнению запроса.

- SC_NOT_IMPLEMENTED: Представляет код состояния 501 и указывает, что сервер не поддерживает данный тип запросов.

Можно отправить ответ с ошибкой или состоянием клиенту независимо от того, где возникла ошибка при работе веб-приложения. `sendError()` и `setStatus()` - два метода объекта `HttpServletResponse`, которые можно использовать для отправки сообщения с ошибкой и состоянием клиенту.

Метод `sendError()`

Метод `sendError()`, определенный в интерфейсе `HttpServletResponse`, можно использовать для отправки сообщения об ошибке клиенту. Таблица 17 описывает сигнатуру перегружаемого метода `sendError()`:

Таблица 17.

Метод	Описание
<code>public void sendError(int status)</code>	Принимает аргумент типа <code>int</code> как аргумент, который описывает тип ошибки, и отправляет ответ с ошибкой клиенту.
<code>public void sendError(int status, String message)</code>	Принимает дополнительный аргумент типа <code>String</code> , который описывает ошибку, и отправляет ответ с ошибкой клиенту.

Метод `sendError()` генерирует исключение `IllegalStateException`, если он вызывается после завершения ответа.

Следующий фрагмент кода используется для отправки сообщения об ошибке клиенту:

```
public void doPost(HttpServletRequest request, HttpServletResponse response) {

    PrintWriter pw=null;
    try {
        pw = response.getWriter ();
        pw.println("This page is going to give an Error");
        response.sendError(HttpServletResponse.SC_NOT_FOUND,
            "Sorry the source you have requested is
not available.");
        /*
        * Не пытайтесь записывать в буфер ответа после вызова метода
        * sendError().
        */
    } catch (Exception ex) {
        pw.println(ex.getMessage());
    }
}
```



```
}
```

В методе `doPost()` фрагмента кода метод `sendError()` вызывается для отправки кода состояния `SC_NOT_FOUND` (ресурс больше недоступен) с сообщением об ошибке.

Метод `setStatus()`

Метод `setStatus()`, определенный в интерфейсе `HttpServletResponse`, устанавливает информацию состояния о сервлете. Таблица 18 описывает сигнатуры перегружаемого метода `setStatus()`.

Таблица 18.

Метод	Описание
<code>public void setStatus (int status)</code>	Принимает аргумент типа <code>int</code> поле в качестве аргумента, представляющего информацию о состоянии и устанавливает ее для ответа.
<code>public void setStatus (int status, String message)</code>	Принимает дополнительный аргумент типа <code>String</code> , который описывает состояние сервлета.

Метод `setStatus()` генерирует исключение `IllegalStateException`, если он вызывается после того, как ответ принят. Следующий фрагмент кода применяется для выполнения метода `setStatus()` для отправки ответа с ошибкой клиенту:

```
public void doPost (HttpServletRequest request, HttpServletResponse response)
{
    PrintWriter pw =null;
    try
    {
        pw=response.getWriter();
        pw.println("This page is going to display a status code");
        response.setStatus (HttpServletResponse.SC_GONE);
        // Можно писать в ответ после вызова setStatus ().
    }
    catch(Exception ex)
    {
        pw.println(ex.getMessage());
    }
}
```

В предложенном фрагменте кода метода `doPost()` код состояния `SC_GONE` (ресурс больше недоступен) записывается в объект ответа.

Настраиваем страницы ошибок

Сервер приложений может предоставлять страницы ошибок для вывода сообщений с исключениями и кодами состояния. Эти страницы ошибок ненаглядны и с их помощью трудно понять причину ошибки. Для устранения этого недостатка можно создать собственную страницу ошибок в веб-приложении для вывода сообщений с исключениями и ошибками. Необходимо отобразить исключения на страницу ошибок во время развертывания приложения. Например, можно отобразить исключение типа `java.lang.ArithmeticException` на страницу ошибок, и, таким образом, где бы исключение типа `ArithmeticException` не было обнаружено приложением, веб-контейнер выведет подготовленную страницу ошибок.

Веб-контейнер использует три поля для отправки информации об исключении, перехватываемом страницей ошибок. Например, чтобы отправить информацию на страницу ошибок, соответствующую исключению `ArithmeticException`, сервер устанавливает поля `javax.servlet.error.status_code`, `javax.servlet.error.message` и `javax.servlet.error.exception_type` как атрибут в объекте запроса и передает объект запроса странице ошибок.

В процессе обработки ошибки на странице ошибок можно получить значения полей из объекта запроса и соответствующим образом сгенерировать ответ, который описывает информацию об исключении.

Следующий фрагмент кода демонстрирует элемент `<error-page>` дескриптора развертывания:

```
<error-page>
<exception-type>java.lang.NumberFormatException </exception-type>
<location>/ErrorServlet</location>
</error-page>
```

Рассмотрим пример создания настраиваемой страницы обработки ошибок.

Постановка задачи

Пользователь, используя онлайн-приложение-калькулятор, вводит нецифровые символы для сложения чисел. Когда сервлет пытается преобразовать значение, введенное пользователем, в целый тип, генерируется исключение типа `NumberFormatException`. Необходимо создать веб-приложение, которое обрабатывает исключение, используя настраиваемую страницу ошибок. Настраиваемая страница ошибок должна предоставлять информацию об исключении пользователю и записывать исключение в файл серверного журнала.

Решение

Чтобы решить поставленную задачу, необходимо выполнять следующие шаги.

1. Создать пользовательский интерфейс.
2. Создать сервлет.
3. Создать настраиваемую страницу ошибок.
4. Определить страницу ошибок в веб-приложении.
5. Протестировать приложение.

1. Создание пользовательского интерфейса

Создадим интерфейс с пользователем, в котором принимается два числа от пользователя и содержится кнопка ADD. Для нахождения суммы введенных чисел, нужно нажать на кнопку ADD. Можно использовать следующий код для создания страницы Calculator.html:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<FORM ACTION = "http://localhost:8080/Servlet/Calculate" METHOD = GET >

    Enter First Number: <INPUT TYPE = TEXT NAME = "num1" ><BR>
    Enter Second Number: <INPUT TYPE = TEXT NAME = "num2"><BR>
        <INPUT TYPE = SUBMIT VALUE = "ADD" >
</FORM>

</body>
</html>
```

2. Создание сервлета

Для создания сервлета Calculate, который получает числа, введенные пользователем, и вычисляет сумму, можно использовать следующий код:

```
package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/Calculate")
public class Calculate extends HttpServlet {
```

```

private static final long serialVersionUID = 1L;
public Calculate() {
    super();
    // TODO Auto-generated constructor stub
}
public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    PrintWriter pw = res.getWriter();
    int number1 = Integer.parseInt(req.getParameter("num1"));
    int number2 = Integer.parseInt(req.getParameter("num2"));
    int sum = number1 + number2;
    pw.println("Sum of the numbers is &nbsp;&nbsp;" + sum);
}
}

```

В предложенном коде сервлет Calculate получает значения num1 и num2 от класса HttpServletRequest, преобразует их в целые числа, находит сумму чисел и выводит результат в браузер клиента.

3. Создание настраиваемой страницы ошибок

Пользователь может ввести нецифровой символ в форме ввода и в этом случае, при преобразовании значения num1 и num2 в целые, сервлет сгенерирует исключение NumberFormatException. Можно представить это исключение на отдельной странице ошибок, которая обрабатывает исключения, генерируемые сервлетом Calculate. Можно использовать следующий код для создания настраиваемой страницы ошибок, которая выводит данные о возникшем исключении и записывает его в файл серверного журнала:

```

package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/ErrorServlet")
public class ErrorServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public ErrorServlet() {
        super();
    }
    final String EXC = "javax.servlet.error.exception";
    final String MSG = "javax.servlet.error.message";
    final String ST = "javax.servlet.error.status_code";
    /* Переопределение метода doGet() в HttpServlet, который реализует функциональность сервлета. */
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {

```

```

ServletContext sc = getServletContext();
/* Получение экземпляра класса PrintWriter. */
PrintWriter pw = response.getWriter();
/* Получение информации сгенерированном исключении. */
Exception exc = (Exception)request.getAttribute(EXC);
/* Получение кода состояния из HttpServletRequest. */
Integer st_cd = (Integer)request.getAttribute(ST);
/* Получение строки сообщения, которая объясняет ошибку, или сгенериро-
ванного исключения. */
String msg = (String)request.getAttribute(MSG);
pw.println("<HTML>");
pw.println("<BODY>");
pw.println("<HR>");
/* Вывод подробной информации об ошибке клиенту. */
pw.println("<H1>Sorry, an error has occurred that has pre-
vented the server from servicing your request.</H1>");
pw.println("<FONT SIZE = 5>");
pw.println("<TABLE ALIGN = CENTER>");
pw.println("<TR BGCOLOR = LIGHTGREY>");
pw.println("<TD><B> Status Code : </B></TD><TD>" + st_cd +
" </TD>");
pw.println("</TR>");
pw.println("<TR>");
pw.println("<TD><B> Type of Exception :</B></TD><TD>" +
exc.getClass() + " </TD>");
pw.println("</TR>");
pw.println("<TR BGCOLOR = LIGHTGREY>");
pw.println("<TD><B> Message Description : </B></TD><TD>" +
msg + " </TD><HR/>");
String str=exc.toString()+st_cd.toString()+msg;
sc.log("Exception occurred", exc);
pw.println("</TR>");
pw.println("</TABLE>");
pw.println("</FONT>");
pw.println("<HR>");
pw.println("<HR>");
pw.println("<CENTER><H1>Please try again...</H1></CENTER>");
pw.println("</BODY>");
pw.println("</HTML>");
}
}

```

В предложенном коде сервлет `ErrorServlet` получает всю информацию об исключении, сгенерированном сервлетом `Calculate` (имя исключения, сообщение в исключении и код исключения) и выводит ее пользователю.

4. Отображение страницы ошибок на веб-приложение

При развертывании приложения необходимо указать для сервера приложений (в нашем случае `GlassFish`), что если генерируется исключение `NumberFormatException` сервлетом `Calculate`, то сервлет `ErrorServlet` обработает его. Для этого нужно определить страницу оши-

бок, представленную сервлетом `ErrorServlet` в `web.xml` вводом следующего текстового фрагмента

```
<error-page>
  <exception-type>java.lang.NumberFormatException</exception-type>
  <location>/ErrorServlet</location>
</error-page>
```

5. Тестирование приложения

Теперь можно проверить развернутое приложение. Чтобы протестировать приложение, необходимо сначала открыть `Calculator.html` и ввести два числа, которые нужно сложить, в предоставляемых текстовых полях. Чтобы убедиться, что возникло исключение, введите цифровой символ в одном поле и нецифровой символ в другом.

Рис. 66 показывает `Calculator.html` с одним значением 2, и другим значением `y`, которое не является числовым.

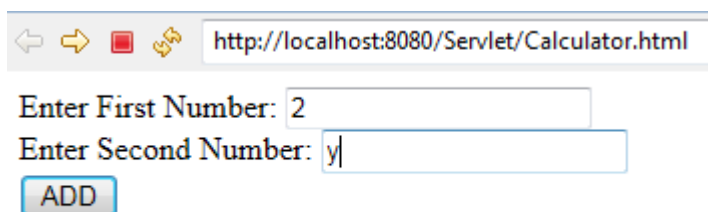


Рис. 66. Результат работы `Calculator.html`

Нажмите кнопку `ADD` в данном окне `Calculator.html` для вызова сервлета `Calculate`. Так как одно из введенных значений нецифровое, сервлет сгенерирует исключение `NumberFormatException`, а сервлет `ErrorServlet` выведет данные об исключении в настраиваемой странице ошибок.

Рис. 67 показывает настраиваемую страницу ошибок, выводимую сервлетом `ErrorServlet`.

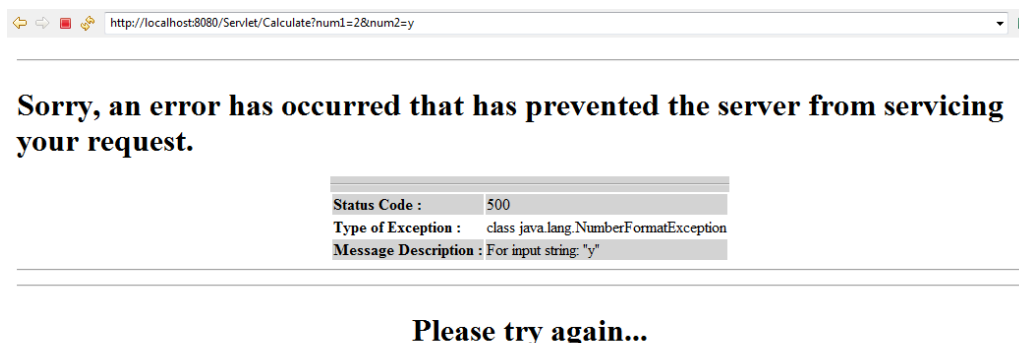


Рис. 67. Вывод настраиваемой страницы ошибок.

Запись исключений и ошибок в файл серверного журнала

Можно записывать сообщения об ошибках и исключениях в файл серверного журнала, что помогает выявлять ошибки, возникающие в веб-приложении. Интерфейс `ServletContext` предоставляет метод `log()` для записи сообщений об ошибках и исключениях и следующий код можно использовать для создания сервлета, который генерирует исключение и записывает данные об исключении в файл серверного журнала:

```
package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/LogDemo")
public class LogDemo extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public LogDemo() {
        super();
    }
    public void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
    {
        ServletContext sc = getServletContext();
        PrintWriter pw = response.getWriter();
        pw.println ("Writing an exception to the server log
file...");
        try
        {
            int x = 9 ;
            int y = x/0 ;
        }
        catch(ArithmeticException ae)
        {
            sc.log("Dividing an integer by zero will not give the
result", ae);
        }
    }
}
```

В предложенном коде сервлет пытается выполнить операцию деления на 0, и в результате генерируется исключение типа `ArithmeticException`. Сервлет получает объект интерфейса `ServletContext` и вызывает метод `log()` для записи исключения в файл журнала сервера.

Рис. 68 показывает вывод сервера.

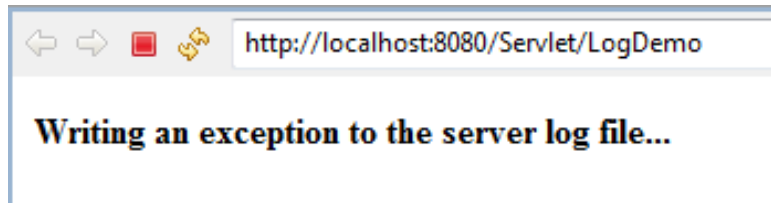


Рис. 68. Вывод сервера LogDemo.

Рис. 69 показывает содержимое файла `server.log` с информацией об исключении `ArithmeticException`.

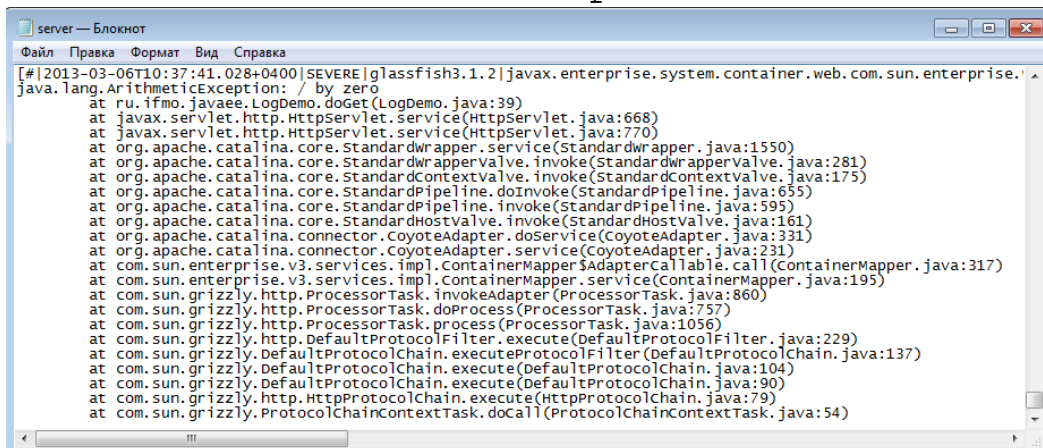


Рис. 69. Содержимое исключения в файле `server.log`.

Взаимодействие сервлетов

В веб-приложении различные сервлеты могут потребовать взаимодействия друг с другом в процессе обработки клиентских запросов. Также один сервлет в веб-приложении может передавать запрос другому сервлету, если возникают соответствующие условия во время обработки запроса. Взаимодействие сервлетов в веб-приложении можно реализовать следующими способами.

1. Используя диспетчер запросов.
2. Используя объект запроса сервлета.

Диспетчер запросов

Диспетчер запросов – это объект интерфейса `javax.servlet.RequestDispatcher`, который обеспечивает возможность взаимодействия сервлетов и позволяет передавать объект `HttpServletRequest` другому ресурсу, который может являться статической HTML-страницей или сервлетом. Кроме того, можно использовать объект `RequestDispatcher` для включения содержимого другого ресурса в сервлет. Интерфейс `ServletContext` предоставляет метод `getRequestDispatcher(String path)`, который воз-

вращает объект `RequestDispatcher`. Аргумент `path` этого метода задает путь из корня контекста. После получения объекта `RequestDispatcher`, можно выполнять два следующих действия.

1. Включить содержимого другого сервлета.
2. Передать объект `request` другому сервлету.

Включение содержимого другого сервлета

Интерфейс `RequestDispatcher` содержит метод `include()`, который можно использовать для включения содержимого другого сервлета. Для этого необходимо сначала получить объект интерфейса `RequestDispatcher`, а затем вызвать метод `include()`. Предположим, имеется сервлет `CopyrightServlet`, который выводит информацию о правах ABC Inc. на веб-сайте этой организации.

Следующий код используется для разработки сервлета `CopyrightServlet`:

```
package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
 * Servlet implementation class CopyrightServlet
 */
@WebServlet("/CopyrightServlet")
public class CopyrightServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public CopyrightServlet() {
        super();
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request,
    HttpServletResponse response)
     */
    public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException
    {
        PrintWriter pw = response.getWriter();
        pw.println("Copyright 2000-2013 ABC, Inc. All Rights Re-
    served.<BR>");
    }
}
```

Для включения содержимого сервлета `CopyrightServlet` в другой сервлет `IncludeServlet` требуется сначала вызвать метод `getRequestDispatcher()` интерфейса `ServletContext`, передав путь к `CopyrightServlet` в качестве параметра и получить объект `RequestDispatcher`. Затем можно вызвать метод `include()` для включения содержимого `CopyrightServlet`. Следующий код используется для создания сервлета `IncludeServlet`:

```

package ru.ifmo.javaee;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
 * Servlet implementation class IncludeServlet
 */
@WebServlet("/IncludeServlet")
public class IncludeServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public IncludeServlet() {
        super();
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request,
    HttpServletResponse response)
     */
    public void doGet(HttpServletRequest request,HttpServletResponse re-
    sponse)
        throws ServletException, IOException
    {
        /*Получение объекта RequestDispatcher */
        RequestDispatcher dispatch =
        getServletContext().getRequestDispatcher("/CopyrightServlet");
        PrintWriter pw = response.getWriter();
        pw.println("<B> The copyright information included from cop-
        yright servlet: </B><BR>");
        /*использование метода include() в RequestDispatcher для включения со-
        держимого*/
        dispatch.include(request, response);
    }
}

```

При запуске сервлета `IncludeServlet` в браузере выводится результат его работы вместе с выводом сервера `CopyrightServlet`, как показано на Рис. 70.

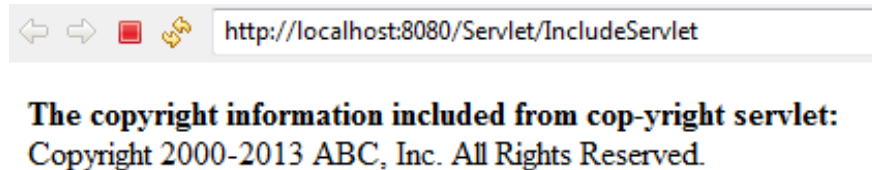


Рис. 70. Вывод сервера IncludeServlet.

Передача запроса другим сервлетам

Объект `RequestDispatcher` может быть использован в приложении для передачи запросов другим сервлетам. Например, рассмотрим компанию ABC, Inc., которая предлагает туристические поездки на время отпуска. Необходимо создать веб-сайт ABC Inc., на котором пользователи могут получать информацию о доступных авиарейсах, транзитных аэропортах и свободных номерах в гостиницах. Можно создать главный сервлет для обработки всех этих типов запросов и формирования ответов пользователям. Однако это существенно усложнит содержание сервлета и создаст проблемы его поддержки. Для устранения этих проблем можно разработать несколько сервлетов для обработки различных типов клиентских запросов. Например, сервлет `HotelInformation` может обрабатывать запросы на гостиничные номера, а сервлет `FlightInformation` может обрабатывать запросы, имеющие отношение к авиаперелетам. Главный сервлет веб-приложения может получать все запросы пользователя и использовать объект `RequestDispatcher` для передачи запроса соответствующему сервлету. Следующий фрагмент кода показывает, как можно использовать `RequestDispatcher` в методе `doGet()` для передачи запросов различным сервлетам приложения:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    String requestType = request.getParameter("Type");
    if (requestType.equals("hotel")) {
        /* Получение объекта RequestDispatcher */
        RequestDispatcher dispatch = getServletContext()
            .getRequestDispatcher("/HotelInformation");
        /*
         * Использование метода forward() объекта
         * RequestDispatcher для передачи запроса
         */
        dispatch.forward(request, response);
    }
    if (requestType.equals("cab")) {
        /* Получение объекта RequestDispatcher */
        RequestDispatcher dispatch = getServletContext()
```

```

        .getRequestDispatcher("/CabInformation");

        /*
         * Использование метода forward() объекта
RequestDispatcher для передачи запроса
         */
        dispatch.forward(request, response);
    }
    if (requestType.equals("flight")) {

        /* Получение объекта RequestDispatcher */
        RequestDispatcher dispatch = getServletContext()

        .getRequestDispatcher("/FlightInformation");
        /*
         * Использование метода forward() объекта
RequestDispatcher для передачи запроса
         */
        dispatch.forward(request, response);
    }
}

```

В предложенном фрагменте программы сервлет проверяет тип запроса, который он получает через объект `RequestDispatcher` и передает его соответствующему сервлету для обработки.

