Программирование Web-приложений

А.Ф. Лепехин

Программирование Web-приложений

А.Ф. Лепехин

Опубликовано 2002

Аннотация к учебному курсу

В последнее время все большую популярность приобретает использование Web-технологий для обеспечения коллективного доступа к базам данных. Эти технологии обеспечивают доступ с помощью Web-браузеров, таких как Internet Explorer или Netscape Navigator, без установки каких-либо дополнительных программ на компьютерах пользователей. Системы с архитектурой Web-приложений могут без каких-либо изменений эксплуатироваться как в локальной, так и в глобальной сети.

В данном курсе рассматривается технология построения Web-приложений, основанных на открытых стандартах Java, SQL, HTML, XML, UML. Кроме перечисленных языков, слушатели будут ознакомлены с архитектурой Enterprise Java Beans (EJB), технологией Java Server Pages (JSP)и платформой Java 2 Enterprise Edition (J2EE). Следование открытым стандартам обеспечит Вас независимостью от конкретных поставщиков программных и инструментальных средств, т.к. все такие средства будут делать одно и тоже - реализовывать данный стандарт.

Данный курс включает изучение следующих тем:

- Введение в языки HTML, JavaScript, XML и XSL-преобразования
- Введение в языки UML и Java
- Программирование распределенных компонент бизнес-логики в виде Enterprise Java Beans
- Программирование сервлетов и Java Server Pages
- Установка приложения на платформе Java 2 Enterprise Edition

Курс расчитан на слушателей, имеющих опыт разработки клиент-серверных приложений для работы с реляционными базами данных и знакомых с языком Java. В результате обучения слушатели должны овладеть средствами Java (Enterprise Edition) платформы для построения распределенных приложений, в том числе научиться:

- Создавать приложения, работающих с базами данных
- Создавать Java-компоненты (Java Beans) и использовать их в сервлетах и Java Server Pages
- Создавать распределенные Java-компоненты (Enterprise Java Beans)
- Создавать пользовательский интерфейс средствами HTML, XML и Java-script.

В процессе обучения слушателями будет построено демонстрационное приложение, моделирующее аренду книг в библиотеке через Internet. Затем будет показано, как можно реализовать получение списка книг в виде Web-сервиса.

Содержание

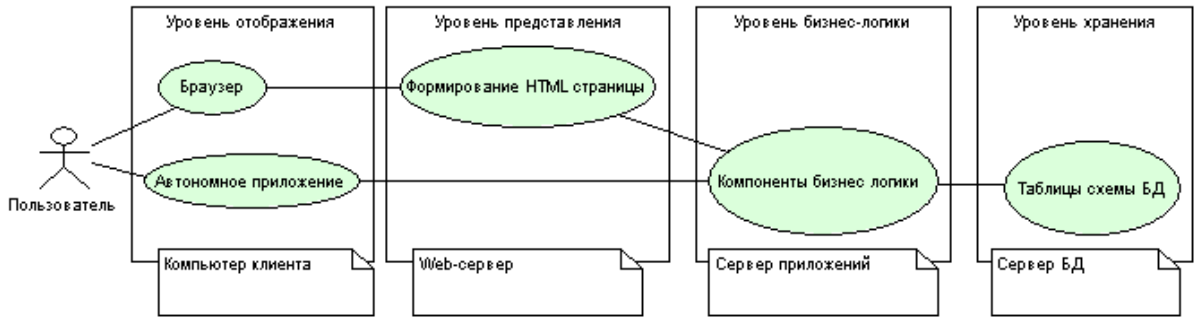
Список иллюстраций

Глава 1. Архитектура Web-приложений

Архитектура Web-приложений

Базовой архитектурой Web-приложения является много-звенная архитектура:

Рисунок 1.1. Базовая архитектура Web-приложения



На уровне клиента доступ к приложению через браузер обеспечивает доступ "в любом месте, в любое время". Более сложный графический интерфейс может быть реализован либо через дополнительные возможности `HTML` [html.xml#HTML] (апплеты или `JavaScript` [javascript.xml]) или путем установки на клиенте автономных программ, выполняющихся вне браузера.

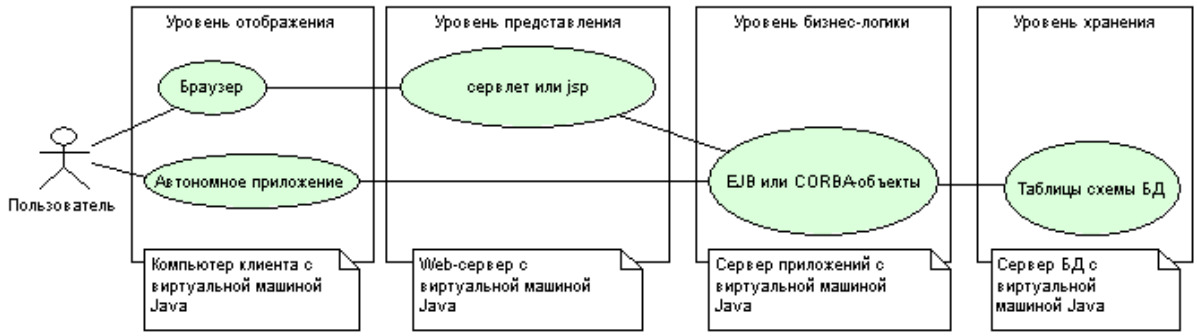
Сервер уровня представления - Web-сервер - обслуживает `HTTP` [html.xml#http]-запросы и динамически генерирует код представления - обычно `HTML` или `XML`-страницы - отображаемые на клиенте (в браузере).

Сервер приложений управляет бизнес-логикой приложения и разделяется множеством клиентов и приложений. Он может иметь компоненты подготовки сложных отчетов и анализа данных. Компоненты бизнес-логики должны быть доступны из других внешних серверов для облегчения требований интеграции. Для этого компоненты бизнес-логики представляются в виде Enterprise Java Beans или Web-сервисов

Сервер баз данных содержит масшабируемую систему управления данными или другие внешние источники данных, которые должны быть интегрированы в систему.

Использование языка Java привело к созданию следующей архитектуры, которая в настоящее время фактически является стандартной платформой Web-приложений [1]:

Рисунок 1.2. Архитектура Web-приложения на языке Java



Клиент - машина клиента (пользователя), оснащенная браузером (Microsoft Internet Explorer, Netscape

Navigator или др.) для просмотра Web-страниц. Для просмотра XML необходим браузер, правильно выполняющий XSL-преобразования, например Microsoft Internet Explorer 6.0

Web-сервер - любой Web-сервер, содержащий JVM (Java Virtual Machine) для выполнения Java-кода. Web-сервер предоставляет браузеру HTML или XML-страницы либо с помощью сервлетов - Java-компонент, расширяющих возможности Web-сервера, либо с помощью JSP (Java Server Pages) - особого стиля написания сервлетов.

Сервер приложений должен поддерживать платформу J2EE для размещения EJB, и SOAP (Simple Object Access Protocol) для реализации Web-сервисов.

Сервер баз данных обслуживает запросы к СУБД. Это может быть реляционная СУБД, но может быть и любая другая, например объектная или объектно-реляционная.

Взаимодействие между компьютерами осуществляется с помощью сети с протоколом TCP/IP - стандартного протокола Internet.

Реализацией стандартной платформы Интернет является Oracle 9i. Он реализует Java-платформу, включающую поддержку ORB и EJB, непосредственно в СУБД. Интернет-платформа Oracle позволяет писать хранимые процедуры на языке Java, являющимся стандарным языком программирования, в отличие от PL/SQL, который был специфичен только для Oracle.

Компоненты Web-приложения

Типичное приложение состоит из следующих компонентов (уровней):

Клиентская часть *(уровень отображения)*, выполняемая в браузере, представляет собой HTML-страницы с кодом, реализованном в виде Java-апплетов или написанным на JavaScript. Клиентская часть также может быть автономной Java-программой.

Web-серверная часть *(уровень представления)* - это сервлеты или jsp, динамически генерирующие HTML страницы уровня отображения. Сервлет может порождать Java Beans (компоненты на языке Java) и использовать их для получения результатов из объекта бизнес-логики

Объекты бизнес-логики *(уровень бизнес-логики)*, порождаемые фабрикой на сервере среднего уровня. Объекты бизнес логики программируются на языке Java в стандартах CORBA или EJB. Они решают основные задачи приложения, не связанные с представлением информации на экране пользователя. Такими задачами являются, например, расчет заработной платы, расчет платежей за электроэнергию и пр.