Использование объекта `Request`

Мы рассмотрели, как объект `RequestDispatcher` обеспечивает взаимодействие сервлетов с помощью передачи запроса другому сервлету и включения ответа от другого сервлета. Однако объект `RequestDispatcher` не позволяет сервлетам совместно использовать данные. Для обмена данными между сервлетами можно использовать объект запроса в сервлетах.

Метод `setAttribute()` интерфейса `javax.servlet.ServletRequest` используется для записи значений данных в объект запроса. Другие сервлеты приложения могут использовать метод `getAttribute()` интерфейса `javax.servlet.ServletRequest` для получения значений данных. Например, рассмотрим веб-приложение, в котором `CalculatorServlet` выполняет арифметические вычисления, а другой сервлет выводит результат вычислений. Чтобы отправить вычисленный результат другому сервлету, `CalculatorServlet` может выполнять следующие шаги.

1. Сохранить вычисленные данные как атрибуты в объекте запроса первого сервлета.

2. Передать запрос второму сервлету, используя `RequestDispatcher`, а второй сервлет получает данные из объекта запроса и выводит результат.

Следующий код можно использовать для создания сервлета `CalculatorServlet`, который складывает два числа, заданных пользователем, и сохраняет результат как атрибут объекта запроса. Затем сервлет передает запрос сервлету `DisplayServlet`:

```
package ru.ifmo.javaee;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
 * Servlet implementation class CalculatorServlet
 */
@WebServlet("/CalculatorServlet")
public class CalculatorServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    /**
     * @see HttpServlet#HttpServlet()
     */
    public CalculatorServlet() {
        super();
    }
    /**
     * @see HttpServlet#doGet(HttpServletRequest request,
    HttpServletResponse
     * response)
     */
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws
    ServletException, IOException {
        /* Получение чисел, введенных пользователем в HTML-форме */
        int num1 = Integer.parseInt(request.getParameter("number1"));
        int num2 = Integer.parseInt(request.getParameter("number2"));
        /* Нахождение суммы чисел, введенных пользователем. */
        int result = num1 + num2;
        /* Запись результата в атрибут объекта запроса */
        request.setAttribute("result", new Integer(result));
        /* Получение объекта ServletContext */
        ServletContext ctx =
        getServletContext().getServletContext();
        /* Получение объекта RequestDispatcher */
        RequestDispatcher reqDispatcher = ctx
        .getRequestDispatcher("/DisplayServlet");
        /* Передача запроса */
        reqDispatcher.forward(request, response);
    }
}
```

```
    }  
}
```

В приведенном коде сервлет `CalculatorServlet` получает числа, отправленные как параметры запроса, в методе `doGet()`. Затем сервлет складывает числа и запоминает результат в объекте запроса, используя метод `setAttribute()`. В завершение сервлет использует объект `RequestDispatcher` для передачи запроса сервлету `DisplayServlet`.

Можно использовать следующий код для разработки сервлета `DisplayServlet`, который выводит результат вычислений:

```
package ru.ifmo.javaee;  
import java.io.IOException;  
import java.io.PrintWriter;  
import javax.servlet.ServletException;  
import javax.servlet.annotation.WebServlet;  
import javax.servlet.http.HttpServlet;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
  
/**  
 * Servlet implementation class DisplayServlet  
 */  
@WebServlet("/DisplayServlet")  
public class DisplayServlet extends HttpServlet {  
    private static final long serialVersionUID = 1L;  
    /**  
     * @see HttpServlet#HttpServlet()  
     */  
    public DisplayServlet() {  
        super();  
    }  
    /**  
     * @see HttpServlet#doGet(HttpServletRequest request,  
     * HttpServletResponse  
     * response)  
     */  
    protected void doGet(HttpServletRequest request,  
        HttpServletResponse response) throws  
ServletException, IOException {  
        /* Получение результата, хранимого в объекте запроса */  
        Integer res = (Integer) request.getAttribute("result");  
        /* Получение объекта PrintWriter */  
        PrintWriter pw = response.getWriter();  
        /* Отправка ответа для вывода результата */  
        pw.println("The result of the calculation is: " +  
res.toString());  
    }  
}
```

В предложенном коде сервлет `DisplayServlet` использует метод `getAttribute()` для получения значения результата, полученно-

го в объекте запроса и затем выводит сообщение о полученном результате пользователю.

Следующую HTML-страницу CalculatorPage.html можно использовать для получения пользовательских данных и передачи этих данных сервлету CalculatorServlet:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Calculate</title>
</head>
<body>
  <form name = frm method="GET" ac-
tion="http://localhost:8080/Servlet/CalculatorServlet">
  <p align="center"><font size="5"><b>Calculation
Form</b></font></p>
  <TABLE ALIGN="center" height="57">
  <TR>
    <TD >
      <b>Enter First Number:</b>
    </TD>
    <TD >
      <input type="text" name="number1" size="20" >
    </TD>
  </TR>
  <TR>
    <TD >
      <b>Enter Second Number:</b>
    </TD>
    <TD >
      <input type="text" name="number2" size="20" >
    </TD>
  <TR align="center">
    <TD colspan=2>
      <input type="Submit" value=" Add " name="B1" >
    </TD>
  </TR>
  </TABLE>
</form>
</body>
</html>
```

Рис. 71 показывает результат работы указанной выше HTML-страницы, которая принимает два числа.

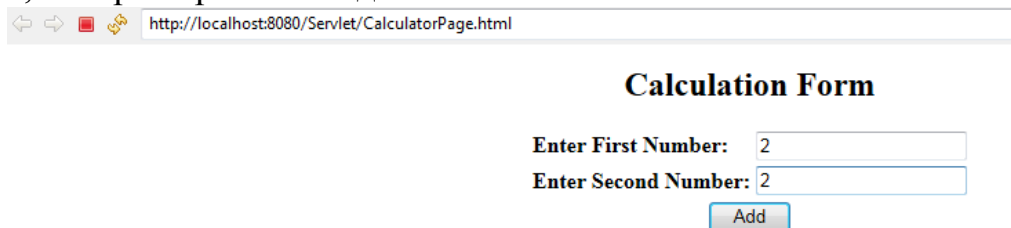


Рис. 71. HTML-страница для ввода двух чисел.

После ввода чисел, которые необходимо сложить, нажмите кнопку **Add** для вызова `CalculatorServlet`. Сервлет `CalculatorServlet` складывает числа и передает результат сервлету `DisplayServlet`, который выводит результат в браузере клиента. Рис. 72 показывает результат работы `CalculatorServlet`, выведенный сервлетом `DisplayServlet`.

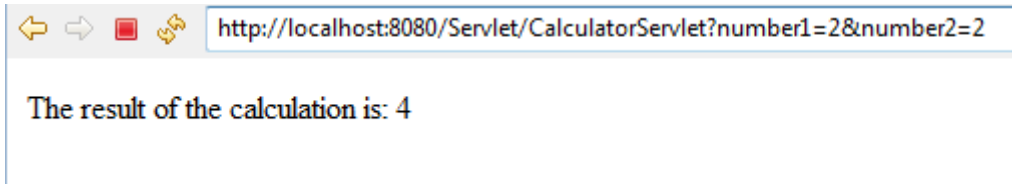


Рис. 72. Результат результата работы сервлета `CalculatorServlet`.

Вопросы для самопроверки

1. Что представляет собой сервлет?
2. Какой компонент системы развертывает JavaEE веб-компонент в веб-контейнере, который предоставляет среду выполнения для сервлетов?
3. В каких пакетах находятся классы и интерфейсы, которые используются для разработки сервлета?
4. Расширением какого класса может быть разработан сервлет, если клиент использует протокол HTTP для отправки запроса сервлету?
5. В каком интерфейсе определяются методы, которые используются для управления жизненным циклом сервлета?
6. Какой интерфейс реализуется веб-контейнером, чтобы передавать конфигурационную информацию сервлету?
7. Какой интерфейс определяет методы жизненного цикла сервлетов, такие как `init()`, `service()` и `destroy()`?
8. Какой метод экземпляра сервлета вызывается при инициализации сервлета веб-контейнером?
9. Какой метод экземпляра сервлета вызывает Веб-контейнер, когда веб-контейнер получает клиентский запрос?
10. Какой метод Веб-контейнер вызывает перед удалением экземпляра сервлета?
11. Какие процедуры необходимо выполнить для создания и запуска сервлета?
12. Какой объект представляет запрос, отправленный клиентом с помощью HTTP?
13. Какой объект представляет ответ, отправленный экземпляром сервлета клиенту с помощью HTTP?

14. Какой интерфейс содержит методы для извлечения заголовка запроса?
15. Какой интерфейс содержит методы для отправки ответ на запрос клиента?
16. Какой интерфейс содержит методы для установки атрибутов контекста для сервлета?
17. Какие интерфейсы содержит пакет javax.servlet.http?
18. Какой интерфейс содержит методы для получения параметров запроса, отправленного с использованием HTTP?
19. Какой интерфейс содержит методы для обработки ответа и кодов статуса для сервлетов, использующих HTTP?
20. Какой интерфейс предоставляет различные методы для сохранения состояния конечного пользователя в веб-приложении?
21. Какие события жизненного цикла сервлета генерируются внутри веб-контейнера?

Введение в технологию JSP

Технология Java Server Pages (JSP) была разработана компанией Sun Microsystems для облегчения создания страниц с динамическим содержанием. Сервлеты наилучшим образом подходят для выполнения контролирующей функции приложения (в модели MVC) в процессе обработки запросов и вычисления варианта ответа клиенту. Страницы JSP выполняют функцию формирования текстовых документов, которые содержат статические данные, и могут быть оформлены в одном из текстовых форматов HTML, SVG, WML или XML. При этом JSP элементы предназначены для формирования динамического содержания текстовых документов. В процессе создания страниц JSP могут использоваться библиотеки JSP тегов, а также EL (Expression Language) для внедрения Java-кода в статичное содержимое JSP-страниц.

Основные возможности JSP

Типичное веб-приложение в соответствии с моделью MVC состоит из логики представления, имеющей статическое содержание, которое используется в дизайне структуры веб-страницы на основе разметки страницы, изображений, цвета и текста. Бизнес-логика, или динамическое содержание, включает вычислительные компоненты предприятия с использованием базы данных и модели бизнес операций. При разработке веб-приложений программисты занимаются кодированием статического содержания логики представления, в то время как этой работой должны заниматься дизайнеры, а не программисты.

Технология JSP содействует разделению работы веб-дизайнера и веб-разработчика. Веб-дизайнер может разрабатывать и формировать разметку веб-страницы, используя HTML. С другой стороны, веб-разработчик, работая независимо, может использовать код Java и теги JSP для кодирования бизнес-логики - динамического содержания. Разделение статического и динамического содержимого способствует разработке более качественных приложений с улучшенной производительностью.

JSP-страница после компиляции генерирует на сервере сервлет и, следовательно, включает в себе все функциональные возможности сервлетов. Таким образом, сервлеты и JSP разделяют общие возможности, основанные на платформенной независимости, использовании баз данных и возможности серверного программирования. Однако существуют некоторые фундаментальные различия между сервлетами и JSP.

В частности, сервлеты объединяют файлы (HTML-файл для статического содержимого и файлы Java для динамического содержимого) для обработки статической логики представления и динамической бизнес-логики. Из-за этого изменение, сделанное в любом файле, требует перекомпиляции сервлета и его развертывания. JSP страницы позволяют коду Java встраиваться непосредственно в HTML-страницу с помощью специальных тегов. При этом содержимое HTML и содержимое Java могут размещаться в отдельных файлах. Любые изменения, сделанные в содержимом HTML, автоматически компилируются и загружаются на сервер.

Программирование сервлетов предполагает активное кодирование, и любое изменение, сделанное в коде сервлета, требует содержимого статического кода (для дизайнера) и содержимого динамического кода (для разработчика) для выполнения соответствующих изменений. С другой стороны, JSP-страница, в силу отдельного размещения статического и динамического содержимого, облегчает и для веб-разработчиков, и для веб-дизайнеров независимую работу. Чтобы облегчить внедрение динамического содержания JSP использует ряд тегов, которые дают возможность проектировщику страницы вставить значение полей объекта JavaBean в файл JSP.

Содержимое Java Server Pages (теги HTML, теги JSP и скрипты) компилируется сервером приложений в код сервлета. Этот процесс выполняет трансляцию как динамических, так и статических составляющих, объявленных внутри файла JSP. Об архитектуре сайтов, использующих JSP/Servlet-технологии, часто говорят как о thin-client (использование ресурсов клиента незначительно), поскольку большая часть логики выполняется на сервере.

Для иллюстрации возможностей JSP рассмотрим пример простого веб-приложения, которое создает пользовательский интерфейс для приложения создания списка электронных адресов. Приложение состоит из двух страниц. Первая страница - `email_list.html` запрашивает пользователя ввести имя, фамилию и электронный адрес и, при нажатии пользователем на кнопку "Ввести", вызывается страница `display_email.jsp` и ей передаются введенные значения. В процессе работы на экране отображается информация, представленная на Рис. 73.

Регистрация в списке email

Для регистрации в списке, введите персональные данные и email адрес ниже. Затем, нажмите кнопку Ввести.

Имя: Ivan

Фамилия: Petrov

Email адрес: petrov@ifmo.ru

Ввести

Рис. 73. Форма для регистрации.

Когда страница JSP получает введенные пользователем три значения, она может обработать полученные значения, например, проверить их на допустимость и записать в файл или базу данных. В нашем же примере JSP просто возвращает введенные значения, которые выводятся в браузере клиента. Из полученной страницы можно вернуться на первую страницу, нажав кнопку (Back) или нажать кнопку "Возврат". Результат работы страницы JSP отображается на Рис. 74.

Спасибо за регистрацию в списке email

Ниже представлена введенная Вами информация:

Имя:	Ivan
Фамилия :	Petrov
Email адрес:	petrov@ifmo.ru

Чтобы ввести другой адрес, нажмите кнопку Back в браузере или кнопку Возврат ниже.

Возврат

Рис. 74. Результат работы страницы JSP.

Ниже представлены тексты страниц email_list.html и display_email.jsp.

email_list.html:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Простое приложение</title>
</head>
<body>
    <%
        // получение параметров из запроса
        String firstName = request.getParameter("firstName");
        String lastName = request.getParameter("lastName");
        String emailAddress = request.getParameter("emailAddress");
    %>
    <h1>Спасибо за регистрацию в списке email </h1>
    <p> Ниже представлена введенная Вами информация:</p>
    <table cellspacing="5" cellpadding=" 5 " border="1">
        <tr>
            <td align="right">Имя:</td>
            <td><%= firstName%></td>
        </tr>
        <tr>
            <td align="right">Фамилия :</td>
            <td><%= lastName%></td>
        </tr>
        <tr>
            <td align="right">Email адрес:</td>
            <td><%= emailAddress%></td>
        </tr>
    </table>
    <p>
        Чтобы ввести другой адрес, нажмите кнопку Back <br>
        в браузере или кнопку Возврат <br> ниже.
    </p>
    <form action="email_list.html" method="post">
        <input type="submit" value="Возврат">
    </form>
</body>
</html>
```

display_email.jsp:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
```

```

<%
    // получение параметров из запроса с помощью скриплетта
    String firstName = request.getParameter("firstName");
    String lastName = request.getParameter("lastName");
    String emailAddress = request.getParameter("emailAddress");
%>
<h1>Спасибо за регистрацию в списке email </h1>
<p>Ниже представлена введенная Вами информация:</p>
<table cellspacing="5" cellpadding=" 5 " border="1">
    <tr>
        <td align="right">Имя:</td>
        <td><%= firstName%></td>
    </tr>
    <tr>
        <td align="right">Фамилия :</td>
        <td><%= lastName%></td>
    </tr>
    <tr>
        <td align="right">Email адрес:</td>
        <td><%= emailAddress%></td>
    </tr>
</table>
<p>
    Чтобы ввести другой адрес, нажмите кнопку Back <br>
    в браузере или кнопку Возврат <br> ниже.
</p>
<form action="join_email_list.html" method="post">
    <input type="submit" value="Возврат">
</form>
<!--

```

В представленной странице JSP содержатся теги HTML и встроенный код Java, для кодирования которого применяются следующие символы:

- Для кодирования кода JSP, т.н. скриплетта, содержащего одно или несколько утверждений Java, используются теги `<%` и `%>`.
- Для кодирования выражений, которые преобразуются в символьную строку можно использовать `<%=` и `%>`.
- Для получения значения параметров, которые передаются в JSP, можно использовать метод `getParameter` неявного определенного объекта запроса с именем `request`.

Синтаксис JSP скриплет - `<% Java утверждения %>`. При этом, в рамках сценариев можно кодировать одно или несколько утверждений Java и каждое утверждение Java необходимо заканчивать точкой с запятой (;).

Синтаксис для выражений JSP - `<%= выражение %>`. В выражении JSP кодируется допустимое выражение Java, которое возвращает значение объекта Java или примитивного типа. Поскольку выражение не является утверждением Java, в конце не ставится точка с запятой.

Синтаксис для получения параметров из неявного объекта запроса - `request.getParameter(parameterName)`.

Рассмотрим несколько примеров, которые используют скриплеты и выражения. Ниже представлен фрагмент кода, в котором используются скриплеты и выражения, отображающие значения параметра `firstName`:

```
<%  
String firstName = request.getParameter("firstName");  
%>  
Имя <%= firstName %>
```

или

```
Имя <%= request.getParameter ("firstName") %>
```

Следующий фрагмент кода JSP показывает два скриплета и выражения, которые отображают строку HTML 5 раз:

```
<%  
int numOfTimes = 1;  
while (numOfTimes <= 5)  
{  
%>  
<h1>This line is shown <%= numOfTimes %> of 5 times in a JSP.</h1>  
<%  
numOfTimes++;  
}  
%>
```

В большинстве случаев, используемый в коде метод `getParameter`, возвращает значение параметра. Для текстового поля, как правило, значение вводится пользователем. Но для группы переключателей или поля со списком, `getParameter` возвращает значение кнопки или элемента, выбранного пользователем.

Для флажков или независимых переключателей, которые содержат значение атрибута, метод `getParameter` возвращает это значение, если флажок или кнопка выбраны и значение `null`, если не выбраны. Для флажков или независимых переключателей, которые не содержат значения атрибута, `getParameter` метод возвращает значение "on", если флажок или кнопка выбраны и нулевое значение `null`, если нет, как в следующем фрагменте кода:

```
<%  
// возвращает значение "on" если выбрано, null в противном случае.  
String rockCheckBox = request.getParameter ( "Rock" );  
if (rockCheckBox != null)  
{  
%>  
Вы выбрали музыку Rock!  
<%  
}  
%>
```

Для получения нескольких значений для одного имени параметра, можно использовать метод `getParameterValues`, как показано в следующем фрагменте кода:

```
<%
// возвращает значения элементов, выбранных в списке
String[] selectedCountries = request.getParameterValues ("country" );
for (int i = 0; i < selectedCountries . length; i++)
{
%>
<%=selectedCountries [i] %> <br>
<%
}
%>
```

Этот метод полезен для элементов управления типа список, которые позволяют множественный выбор. После использования `getParameterValues` метод возвращает массив строковых объектов и можно использовать цикл для получения значений из массива.

Для получения всех имен параметров, отправленных вместе с запросом, можно использовать метод `getParameterNames`, возвращающий объект `Enumeration`, который содержит имена параметров. Затем можно просматривать `Enumeration` объекты для получения имен параметров, и можно использовать метод `getParameter`, возвращающий значение для каждого имени параметра, как показано в следующем фрагменте кода, в котором скриптлет читает и показывает все параметры `request` и их значения:

```
<%
Enumeration<String> parameterNames = request.getParameterNames();
while (parameterNames.hasMoreElements()) {
String parameterName = parameterNames.nextElement();
String parameterValue = request.getParameter(parameterName);

%>
<%=parameterName%>
имеет значение
<%=parameterValue%>.
<br>
<%
}
%>
```

В большинстве случаев в отношении использования класса `Enumeration` достаточно лишь знать, что объект `Enumeration` является коллекцией, в которой можно просматривать элемент за элементом. Чтобы определить, есть ли еще элементы в коллекции, следует использовать метод `hasMoreElements`, который возвращает `Boolean` значение. А для того, чтобы получить следующий элемент в коллекции, следует использовать метод `nextElement`.

Использование регулярных классов в JSP

Рассмотрим, как можно использовать регулярные классы Java для выполнения требуемой обработки в JSP. В частности, рассмотрим, как с помощью двух классов User и UserIO выполняется обработка данных JSP-приложения для создания списка электронных адресов. Утверждение package в начале каждого класса показывает, где каждый класс сохранен. Класс User находится в подкаталоге business, поскольку он определяет объект бизнеса, а класс UserIO хранится в подкаталоге data, поскольку он обеспечивает доступ к данным для приложения.

Ниже представлены код для бизнес-класса User и код для класса UserIO, выполняющего операции ввода/вывода:

```
package business;
public class User {
    private String firstName;
    private String lastName;
    private String emailAddress;
    public User() {
        firstName = "";
        lastName = "";
        emailAddress = "";
    }
    public User(String firstName, String lastName, String
emailAddress) {
        super();
        this.firstName = firstName;
        this.lastName = lastName;
        this.emailAddress = emailAddress;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getEmailAddress() {
        return emailAddress;
    }
    public void setEmailAddress(String emailAddress) {
        this.emailAddress = emailAddress;
    }
}
```

Класс User определяет пользователя приложения. Этот класс содержит три переменных экземпляра класса: firstName, lastName и emailAddress. Он содержит конструктор, который принимает три значения для этих переменных. Он также содержит методы get и set для каждой переменной экземпляра класса.


```

package data;
import java.io.*;
import business.User;
public class UserIO {
    public static void add(User user, String filepath) throws
IOException {
        File file = new File(filepath);
        PrintWriter out = new PrintWriter(new FileWriter(file,
true));
        out.println(user.getEmailAddress() + "|" + us-
er.getFirstName() + "|"
+ user.getLastName());
        out.close();
    }
}

```

Класс UserIO содержит один статический метод с именем add, который записывает данные, хранящиеся в объекте User, в текстовый файл. Этот метод получает два параметра: объекты user и filepath, который определяет путь к файлу. Если необходимый файл уже существует, то метод будет добавлять пользовательские данные в конец, а если файл не существует, метод создает файл и добавляет данные в начало файла.

В структуре проекта Eclipse вновь созданные классы будут представлены с соответствующих пакетах в окне Project Explorer как показано на Рис. 75.

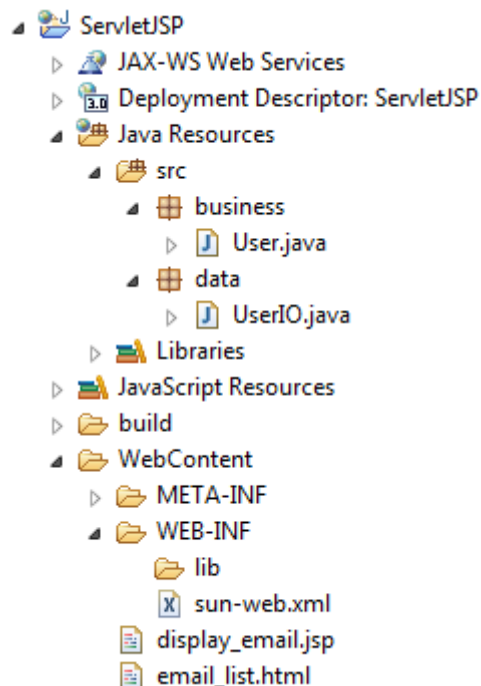


Рис. 75. Структура проекта ServletJSP регистрации email

Ниже представлен код JSP-страницы display_email.jsp после расширения функциональности приложения по обработке полученных параметров с использованием классов User и UserIO.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
```

```

        pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Простое приложение</title>
</head>
<body>
        <!-- импорт пакетов и классов необходимых для скриптов -->
<%@ page import="business.*,data.*"%>
<%
        // получение параметров из объекта request
String firstName = request.getParameter("firstName");
String lastName = request.getParameter("lastName");
String emailAddress = request.getParameter("emailAddress");
// получение реального пути для файла EmailList.txt
ServletContext sc = this.getServletContext();
String path = sc.getRealPath("/WEB-INF/EmailList.txt");
// использование регулярных объектов Java
User user = new User(firstName, lastName, emailAddress);
UserIO.add(user, path);
%>
<h1>Спасибо за регистрацию в списке email</h1>
<p>Ниже представлена введенная Вами информация:</p>
<table cellspacing="5" cellpadding=" 5 " border="1">
    <tr>
        <td align="right">Имя:</td>
        <td><%=user.getFirstName() %></td>
    </tr>
    <tr>
        <td align="right">Фамилия :</td>
        <td><%=user.getLastName() %></td>
    </tr>
    <tr>
        <td align="right">Email адрес:</td>
        <td><%=user.getEmailAddress() %></td>
    </tr>
</table>
<p>
        Чтобы ввести другой адрес, нажмите кнопку Back <br> в браузере
        или кнопку Возврат <br> ниже.
</p>
<form action="email_list.html" method="post">
    <input type="submit" value="Возврат">
</form>
</body>
</html>

```

В первом утверждении раздела body используется специальный тег JSP, предназначенный для импорта пакетов business и data, содержащих классы User и UserIO. Более подробно мы рассмотрим кодирование тега этого типа позже.

В скрипте JSP метод getParameter используется для получения значений трех параметров, которые передаются на странице JSP. Эти

значения сохраняются в соответствующих объектах String. Затем, следующие два утверждения используют метод `getRealPath` объекта `ServletContext` для получения реального пути к файлу `EmailList.txt`, который размещается в подкаталоге `WEB-INF` приложения. Поскольку подкаталог `WEB-INF` непосредственно не доступен через интернет, пользователи не имеют возможности получить доступ к этому файлу.

Два последних утверждения скриптлета создают объект `User` на основе полученных значений трех параметров и вызывают метод статический `add` класса `UserIO` для записи этого объекта в файл `EmailList.txt`. Отметим, что реальный путь к файлу `EmailList.txt` передается методу `add` класса `UserIO`.

После выполнения скриптлета, код в JSP определяет макет страницы в формате `html`, состоящий из таблицы, в которой выражения JSP показывают значения имени, фамилии и адреса электронной почты, используя методы `get` объекта `User`. Здесь в выражениях JSP можно использовать соответствующие объекты параметров, но показывается возможность применения `get` методов для доступа к значениям переменных. Кроме того, можно убедиться, что данные, показанные в JSP точно такие же, как записанные в файл `EmailList.txt`.

Файла `EmailList.txt` располагается в подкаталоге `c:\Glassfish3\glassfish\domains\domain1\eclipseApps\ServletJSP\WEB-INF\`.

Мы рассмотрены, как использовать два наиболее распространенных вида JSP тегов: теги для скриптлетов и для выражений. В последующих разделах будут рассмотрены дополнительные возможности JSP более детально.

Жизненный цикл JSP

Когда браузер клиента запрашивает конкретную JSP-страницу, сервер отправляет запрос контейнеру JSP. Контейнер JSP – это часть веб-контейнера, который компилирует JSP-страницу в сервлет. Рис. 76 показывает поток событий, который возникает после запроса клиента к JSP-странице.

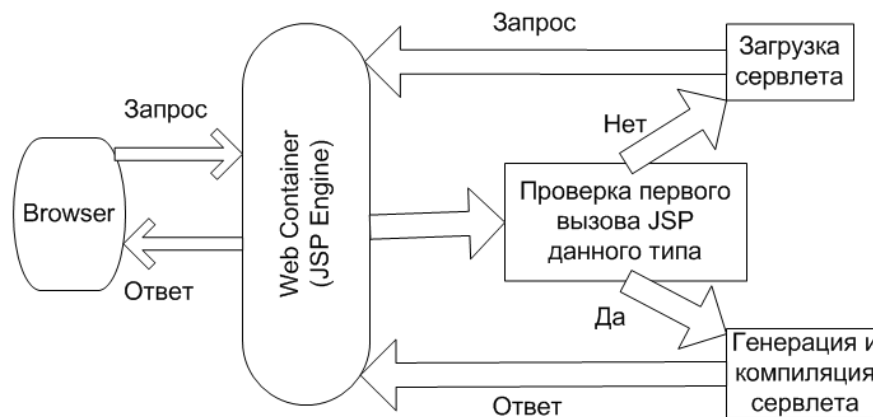


Рис. 76. Цикл запрос-ответ JSP-страницы.

Цикл запрос-ответ, в сущности, включает в себе две фазы, а именно фазу трансляции и фазу обработки запроса. Фаза трансляции реализуется контейнером JSP и заключается в генерировании сервлета, что приводит к созданию файла класса JSP-страницы, который реализует интерфейс сервлета. Во время фазы обработки запроса генерируется ответ по спецификации запросов. После того, как сервлет загружается первый раз, он остается активным, обрабатывая все последующие запросы, и экономит время, которое в противном случае тратилось бы на загрузку сервлета при каждом запросе.

После того, как JSP транслируется в сервлет, контейнер вызывает следующие методы жизненного цикла сервлета, которые определены в интерфейсе `javax.servlet.jsp.JspPage`:

`jspInit()`: Этот метод вызывается каждый раз, когда инициализируется сервлет.

`jspService()`: Этот метод вызывается, когда получен запрос на JSP-страницу.

`jspDestroy()`: Этот метод вызывается перед тем, как удаляется сервлет.

Структура JSP-страницы

Итак, рассматривая основные возможности JSP, мы выяснили, что JSP-страница состоит из обычных HTML-тегов, представляющих статическое содержимое, и кода, заключенного в специальные теги, представляющие динамическое содержимое. Эти теги начинаются с символа “<%” и заканчиваются символом “%>”. Элементы сценария и директивы записываются между символами “<%” и “%>”. Элементы сценария состоят из фрагментов кода `java`, в то время как директивы используются для определения спецификации всей JSP-страницы и кодируются

<%@ директива %>. Например, директива используется для импорта необходимых классов, используемых JSP, следующим образом:

```
<%@ page import="business.*,data.*,java.util.Date"%>
```

В рассмотренном примере директива, которая используется для выполнения импорта, называется *page directive*, и эта директива кодируется, начиная с начального тега " <%@" и слова *page*, за которым следует атрибут *import*. В кавычках после знака равенства для этого атрибута кодируются имена классов Java, которые импортируются также, как в утверждениях *import* в Java.

Комментарии, которые предоставляют дополнительную информацию о различных разделах кода, заключаются в символы <%-- JSP-комментарий --%>.

Следующий фрагмент кода показывает пример кодирования комментариев на странице JSP:

```
<!--  
<p> This email address was added to our list on <%= new Date() %></p>  
-->
```

Здесь комментарии HTML используются для строки, которая содержит выражение, представляющее дату.

Следующий фрагмент кода показывает использование комментариев на странице JSP:

```
<%--  
<p>This email address was added to our list on <%= new Date() %></p>  
--%>
```

Здесь комментарии JSP используются для строки, также содержащей выражение JSP, показывающее дату.

При кодировании комментариев HTML и JSP необходимо понимать, как они работают, чтобы использовать их корректно. В частности, любой код Java в комментарии HTML компилируется и запускается, но браузер не отображает его. Действительно, в первом примере создается новый объект *Date*, хотя он и не отображает дату в браузере. При этом значение объекта даты возвращается в браузер, но комментируется HTML.

И наоборот, любой код Java в JSP комментариях не компилируется, не выполняется и не возвращается в браузер. Например, второй пример не создает объект *Date* и не возвращает его в браузер в качестве комментария.

В заключение рассмотрим, как кодируются комментарии внутри скриптлета на следующем примере:

```
<%  
// получение параметров из request  
String firstName = request.getParameter("firstName");  
String lastName = request.getParameter("lastName");  
String emailAddress = request.getParameter("emailAddress");
```

```

/*
User user = new User(firstName, lastName, emailAddress);
UserIO.add(user, path);
*/
%>

```

Здесь кодируются комментарии одной строкой перед тремя утверждениями, которые получают параметры из запроса. Затем многострочные комментарии используются для двух утверждений, которые создают объект `User` и записывают его в текстовый файл. Поскольку эти комментарии точно такие же, как в языке Java, не должно быть проблем в их использовании.

Теги `<%! %>` предназначены для объявления переменных и методов в JSP. Как известно, при первом обращении к JSP создается один экземпляр JSP страницы, загружается в память и запускается поток, который выполняет Java-код в JSP. Для каждого последующего запроса JSP страницы, запускается другой поток, который может получать доступ к одному экземпляру JSP. Когда кодируются переменные скриплетта, они рассматриваются как локальные переменные, и каждый поток получает свою собственную копию каждой локальной переменной, что, как правило, и необходимо для разработчика.

Однако, также можно объявить переменные в JSP, которые будут общими для всех потоков, имеющих доступ к JSP, как это представлено в следующем фрагменте кода:

```

<!-- импорт пакетов, необходимых для страницы--%>
<%@ page import= "business.*, data.*, java.util.Date, java.io.*"%>
<%!
// объявление переменной экземпляра для страницы
int globalCount = 0; // не безопасно для нескольких потоков
%>
<%!
// объявляется метод для page
public void add (User user, String filename) throws IOException
{
    PrintWriter out = new PrintWriter (
        new FileWriter (filename, true) );
    out.println (user.getEmailAddress () + " | " + user.getFirstName () + " |
" + user.getLastName () );
    out.close() ;
}
%>
<%
String firstName = request.getParameter("firstName");
String lastName = request.getParameter("lastName");
String emailAddress = request.getParameter("emailAddress");
ServletContext sc = getServletContext();
String path = sc.getRealPath("/WEB-INF/EmailList.txt");
User user = new User(firstName, lastName, emailAddress);
// используется объявленный метод
this.add(user, path);

```

```
// изменяется значение переменной экземпляра
globalCount++; // не безопасно для нескольких потоков
%>
<p>
This email address was added to our list on <%= new Date() %><br>
This page has been accessed <%= globalCount %> times. </p>
```

В данном фрагменте кода объявленная переменная экземпляра `globalCount` инициализируется при первом запросе к JSP, и затем каждый поток может получать доступ к этой переменной экземпляра. При этом переменная `globalCount` увеличивается на единицу при каждом запросе этой JSP страницы, и затем накопленное значение выводится, представляя количество обращений к странице.

Следует отметить, что использование переменной экземпляра может привести к неправильным результатам и не является хорошей практикой. Проблемы возникают при выполнении серии операций увеличения переменной `globalCount`. На практике операция увеличения реализуется последовательностью операций чтения значения переменной, его модификации и записи обратно в переменную увеличенного значения. При нескольких потоках все имеют доступ к этой переменной, и два потока могут попытаться прочитать переменную в одно то же время, что может привести к потерянными обновлениям. Иными словами, переменная экземпляра в JSP не является потокобезопасной и следует избирательно использовать описанную возможность.

Хотя существует несколько методов для создания переменных экземпляра потокобезопасными, ни один из них не реализуется простыми методами. В итоге, если необходимо обеспечить безопасность потокоориентированных операций, следует использовать локальные переменные, а не переменные экземпляра, когда такое возможно.

Кроме объявления переменных экземпляра в JSP страницах можно объявлять методы, как это сделано в рассмотренном фрагменте кода. В этом коде JSP объявляет и вызывает метод `add`, который записывает объект пользователя в файл. Это также является плохой практикой, и вместо объявления методов в JSP следует рассмотреть возможность реорганизации приложения. В некоторых случаях можно использовать обычные классы Java, как класс `UserIO`, в других случаях, предпочтительнее использовать сервлет, но в большинстве случаев следует использовать комбинацию сервлетов, JSP страниц и обычных классов Java.

Директивы JSP

Директива в JSP-странице предоставляет глобальную информацию о конкретной JSP-странице, которая может быть трех типов:

- директива `page`;
- директива `include`;
- директива `taglib`.

Для директивы включаются символы `<%@` в качестве префикса, и символы `%>` в качестве суффикса имени директивы. Директива может иметь более одного атрибута. Синтаксис определения директивы:

```
<% @ directive attribute="value" %>;
```

Директива `page`

Директива `page` определяет атрибуты, которые уведомляют веб-контейнер о главных установках JSP-страницы. Можно задавать различные атрибуты в директиве `page` в соответствии с синтаксисом директивы:

```
<% @ page attribute_list %>, например,
```

```
<% @ page import = "business.*, data.*, java.util.Date" language="java" session="true" %>
```

Таблица 19 описывает атрибуты, поддерживаемые директивой `page`, представляя их возможными значениями:

Таблица 19.

Имя атрибута	Описание
<code>language</code>	Определяет язык сценария JSP-страницы.
<code>extends</code>	Определяет родительский класс, который расширяет сгенерированный JSP сервлет.
<code>import</code>	Импортирует список пакетов, классов или интерфейсов в генерируемый сервлет.
<code>session</code>	Определяет, может ли генерируемый сервлет использовать сеанс или нет. Генерируется неявный объект <code>session</code> , если значение установлено в <code>true</code> . По умолчанию значением атрибута <code>session</code> является <code>true</code> .
<code>buffer</code>	Задаёт размер буфера вывода. Если размер установлен в <code>none</code> , буферизация не выполняется. Значением по умолчанию <code>buffer</code> является 8 КВ.
<code>autoflush</code>	Задаёт автоматическую очистку буфера вывода, если значение установлено в <code>true</code> . Если значение установлено в <code>false</code> , возникает исключение при заполнении буфера. Значением по умолчанию атрибута <code>autoFlush</code> является

Имя атрибута	Описание
	true.
<code>isThreadSafe</code>	Задаёт, обеспечивает ли JSP-страница безопасное выполнение потоков или нет.
<code>errorPage</code>	Задаёт, что любое необработанное сгенерированное исключение будет передано данному URL.
<code>isErrorPage</code>	Задаёт, что текущая JSP-страница является страницей ошибок, если значение атрибута установлено в true. Значением по умолчанию атрибута <code>isErrorPage</code> является false.
<code>contentType</code>	Определяет тип MIME ответа. Значением по умолчанию атрибута <code>contentType</code> является text/html.

Директива `include`

Директива `include` используется для задания имен файлов (в виде их относительных URL), которые присоединяются при компиляции JSP-страницы. Содержимое файлов, добавленных таким образом, становится частью JSP-страницы. Директива `include` также может быть использована для включения части кода, который является общим для нескольких страниц. Это помогает избежать использования компонентов отдельно для каждой страницы. Следующий синтаксис применяется для определения директивы `include`: `<%@ include file = "URLname" %>`

Например, строка кода для включения файла HTML (`Superstore.html`) в JSP-страницу, содержащего название и логотип магазина Superstore Online Shopping, может быть написана следующим образом: `<%@ include file = "Superstore.html" %>`

Директива `taglib`

Директива `taglib` импортирует нестандартный тег в текущую JSP-страницу. Нестандартный тег – это определенный пользователем тег, который используется для выполнения повторяющихся задач на JSP-странице. Файл дескриптора библиотеки тегов (Tag Library Descriptor - TLD) определяет функциональность нестандартного тега. Директива `taglib` ассоциируется с URI для уникальной идентификации нестандартного тега. Также ссылка связывает строку префикса тега,

который служит отличительным признаком нестандартного тега, с другой библиотекой тегов, используемой в JSP-странице. Синтаксис для импорта директивы `taglib` на JSP-странице:

```
<%@ taglib uri="tag_lib_URI" prefix="prefix" %>
```

Директива `taglib` использует два атрибута, описанных в таблице 20.

Таблица 20.

Атрибут	Описание
<code>uri</code>	Определяет местонахождение файла TLD нестандартного тега.
<code>prefix</code>	Определяет строку префикса, используемую для различения экземпляра нестандартного тега.

В настоящем пособии не рассматриваются технологии создания нестандартных тегов.

Кодирование элементов сценария JSP

Как мы уже рассмотрели, элементы сценария JSP позволяют вставлять код Java непосредственно в HTML-страницу с использованием тегов Объявление, Выражение и Скриптлет. В дополнение к реализации таких возможностей следует отметить, что можно использовать синтаксис в формате XML для записи утверждений JSP, который представлен в таблице 21.

Таблица 21.

Тип	Тег JSP	Тег JSP XML
Директива <code>page</code>	<code><% page ... %></code>	<code><jsp:directive.page ... /></code>
Директива <code>include</code>	<code><%@ include ... %></code>	<code><jsp:directive.include ... /></code>
Директива <code>taglib</code>	<code><%@ taglib ... %></code>	<code><jsp:root xmlns:prefix="taglibURI"> ... </jsp:root></code>
Объявление	<code><%! declaration %></code>	<code><jsp:declaration> declaration </jsp:declaration></code>
Выражение	<code><%= expression %></code>	<code><jsp:expression> expression </jsp:expression></code>
Скриплеты	<code><% scriptlet_code %></code>	<code><jsp:scriptlet> scriptlet_code </jsp:scriptlet></code>

Неявные объекты JSP

В сервлетах в процессе обработки запросов явно создаются объекты ServletContext, ServletConfig, Throwable, HttpSession, PageContext и JspWriter, но в страницах JSP эти объекты можно использовать неявно. Например, для отслеживания сессии пользователя в JSP странице используется неявный объект session без необходимости создания экземпляра интерфейса HttpSession. Также можно использовать метод setAttribute() для установки значения переменной в сессии, и метод getAttribute() для получения значения переменной сессии. Синтаксис для установки пользовательского имени в сессии:

```
session.setAttribute("userName", userName);
```

а для получения значения имени пользователя из сеанса:

```
session.getAttribute("userName", userName);
```

Таким образом, объекты в JSP могут быть созданы неявно с помощью директив или с помощью стандартных действий, или явно путем объявления их внутри скриплетов. Неявные объекты JSP – это набор предопределенных переменных, которые могут быть включены в выражения JSP и скриплеты. Неявные объекты JSP реализуются классами сервлетов и интерфейсов, которые описываются в таблице 22 с указанием соответствующих классов.

Таблица 22.

Неявный объект	Класс	Описание
application	javax.servlet.ServletContext	Объект application описывает веб-приложение.
config	javax.Servlet.ServletConfig	Представляет объект класса ServletConfig.
exception	java.lang.Throwable	Представляет исключение Throwable на JSP-странице.
out	javax.servlet.jsp.JspWriter	Представляет объект JspWriter для отправки ответа клиенту. JspWriter расширяет класс PrintWriter и используется JSP-страницами для отправки ответов клиентам.
page	java.lang.Object	Представляет текущий экземпляр JSP-страницы, который используется для ссылки на текущий экземпляр сгенерированного сервлета.
session	javax.servlet.http.HttpSessi	Представляет объект сессии интерфей-

Неявный объект	Класс	Описание
	<code>on</code>	с <code>HttpSession</code> .
<code>response</code>	<code>javax.servlet.http.HttpServletResponse</code>	Представляет объект <code>response</code> класса <code>HttpServletResponse</code> , который используется для отправки результата HTML клиенту.
<code>request</code>	<code>javax.servlet.http.HttpServletRequest</code>	Представляет объект <code>request</code> класса <code>HttpServletRequest</code> . Он используется для получения данных, представленных вместе с запросом.
<code>pageContext</code>	<code>javax.servlet.jsp.PageContext</code>	Представляет контекст JSP-страницы.