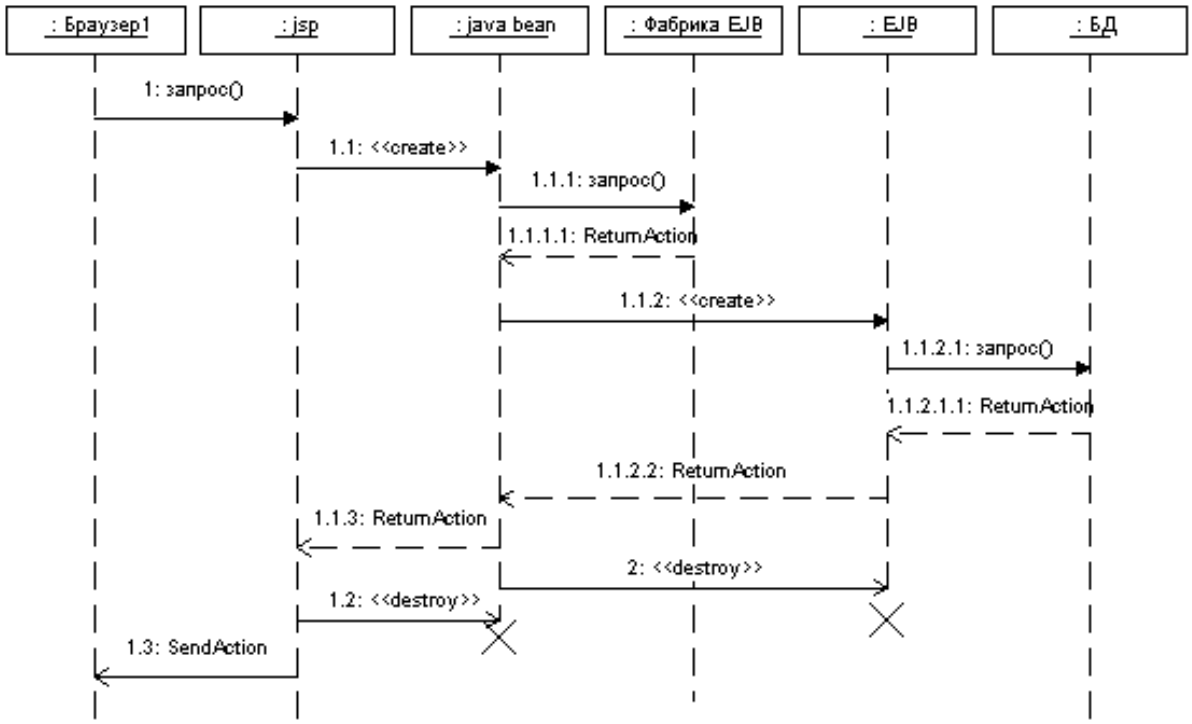
Схема БД *(уровень хранения)* - совокупность таблиц и других объектов БД, необходимых для реализации бизнес-логики. Уровень хранения обеспечивает так называемые "постоянство" объектов.

Объектов бизнес-логики, так же как и bean'ов может быть много - столько, сколько надо, чтобы реализовать бизнес логику. Заметьте, что первоначально их нет - они создаются динамически при решении конкретной задачи бизнеса.

Последовательность выполнения

Рассмотрим решение некоторой задачи из описанных в бизнес-логике. Последовательность решения задачи такова:

Рисунок 1.3. Web-приложение на языке Java



- Браузер вызывает сервлет или jsp с Web-сервера для решения некоторой задачи, описанной в бизнес логике
- Сервлет (jsp) может создать объект Java Bean для получения решения
- Java-bean обращается к фабрике EJB с требованием создания нужного объекта бизнес - логики - того, который решает поставленную задачу
- Фабрика создает объект бизнес-логики и ссылку на него возвращает в bean
- Bean обращается к объекту бизнес-логики, вызывая тот или иной метод для решения задачи
- Объект бизнес-логики обращается к БД для получения или сохранения данных, участвующих в решении задачи
- Объект бизнес-логики получает данные из БД
- Объект бизнес-логики возвращает полученные данные в bean
- Bean возвращает полученные данные в сервлет (jsp)
- Сервлет (jsp) формирует HTML-страницу ответа на задачу и посылает ее в браузер.

После решения задачи автоматически уничтожается порожденный bean и объект бизнес-логики. Остается фабрика и сервлет (jsp). Конечно, все на самом деле гораздо сложнее, но детали мы отложим до изучения CORBA и Enterprise Java Beans.

Преимущества

Достоинствами данной архитектуры являются:

- Разделение бизнес-логики и визуального представления приложения. Это позволяет, во-первых, сконцентрироваться на каждой задаче в отдельности, и, во-вторых, реализовать бизнес-логику в виде повторно-используемых компонент. Одним из недостатков клиент-серверных приложений является их "монолитность". Бизнес-логика таких приложений вплетена в уровень представления ("формы"), в результате требуется тестирование всего приложения при изменении какой-либо части бизнес-логики.
- Централизованное хранение объектов приложения. Все изменения сразу становятся доступными всем пользователям. В этом состоит основное отличие данной архитектуры от архитектуры клиент-сервер. В последней большая часть логики приложения сосредоточена на клиенте, т.е. компьютере пользователя - следовательно изменения, если они есть, должны быть перенесены на каждое рабочее место.