Неявные объекты JSP `request` и `response`, как мы уже убедились в предыдущих разделах, могут применяться для получения запроса от клиента и отправки ему ответа. В дальнейшем будет рассмотрено применение других неявных объектов.

Действия JSP

Действия JSP используются для выполнения задач для вставки файлов, использования bean-компонентов, передачи запроса другой странице и создания объектов. Синтаксис использования действий JSP в JSP-странице: `<jsp:attribute>`.

При использовании действий JSP необходимо задавать значение атрибута. Таблица 23 представляет различные теги действий JSP вместе с описанием поддерживаемых атрибутов:

Таблица 23.

Действие JSP	Описание	Атрибут	Описание атрибута
<code><jsp:forward></code>	Используется для передачи запроса целевой странице.	<code>page</code>	Задает URL целевой страницы.
<code><jsp:include></code>	Подключает файл в текущую JSP-страницу.	<code>page</code> <code>flush</code>	Задает URL включаемого ресурса. Задает, должен ли освобождаться буфер или нет. Значение <code>flush</code> может быть <code>true</code> или <code>false</code> .

Действие JSP	Описание	Атрибут	Описание атрибута
<code><jsp:param></code>	Определяет параметр, передаваемый включенной или перенаправленной странице.	<code>name</code> <code>value</code>	Определяет имя ссылочного параметра. Определяет значение указанного параметра.
<code><jsp:plugin></code>	Выполняет апплеты Java или JavaBean.	<code>type</code> <code>code</code> <code>codebase</code>	Определяет тип включаемого плагина. Определяет имя выполняемого плагином класса. Определяет путь к классу.

Действие `<jsp:include>`

Действие JSP `<jsp:include>` используется для включения результата работы одной JSP-страницы в другую JSP-страницу. Также можно добавить файл в текущую JSP-страницу. Следует указать путь к файлу и значение параметра `flush` в теге `<jsp:include>`. Например, чтобы вывести системную дату на веб-браузер, можно включить вывод другой JSP-страницы, которая имеет доступ к системной дате. Следующие два файла используются для вывода системной даты.

`IncludeDatePage.jsp`: Использует действие `<jsp:include>`, чтобы включить результат работы файла `JSP Date_JSP.jsp`.

`Date_JSP.jsp`: Использует выражение JSP для отображения текущей системной даты.

Чтобы запустить это приложение введите `http://localhost:8080/JSP/IncludeDatePage.jsp` в веб-браузере. Следующий код демонстрирует содержимое JSP-страницы `IncludeDatePage.jsp`, которая содержит действие JSP `<jsp:include>` для включения вывода JSP-страницы `DatePage`:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
```

```

<!-- Использование действия <jsp: include --> -->
<h1> Сегодня: <jsp:include page="DateJSP.jsp" flush="true"/></h1>
<%
    /* Вывод сообщения пользователю */
    out.println("<h3> Вывод файла DatePage.jsp показан выше</h3>");
%>
</body>
</html>

```

Действие JSP `<jsp:include page="Date_JSP.jsp" flush="true"/>`, данное на JSP-странице IncludeDatePage, вызывает JSP-страницу DatePage, которая выводит системную дату. Следующий код показывает содержание файла JSP DatePage:

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ page import="java.util.Calendar"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
    <!-- Это содержимое JSP, которое выводит серверное время, используя
    класс Calendar пакета java.util.-->
    <%
        Calendar calendar = Calendar.getInstance();
    %>
    <%=calendar.get(Calendar.DAY_OF_MONTH)%>/<%=calendar.get(Calendar.
MONTH)+1%>/<%=calendar.get(Calendar.YEAR)%>
    <H1>Этот код JSP для показа серверного времени</H1>
    <H1>Текущее время </H1>
    <H2>
        <%=calendar.get(Calendar.HOUR_OF_DAY)%>:<%=calendar.get(Calendar.M
INUTE)%>:<%=calendar.get(Calendar.SECOND)%>
    </H2>
</body>
</html>

```

Рис. 77 показывает результат работы приложения JSP IncludeDatePage.jsp.

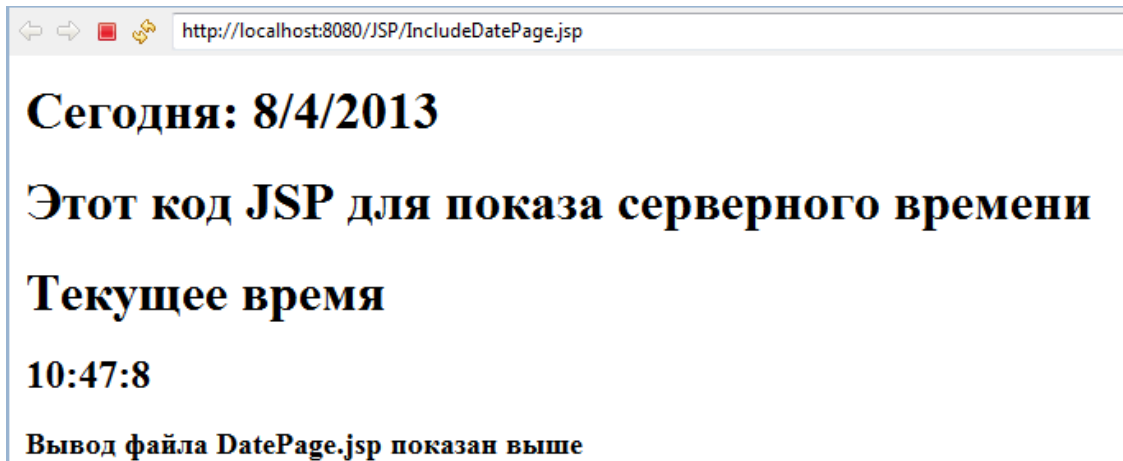


Рис. 77. Результат работы JSP-страницы IncludeDatePage.jsp.

Программирование JSP

Для успешного создания программ на JSP следует понять использование различных методов, определенных в классах и интерфейсах JSP API. Кроме того, необходимо знать различные компоненты JSP (действия JSP, директивы JSP и сценарии JSP), которые рассмотрены ранее. Классы, определенные в JSP API, содержат методы, которые доступны из JSP-страниц через использование неявных объектов.

Классы JSP API

JSP API – это набор классов и интерфейсов, которые можно использовать при создании JSP-страниц. Эти классы и интерфейсы хранятся в пакете `javax.servlet.jsp`, который определяет связь между JSP-страницей и ее средой времени выполнения. Ниже представлены некоторые из классов, определенных в пакете `javax.servlet.jsp`:

- `ErrorData`
- `JspWriter`
- `PageContext`

Класс `ErrorData`

Класс `ErrorData` содержит информацию об ошибках для страниц ошибок. Необходимо установить значение директивы `page` с атрибутом `isErrorPage` в значение `true` для указания того, что страница является страницей ошибок. Класс `ErrorData` расширяет класс `java.lang.Object`. Ниже рассмотрен ряд методов, определенных в классе `ErrorData` и используемых в JSP-странице:

`getRequestURL()`: Возвращает запрашиваемый URL типа `String`.

`setServletName()`: Возвращает имя вызываемого сервлета типа `String`.

`getStatusCode()`: Возвращает код состояния ошибки типа `int`.

`getThrowable()`: Возвращает исключение `Throwable`, которое вызвало ошибку.

Класс `JspWriter`

Класс `JspWriter` используется для записи действий и шаблонных данных на JSP-странице. Переменная `out` неявно ссылается на объект класса `JspWriter`. Класс `JspWriter` расширяет класс `java.io.Writer`. Ниже представлен ряд методов, определенных в классе `JspWriter`, которые можно использовать в JSP-странице:

`clear()`: Удаляет содержимое буфера. Метод `clear()` генерирует исключение `IOException`, если буфер уже пустой.

`close()`: Закрывает и очищает поток.

`flush()`: Очищает содержимое буфера. Метод `flush()` очищает все буферы в цепи `Writers` и `OutputStreams`. Он генерирует исключение `java.io.IOException`, если сделан вызов метода `write()` или `flush()` после закрытия потока.

`getBufferSize()`: Возвращает размер буфера, используемый `JspWriter`.

`print()`: Выводит значение логического, целого, символьного, длинного целого, с плавающей точкой, с плавающей точкой двойной точности числовых типов, а также массивы символов, строки и объекты. Метод `print()` генерирует исключение `java.io.IOException`, если во время печати возникает ошибка.

`println()`: Выводит значение логического, целого, символьного, длинного целого, с плавающей точкой, с плавающей точкой двойной точности числовых типов, а также массивы символов, строки и объекты. Этот метод выводит разделитель строк после завершения текущей строки. Метод `println()` генерирует исключение `java.io.IOException`, если во время печати возникает ошибка.

Класс `PageContext`

Класс `PageContext` предоставляет информацию о контексте, когда технология JSP используется в среде сервлетов. Класс `PageContext` расширяет класс `JspContext`. Экземпляр `PageContext` предоставляет доступ к пространствам имен, связанным с JSP-страницей. Ниже представлены некоторые из методов, определенные в классе `PageContext`:

`forward()`: Перенаправляет текущий запрос и ответ сервлета другой странице. Этот метод получает URL целевой страницы как аргумент.

`getPage()`: Возвращает текущее значение объекта страницы.

`getRequest()`: Возвращает текущее значение объекта запроса. Возвращаемым типом метода `getRequest()` является запрос сервлета.

`getResponse()`: Возвращает текущее значение объекта ответа. Возвращаемым типом метода `getResponse()` является ответ сервлета.

`getServletConfig()`: Возвращает объект `ServletConfig` текущей страницы.

`getServletContext()`: Возвращает объект `ServletContext` текущей страницы.

`getSession()`: Возвращает объект `HttpSession` для текущего объекта `PageContext`.

`include()`: Обрабатывает текущий запрос сервлета и вставляет ресурс, указанный в URL. Метод `include()` принимает два аргумента: путь к URL типа `string` и `flush` типа `Boolean`.

Этапы создания приложения JSP

Различные методы, определенные в классах JSP API, могут быть использованы для создания приложения JSP, которое может состоять из файлов HTML, JavaBean или JSP. При создании приложения JSP обычно необходимо выполнить следующие шаги.

1. Создать пользовательский интерфейс. Пользовательский интерфейс используется для принятия входных данных от пользователя, которые передаются как запрос JSP-странице.
2. Создать JSP-страницу. JSP-страница предоставляет бизнес-логику для обработки запросов, переданных из интерфейса HTML. Также JSP-страница отправляет ответ веб-браузеру клиента.
3. Упаковать и развернуть компоненты приложения (пользовательский интерфейс и JSP-страницу).

4. Протестировать работу созданного приложения JSP, используя веб-браузер.

Постановка задачи

Технический директор фирмы, поручил создать приложение, которое проверяет id и пароль каждого клиента перед получением доступа к данным его учетной записи. Идентификатор клиента (id) должен быть в числовом виде. Также необходимо выводить сообщение об ошибке, если введенные клиентом id или password некорректны. Перед применением данной функциональности во всем приложении, необходимо проверить его работу, сделав прототип для конкретного клиента. Программисту необходимо реализовать функциональность прототипа с использованием JSP.

Решение

Для выполнения поставленной задачи, необходимо выполнить следующие шаги.

1. Создать входную страницу, используя HTML.
2. Создать страницу аутентификации, используя JSP.
3. Создать страницу ошибок, используя JSP.
4. Упаковать приложение JSP.
5. Развернуть приложение JSP.
6. Выполнить приложение JSP.

1. Создание входной страницы с использованием HTML

Можно создать входную страницу онлайн-банковской системы, используя HTML. Эта входная страница принимает id и password от клиента банка. При нажатии клиентом кнопки Login, вызывается JSP-страница AuthenticateCustomer для проверки id и password клиента. Следующий код демонстрирует содержимое HTML-страницы LoginPage, принимающая id и password клиента и устанавливающая имя JSP-страницы, которая должна вызываться для аутентификации данных клиента:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
```

```

        <form name="f1" action="AuthenticateCustomer.jsp">
            <font size=4 face="Verdana" color=#120292> <marquee>
                Online Banking System </marquee> <br> <br>
                <table cellspacing=5 cellpadding=5 bgcolor=#959999
                    colspan=2
                        rowspan=2 align="center">
                            <tr>
                                <td>Bank Customer Authentication
                            </td>
                            </tr>
                            <tr>
                                <td>Enter Customer Id :</td>
                                <td><input type=text name="uname"></td>
                            </tr>
                            <tr>
                                <td>Enter Password:</td>
                                <td><input type=password
                                    name="password"></td>
                            </tr>
                        </table> <br>
                        <table align="center">
                            <tr>
                                <td><input type="submit" value=" Login
                                </td>
                                <td><input type="Reset" value=" Cancel
                                </td>
                            </tr>
                        </table>
                    </font>
                </form>
            </body>
        </html>

```

При нажатии на кнопку Login, данные клиента передаются JSP-странице AuthenticateCustomer. При нажатии на кнопку Cancel, поля на входной странице устанавливаются в исходное состояние.

2. Создание страницы аутентификации, используя JSP

Входная страница, которая создана на предыдущем этапе, отправляет id и пароль клиента JSP-странице AuthenticateCustomer для аутентификации. Клиент банка перенаправляется на страницу ошибок, если введенный клиентом id является строкой. Следующий код представляет содержимое JSP-страницы AuthenticateCustomer, которая проверяет введенные клиентом id и password:

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>

```

```

</head>
<body>
  <!-- Задание имени страницы ошибок -->
  <%@ page errorPage="LoginErrorPage.jsp"%>
  <font face="verdana"> <%
  /* Получение значения текстового поля uname */
  String user = request.getParameter("uname");
  /* Преобразование строкового значения в целое значение */
  int customerID = Integer.parseInt(user);
  /* Получение значения текстового поля password */
  String pass = request.getParameter("password");
  /* Проверка ID и пароля клиента */
  if (customerID == 1010 && pass.equals("Customer")) // Такой Log-
in/Password вводится
  {
    out.println("Welcome to Online Banking System");
  %> <br> <br> <%
  out.println("Login Successful");
  } else {
    out.println("Login Unsuccessful");
  }
  %>
  </font>

</body>
</html>

```

В предложенном коде метод `getParameter()` сохраняет `id` и `password` клиента в переменных `user` и `pass`. Эти два значения сравниваются с predetermined значениями клиента `id` равным 1010 и `password` равным `Customer`. При успешном сравнении `id` и `password` клиента выводится сообщение “Login Successful”. Если введенные клиентом `id` или `password` не верны, то JSP-страница генерирует исключение, и вызывается JSP страница ошибок `LoginErrorPage`.

3. Создание страницы ошибок, используя JSP

Можно создать страницу ошибок для вывода сообщения с ошибкой пользователю о неудачной аутентификации. Вам нужно установить значение атрибута `page isErrorPage` в `true`. Следующий код демонстрирует содержимое JSP-страницы `LoginErrorPage`, которая выводит класс исключения, сообщение исключения и сообщение об ошибке:

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
  pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>

```

```

<h3> An exception has occurred</h3>
  <%@ page isErrorPage="true" %>
  <table>
    <tr>
      <td>Exception Class:</td>
      <!-- Получение имени класса исключения, где 'exception' является неявным объектом, предоставляемым JSP --%>
      <td><%= exception.getClass() %></td>
    </tr>
    <tr>
      <td>Message:</td>
      <!-- Получение сообщения исключения, где 'exception' является неявным объектом, предоставляемым JSP --%>
      <td><%= exception.getMessage() %></td>
    </tr>
  </table>
  <br>
  To go back to the login page click Login Page button
  <form name="f2" action="LoginPage.html">
  <input type="submit" name="button1" value="Login Page">
  </form>
</body>
</html>

```

В предложенном коде выводится сообщение с ошибкой “An exception has occurred”. При нажатии пользователем на кнопку Login Page, обработка перенаправляется на страницу HTML LoginPage.

4/5. Упаковка JavaEE приложения

Страницы LoginPage, AuthenticateCustomer и LoginErrorPage упаковываются и развертываются на сервере приложений с применением IDE Eclipse автоматически.

6. Запуск JavaEE приложения

Теперь можно запустить приложение, которое автоматически развернуто на сервере GalassFish, выполняя следующие шаги:

1. Откройте веб-браузер, который является внутренним для IDE Eclipse или любой другой, установленный на вашем компьютере.
2. Введите `http://localhost:8080/JSP/LoginPage.html` в адресной строке веб-браузера, чтобы открыть HTML-страницу LoginPage в веб-браузере, как показано на Рис. 78:

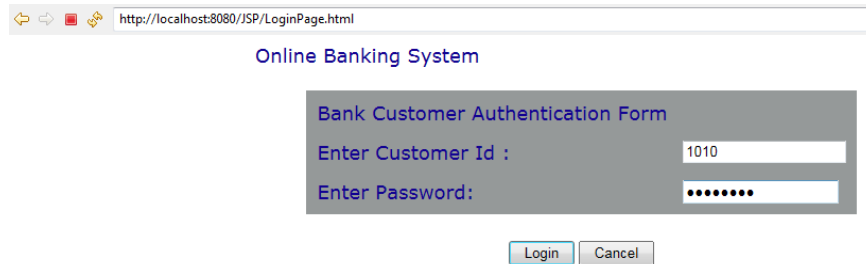


Рис. 78. Результат работы HTML-страницы LoginPage.

3. Ведите id и пароль клиента, 1010 и Customer, в полях Enter Customer Id и Enter Password соответственно, и нажмите кнопку Login, чтобы вызвать JSP-страницу AuthenticateCustomer. На экране отображается сообщение “Login Successful”, если введенные id и password клиента верны, как показано на Рис. 79.

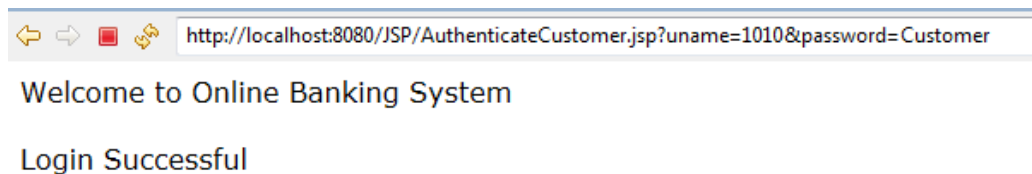


Рис. 79. Результат работы JSP-страницы AuthenticateCustomer.

Если введенные клиентом id или пароль неверны, будет выведено сообщение об ошибке, как показано на Рис. 80.

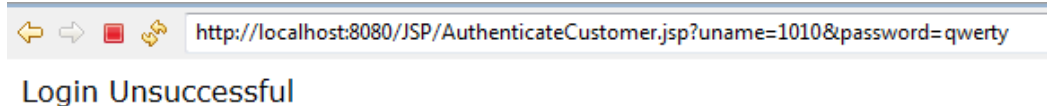


Рис. 80. Результат работы JSP-страницы AuthenticateCustomer

JSP-страница LoginPage вызывается, когда введенный клиентом id содержит нечисловые символы и возникает исключение на JSP-странице AuthenticateCustomer и Рис. 81 показывает сообщения, выведенные JSP-страницей LoginPage:

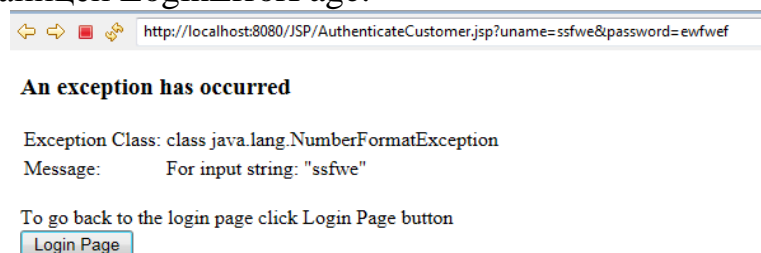


Рис. 81. Результат работы JSP-страницы LoginPage

При нажатии пользователем на кнопку Login Page, обработка перенаправляется на HTML-страницу LoginPage.

Понятие и работа с JavaBeans

Компоненты JavaBeans представляют собой многократно используемые классы Java, позволяющие разработчикам существенно ускорить процесс разработки WEB-приложений путем их сборки из программных компонентов. JavaBeans, как и другие компонентные технологии, используют концепцию сборки приложений из компонентов, при которой разработчик должен знать только сервисы компонентов, а детали реализации компонентов не играют никакой роли.

Компоненты JavaBean – это объекты, используемые для инкапсуляции в одном объекте сложного кода и данные. Компонент JavaBean может иметь свойства, методы и события, открытые для удаленного доступа. Компонент JavaBean представляет собой java-класс, удовлетворяющий определенным соглашениям о наименовании методов и экспортируемых событиях. Одним из важных понятий технологии JavaBeans является внешний интерфейс компонента - properties (свойства). Property – это пара методов класса (getter и setter) для доступных пользователю свойств, обеспечивающих информацию о внутреннем состоянии компонента JavaBean. Когда используется класс, который представляет собой JavaBean (или Bean), можно использовать для работы с ним специальные теги JSP. Рассмотрим, как определяется JavaBean, а также три типа JSP тегов, которые можно использовать для создания Bean и обработки его свойств.

Ниже представлен код класса Employee, который был рассмотрен ранее, и является JavaBean, поскольку удовлетворяет трем обязательным требованиям.

```
package ru.ifmo.javaee;
import java.io.Serializable;
public class Employee implements Serializable {
    private Long id;
    private String firstName;
    private String lastName;
    private String designation;
    private String phone;
    public Employee() {
        id = (long) 0;
        firstName = "";
        lastName = "";
        designation = "";
        phone = "";
    }
    public Employee(Long id, String firstName, String lastName,
        String designation, String phone) {
        super();
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.designation = designation;
        this.phone = phone;
    }
}
```

```

    }
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getDesignation() {
        return designation;
    }
    public void setDesignation(String designation) {
        this.designation = designation;
    }
    public String getPhone() {
        return phone;
    }
    public void setPhone(String phone) {
        this.phone = phone;
    }
}

```

Класс `Employee` удовлетворяет следующим трем правилам `JavaBeans`. Во-первых, `JavaBean` должен содержать конструктор без аргументов. В коде, в частности, конструктор без аргументов присваивает начальные значения пяти объявленным переменным экземпляра. В результате работы конструктора по умолчанию вновь созданный объект `Employee` для четырех переменных экземпляра класса содержит пустые строки и для одной переменной - 0. Также возможны другие значения, которые могут оказаться полезными разработчику при инициализации объекта.

Во-вторых, в `JavaBean` не могут объявляться переменные типа `public`, например, в рассмотренном коде все переменные объявлены как `private`. Однако можно объявлять переменные со спецификатором доступа `protected`.

В-третьих, `JavaBean` должен содержать методы `get` и `set` для всех свойств, которые должны быть доступны для `JSP` страницы. В нашем коде, например, методы обеспечивают доступ ко всем переменным эк-

земпляра класса Employee. В результате, класс Employee квалифицируется как JavaBean.

Для обеспечения доступа к свойствам типа Boolean кодируются методы is и set, вместо методов get и set. Например, можно написать методы с именем isEmailUpdated и setEmailUpdated для обеспечения доступа к Boolean свойству emailUpdated.

При кодировании методов get, set и is, необходимо следовать правилам применения больших букв в обозначениях имен, используемых в коде JavaBean. Иными словами, каждое имя метода должно начинаться со строчной буквы и каждое имя свойства должно начинаться с заглавной буквы. Например, setFirstName метод использует строчную букву s в начале имени метода и прописную букву F в начале имени свойства.

В дополнение к упомянутым обязательным трем правилам, JavaBean должен реализовать интерфейс Serializable. Интерфейс Serializable является интерфейсом в пакете java.io, который указывает, что класс содержит get, set и is методы, которые любой другой класс может использовать для чтения и записи переменных объекта от источника данных. В предыдущем коде, например, класс Employee реализует интерфейс Serializable и содержит все необходимые методы получения и установки свойств. В результате, некоторые сервлеты могут сохранять и восстанавливать этот объект, если это необходимо. Например, веб-контейнер может сохранить состояние объекта Employee, перед завершением работы и восстанавливать состояние объекта Employee, когда сервер запускается в следующий раз.

При кодировании веб-приложений, как правило, JavaBeans используется для определения бизнес-объектов приложения, которые можно назвать также невидимыми компонентами, поскольку они не предназначены для вывода. В этом разделе мы будем рассматривать именно такой тип JavaBeans.

Необходимо однако учитывать, что JavaBeans могут реализовать значительно большую функциональность, чем просто определять бизнес-объекты. Например, JavaBeans могут быть использованы для определения кнопок и других элементов управления пользовательского интерфейса.

Также необходимо иметь в виду, что есть еще один тип JavaBeans, который называется Enterprise JavaBean (EJB). Несмотря на то, EJB, в некотором смысле аналогичны JavaBeans, классы EJB являются более сложными и трудными для разработки, чем JavaBeans. Для более детального знакомства с EJB необходимо обращаться к специальной литературе.

Для обращения к компонентам JavaBeans на странице JSP необходимо использовать следующее описание тега:

```
<jsp:useBean id="BeanID" [scope="page | request  
| session | application"] class="BeanClass" />
```

Тег `jsp:useBean` позволяет ассоциировать экземпляр Java-класса, определенный в данном диапазоне видимости `scope`, с заданным внутренним идентификатором этого класса `id` для данной странице JSP. Прежде, чем использовать свойства компонента JavaBean (`setProperty` и `getProperty`), необходимо объявить тег `jsp:useBean`. При выполнении тега `jsp:useBean` сервер приложений обеспечивает поиск (lookup) экземпляра данного Java-класса, используя значения, определенные в атрибутах:

- `id` - идентификатор экземпляра класса внутри страницы JSP;
- `scope` - диапазон видимости (`page`, `request`, `session`, `application`).

`BeanID` определяет имя компонента JavaBean, являющееся уникальным в области видимости, заданной атрибутом `scope`. По умолчанию принимается область видимости `scope="page"`, т.е. текущая страница JSP.

Значение `scope` кодируется из диапазона `page | request | session | application`, который определяется следующим образом.

Page (страница). Объект, определенный с диапазоном видимости `page`, доступен до тех пор, пока не будет отправлен ответ клиенту или пока запрос к текущей странице JSP не будет перенаправлен другому ресурсу. Ссылки на объект возможны только в пределах страницы, в которой этот объект определен. Объекты, объявленные с атрибутом `page`, сохраняются в объекте `pageContext`.

Request (запрос). Объект, имеющий диапазон видимости `request`, существует и доступен в течение текущего запроса, и остается видимым, даже если запрос перенаправляется другому ресурсу в том же самом цикле выполнения. Объекты, объявленные с атрибутом `request`, сохраняются в объекте `request`.

Session (сессия). Объект, имеющий диапазон видимости `session` доступен в течение текущей сессии, если страница JSP "знает" о сессии.

Application (приложение). Объект, имеющий диапазон видимости `application` доступен страницам, обрабатывающим запросы в одном и том же приложении Web, и существует до тех пор, пока сервер приложений поддерживает объект `ServletContext`. Объекты, объявленные с атрибутом области видимости `application`, сохраняются в объекте `application`.

Обязательный атрибут класса компонента `"class"` может быть определен следующим способом:

```
class="имя класса" [type="полное имя суперкласса"]
```

Свойство компонента `JavaBean` с именем `myBean` устанавливается тегом:

```
<jsp:setProperty name="myBean" property="Имя свойства" value="Строка или выражение JSP" />
```

Для чтения свойства компонента `JavaBean` с именем `myBean` используется тег:

```
<jsp:getProperty name="myBean" property="Имя свойства" />
```

Рассмотрим примеры кодирования применительно к `JavaBean Employee`, представленные в коде ниже:

```
<jsp:useBean id="user" scope="session" class="ru.ifmo.javaee.Employee" />


|                                                                                                        |               |                                                      |
|--------------------------------------------------------------------------------------------------------|---------------|------------------------------------------------------|
| <td align="right">First name:</td> <td>&lt;jsp:getProperty name="user" property="firstName" /&gt;</td> | First name:   | <jsp:getProperty name="user" property="firstName" /> |
| <td align="right">Last name:</td> <td>&lt;jsp:getProperty name="user" property="lastName" /&gt;</td>   | Last name:    | <jsp:getProperty name="user" property="lastName" />  |
| <td align="right">Phone number:</td> <td>&lt;jsp:getProperty name="user" property="phone" /&gt;</td>   | Phone number: | <jsp:getProperty name="user" property="phone" />     |


```

В представленном фрагменте кода тег `useBean` объявляет доступ к `JavaBean Employee`. При этом, если объект `Employee` с таким именем уже существует внутри объекта `session`, то осуществляется доступ к этому объекту, в противном случае, создается объект с таким именем путем вызова конструктора класса `Employee` без аргументов, и свойствам объекта присваиваются начальные значения. Затем трижды тег `getProperty` показывает значения свойств, которые установлены в `bean`. Аналогичным образом можно было бы использовать тег `setProperty`.

Следующий фрагмент кода показывает те же действия JSP, но без использования тегов JSP для доступа к `bean`:

```
<%@ page import="ru.ifmo.javaee.Employee"%>
<%
Employee user = (Employee) session.getAttribute("user");
if (user == null) {
    user = new Employee();
}
%>
```

```

<table cellpadding="5" cellspacing="5" border="1">
  <tr>
    <td align="right">First name:</td>
    <td> <%= user.getFirstName() %> </td>
  </tr>
  <tr>
    <td align="right">Last name:</td>
    <td>
      <td><%= user.getLastName() %></td>
    </td>
  </tr>
  <tr>
    <td align="right">Phone number:</td>
    <td>
      <td>
      <td><%= user.getPhone() %></td>
    </td>
  </tr>
</table>

```

В этом фрагменте кода показан JSP, выполняющий ту же самую работу, но с использованием скриплетов и выражений. Поскольку этот код совмещает коды Java и HTML, не трудно убедиться, что при использовании JavaBeans вместо тегов JSP создание страниц JSP и их поддержка будут выполняться легче. Фактически JSP с использованием тегов useBean не требует применения кода Java, что упрощает разработку JSP непрограммистами.

Разработка JSP-страницы

Проект будет содержать всего одну страницу index.jsp, которую мы добавим к ранее созданному проекту Servlet, и она будет использоваться как для ввода фамилии, так и для отображения результатов поиска.

- Для создания новой JSP-страницы, нажмите правой кнопкой мыши на проект Servlet и выберите пункт меню New/Other, затем выберите Web/JSP и нажмите Next, как показано на Рис. 82.

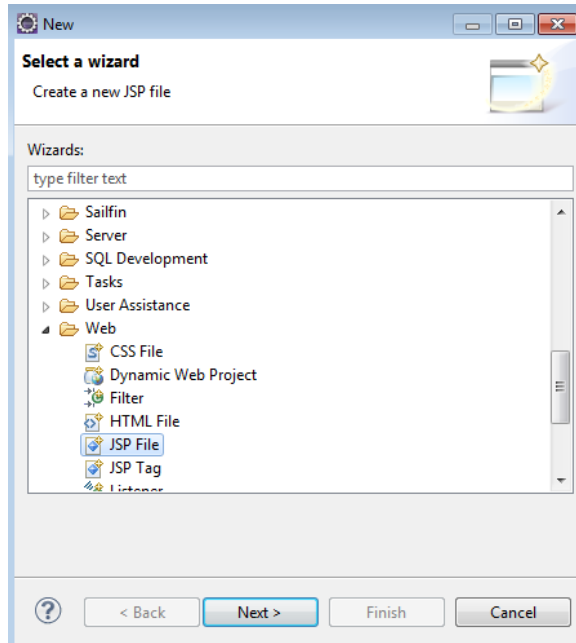


Рис. 82. Диалоговое окно создания файла JSP.

- Укажите имя файла `find.jsp` и убедитесь, что в дереве структуры проекта выбран каталог `WebContent`. Нажмите `Finish` (Рис. 83.).

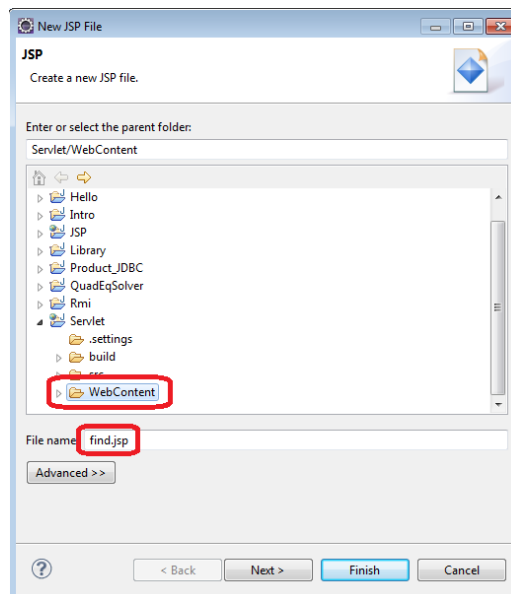


Рис. 83. Диалоговое окно определения имени файла JSP.

- Созданный JSP-файл уже содержит базовый набор тэгов заголовка и тела веб-страницы. Нам необходимо добавить следующие элементы кода:
 - поменять кодировку страницы на UTF-8 для корректного отображения символов кириллицы, если она уже не установлена;
 - поменять заголовок страницы на «Поиск сотрудников»;

- создать форму, включающую в себя текстовое поле lastname для ввода фамилии и кнопку подтверждения (Submit). При выполнении подтверждения форма передает значение параметра lastname сервлету EmployeeServlet;
- сигналом того, что поиск был выполнен и имеются результаты для отображения, будет являться наличие параметра http-запроса employeesFound, который будет передан сервлетом странице find.jsp в случае успешного выполнения поиска. Далее, проверив размер коллекции найденных сотрудников, мы определим, был ли найден хотя бы один сотрудник. В случае утвердительного ответа обработаем коллекцию, и отобразим все свойства каждого найденного сотрудника в виде таблицы.

Ниже приведен полный код JSP-страницы:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@page import="java.util.ArrayList"%>
<%@page import="ru.ifmo.javaee.Employee"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Поиск сотрудников</title>
</head>
<body>
    <form action="EmployeeServlet">
        Фамилия сотрудника <input type="text" name="lastname"> <input
put
        type="submit" value="наиск">
    </form>
    <%
        // Получение значения параметра employeesFound
        ArrayList<Employee> employees = (ArrayList) request
            .getAttribute("employeesFound");
        // Если параметр задан, начинаем обработку
        if (employees != null) {
            // Если не найдено ни одного сотрудника - вывод сооб-
щения
            if (employees.size() == 0)
                out.print("Сотрудники не найдены");
            else {
                out.print("<TABLE border=\"1\">");
                // Заголовок таблицы
                out.print("<TR><TD>Id</TD><TD>Имя</TD><TD>Фамилия</TD>"
                    +
                "<TD>Должность</TD><TD>Телефон</TD></TR>");
                for (int i = 0; i < employees.size(); i++) {
                    // По каждому найденному сотруднику
                    // формируется строка таблицы
                    out.print("<TR>");
```

```

коллекции
ees.get(i);
сотрудника
"</TD>");
"</TD>");
"</TD>");
+ "</TD>");
"</TD>");

// Получение очередного сотрудника из
Employee emp = (Employee) employ-
// Заполнение строки таблицы свойствами
out.print("<TD>" + emp.getId() +
out.print("<TD>" + emp.getFirstName() +
out.print("<TD>" + emp.getLastName() +
out.print("<TD>" + emp.getDesignation()
out.print("<TD>" + emp.getPhone() +
out.print("</TR>");
}
out.print("</TABLE>");
}
}
}
%>
</body>
</html>

```

Доработка сервлета

Метод сервлета `doGet()` выполняется после того как пользователь выполнил подтверждение (Submit) формы. Необходимо создать код сервлета `EmployeesFound`, изменив код метода `doGet()` сервлета `EmployeeServlet` таким образом, чтобы результаты поиска отправлялись jsp-странице в виде параметра `EmployeesFound`. Для этого откроем исходный сервлет для редактирования, прокомментируем часть кода, связанную с выводом коллекции сотрудников в поток вывода (`response.getWriter()`), и добавим код помещения этой коллекции в параметр запроса и перенаправление запроса к странице `find.jsp`:

```

// Выводим информацию о найденных сотрудниках
/*
// Выводим информацию о найденных сотрудниках
PrintWriter out = response.getWriter();
out.println("Найденные сотрудники<br>");
for (Employee emp: employees) {
    out.print(emp.getFirstName() + " " +
        emp.getLastName() + " " +
        emp.getDesignation() + " " +
        emp.getPhone() + "<br>");
}
*/

// Помещение результатов в параметр запроса employeesFound
request.setAttribute("employeesFound", employees);
// Перенаправление http-запроса на страницу find.jsp

```

```
RequestDispatcher dispatcher =
getServletContext().getRequestDispatcher("/find.jsp");
dispatcher.forward(request, response);
```

Упаковка и развертывание приложения

Упаковка и развертывание выполняются автоматически.

Тестирование приложения

Запустите браузер и перейдите по адресу `http://localhost:8080/Servlet/Find.jsp`.

Введите фамилию сотрудника, например "Ivanov" и нажмите «Поиск». В результате на странице отображается информация о найденных сотрудниках, как показано на Рис. 84.

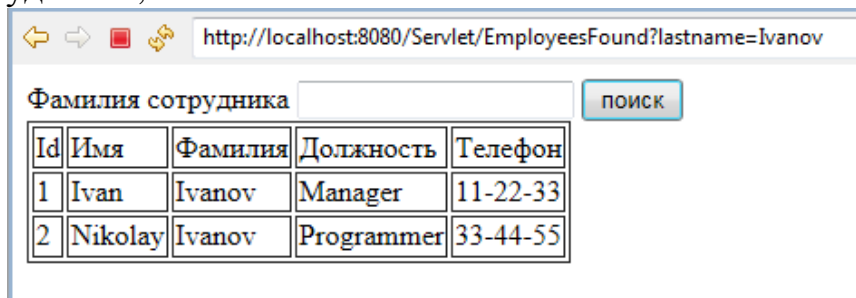


Рис. 84. Результат работы страницы Find.jsp.

Введение в JSP Expression Language (EL)

Expression Language (EL) - скриптовый язык выражений, который позволяет получить доступ к Java компонентам (JavaBeans) из страниц JSP. Начиная с JSP 2.0 язык EL используется внутри JSP тегов для отделения Java кода от тегов JSP с целью обеспечения упрощенного доступа к Java компонентам.

Появление EL упростило работу веб-дизайнеров, которые не имеют профессиональных навыков в программировании на языке Java и которые должны знать исключительно способ вызова соответствующих java-методов на языке EL.

EL обеспечивает компактный синтаксис, позволяющий получать данные из JavaBeans, map, array и list, которые хранятся в качестве атрибутов веб-приложения. Например, ниже представлено два фрагмента кода, в которых данные получаются от объекта User с именем user, сохраненный в качестве атрибута объекта session. В обоих представленных примерах предполагается, что класс User удовлетворяет всем правилам создания JavaBean. Если первый пример использует EL, для получения свойств User bean, то во втором примере используются стандартные теги JSP, для получения этих свойств.

Фрагмент кода с использованием EL:

```
<table cellpadding="5" cellspacing="5" border="1">
```



```

        <tr>
            <td align="right">First name:</td>
            <td>${user.firstName}</td>
        </tr>
        <tr>
            <td align="right">Last name:</td>
            <td>${user.lastName}</td>
        </tr>
        <tr>
            <td align="right">Email address:</td>
            <td>${user.emailAddress}</td>
        </tr>
    </table>

```

Фрагмент кода с использованием тегов JSP:

```

<jsp:useBean id="user" scope="session" class="business.User" />
<table cellspacing="5" cellpadding="5" border="1">
    <tr>
        <td align="right">First name:</td>
        <td><jsp:getProperty name="user" property="firstName" />
        <td>
    </tr>
    <tr>
        <td align="right">Last name:</td>
        <td><jsp:getProperty name="user" property="lastName" /></td>
    </tr>
    <tr>
        <td align="right">Email address:</td>
        <td><jsp:getProperty name="user" property="emailAddress" />
        <td>
    </tr>
</table>

```

Из примеров видно преимущества удобства кодирования, которые может предоставить EL для расширения возможностей JSP, в частности:

- EL имеет более элегантный и компактный синтаксис, чем стандартные теги JSP.
- EL позволяет получить доступ к вложенным свойствам.
- EL позволяет получить доступ к коллекциям Maps, Array и List.
- EL улучшает работу по обработке null значений.
- EL обеспечивает большую функциональность.

К недостаткам EL следует отнести:

- EL не обеспечивает создания объектов JavaBean, если они еще не существуют.
- EL не обеспечивает возможности установки свойств.

Ранее рассматривалось использование метода SetAttribute объектов HttpServletRequest и HttpSession для хранения объекта в области видимости запроса или сессии. Если необходимы более широкие диапазоны, то можно использовать метод SetAttribute объекта ServletContext для хранения объекта с областью видимости application. Или наоборот, если

необходим меньший диапазон, можно использовать `setAttribute` метод неявного объекта `PageContext` для хранения объектов в области видимости `page`. В дальнейшем, можно использовать метода `getAttribute` соответствующего объекта для извлечения атрибута.

Следующий фрагмент кода показывает, как использовать EL для доступа к атрибуту веб-приложения. При использовании EL кодирование начинается со знака доллара (`$`), за которым следует открывающая фигурная скобка (`{`) и закрывающая скобки (`}`), внутри скобок кодируется EL-выражение.

Следующий фрагмент кода показывает получения атрибута для простого объекта типа `String` или объекта типа `Date`:

Код сервлета:

```
Date currentDate = new Date();
request.setAttribute("currentDate", currentDate);
```

код JSP:

```
<p>The current date is ${currentDate}</p>
```

В коде сервлет создает объект типа `Date` с именем `CurrentDate`, который содержит текущую дату. Затем сервлет сохраняет этот объект в качестве атрибута объекта `request`, а код JSP использует EL для доступа к этому атрибуту, преобразования его в строку и показывает значение.

Следует отметить, что в этом фрагменте не определен диапазон видимости, и в этом случае EL автоматически просматривает все уровни диапазона, начиная с самого маленького (`page` диапазона), двигаясь по направлению к наибольшему диапазону (`application`).

Следующий фрагмент кода используется для представления свойства атрибута из более сложного объекта `JavaBean` или `map`:

Код сервлета:

```
User user = new User(firstName, lastName, emailAddress);
session.setAttribute("user", user);
```

Код JSP:

```
<p>Hello ${user.firstName}</p>
```

В коде сервлет создает объект `JavaBean` `user` и сохраняет его как атрибут `session`. После чего код JSP использует EL для доступа к этому атрибуту и использует оператор (`.`), чтобы специфицировать свойство, которое необходимо вывести. Аналогичным образом можно использовать этот подход для работы с `map`. В этом случае необходимо закодировать имя `key` после оператора (`.`) для доступа к объекту, сохраненному в объекте типа `map` (отображение).