- Использование пользователями стандартных средств доступа в Internet, без необходимости что-либо дополнительно устанавливать на рабочих местах.
- Легкость расширения системы (масштабируемость) за счет подключения новых серверов и распределения объектов бизнес-логики между ними.
- Легкость подключения новых пользователей.

Литература

1. Building Java Applications for the Oracle Internet Platform With Oracle JDeveloper. An Oracle White Paper October 1999

2.Steven Gould, "Develop n-tier applications using J2EE" [http://www.javaworld.com/javaworld/jw-12-2000/jw-1201-weblogic.html], Java World, december 2000.

3. Значения терминов и всевозможных сокращений (например, JSP) можно найти в Технической энциклопедии [http://www.techweb.com/encyclopedia]

Глава 2. HTML - язык разметки гипертекста

HTML - язык разметки гипертекста

Я думаю, что методически правильно начать с HTML. Это - очень простой язык и знакомство с ним не требует никакой предварительной подготовки.

HTML - Hyper Text Markup Language - язык *разметки гипертекста*. Его сутью является разметка и гипертекст. *Разметка* - это команды форматирования документа. Большинство из вас скорее всего привыкло работать с редакторами WYSIWYG (what you see is what you get) – типа Word, в которых результат форматирования текста сразу отображается на экране ("что Вы видите, то и получаете"). Однако, есть и другие редакторы, а точнее способы подготовки документов, когда в текст документа непосредственно вставляются команды форматирования. Те, кто работал в свое время в операционных системах типа RSX-11M или ОСРВ-2, возможно помнят системы подготовки текстов runoff или DOC. Современным примером такой системы является TeX, написанный Д.Кнутом. Текст в такой системе создается обычным редактором (типа Notepad) и затем обрабатывается некоторым постпроцессором, для получения окончательного вида документа. HTML относится именно к таким системам. Постпроцессорм для HTML является браузер, интерпретирующий команды разметки и выводящий документ на экран в отформатированном виде. Например, если какую-либо часть текста необходимо выделить жирным шрифтом, мы записываем этот текст так:

```
<B>Текст жирным шрифтом</B>
```

Как Вы видите, перед требуемым отрезком текста вставляется команда форматирования, указывающая, что дальнейший текст должен быть выведен жирным шрифтом. Команды HTML называют тегами. Теги пишутся попарно – вначале открывающий тег, потом – закрывающий. В нашем случае `</B>` - это закрывающий тег, заканчивающий действие тега `<B>`. Открывающий тег, закрывающий тег и все, что находится между ними называется элементом. HTML не требует наличия некоторых закрывающих тегов, но их рекомендуется писать для облегчения перехода к XML (Extended Markup Language), о котором мы поговорим позднее. Заодно рекомендуется писать теги HTML большими буквами (а теги XML – маленькими). Открывающие теги могут иметь атрибуты, уточняющие действие тега:

```
<Тег атрибут=”значение” ... > Текст </Тег>
```

Под *гипертекстом* понимается возможность переходить от одного документа к другому, возможно лежащему на другом компьютере. Это делается с помощью "ссылок", с которыми Вы возможно знакомы по редактору Word. Кроме того, в понятие "гипертекст" включается и "гипермедиа" – возможность включения в документ разрозненных графических, аудио и видео фрагментов. Для включения ссылок в текст документа существует специальный тег `<A>`, в котором указывается узел и имя связанного документа (адрес ссылки):

```
<A HREF="Адрес ссылки" > Это ссылка </A>
```

Здесь `HREF` – атрибут тега `<A>`. Можно указать конкретный элемент внутри документа ссылки, к которому надо перейти. В этом случае `ID` элемента указывается в адресе после знака #. Например,

```
<A HREF="ООП.html#Наследование" > Это ссылка на элемент с ID="Наследование"  в документе ООП.html
```

Истории гипертекста более 50-и лет. Впервые концепция гипертекста была опубликована в 1945 г. в статье Ванневара Буша "Как мы можем думать" в журнале Atlantic Monthly. В этой статье описывается машина для "просмотра и составления заметок в обширной текстовой и графической системе, подключенной к Сети". Однако в реальной жизни гипертекст появился только в 1987 г., когда Apple и Microsoft

ввели свои справочные системы, в которых для перехода от одной темы к другой достаточно было щелкнуть на гипертекстовой ссылке.

Язык HTML был разработан в 1990 г. в физической лаборатории CERN в Швейцарии для доступа к документам в сети лаборатории. Тогда же была изобретена "паутина" (Web) и первый HTML браузер, как часть этого проекта. В 1991 году "паутина" была опробована в CERN, а в 1992 г. стала доступна пользователям Интернет. При пересылке HTML документов по сети используется протокол *HTTP* (Hyper Text Transfere Protocol). В дальнейшем HTML был усовершенствован, пройдя ряд стандартов. Сейчас действует стандарт HTML 4.0.