Код сервлета:

```
ArrayList <User> users = UserIO.getUsers(path);
session.setAttribute("users", users);
```

Код JSP:

```
<p>The first address on our list is ${users[0].emailAddress}<br>
```

```
The second address on our list is ${users[1].emailAddress} </p>
```

или

```
<p>The first address on our list is ${users["0"].emailAddress}<br>
The second address on our list is ${users["1"].emailAddress} </p>
```

EL позволяет использовать оператор [] при работе с JavaBeans и maps, но чаще всего он используется для работы с массивами и списками. При этом используется синтаксис: `${attribute["propertyKeyOrIndex"]}`

Ниже представлен фрагмент кода по работе со свойствами JavaBean:

Код сервлета:

```
User user = new User ("John", "Smith", "jsmith@gitiail.com");
session.setAttribute("user", user);
```

Код JSP:

```
<p>Hello ${user["firstName"]}</p>
```

В представленном фрагменте кода с помощью оператора [] осуществляется доступ к свойству firstName атрибута с именем user. Аналогичный эффект доступа к свойству может быть достигнут иными средствами языка EL, например, `<p>Hello ${user.firstName}</p>`, однако, в приведенном фрагменте кодирование легче и нагляднее и поэтому оператор (.) обычно используется для доступа к свойствам JavaBeans и значениям maps.

Следующий фрагмент кода использует оператор [] для доступа к элементам массива строк:

Код сервлета:

```
String[] colors = {"Red", "Green", "Blue"};
ServletContext app = this.getServletContext();
app.setAttribute("colors", colors);
```

Код JSP:

```
<p>The first color is ${colors[0]}<br>
The second color is ${colors[1]}
</p>
```

В коде создается символьный массив colors с тремя значениями цвета. Затем код получает объект ServletContext и сохраняет массив в этом объекте, что позволяет получить доступ к нему из всего приложения. В заключение, код JSP использует EL для получения первых двух элементов, которые сохранены в массиве.

Следует заметить, что представленный код аналогичен синтаксису Java для доступа к элементам, сохраненным в массиве, и что значения индекса могут быть заключены в кавычки. Хотя кавычки необходимы для использования оператора [] для доступа к свойству JavaBean или ключа в map, они не являются обязательными для определения индекса массива или списка.

С помощью оператора (.) можно получать доступ к вложенным свойствам, используя синтаксис `${attribute.property1.property2}`.

Например, следующий фрагмент кода показывает наиболее общий способ доступа к вложенным свойствам:

Код сервлета:

```
Product p = new Product();
p.setCode("p001");
LineItem lineItem = new LineItem(p,10);
session.setAttribute("item", lineItem);
```

Код JSP:

```
<p>Product code: ${item.product.code}</p>
```

В представленном коде сервлет создает объект `p` класса `Product` и устанавливает для объекта значение свойства `code`. Затем код сервлета запоминает объект `p` в создаваемом объекте `lineItem` класса `LineItem`, и сохраняет объект `LineItem` в атрибуте сессии `item`.

Поскольку классы `LineItem` и `Product` удовлетворяют правилам `JavaBeans`, то код JSP может использовать EL для получения свойства `code` объекта `Product`, который хранится в атрибуте `item`. Хотя в примере показано программирование одного уровня вложенности свойств, не существует ограничений на количество уровней вложенности.

Следующий фрагмент кода показывает использование другого синтаксиса с оператором `(.)` после оператора `[]` - `${attribute["property1"].property2}`:

Код сервлета:

```
Product p = new Product();
p.setCode("pf01");
LineItem lineItem = new LineItem(p,10);
session.setAttribute("item", lineItem);
```

Код JSP:

```
<p>Product code: ${item["product"].code}</p>
```

Единственное отличие здесь состоит в том, что возвращаемый объект оператора `[]` должен быть `JavaBean` или `map`. В данном случае это условие выполняется, потому что оператор `[]` возвращает тот же `Product bean`, как и предыдущем фрагменте.

EL позволяет использовать несколько иные неявные объекты, с помощью которых можно выполнять другие задачи в JSP. Список этих объектов представлен в таблице 24.

Таблица 24.

Неявный объект	Описание
<code>param</code>	Представляет объект <code>map</code> , который возвращает значение для заданного имени параметра <code>request</code> .
<code>paramValues</code>	Представляет объект <code>map</code> , который возвращает массив значений для заданного имени параметра <code>request</code>
<code>header</code>	Представляет объект <code>map</code> , который возвращает зна-

Неявный объект	Описание
	чение для заданного заголовка HTTP request.
headerValues	Представляет объект map, который возвращает массив значений для заданного заголовка HTTP request.
cookie	Представляет объект map, который возвращает объект Cookie для заданного cookie.
initParam	Представляет объект map, который возвращает значение для заданного имени параметра в элементе context-param файла web.xml
pageContext	Представляет ссылку на неявный объект pageContext, который доступен из JSP.

Ниже представлен фрагмент кода, который использует неявные объекты для работы параметрами:

Код HTML

```
<form action="addToEmailList" method="post">
<p> First name: <input type="text" name="firstName"> </p>
<p> Email address 1: <input type="text" name="emailAddress"> </p>
<p> Email address 2: <input type="text" name="emailAddress"> </p>
</form>
```

Код JSP

```
<p>First name: ${param.firstName}<br>
Email address 1: ${paramValues.emailAddress[0]}<br>
Email address 1: ${paramValues.emailAddress[1]}<br>
</p>
```

Тег form, в представленном фрагменте кода HTML, определяет параметр firstName для ввода имени в текстовое поле и параметр emailAddress для следующих двух текстовых полей. Иными словами, форма позволяет ввести два электронных адреса в одно имя. В этом случае код JSP показывает, как можно использовать объект param для получения значения параметра firstName. Кроме того, код JSP показывает, как использовать объект paramValues для получения массива строк, который содержит значения для параметра emailAddress.

Ниже представлен фрагмент кода использования объекта header для получения данных из HTTP header.

Код JSP:

```
<p>Browser MIME types: ${header.accept}<br><br>
Browser compression types: ${header["accept-encoding"]}</p>
```

Вывод браузера представлен на Рис. 85:

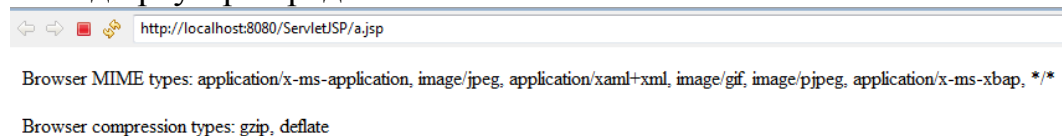


Рисунок 85 Вывод браузера.

В коде оператор (.) используется для получения значения любого заголовка запроса, который содержит единственное слово name. Например, можно использовать оператор (.) для получения значения заголовка Ассерт.

Если заголовок запроса содержит более одного слова в имени, то следует применять оператор [] для получения его значения как в случае со значением заголовка Ассерт-Encoding.

Ниже представлен фрагмент кода, который использует неявные объекты Cookie для получения cookie:

Код сервлета:

```
Cookie c = new Cookie("emailCookie", emailAddress);
c.setMaxAge(60*60); // устанавливает время жизни 1 час
c.setPath("/") ; //разрешает всему приложению доступ к нему
response.addCookie(c) ;
```

Код JSP:

```
<p>The email cookie: ${cookie.emailCookie.value}</p>
```

В представленном фрагменте кода сервлет создает объект с класса Cookie с именем emailCookie, который содержит адрес электронной почты. Затем код JSP использует неявный объект cookie для получения объекта Cookie, который используется для получения значение свойства cookie с именем emailCookie.

Этот фрагмент работает, поскольку класс Cookie удовлетворяет всем правилам JavaBean и содержит метод getValue, который может быть использован для получения значения, хранящегося в cookie. При необходимости, можно использовать аналогичный код для получения других свойств cookie. Например, можно использовать свойство MaxAge для получения максимального времени жизни для cookie. Ниже представлен фрагмент JSP кода для получения параметра инициализации контекста.

```
<p>The context init param: ${initParam.custServEmail}</p>
```

При этом предполагается, что файл web.xml в составе прочего содержит следующие строки:

```
<context-param>
<param-name>custServEmail</param-name>
<param-value>custserv@mail.com</param-value>
</context-param>
```

В коде JSP используется объект initParam для получения значения параметра custServEmail. Здесь следует отметить, что используется параметр инициализации контекста, который доступен не только для сервлета, но и для всего приложения. Ниже представлен фрагмент JSP кода использования объекта pageContext, который доступен из любой JSP-страницы.

```
<p>HTTP request method: ${pageContext.request.method}<br>
HTTP response type: ${pageContext.response.contentType}<br>
```

```

HTTP session ID: ${pageContext.session.id}<br>
HTTP contextPath: ${pageContext.servletContext.contextPath}<br>
</p>

```

Поскольку объект PageContext удовлетворяет правилам JavaBean, можно легко получать доступ к любому из его свойств. Кроме того, свойства объекта PageContext позволяют получить доступ к объектам в области видимости request, response, session и application.

Например, в предыдущем фрагменте кода используется свойство request объекта PageContext для получения объекта HttpServletRequest, который позволяет получить информацию о текущем методе. Также используется свойство response для получения объекта HttpServletResponse, который позволяет получить информацию о текущем ответе. Далее используется свойство session для получения объекта HttpSession, который позволяет получить информацию о текущей сессии. Также в коде используется свойство ServletContext для получения объекта ServletContext, который позволяет получить информацию о контексте приложения. В результате выполнения представленного фрагмента кода может быть получен результат аналогичный Рис. 86.



Рис. 86. Результат работы фрагмента кода.

Рассмотрим как можно использовать другие операторы языка EL, которые показаны в таблице 25, для выполнения вычислений и сравнений.

Таблица 25.

Операция	Альтернатива	Описание
+		Сложение
-		Вычитание
*		Умножение
/	div	Деление
%	mod	Модуль (остаток)

Ниже представлены утверждения EL, использующие математические операции для выполнения вычислений и результаты их выполнения:

Операция	Результат
<code>\${1+2}</code>	3

```

${16.5+10}          26.5
${3.5E3}             3500.0
${2.5E3+18.4}       2518.4
${3-1}              2
${7*4}              28
${1/5}              0.2
${1 div 5}          0.2
${10 % 8}           2
${10 mod 8}         2
${1+2*4}            9
${(1 +2) * 4}       12
${userID +1}        9 если userID = 8;
                    1 если userID = 0

```

Приведенные примеры показывают, как можно использовать научные обозначения различных чисел, а также использовать скобки для определения порядка выполнения операций.

Также можно использовать операторы отношений для сравнения двух операндов, которые возвращают значения true или false. Хотя эти операторы работают как стандартные операторы Java, можно использовать альтернативный синтаксис, который использует сочетания двух букв. Например, вы можете использовать eq вместо ==, как показано в таблице 26.

Таблица 26

Операция	Альтернатива	Описание
==	eq	Равно
!=	ne	Не равно
<	lt	Меньше
>	gt	Больше
<=	le	Меньше или равно
>=	ge	Больше или равно)

Ниже показаны утверждения EL, использующие операции отношения и результаты их выполнения:

```

Операция                Результат
${"s1" == "s1"}           true
${"s1" eq "s1"}           true
${1 == 1}                  false
${1 != 1}                  false
${1 ne 1}                  false
${3 < 5}                   true
${3 lt 5}                  true

```



```

${2 > 4}                false
${2 gt 4}               false
${3 <= 5}               true
${3 >= 5}               false
${user.firstName==null true, если firstName
                        возвращает null
${user.firstName==" "  true, если firstName
                        возвращает пустую
                        строку
${isDirty == true}     true, если isDirty - true,
                        false если isDirty=false,
                        false, если isDirty - null

```

Рассмотренные примеры также показывают, что EL рассматривает любое null значение равным нулю. Предположим, что имеется атрибут `userID`, который содержит целочисленное значение. Тогда, если атрибут содержит целочисленное значение, то EL будет использовать его в вычислениях. Однако, если атрибут содержит значение `null`, то EL будет использовать в вычислениях нуль.

При создании реляционных выражений, можно использовать ключевое слово `null`, чтобы указать null значение, ключевое слово `true`, чтобы определить значение `true`, а также ключевое слово `false`, чтобы указать `false` значение. Кроме того, при создании реляционных выражений, EL рассматривает значение `null` как значение `false`. Чтобы проиллюстрировать это, предположим, что существует атрибут `IsDirty`, который содержит `Boolean` значение. Если атрибут содержит `true` или `false` значение, EL будет использовать это значение в выражении. Однако, если атрибут содержит значение `null`, EL будет использовать для атрибута `IsDirty` значение `false`.

Логические операторы можно использовать для объединения нескольких реляционных выражений, которые возвращают `true` или `false`. Примеры, приведенные ниже показывают, как использовать все три типа логических операторов `and`, `or` и `not`. Оператор `And` (также можно использовать `&&`) используется для указания того, что оба реляционных выражения должны быть `true`, чтобы все выражение равнялось `true`. Оператор `Or` (также можно использовать `||`) используется для указания того, что по крайней мере одно из реляционных выражений должно быть `true`, чтобы все выражение равнялось `true`. И, наконец, оператор `Not` используется для отрицания значения реляционного выражения.

Операция	Результат
<code>\${"s1" == "s1" && 4 > 3}</code>	true
<code>\${"s1" == "s1" and 4 > 3}</code>	true
<code>\${"s1" == "s1" && 4 < 3}</code>	false
<code>\${"s1" == "s1" 4 < 3}</code>	true
<code>\${"s1" != "s1" 4 < 3}</code>	false
<code>\${"s1" != "s1" or 4 < 3}</code>	false
<code>\${!true}</code>	false
<code>\${not true}</code>	false

Существует еще два оператора, которые полезно использовать в выражениях EL. Первый из них – оператор `empty` *x* - проверяет, содержит ли переменная *x* значение `null` или пустую строку, и возвращает значение `true`. Вторым оператором – `x?y:z` – представляет собой оператор `if`, который выполняется следующим образом: проверяется условие *x* и возвращается значение *y*, если условие *x* выполняется и *z*, в противном случае. Примеры применения описанных операторов приведены ниже:

Операция	Результат
<code>\${empty firstName}</code>	true, если <code>firstName</code> равно <code>null</code> или пустой строке
<code>\${true ? "s1" : "s2"}</code>	<code>s1</code>
<code>\${false ? "s1" : "s2"}</code>	<code>s2</code>

Введение в JSTL

JSP Standard Tag Library (JSTL) представляет собой набор тегов для решения типовых задач, которые обычно выполняются на страницах JSP. В таблице 27 представлены пять библиотек тегов, которые входят в JSTL 1.2.

Таблица 27.

Имя	Префикс	URI	Описание
Core	c	http://java.sun.com/jsp/jstl/core	Содержит основные теги общего назначения (циклы, условные операторы)
Formatting	fmt	http://java.sun.com/jsp/jstl/fmt	Содержит теги для форматирования чисел, времени и даты для правильной работы с интернационализацией
SQL	sql	http://java.sun.com	Содержит теги для рабо-

<i>Имя</i>	<i>Префикс</i>	<i>URI</i>	<i>Описание</i>
		/jsp/jstl/sql	теги с SQL запросами и источниками данных.
XML	X	http://java.sun.com/jsp/jstl/xml	Содержит теги для работы с XML - документами.
Functions	fn	http://java.sun.com/jsp/jstl/functions	Предоставляет функции, которые могут быть использованы для работы со строками.

В настоящем разделе будет рассмотрена работа с общими теги в основной библиотеке. Эта библиотека содержит теги, которые можно использовать для кодирования URL, выполнения цикла по коллекции, а также кодирование условного выполнения утверждений. Если используется модель MVC, теги основной библиотеки зачастую являются единственными тегами JSTL, которые необходимы для совершенствования JSP-страниц.

Для использования JSTL совместно с Eclipse Indigo и GlassFish, которые используются для реализации примеров, никаких дополнительных библиотек не требуется - они предустановлены в IDE Eclipse и на сервере приложений GlassFish. Если же все-таки возникнут проблемы с применением JSTL необходимо загрузить файл jstl-1.2.jar с сайта <http://repo1.maven.org/maven2/javax/servlet/jstl/1.2/> и скопировать его в проекты, в подкаталог WEB-INF/lib. В дальнейшем необходимо включить соответствующие теги в JSP - файлы перед их использованием в JSTL, в частности:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="X" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
```

Следующий фрагмент кода показывает, как тег URL используется для кодирования URL, который ссылается на файл index.jsp в корневом каталоге веб-приложения.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<a href = "<c:url value='/index.jsp' />">Continue Shopping</a>
```

Следующий фрагмент кода выполняет подобную задачу с помощью скриптата JSP:

```
<a href="<%=response.encodeURL("index.jsp")%>">Continue Shopping</a>
```

Обратите внимание, что префикс для данного тега "c". Также обратите внимание, что этот тег больше похож на тег HTML, что существ-

венно облегчает его кодирование и понимание, нежели эквивалентные сценарии JSP, что особенно важно для веб-дизайнеров, которые традиционно используют синтаксис HTML.

При необходимости получения дополнительных сведений по применению средств JSTL, можно обращаться к документации по любому из тегов в этой библиотеке по ссылке <http://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/>, как показано на Рис. 87.

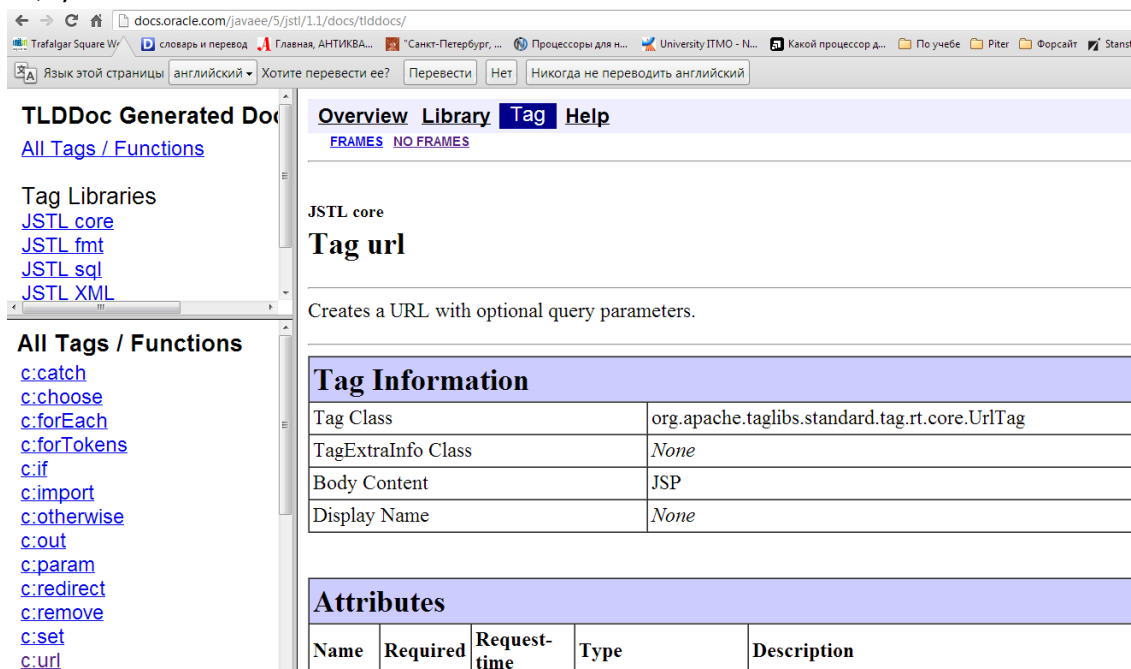


Рис. 87. Пример сайта помощи JSTL

Так если, например, необходимо узнать больше о теге URL в библиотеке core, можно нажать на кнопку "JSTL core" в левом верхнем окне и затем нажать на "C:URL" в левом нижнем углу окна, чтобы отобразить необходимую документацию в окне справа. Этот документ содержит общее описание тегов, список всех доступных атрибутов тега, и подробные сведения о каждом из этих атрибутов.

Рассмотрим еще раз следующий пример кодирования тега URL:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<a href = "<c:url value='/index.jsp' />">Continue Shopping</a>
```

Здесь тег URL кодирует относительную ссылку, которая указывает на файл index.jsp в корневом каталоге веб-приложения. При кодировании тегов JSTL следует учитывать, что в них используется синтаксис XML, а не HTML, что означает необходимость использования правильной капитализации указания значений. Кроме того, атрибуты должны быть заключены в одинарные или двойные кавычки. В представленном фрагменте используются как одинарные, так и двойные кавычки для от-

личия атрибута тега HREF (который использует двойные кавычки) от значения атрибута тега URL (который использует одинарные кавычки).

Еще один пример, представленный ниже, показывает, как использовать тег URL для кодирования URL. Тег URL содержит параметр с именем ProductCode с закодированным значением равным p001 с применением JSTL и соответствующий эквивалент скрипттета:

```
<a href="<c:url value= '/cart?productCode=p001'  
Add To Cart/>">  
</a>
```

```
<a href = "<%=response.encodeURL ("cart?productCode=p001") %>">  
Add To Cart  
</a>
```

Следующие фрагменты кода показывают использование тега URL для кодирования ссылки, которая содержит параметр с именем ProductCode со значением, подставляемым в выражении EL:

```
<a href="<c:url value= '/cart?productCode=${product.code}' />">  
Add To Cart </a>
```

И тот же самый код с применением JSTL тега param:

```
<a href="<c:url value= '/cart'  
<c:param name= 'productCode' value= '${product.code}' />  
</c:url> ">Add To Cart</a>
```

В представленных фрагментах выражение EL получает значение свойства code объекта product класса Product. Также показывается, как записывается JSTL тег param в теге URL для указания имени и значения параметра. Преимущество использования тега param состоит в том, что автоматически кодируются любые небезопасные символы в URL, такие как пробелы и специальные символы, например, знаки плюс и др.

Ниже представлен фрагмент кодирования выше описанных действий с помощью сценариев:

```
<%@ page import= "business.Product" %> <%  
Product product = (Product) session.getAttribute("product");  
String cartUrl = "cart?productCode=" + product.getCode(); %>  
<a href="<%=response.encodeURL(cartUrl)%>">  
Add To Cart</a>
```

При сравнении кодирования тега URL в этих фрагментах кода с эквивалентным сценарием, очевидно, что JSTL теги проще кодируются, изучаются и поддерживаются.

Использование тега forEach

JSTL позволяет использовать теги ForEach для просмотра элементов, которые хранятся в большинстве коллекций, включая массивы. Следующий фрагмент кода показывает, как использовать теги ForEach для организации цикла по объектам LineItem, доступным через свойство items атрибута cart :

```
<c:forEach var= "item" items= "${cart.items}">
```

```

<tr valign="top">
  <td>${item.quantity}</td>
  <td>${item.product.description}</td>
  <td>${item.product.priceCurrencyFormat}</td>
  <td>${item.totalCurrencyFormat}</td>
</tr>
</c:forEach>

```

Здесь, атрибут `var` определяет имя переменной `item` для доступа каждого элемента в коллекции. Затем атрибут `items` использует EL для указания коллекции, в которой хранятся данные. В данном случае, коллекция является объектом `ArrayList<LineItem>`, который возвращается методом `getItems` объекта `Cart` для текущей сессии и, который был сохранен в качестве атрибута с именем `cart`.

Внутри цикла `forEach`, код JSP создает одну строку, состоящую из четырех столбцов для каждого элемента в корзине. При этом каждый столбец использует EL для отображения данных, доступных из объекта `LineItem`. В частности, первый столбец отображает количество, второй столбец отображает описание продукта, в третьем столбце отображается цена за единицу, а в четвертом столбце отображается общая сумма (количество умножается на цену). В результат выполнения фрагмента кода результат может выглядеть, как представлено на Рис. 88.

Количество	Описание	Цена	Сумма
1	Thinking in Java, Bruce Eckel	14,95 руб.	14,95 руб.
5	Struts 2 Black Book, Kogent Solutions Inc.	14,95 руб.	74,75 руб.

Рис. 88. Результат работы тега `ForEach`

Ниже представлен эквивалентный код JSP, не использующий JSTL.