Основные команды

Если не считать "пролога", с которого начинается HTML документ, то весь документ – это набор элементов. Элементы могут быть вложены один в другой. Весь текст HTML документа должен находиться внутри тегов:

```
<HTML>
........
</HTML>
```

Обычно текст делится на части – заголовок и тело. Они, соответственно, должны находиться между тегами

```
<HEAD>
.....
</HEAD>
```

и

```
<BODY>
....
</BODY>
```

Теперь мы уже можем составить первый HTML документ. Будем их писать в файлах каталога Html:

```
<HTML>
<HEAD>
Пример 1
</HEAD>
<BODY>
Привет этому миру!
</BODY>
</HTML>
```

Создайте этот файл под именем `Hello.html` и откройте его в браузере. Получите:

```
Пример 1 Привет этому миру!
```

Как видите строки слились в одну, хотя мы писали эти фразы на разных строках. Все написано одним шрифтом и размером. В общем, хотелось бы как-то по другому. Фразу "Пример 1" хочется иметь посередине и побольше. "Привет этому миру" хочется иметь пониже.

Чтобы разделить текст на строки необходимо указать тег перехода на новую строку:

```
<BR>
```

Для начала нового абзаца необходимо написать тег начала абзаца:

```
<P>
```

Добавим тег <P> к нашему тексту:

```
<HTML>
<HEAD>
Пример 1
</HEAD>
<BODY>
<P>
Привет этому миру!
</P>
</BODY>
</HTML>
```

получим:

```
Пример 1
Привет этому миру!
```

Уже лучше. Теперь отцентрируем заголовок. Для этого есть атрибут

```
ALIGN = "CENTER" | "LEFT" | "RIGHT"
```

- выравнивание по центру, влево, вправо. Вертикальная черта обозначает "или". Этот атрибут можно писать почти у всех тегов, а вот у <HEAD> нельзя! Чтобы все-таки разместить заголовок по центру обрамим его фиктивным тегом <DIV>. Этот тег служит для выделения какой-либо части документа:

```
<HTML>
<HEAD>
<DIV ALIGN="CENTER">
Пример 1
</DIV>
</HEAD>
<BODY>
<P>
Привет этому миру!
</P>
</BODY>
</HTML>
```

получим отцентрированный заголовок. Теперь можно изменить размеры окна браузера – центрирование будет сохраняться, это приятно – значит браузер сам отслеживает изменение размеров окна и переформатирует текст.

Как видите – HTML очень простой язык. Тем более, что браузер делает все, что может, чтобы правильно отобразить документ. Исходный текст пишется в свободной форме – переход на новую строку не обрабатывается. Тот же текст можно было бы написать вообще в одну строку или выровнять так, чтобы была видна структура документа. Например, так:

```
<HTML>
```

```
<HEAD>
  <DIV ALIGN="CENTER">Пример 1</DIV>
</HEAD>
<BODY>
  <P>Привет этому миру!</P>
</BODY>
</HTML>
```

Все, что надо – выучить существующие теги, а потом их открывать и закрывать. Полный список тегов, их атрибутов, методов и событий, которые они поддерживают, приведен в [1]. Нам будут необходимы следующие теги:

<HTML>	Начинает документ HTML
<HEAD>	Начинает раздел заголовка документа. Команды, помещенные в этот раздел гарантированно выполняются до показа тела документа
<BODY>	Начинает тело документа
<TITLE>	Начинает заголовок документа, показываемый на рамке окна браузера
<H1>	Начинает заголовок 1-го уровня
<title>	Начинает заголовок 2-го уровня и т.д.
<Hn>	Начинает заголовок n-го уровня
<BIG>	Увеличивает на единицу размер текущего шрифта
<SMALL>	Уменьшает на единицу размер текущего шрифта
<B>	Выделяет текст жирным шрифтом (bold)
<U>	Подчеркивает текст (underline)
<I>	Выделяет текст курсивом (italic)
<LISTING>	Выводит текст так, как он есть, шрифтом фиксированного размера
<KBD>	Выводит текст шрифтом фиксированного размера
<emphasis>	Выделение определения
<DIV>	Начало раздела
<P>	Начало абзаца
 	Вставляет конец строки
<NOBR>	Текст выводится в одну строку. Если текст не поместится на экране, придется пользоваться горизонтальной полосой прокрутки
<BLOCKQUOTE>	Обозначает отступ в тексте
<CENTER>	Начинает центрированный текст
<OL>	Преобразует следующие строки с тегом <LI> в нумерованный список
<UL>	Преобразует следующие строки с тегом <LI> в маркированный список
<LI>	Начинает строку списка
<FORM>	Начинает форму ввода данных
<INPUT>	Описывает поле формы ввода данных
<FRAMESET>	Объявляет начало набора фреймов