```

<%@ page import="business.*, java.util.ArrayList" %>
<%
  Cart cart = (Cart) session.getAttribute("cart");
  ArrayList<LineItem> items = cart.getItems();
  for (LineItem item : items)
  {
%>
  <tr valign="top">
    <td><%=item.getQuantity()%></td>
    <td><%=item.getProduct().getDescription()%></td>
    <td><%=item.getProduct().getPriceCurrencyFormat()%></td>
    <td><%=item.getTotalCurrencyFormat()%></td>
  </tr>
<% } %>

```

Использование тега `forTokens`

Теги `forTokens` обеспечивают возможность выделения элементов, которые хранятся в строке и разделяются одним или несколькими сим-

волами-разделителями. Следующий фрагмент кода показывает работу со строкой, в которой используется запятая в качестве разделителя:

Код сервлета:

```
session.setAttribute("productCodes", "p001,p002,p003,p004");
```

Код JSP:

```
<p>Product codes<br>
<c:forTokens var="productCode" items="${productCodes}" delims="," >
  <li>${productCode}</li>
</c:forTokens>
</p>
```

В коде JSP показано использование тега `forTokens` в цикле по четырем значениям, которые хранятся в строке. Здесь, атрибут `var` определяет имя переменной `ProductCode` для идентификации каждого элемента в списке. Затем атрибут `items` на языке EL указывает атрибут `productCodes` как строку, которая содержит элементы. И наконец, атрибут `delims` указывает запятую в качестве разделителя. Результат работы кода представлен ниже:

Product codes

- p001
- p002
- p003
- p004

Следующий фрагмент кода работает аналогично предыдущему, но использует два разделителя вместо одного:

Код сервлета:

```
session.setAttribute("emailAddress", "customer@gmail.com");
```

Код JSP

```
<p>Email parts<br>
<c:forTokens var="part" items="${emailAddress}"
  delims="@. ">
  <li>${part}</li>
</c:forTokens>
</p>
```

В частности, атрибут `delims` указывает на символ (`@`) в качестве первого разделителя и точку (`.`) в качестве второго разделителя. В результате, цикл обрабатывает три элемента, по одному для каждой части адреса электронной почты. Результат работы кода представлен ниже:

Email parts

- customer
- gmail
- com

При необходимости, можно использовать вложение одного тега `forTokens` в другой, а также можно вкладывать теги `forTokens` внутри тегов `ForEach`.

Программирование расширенного цикла

При работе с коллекциями, сервлет обычно создает коллекции и передает их в JSP, где они могут отображаться для пользователя. В этом случае в JSP используется теги `ForEach` для организации цикла и выводу его значений. Однако, возникают случаи, когда JSP нужно выполнить некоторую дополнительную обработку. Например, в процессе обработки JSP необходимо учитывать, является ли элемент первым или последним, с тем, чтобы применить к нему специальное форматирование. Или необходимо знать номер элемента, чтобы можно было применять затенение чередующихся элементов. В этом случае, для организации цикла можно использовать атрибуты, описанные в Таблице 28.

Таблица 28.

Атрибуты	Описание
<code>begin</code>	Определяет первый индекс цикла.
<code>end</code>	Определяет последний индекс цикла.
<code>step</code>	Определяет величину приращения к индексу каждый раз в цикле.
<code>varStatus</code>	Определяет имя переменной, которая может быть использована для получения информации о состоянии цикла. Эта переменная поддерживает свойства <code>first</code> , <code>last</code> , <code>index</code> и <code>count</code> .

Указанные в Таблице 28 атрибуты работают так же для тегов `ForEach` и `forTokens`.

Следующий фрагмент кода показывает работу с атрибутами `BEGIN`, `END` и `STEP`:

Код сервлета:

```
int[] numbers = new int[30];
    for (int i = 0; i < 30; i++)
    {
        numbers[i] = i+1;
    }
    session.setAttribute("numbers", numbers);
```

Код JSP:

```
<p>Numbers<br>
<c:forEach items="${numbers}" var="number"
    begin="0" end="9" step="1"
    varStatus="status">
    <li>${number} | First: ${status.first}
        | Last: ${status.last} |
        Index: ${status.index} | Count: ${status.count}
    </li>
```



```
</c:forEach>
</p>
```

Здесь атрибут `Begin` определяет начальный индекс цикла, атрибут `END` задает последний индекс цикла, а атрибут `STEP` определяет величину, на которую индекс увеличивается при каждом прохождении цикла. Если известно, как работает цикл в Java, то не должно быть особых проблем в понимании этих атрибутов.

Результат работы рассмотренного кода представлен ниже:

Numbers

- 1 | First: true | Last: false | Index: 0 | Count: 1
- 2 | First: false | Last: false | Index: 1 | Count: 2
- 3 | First: false | Last: false | Index: 2 | Count: 3
- 4 | First: false | Last: false | Index: 3 | Count: 4
- 5 | First: false | Last: false | Index: 4 | Count: 5
- 6 | First: false | Last: false | Index: 5 | Count: 6
- 7 | First: false | Last: false | Index: 6 | Count: 7
- 8 | First: false | Last: false | Index: 7 | Count: 8
- 9 | First: false | Last: false | Index: 8 | Count: 9
- 10 | First: false | Last: true | Index: 9 | Count: 10

В рассмотренном примере, цикл используется для вывода первых 10 чисел, которые хранятся в массиве из 30 целых значений. Пример также показывает, как использовать `varStatus` атрибут. Этот атрибут определяет имя переменной, которая может быть использована для получения информации о состоянии цикла. В частности, эта переменная содержит четыре свойства с именами `first`, `last`, `index` и `count`, которые можно использовать в теле цикла. Например, можно использовать свойства `first` и `last`, возвращающие `Boolean` значение, которое указывает является ли элемент первым или последним в коллекции. Или, можно использовать свойства `index` и `count`, возвращающие целые значения для данного элемента.

Использование тега `if`

В процессе разработки страниц JSP возникает необходимость использования условного выполнения утверждений в зависимости от значений атрибутов, которые доступны на странице. Для этого можно использовать теги `if`, как показано на следующем фрагменте кода:

```
<c:if test="{cart.count == 1}">
  <p>You have 1 item in your cart.</p>
</c:if>
<c:if test="{cart.count > 1}">
  <p>You have {cart.count} items in your cart.</p>
</c:if>
```

В начале кодируется открывающий тег `if`, который содержит проверочный атрибут. В коде `test` используется атрибут `EL` для получения свойства `count` атрибута `cart`, который содержит количество элементов в корзине. Затем внутри тегов `if` выводятся сообщения, соответствующие количеству товаров в корзине. В частности, первый тег `if` выводит сообщение, если в корзине содержится 1 элемент, а второй тег `if`, если в корзине содержится более одного элемента. Основная разница между этими двумя сообщениями в том, что второе сообщение выводит конкретное значение элементов корзины. Результат работы кода представлен на Рис. 89.

You have 2 items in your cart.

Количество	Описание	Цена	Сумма
1	Thinking in Java, Bruce Eckel	14,95 руб.	14,95 руб.
5	Struts 2 Black Book, Kogent Solutions Inc.	14,95 руб.	74,75 руб.

Рис. 89. Результат работы тега `if`

Ниже представлен эквивалентный предыдущему скриптовый код JSP:

```
<%@ page import="business.Cart, java.util.ArrayList" %>
<%
    Cart cart = (Cart) session.getAttribute("cart");
    if (cart.getCount() == 1)
        out.println("<p>You have 1 item in your cart.</p>");
    if (cart.getCount() > 1)
        out.println("<p>You have " + cart.getCount() +
            " items in your cart.</p>");
%>
```

Для кодирования аналога `is/else` оператора можно использовать тег `choose`, фрагмент применения которого представлен ниже:

```
<c:choose>
    <c:when test="${cart.count == 0}">
        <p>Your cart is empty.</p>
    </c:when>
    <c:when test="${cart.count == 1}">
        <p>You have 1 item in your cart.</p>
    </c:when>
    <c:otherwise>
        <p>You have ${cart.count} items in your cart.</p>
    </c:otherwise>
</c:choose>
```

В начале кодируются теги открытия и закрытия `choose`. Внутри этих тегов кодируется один или более тегов `when`. В представленном фрагменте кода первый тег `when` выполняет проверку атрибута на наличие элементов в корзине. Затем, второй тег `when` проверяет атрибут на

наличие в корзине одного элемента. В противном случае, тег `when` выводит сообщение с количеством элементов.

После тега `when`, но до закрытия тега `choose`, можно кодировать один тег `otherwise`, который выполняется, если ни одно из условий тегов `when` не выполнилось. В этом примере тэг `otherwise` выводит соответствующее сообщение, если в корзине больше чем один.

Ниже представлен эквивалентный предыдущему скриптовый код JSP:

```
<%@ page import="business.Cart, java.util.ArrayList" %>
<%
    Cart cart = (Cart) session.getAttribute("cart");
    if (cart.getCount() == 0)
        out.println("<p>Your cart is empty.</p>");
    else if (cart.getCount() == 1)
        out.println("<p>You have 1 item in your cart.</p>");
    else
        out.println("<p>You have " + cart.getCount() +
            " items in your cart.</p>");
%>
```

Использование тега `import`

Тег `import`, пример использования которого представлен на следующем фрагменте кода, предоставляет еще один способ работы с включением файлов, и работает как стандартные `include` теги JSP, предоставляя возможность выполнять включение файлов во время исполнения, а не компиляции:

Код JSP с JSTL:

```
<c:import url="/includes/header.html" />
```

Эквивалентный стандартный тег JSP:

```
<jsp:include page="/includes/header.html" />
```

Одно из преимуществ тегов `import` по сравнению со стандартными JSP тегами `import` состоит в том, что он позволяет включать файлы из других приложений и веб-серверов, как показано во фрагменте кода ниже:

```
<c:import url="http://localhost:8080/Store/includes/footer.jsp"/>
<c:import url="www.ifmo.ru/includes/footer.jsp"/>
```

Другие теги библиотеки `core JSTL`

В таблице 29 представлены дополнительные шесть тегов библиотеки `core JSTL`.

Таблица 29.

Имя тега	Описание
<code>out</code>	Использует EL для вывода значений, автоматически обрабатывая большинство специальных символов, таких как левая треугольная скобка (<code><</code>) и правая треугольная скобка (<code>></code>).

Имя тега	Описание
set	Устанавливает значение для атрибута в диапазоне видимости
remove	Удаляет атрибут из диапазона видимости.
catch	Перехватывает исключения, которые происходят в процессе выполнения, и создает переменную EL, которая ссылается на объект Throwable.
redirect	Перенаправляет браузер клиента на новый URL.
param	Добавляет параметр к родительскому тегу.

Следующий фрагмент кода позволяет отображать специальные символы в JSP:

Использование атрибута value:

```
<c:out value="${message}" default="No message" />
```

Использование тега с телом:

```
<c:out value="${message}">
  No message
</c:out>
```

Тег out автоматически обрабатывает специальные символы перед их отображением на странице JSP. Если, например, использовать EL для отображения строки, содержащей левые и правые угловые скобки (<>), JSP будет интерпретировать эти скобки как теги HTML и строка отображается некорректно. Однако, при использовании тега out, эти символы отображаются правильно на JSP.

В случае необходимости установить значение атрибута в допустимом диапазоне видимости, используется тег set. Следующий фрагмент кода показывает, как установить значение «Test message» атрибуту с именем message в диапазоне session:

```
<c:set var="message" scope="session" value="Test message" />
```

Также имеется возможность использовать теги set для установки значения свойства атрибута в указанной области. Однако, вместо использования атрибута var для указания имени атрибута, можно использовать атрибут, который непосредственно содержит свойство. Для этого используется EL внутри атрибута для указания ссылки на атрибут. Это иллюстрирует следующий фрагмент кода, который устанавливает значение в JavaBean:

Код JSP с JSTL

```
<c:set target="${user}" property="firstName" value="Fill" />
```

Эквивалентный стандартный тег JSP

```
<jsp:setProperty name="user" property="firstName" value="Fill"/>
```

В следующем фрагменте кода показано, как использовать тег remove для удаления атрибута в диапазоне:

```
<c:remove var="message" scope="session" />
```

При использовании тега `remove`, можно использовать атрибут `var` для указания имени удаляемого атрибута и атрибут `scope` для указания области видимости, которая содержит атрибут.

Если JSP-страница содержит код, который может вызвать исключение, то можно использовать теги `catch` для перехвата исключения, как показано в следующем фрагменте кода:

```
<c:catch var="e">
  <% // this statement will throw an exception
      int i = 1/0;
  %>
  <p>Result: <%= i %></p>
</c:catch>
<c:if test="{e != null}">
  <p>An exception occurred. Message: ${e.message}</p>
</c:if>
```

Здесь открытие и закрытие тега `catch` кодируются вокруг Java скриптата, который вызывает исключение `ArithmeticException` из-за деления на ноль. Когда вызывается исключение, выполнение переходит на выражение Java, которое отображает результат вычисления. Однако тег `catch` также представляет исключение в переменной с именем `e`. В результате тег `if`, который следует за тегом `catch` отображает соответствующее сообщение об ошибке.

Следующим фрагмент кода показывает, как использовать тег `redirect` для перенаправления клиента на новый URL:

```
<c:if test="{e != null}">
  <c:redirect url="/error_java.jsp" />
</c:if>
```

В этом случае тег `redirect` кодируется внутри тега `if`, чтобы клиент не перенаправлялся, если не происходит ошибки.

Расширенный пример с использованием EL и JSTL

Постановка задачи

Необходимо разработать веб-приложение для книжного магазина, предоставляющего возможность просмотра списка книг и помещения выбранных книг в корзину. Корзина накапливает информацию о добавленных книгах и позволяет изменять количество книг и удалять книги из корзины. Данные о книгах содержат три параметра: код, описание и цену.

Примечание: для упрощения приложения формирование списка книг производится статически, а в процессе обработки данные о книгах считываются из файла `products.txt`, расположенного в подкаталоге `WEB-INF` следующего содержания:

```
P001|Thinking in Java, Bruce Eckel|14.95
P002|Object-Oriented Design Patterns, Erich Gamma|12.95
```

P003 | Struts 2 Black Book, Kogent Solutions Inc. | 14.95

P004 | Microsoft C# Projects, Alfred C Thompson | 13.91

Использование для этих целей СУБД было описано в предыдущих разделах, и не должно составить трудностей применения.

Для решения поставленной задачи необходимо выполнить следующие шаги.

1. Создать новый проект.
2. В проекте разработать три класса, в частности:

Класс *Product* - содержит информацию о книге, доступной в магазине. Этот класс должен содержать методы `get` и `set` для полей `code`, `description` и `price`. Кроме того, класс содержит метод `getPriceCurrencyFormat`, который возвращает форматированное значение цены. Например, для значения цены равной 11,5 метод возвращает "11.5 руб."

Класс *LineItem* - содержит информацию о книге, помещенной в корзину. Класс использует объект класса *Product* в качестве экземпляра переменной для хранения соответствующей информации. Кроме того, этот класс всегда вычисляет значение поля `total`, умножая цену продукта на количество и поэтому нет необходимости кодировать метод `set` для этого поля. Кроме того, класс должен содержать метод `getTotalCurrencyFormat`, который форматирует значение, полученное с помощью метода `getTotal`.

Класс *Cart* - содержит строки заказа книги, которые добавлены в корзину на основе использования *ArrayList*, который содержит объекты *LineItem*. При создании объекта *Cart* выполняется конструктор, инициализирующий объект *ArrayList*, к которому в дальнейшем можно добавлять элементы, используя метод `addItem`, и удалять элементы, используя метод `removeItem`. Кроме того, предусматривается метод `getItems`, возвращающий объект *ArrayList* и метод `getCount`, для получения количества продуктов в корзине.

3. Разработать JSP страницу `index.jsp`, которая выводится в начале работы приложения и является реализацией *View* в модели MVC. На этой странице кодируется директива, включающая `taglib`, которая импортирует библиотеку *JSTL core*. Далее на этой странице представляется таблица, где в каждой строке отображается один продукт и ссылка `Add To Cart`, которая использует *JSTL* тег `URL` для указания ссылки, используемой для помещения продукта в корзину. Эта ссылка указывает на URL сервлет `"/CartServlet"`, который является реализацией *Controller*

в модели MVC.

Хотя все четыре продукта закодированы непосредственно на странице, данные о продуктах могут быть получены из базы данных и сохранены в объекте `ArrayList`. Как уже отмечалось, это делается для упрощения задачи при рассмотрении возможностей JSP.

4. Разработать сервлет `CartServlet`. В процессе работы сервлет получает значение параметра `ProductCode` из объекта `request`. Этот параметр однозначно идентифицирует объект `Product`. Затем код получает значение параметра `quantity`, если он задан. Однако, если пользователь нажал на кнопку Обновление на странице `Cart`, этот параметр будет равен значению `null`. После получения значений параметров из запроса, сервлет использует метод `getAttribute` для получения объекта `Cart` из атрибута `session` с именем `cart`. Если этот метод возвращает значение `null`, то сервлет создает новый объект `Cart`.

После того, как объект `Cart` был получен или создан, сервлет устанавливает значение переменной `quantity`, начиная с установки величины переменной значению 1 по умолчанию. Кроме того, если переменная `quantityString` содержит недопустимое целое значение, например, значение `null`, метод `parseInt` класса `Integer` будет генерировать исключение, также устанавливая значение 1 переменной `quantity`. При этом, если пользователь вводит целое число, то переменной `quantity` будет установлено введенное значение. И наконец, если введенная величина отрицательна, то переменной `quantity` будет присвоено 1.

После того, как значение переменной `quantity` было установлено, сервлет использует `getProduct` метод класса `ProductIO` для чтения объекта `Product`, соответствующего переменной `ProductCode` из текстового файла с именем `products.txt`, который хранятся в WEB-INF каталога приложения. Для этого код сервлета устанавливает переменную `ProductCode` в качестве первого аргумента метода `getProduct`. Хотя это приложение для упрощения хранит данные в текстовом файле, в реальной ситуации используются базы данных.

После чтения объекта `Product` из текстового файла, сервлет создает объект `LineItem` и устанавливает значения объекта `Product` и `quantity`. При этом, если значение `quantity` больше 0, этот сервлет добавляет объект `LineItem` в объект `Cart`, в противном случае, элемент удаляется из объекта `Cart`.

В заключение, сервлет устанавливает объект `cart` как атрибут

session с именем cart, и перенаправляет request и response на страницу cart.jsp.

5. Разработать страницу cart.jsp. Как и страница index, здесь используется TagLib директива для импорта JSTL библиотеки core. Затем на странице формируется таблица, где в каждой строке отображается один элемент корзины. Для этой цели используется тег цикла ForEach по элементам Lineltem в ArrayList, который возвращает свойства элементов атрибута cart и использует EL для отображения данных в каждой позиции и кнопок "Обновления" и "Удалить книгу". После создания таблицы с элементами корзины кодируется две формы с кнопками: "Продолжить покупку" и "Завершить покупку". При нажатии на кнопку "Продолжить покупку" выполнение перенаправляет request и response на страницу index.jsp, а при нажатии "Завершить покупку" выполнение перенаправляет request и response на страницу order.jsp.
6. . Разработать страницу order.jsp, которая выводит содержимое корзины и выполняет расчет полной стоимости заказа. Здесь также используется TagLib директива для импорта JSTL библиотеки core. Затем на странице формируется таблица, где в каждой строке отображается один элемент корзины. Для этой цели используется тег ForEach для выполнения цикла по элементам Lineltem в ArrayList, который возвращает свойства элементов атрибута cart и использует EL для отображения данных в каждой позиции. Одновременно в цикле вычисляется стоимость заказа, которая выводится в последнюю строку таблицы.
7. Протестировать работу приложения в браузере.

Создание нового проекта

Выберите пункт меню File/New/Project, в окне выбора типа проекта укажите Web/Dynamic Web Project и нажмите Next. Затем укажите имя проекта CartApp и нажмите Finish.

Создание трех бизнес классов

Прежде всего необходимо создать пакет business. Для этого раскройте в окне Project Explorer объект Java Resources, выберите пакет src и нажмите правую кнопку. В меню выберите New/Package, и в диалоговом окне New Java Package в поле Name введите имя пакета business. Нажмите кнопку Finish.

Затем в образованном пакете последовательно создайте три простых класса Product, LineItem и Cart и поместите в них расположенные ниже исходные тексты классов Java:

Класс Product.java


```

package business;
import java.io.Serializable;
import java.text.NumberFormat;
public class Product implements Serializable
{
    private static final long serialVersionUID = 1L;
    private String code;
    private String description;
    private double price;
    public Product()
    {
        code = "";
        description = "";
        price = 0;
    }
    public void setCode(String code)
    {
        this.code = code;
    }
    public String getCode()
    {
        return code;
    }
    public void setDescription(String description)
    {
        this.description = description;
    }
    public String getDescription()
    {
        return description;
    }

    public void setPrice(double price)
    {
        this.price = price;
    }
    public double getPrice()
    {
        return price;
    }

    public String getPriceNumberFormat()
    {
        NumberFormat number = NumberFormat.getNumberInstance();
        number.setMinimumFractionDigits(2);
        if (price == 0)
            return "";
        else
            return number.format(price);
    }

    public String getPriceCurrencyFormat()
    {
        NumberFormat currency = NumberFormat.getCurrencyInstance();
        return currency.format(price);
    }
}

```

```

}
Класс LineItem.java
package business;
import java.io.Serializable;
import java.text.NumberFormat;
public class LineItem implements Serializable {
    private static final long serialVersionUID = 1L;
    private Product product;
    private int quantity;
    public LineItem() {
    }
    public Product getProduct() {
        return product;
    }
    public void setProduct(Product product) {
        this.product = product;
    }
    public int getQuantity() {
        return quantity;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
    public double getTotal() {
        double total = product.getPrice() * quantity;
        return total;
    }
    public String getTotalCurrencyFormat() {
        NumberFormat currency = NumberFormat.getCurrencyInstance();
        return currency.format(this.getTotal());
    }
}

```

Класс Cart.java

```

package business;
import java.io.Serializable;
import java.util.ArrayList;
public class Cart implements Serializable {
    private ArrayList<LineItem> items;
    public Cart() {
        items = new ArrayList<LineItem>();
    }
    public ArrayList<LineItem> getItems() {
        return items;
    }
    public void setItems(ArrayList<LineItem> items) {
        this.items = items;
    }
    public int getCount(){
        return items.size();
    }
    public void addItem(LineItem item) {
        String code = item.getProduct().getCode();
        int quantity = item.getQuantity();
        for (int i = 0; i < items.size(); i++) {
            LineItem lineItem = items.get(i);

```

```

        if (lineItem.getProduct().getCode().equals(code)) {
            lineItem.setQuantity(quantity);
            return;
        }
        items.add(item);
    }
}

public void removeItem(LineItem item) {
    String code = item.getProduct().getCode();
    for (int i = 0; i < items.size(); i++) {
        LineItem lineItem = items.get(i);
        if
(lineItem.getProduct().getCode().equals(code)) {
            items.remove(i);
            return;
        }
        items.add(item);
    }
}
}
}

```

Далее необходимо ранее описанным способом создать в src подкаталоге пакеты cart и data и поместить в них соответственно содержимое сервлета CartServlet.java и ProductIO.java, представленных ниже:

CartServlet.java

```

package cart;
import javax.servlet.*;
import javax.servlet.http.*;
import business.*;
import data.*;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@WebServlet("/CartServlet")
public class CartServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws
ServletException, IOException {
        String productCode = request.getParameter("productCode");
        String quantityString = request.getParameter("quantity");
        HttpSession session = request.getSession();

        Cart cart = (Cart) session.getAttribute("cart");
        if (cart == null)
            cart = new Cart();
        int quantity = 1;
        try {
            quantity = Integer.parseInt(quantityString);
            if (quantity < 0)
                quantity = 1;
        } catch (NumberFormatException nfe) {
            quantity = 1;
        }
    }
}

```

```

    }
    ServletContext sc = getServletContext();
    String path = sc.getRealPath("WEB-INF/products.txt");
    Product product = ProductIO.getProduct(productCode, path);
    LineItem lineItem = new LineItem();
    lineItem.setProduct(product);
    lineItem.setQuantity(quantity);
    if (quantity > 0)
        cart.addItem(lineItem);
    else if (quantity == 0)
        cart.removeItem(lineItem);
    session.setAttribute("cart", cart);
    String url = "/cart.jsp";
    RequestDispatcher dispatcher = getServletContext()
        .getRequestDispatcher(url);
    dispatcher.forward(request, response);
}
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws
ServletException, IOException {
    doGet(request, response);
}
}

```

ProductIO.java

```

package data;
import java.io.*;
import java.util.*;
package data;
import java.io.*;
import java.util.*;
import business.*;
public class ProductIO {
    public static Product getProduct(String code, String filepath) {
        try {
            File file = new File(filepath);
            BufferedReader in = new BufferedReader(new
FileReader(file));
            String line = in.readLine();
            while (line != null) {
                StringTokenizer t = new StringTokenizer(line,
"|");
                String productCode = t.nextToken();
                if (code.equalsIgnoreCase(productCode)) {
                    String description = t.nextToken();
                    double price = Dou-
ble.parseDouble(t.nextToken());
                    Product p = new Product();
                    p.setCode(code);
                    p.setDescription(description);
                    p.setPrice(price);
                    in.close();
                    return p;
                }
                line = in.readLine();
            }
            in.close();
        }
    }
}

```

```

        return null;
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}

public static ArrayList<Product> getProducts(String filepath) {
    ArrayList<Product> products = new ArrayList<Product>();
    File file = new File(filepath);
    try {
        BufferedReader in = new BufferedReader(new
FileReader(file));
        String line = in.readLine();
        while (line != null) {
            StringTokenizer t = new StringTokenizer(line,
"|");

            String code = t.nextToken();
            String description = t.nextToken();
            String priceAsString = t.nextToken();
            double price = Double
ble.parseDouble(priceAsString);
            Product p = new Product();
            p.setCode(code);
            p.setDescription(description);
            p.setPrice(price);
            products.add(p);
            line = in.readLine();
        }
        in.close();
        return products;
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}
}

```

Разработка страницы index.jsp

Откройте страница index.jsp для редактирования и скопируйте в нее код, представленный ниже:

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Книжный магазин</title>
</head>
<body>
    <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
    <h1>Список книг для выбора</h1>
    <table cellpadding="5" border="1">
        <tr valign="bottom">
            <td align="left"><b>Описание</b></td>
            <td align="left"><b>Цена</b></td>

```

```

        <td align="left"></td>
    </tr>
    <tr valign="top">
        <td>Thinking in Java, Bruce Eckel</td>
        <td>14.95 py6.</td>
        <td><a href="CartServlet?productCode=p001">Add To
Cart</a></td>
    </tr>
    <tr valign="top">
        <td>Object-Oriented Design Patterns, Erich Gamma</td>
        <td>12.95 py6.</td>
        <td><a href="CartServlet?productCode=p002">Add To
Cart</a></td>
    </tr>
    <tr valign="top">
        <td>Struts 2 Black Book, Kogent Solutions Inc.</td>
        <td>14.95 py6.</td>
        <td><a href="CartServlet?productCode=p003">Add To
Cart</a></td>
    </tr>
    <tr valign="top">
        <td>Microsoft C# Projects, Alfred C Thompson</td>
        <td>13.91 py6.</td>
        <td><a href="CartServlet?productCode=p004">Add To
Cart</a></td>
    </tr>
</table>
</body>
</html>

```

Разработка страницы cart.jsp

Создайте новый файл cart.jsp и скопируйте в него код, представленный ниже:

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Книжный магазин</title>
</head>
<body>
    <h1>Ваша корзина</h1>
    <table border="1" cellpadding="5">
        <tr>
            <th>Количество</th>
            <th>Описание</th>
            <th>Цена</th>
            <th>Сумма</th>
        </tr>
        <%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
        <c:forEach var="item" items="${cart.items}">
            <tr valign="top">
                <td>

```

```

        <form action="<c:url val-
ue= '/CartServlet' />">
            <input type="hidden"
name="productCode"
            val-
ue= "${item.product.code}"> <input type="text" size=2
            name="quantity" val-
ue= "${item.quantity}"> <input
            type="submit"
value="Обновление">
        </form>
    </td>
    <td>${item.product.description}</td>
    <td>${item.product.priceCurrencyFormat}</td>
    <td>${item.totalCurrencyFormat}</td>
    <td>
        <form action="<c:url val-
ue= '/CartServlet' />">
            <input type="hidden"
name="productCode"
            val-
ue= "${item.product.code}"> <input type="hidden"
            name="quantity" value="0">
            <input type="submit"
            value="Удалить книгу">
        </form>
    </td>
</tr>
</c:forEach>
<tr>
    <td colspan="3">
        <p>
            <b>Для изменения количества книг</b>,
введите новое значение и
            нажмите кнопку Обновление.
        </p>
    </td>
</tr>
</table>

<br>
<form action="<c:url value= '/index.jsp' />" method="post">
    <input type="submit" value="Продолжить покупку">
</form>
<form action="<c:url value= '/order.jsp' />" method="post">
    <input type="submit" value="Завершить покупку">
</form>

</body>
</html>

```

Разработка страницы order.jsp

Создайте новый файл order.jsp и скопируйте в него код, представленный ниже:

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>

```

```

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
  <h1>Ваш заказ</h1>
  <%@ page im-
port="business.*,java.util.ArrayList,java.math.BigDecimal"%>
  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
  <table border="1" cellpadding="5">
    <tr>
      <th>Количество</th>
      <th>Описание</th>
      <th>Цена</th>
      <th>Сумма</th>
    </tr>
    <%
      Cart cart = (Cart) session.getAttribute("cart");
      ArrayList<LineItem> items = cart.getItems();
      double itog = 0;
      for (LineItem item : items) {
    %>
    <tr valign="top">
      <td><%=item.getQuantity()%></td>
      <td><%=item.getProduct().getDescription()%></td>
      <td><%=item.getProduct().getPriceCurrencyFormat()%></td>
      <td><%=item.getTotalCurrencyFormat()%></td>
    </tr>
    <%
      itog = itog + item.getTotal();
    %>
    <tr valign="top">
      <th>ИТОГО ${cart.count} наименование(й) книг(и) на
сумму</th>
      <td><%=BigDecimal.valueOf(itog).setScale(2,
BigDecimal.ROUND_HALF_DOWN).doubleValue()%> руб.</td>
    </tr>
  </table>
</body>
</html>

```

Тестирование приложения

Запустите браузер и перейдите по ссылке <http://localhost:8080/CartApp> откроется страница index.jsp для добавления книг в корзину, как показано на Рис. 90.

Список книг для выбора

Описание	Цена	
Thinking in Java, Bruce Eckel	14.95 руб.	Add To Cart
Object-Oriented Design Patterns, Erich Gamma	12.95 руб.	Add To Cart
Struts 2 Black Book, Kogent Solutions Inc.	14.95 руб.	Add To Cart
Microsoft C# Projects, Alfred C Thompson	13.91 руб.	Add To Cart

Рис. 90. Стартовая страница приложения книжного магазина.

Для проверки работоспособности режима добавления товаров в корзину нажмите кнопку Add to Cart в строке с выбранной книгой, и на экране отобразится информация, представленная на Рис. 91.

Ваша корзина

Количество	Описание	Цена	Сумма	
1 <input type="button" value="Обновление"/>	Thinking in Java, Bruce Eckel	14,95 руб.	14,95 руб.	<input type="button" value="Удалить книгу"/>

Для изменения количества книг, введите новое значение и нажмите кнопку Обновление.

Рис. 91. Содержимое корзины покупателя.

Для проверки работоспособности режима изменения количества измените в столбце Количество значение поля с 1, например, на 5 и нажмите кнопку "Обновление". На экране произойдет изменение поля Сумма, как показано на Рис. 92.

Ваша корзина

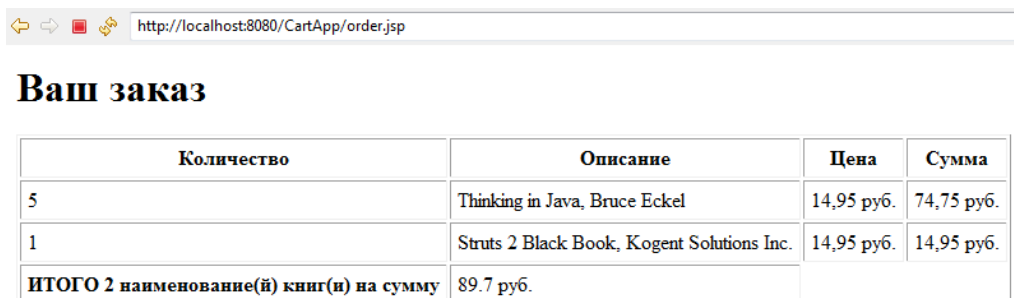
Количество	Описание	Цена	Сумма	
5 <input type="button" value="Обновление"/>	Thinking in Java, Bruce Eckel	14,95 руб.	74,75 руб.	<input type="button" value="Удалить книгу"/>

Для изменения количества книг, введите новое значение и нажмите кнопку Обновление.

Рис. 92. Страница изменения количества книг.

Нажмите кнопку "Продолжить покупку", добавьте в корзину еще одну книгу и проверьте работоспособность режима удаления книги из корзины путем нажатия кнопки "Удалить книгу".

При нажатии на кнопку "Завершить покупку" на экране отобразится Рис. 93.



Количество	Описание	Цена	Сумма
5	Thinking in Java, Bruce Eckel	14,95 руб.	74,75 руб.
1	Struts 2 Black Book, Kogent Solutions Inc.	14,95 руб.	14,95 руб.
ИТОГО 2 наименование(й) книг(и) на сумму		89.7 руб.	

Рис. 93. Итоговое содержание корзины.

Вопросы для самопроверки

1. Разделению каких ролей (видов деятельности) разработки веб-приложения способствует технология JSP?
2. Какой компонент и функциональность веб-приложения формирует JSP-страница после компиляции?
3. Какие две фазы включает в себя цикл запрос-ответ JSP?
4. Из каких элементов состоит JSP-страница для представления статического и динамического?
5. Какие различные элементы используются в JSP-страницах?
6. Какими тремя способами могут быть созданы объекты в JSP?
7. Какие классы JSP API использует JSP-страница, которые определены в пакете javax.servlet?
8. Каких этапов необходимо придерживаться при разработке приложения JSP?
9. Как можно указать контейнеру, что он должен ставить в очередь многочисленные запросы к JSP-странице и отправлять только один запрос за раз?
10. Как можно указать размер буфера объекта out?
11. Экземпляром какого класса является неявный объект JSP exception?
12. Укажите неявный объект и соответствующий метод, используемые для записи в журнал сообщений с ошибками в JSP-странице.
13. Что такое JavaBeans и каким требованиям они должны удовлетворять?
14. Приведите пример описания тега JSP обращения к компоненту JavaBean.

15. Приведите пример описания тега JSP для доступа к свойству JavaBean.
16. Для каких целей предназначен язык EL и перечислите его достоинства и недостатки.
17. Для чего предназначен JSP Standard Tag Library (JSTL) и 5 библиотек тегов он содержит?
18. Приведите формат тега JSP для включения библиотеки JSTL.
19. Приведите пример использования JSTL тега forEach.
20. Приведите пример использования JSTL тега forEachTokens.
21. Приведите пример использования расширенного цикла JSTL.
22. Приведите пример использования JSTL тега if.

Заключение

Автор старался в рамках пособия предоставить информацию по различным технологиям сетевого взаимодействия приложений с использованием средств Java. Однако даже в сфере применения технологии Java для разработки сетевого взаимодействия приложений не рассмотренными оказались огромные возможности. Кроме того, в настоящее время наряду с реализациями спецификации JavaEE разработано большое количество API, которые в значительной степени упрощают и ускоряют разработку различных элементов инфокоммуникационных систем. Вот только краткий перечень наименований систем, которые можно применять совместно и вместо технологии JavaEE: Java Server Faces (JSF), Hibernate, Spring, Struts, Tapestry, Axis2, Seam, OSGi и мн. др. другие.

Автор надеется, что знания и навыки, полученные в процессе изучения настоящего пособия, будут стимулом к более серьезному и глубокому изучению Java технологий.

ЛИТЕРАТУРА

1. Марти Холл, Лэрри Браун. Программирование для Web. Библиотека профессионала, 2002 г. Издательство: "Вильямс"
2. Elliotte Rusty Harold, Java™ Network Programming, Third Edition, Published by O'Reilly Media, Inc., 2005, 738 p.
3. Joel Murach and Andrea Steelman, Murach's Java Servlets and JSP (2nd Edition), 2008, 729 p.
4. Сью Шпильман JSTL. Практическое руководство для JSP-программистов Издательство: КУДИЦ-Образ, 2004 г.
5. Х. М. Дейтел, П. Дж. Дейтел, С. И. Сантри Технологии программирования на Java 2. Книга 3. Корпоративные системы, сервлеты, JSP, Web-сервисы Advanced Java 2 Platform. How to Program Издательство: Бином-Пресс, 2003. – 672 с.: ил.
6. Кришнамурти, Дж. Рексфорд Web-протоколы. Теория и практика. HTTP/1.1, взаимодействие протоколов, кэширование, измерение трафика Издательство: Бином, 2002.
7. Kogent Solutions Inc., Java Server Programming Java Ee5 Black Book, Platinum Ed (With Cd), 2008, 1748 p.
8. Java портал Oracle –<http://www.oracle.com>.



В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена программа его развития на 2009–2018 годы. В 2011 году Университет получил наименование «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

КАФЕДРА СЕРВИСОВ И УСЛУГ В ИНФОКОММУНИКАЦИОННЫХ СИСТЕМАХ

Кафедра "Сервисов и услуг в инфокоммуникационных системах связи" образована в 2010 году в составе факультета "Инфокоммуникационные технологии" и обеспечивает специализацию по профилю "Инфокоммуникационные технологии в сервисах и услугах связи" по направлению подготовки 210700 "Инфокоммуникационные технологии и системы связи".

Полученное образование позволяет быть востребованным специалистом в организациях и предприятиях, связанных с проектированием и производством инфокоммуникационных систем (ИКС) и устройств, учреждениях и фирмах, занимающихся предоставлением инфокоммуникационных сервисов и услуг, в том числе, занимающихся разработкой, эксплуатацией и сопровождением распределенных ИКС и корпоративных систем связи, а также в подразделениях различных организаций, занимающихся техническим сопровождением и администрированием инфокоммуникационных ресурсов и их защитой.

Области профессиональной деятельности бакалавров включают: сервисно-эксплуатационную - в сфере современных ИКС; научно-исследовательскую и проектную - в области новых технологий и сервисов в инфокоммуникационной среде; организационно-управленческую – в сфере информационного менеджмента в ИКС.

Базовая подготовка студентов, ведется по дисциплинам:

- Естественнонаучный цикл – математика, физика, информатика, вычислительная математика, электроника, моделирование и проектирование ИКС и др.
- Профессиональный цикл – электромагнитные поля и волны, теория электрических цепей, цифровая обработка сигналов, основы построения ИКС и сетей, технологии баз данных, распределенное программирование, администрирование сетей, системы хранения данных, инфокоммуникационные технологии в Интернете, технологии облачных вычислений и услуг, технологии программирования, и др.

В ходе подготовки студенты получают навыки работы с различными системами программирования, объектно-ориентированные технологии программирования на языках С++ и Java; получают глубокие знания и практический опыт в области разработки, конфигурирования и тонкой настройки аппаратных и программных компонентов ИКС, разработки программных систем на основе применения свободно распространяемого программного обеспечения в инфокоммуникационных приложениях, осваивают методы и технологии проектирования и обработки баз данных, принципы организации и архитектуру распределенных вычислений по технологии JavaEE, а также получают навыки создания WEB-сервисов и предоставления инфокоммуникационных услуг на основе применения сервис-ориентированной архитектуры.

В рамках образовательных программ, связанных с коммуникационными технологиями и сетевым администрированием, студенты получают знания по проектированию, развертыванию, техническому сопровождению и администрированию локальных и глобальных сетей на крупных предприятиях, конфигурированию и маршрутизации сетевого оборудования на предприятии, в том числе, оборудования корпорации Cisco.

При организации учебного процесса существенное внимание уделяется современным технологиям и услугам связи, которые предоставляются пользователям, в том числе: облачные технологии и услуги в облаках, виртуализация ресурсов, сетевые мультимедийные технологии, современные технологии мониторинга и управления качеством передачи данных в сетевой инфраструктуре, обеспечение безопасности и защиты в сетях передачи данных, а также многие другие технологии развивающейся индустрии инфокоммуникаций.

Кроме того, учебный план по профилю предусматривает изучение дисциплин, преподаваемых в Центре Авторизованного Обучения IT-технологиям, в котором проводится обучение по новейшим продуктам и технологиям таких известных фирм как Microsoft, Novell, Oracle, Cisco,

НР и др. Полученные в рамках направления знания позволят студентам успешно подготовиться к экзаменам и получить сертификаты ведущих транснациональных компаний, что в значительной степени повышает конкурентоспособность выпускников при приеме на работу.

Анатолий Алексеевич Дубаков

Сетевое программирование

Учебное пособие

В авторской редакции

Редакционно-издательский отдел НИУ ИТМО

Зав. РИО

Н.Ф. Гусарова

Лицензия ИД № 00408 от 05.11.99

Подписано к печати

Заказ №

Тираж 100 экз.

Отпечатано на ризографе