<FRAME>	Объявляет фрейм – часть окна, в котором показывается другой документ HTML
<NOFRAMES>	Определяет, что будет показывать браузер, не поддерживающий фреймы
<BASE>	Устанавливает URL исходного документа
<A>	Описывает гиперссылку
<IMG>	Вставляет графический элемент (рисунок) в текст
<FONT>	Определяет вид, размер и цвет шрифта текста
<TABLE>	Начинает вывод таблицы
<TR>	Начинает новую строку таблицы
<TD>	Начинает элемент строки таблицы (ячейку)

Каждый из этих тегов поддерживает многочисленные атрибуты, методы и события.

Пример HTML страницы

В качестве примера, рассмотрим несколько HTML-страниц. Первая страница – заголовок:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<LINK REL=STYLESHEET TYPE="text/css" HREF="lessons.css">
<HEAD>
<META http-equiv="Content-Type" content="text/html" charset="windows-1251">
<TITLE>Разработка Web-приложений на языке Java - Библиотека</TITLE>
</HEAD>
<BODY>
  <TABLE WIDTH="100%">
  <TR>
  <TD WIDTH=130 ALIGN="CENTER"><IMG SRC="gifs/javalogo.gif" ALT="Это - кофе!"></TD>
  <TD ALIGN="CENTER">
  <title>Разработка Web-приложений на языке Java</title>
  <H1>Библиотека</H1>
  </TD>
  <TR>
  </TABLE>
</BODY>
</HTML>
```

Браузеры очень либерально относятся к тегам – про ошибки не сообщают – просто игнорируют ошибочные теги, про ошибки атрибутов – то же, и стараются то, что "понимают", отобразить как надо.

Вторая страница – пример составления таблицы:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<LINK REL=STYLESHEET TYPE="text/css" HREF="aqua.css">
<HEAD>
<META http-equiv="Content-Type" content="text/html" charset="windows-1251">
</HEAD>
<BODY  BGCOLOR="#FFFFFF">
<DIV ALIGN="CENTER">
<TABLE BORDER="1" WIDTH=100% class=vrTable>
<THEAD ALIGN=CENTER class=vrTableHeader>
```

```
<TR CLASS=vrTableHeaderRow>
  <TD>Автор</TD><TD>Название</TD><TD>Издание</TD>
</TR>
</THEAD>
<TBODY ALIGN=CENTER>
<TR>
  <TD>Romain Guy</TD><TD>Jext Editor</TD><TD>www.jext.org</TD>
</TR>
<TR>
  <TD>В.К.Мюллер</TD><TD>Англо-русский словарь</TD><TD>1989</TD>
</TR>
<TR>
  <TD>Elliotte Rusty Harold</TD><TD>Java I/O</TD><TD>O'Reily</TD>
</TR>
<TR>
  <TD>Elliotte Rusty Harold</TD><TD>Java Network Programming</TD><TD>O'Reily</TD>
</TR>
<TR>
  <TD>Elliotte Rusty Harold</TD><TD>Java Secrets</TD><TD>IDG Books</TD>
</TR>
<TR>
  <TD>Elliotte Rusty Harold</TD><TD>Java Beans</TD><TD>IDG Books</TD>
</TR>
<TR>
  <TD>Elliotte Rusty Harold</TD><TD>XML Bible</TD><TD>IDG Books</TD>
</TR>
<TBODY>
</TABLE>
</BODY>
</HTML>
```

Пример HTML формы

Формы используются для передачи значений на Web-сервер. Форма начинается тегом <FORM>. Форма может содержать другие компоненты и элементы управления. Тег <FORM> может иметь следующие атрибуты:

Свойство	Атрибут
action	ACTION=string
className	CLASS=string
encoding	ENCTYPE=string
event	<название события>=<название сценария>
id	ID=string
lang	LANG=string
language	LANGUAGE=JAVASCRIPT|JSCRIPT|VBSCRIPT|VBS
method	METHOD=GET|POST
name	NAME=string
style	STYLE=string
target	TARGET=<имя окна>|_parent|_blank|_top|_self
title	TITLE=string

Обычно форма состоит из фиксированного текста и полей ввода-вывода. Поля формы описываются тегом <INPUT>. Он имеет следующие атрибуты:

Свойство	Атрибут
accessKey	ACCESSKEY=string
className	CLASS=string
disabled	DISABLED
event	<название события>=<название сценария>
id	ID=string
lang	LANG=string
language	LANGUAGE=JAVASCRIPT|JSCRIPT|VBSCRIPT|VBS
maxLength	MAXLENGTH=число
name	NAME=string
size	SIZE=число
src	SRC=url
style	STYLE=string
tabIndex	TABINDEX=число
title	TITLE=string
type	TYPE=BUTTON|CHECKBOX|FILE|HIDDEN|IMAGE|PASSWORD|RADIO|RESET|SUBMIT|TEXT
value	VALUE=string

Пример формы:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<LINK REL=STYLESHEET TYPE="text/css" HREF="aqua.css">
<HEAD>
 <META http-equiv="Content-Type" content="text/html" charset="windows-1251">
</HEAD>
<BODY  BGCOLOR="#FFFFFF">
  <DIV ALIGN="CENTER">
    <FORM NAME="find" ACTION="LibraryBookList.html">
      <TABLE BORDER="1" WIDTH=100% class=vrTable>
        <THEAD ALIGN=CENTER class=vrTableHeader>
          <TR CLASS=vrTableHeaderRow>
          <TD COLSPAN=2>Введите образец для поиска в базе данных</TD>
          </TR>
        </THEAD>
        <TBODY ALIGN=LEFT>
          <TR><TD>Автор</TD><TD><INPUT TYPE="TEXT" NAME="author" SIZE=60></TD></TR>
          <TR><TD>Название</TD><TD><INPUT TYPE="TEXT" NAME="name" SIZE=60></TD></TR>
          <TR><TD>Издание</TD><TD><INPUT TYPE="TEXT" NAME="published" SIZE=60></TD></TR>
          <TR><TD>Читатель</TD><TD><INPUT TYPE="TEXT" NAME="reader" SIZE=60></TD></TR>
          <TR ALIGN="CENTER">
          <TD COLSPAN=2><INPUT TYPE="SUBMIT" VALUE=" Найти  ">
          <INPUT TYPE="SUBMIT" VALUE="Очистить"></TD>
          </TR>
        <TBODY>
```

```
      </TABLE>
    </FORM>
  </DIV>
</BODY>
</HTML>
```

Пример страницы со фреймами

Фреймы – это отдельные окна в основном окне документа. В каждом окне может быть отбражена отдельная страница HTML. Обычно окно документа делят на три фрейма: Я думаю, что Вы видели многочисленные примеры в Интернет. Кстати, тексты HTML страниц Интернета доступны для просмотра в браузере. Рассмотрим в качестве примера страницу Library.html. В ней – три фрейма. Откройте файл Library.html и посмотрите. Вот ее исходный текст:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<LINK REL=STYLESHEET TYPE="text/css" HREF="lessons.css">
<HEAD>
<META http-equiv="Content-Type" content="text/html" charset="windows-1251">
<TITLE>Разработка Web-приложений на языке Java - Библиотека</TITLE>
</HEAD>
<FRAMESET FRAMESPACING="0" FRAMEBORDER="0" ROWS="130,*">
  <FRAME NAME="header" SRC="LibraryHeader.html" SCROLLING="NO">
<FRAMESET COLS="200,*" FRAMEBORDER="1">
  <FRAME NAME="menu" SRC="LibraryMenu.html">
  <FRAME NAME="main" SRC="LibraryWelcome.html">
</FRAMESET>
</FRAMESET>
<NOFRAMES>
  <BODY>
  <P>Get a browser with frames!</P>
  </BODY>
</NOFRAMES>
</HTML>
```

В ней объявляется два набора фреймов. Первый состоит из одного фрейма – заголовка. Второй – из двух фреймов – левого меню и основной части документа.

Таблицы стилей

Указывать у каждого тега форматирования, каким шрифтом, размером и т.д. должен быть выведен текст, довольно утомительно. Еще труднее изменить вид вывода – надо отредактировать огромное число тегов во всех выводимых страницах. HTML 4.0 решает эту проблему стандартным способом – пишется некоторый файл, называемый таблицей стилей, в котором указывается, как выводить те или иные элементы. Потом эта таблица стилей используется в документе HTML. Этот подход похож на “стиль” в редакторе Word – изменив стиль мы меняем вид всех, использующих его документов. Таблицы стилей HTML называют “каскадными”, потому, что можно использовать несколько таблиц стилей для одного документа. Браузер выбирает один из стилей (или свой стиль по умолчанию) основываясь на некоторых приоритетах, описанных в [1]. Элементы таблиц стиля могут иметь более 70-и свойств. Мы рассмотрим вкратце только некоторые из них и то на примере. Стандартное расширение у файлов таблиц стилей css – cascading style sheet. Вот часть текста файла aqua.css:

```
.clsPageToolbar
{
        background:#99cccc;
        border-style:none;
```

```
        borderstyle:none;
}

.clsEditRecordButton
{
        background:#99cccc;
        border-style:none;
        borderstyle:none;
        width:100%;
}


.clsPageTitle
{
        color:#ffffff;
        background:#336699;
        background-color:#336699;
        font-family:Arial;
        font-size:large;
        fontfamily:Arial;
        fontSize:large;
}
```

Из этого примера видна структура файла и атрибуты, которые употребляются в таблицах стилей. Строка, начинающаяся с точки дает имя класса. Кроме того, объявления css могут быть записаны в виде

```
Тег {атрибут=значение ...}
```

Например, объявление css вида

```
H1 {color=white}
```

Окрашивает все заголовки уровня 1 в белый цвет.

Подключение таблиц стиля

Подключение таблицы стиля производится командой `LINK` или `STYLE`. Например,

```
<HTML>
<HEAD>
<LINK REL=STYLESHEET
      HREF="aqua.css"
</LINK>
<DIV ALIGN="center" CLASS="clsPageTitle">
Пример 1
</DIV>
</HEAD>
<BODY>
<P CLASS="clsPageContents">
Привет этому миру!
</P>
</BODY>
</HTML>
```

Выполнив этот пример, мы увидим, как изменился вид документа на экране. Теперь достаточно заменить таблицу стиля, чтобы изменился вид всех документов, где на нее есть ссылки. Конечно, ссылку HREF надо изменить, чтобы получить доступ к таблице стиля. При работе в сети вместо `"aqua.css"` можно написать URL местонахождения файла на Web-сервере. Т.о. можно обеспечить централизованное хранение таблиц стилей.

Литература

1. Алекс Хоумер, Крис Улмен, “Dynamic HTML”, “Питер”, Санкт-Петербург, 2000.

2. Спецификация HTML 4.01 [http://www.w3.org/TR/html401/]

Глава 3. Язык JavaScript

Динамический HTML

Те теги и атрибуты HTML, которые мы разобрали на предыдущем уроке, существовали уже в самых ранних версиях языка. Они обеспечивают только разметку (форматирование) документа и гиперссылки. Этого было достаточно, пока физики пересылали друг другу результаты своей работы в виде статических текстов. Использование HTML для ввода и корректировки данных ставит перед языком другие задачи. Например, необходимо контролировать правильность введенных значений, производить какие-то вычисления и пр. Другими словами, в языке не хватает того, что называется динамичностью – реального языка программирования, позволяющего что-либо изменять во время исполнения. В версии HTML 4.0 такой язык появился, и даже два – VBScript и JavaScript. VBScript (некоторое подобие Visual Basic) разработан Microsoft и поддерживается Internet Explorer'ом. JavaScript (некоторое подобие Java) является стандартом и поддерживается Internet Explorer'ом и Netscape - двумя наиболее популярными браузерами. Поэтому мы рассмотрим только JavaScript. Сам язык исключительно прост. В нем нельзя содавать собственные классы, пользоваться библиотеками, нет контроля за типами. Силу этим языкам придает объектная модель документа – новое, что появилось в так называемом "динамическом" HTML.

Объектная модель документа

Объектная модель документа состоит из иерархии объектов и коллекций (массивов). На вершине иерархии находится объект `document`.

Каждый объект этой модели обладает атрибутами (свойствами), методами и событиями. Объект `document` имеет 17 свойств, 13 методов и 17 событий. Доступ к свойствам и методам осуществляется указанием их через точку, например,

```
document.title – дает название документа, определенное в его теге <TITLE>
document.open("text.html") – открывает документ text.html
```

Доступ к элементу коллекции осуществляется указанием индекса (номера) элемента или его имени, определенного в атрибуте `NAME`:

```
document.images["MyImage"] дает ссылку на графический объект с именем MyImage.
```

Затем можно использовать свойства, методы и события полученного объекта, например, свойство `src` дает название файла, из которого загружена картинка, что эквивалентно значению атрибута `SRC` в теге `<IMG>`:

```
document.images["MyImage"].src
```

В связи со спецификой нашей задачи, наибольший интерес представляет коллекция `forms`. Формы в документе могут содержать поля, кнопки, фиксированный текст и т.д. – то есть действовать как контейнеры для других объектов. Чтобы управлять этим, каждый элемент коллекции `forms` (т.е. каждая форма) может иметь собственную коллекцию элементов `elements`, например,

```
document.forms[0].element[2] – доступ к третьему элементу первой формы страницы
document.forms[1].element["MyText"] – доступ к  элементу "MyText"″ второй формы страницы
document.forms["MyForm"].element["MyText"] – доступ к элементу "MyText"  формы "MyForm"
```

После того, как доступ к объекту получен, мы как всегда, можем использовать его свойства, методы и события. На предыдущем уроке мы определили элемент как открывающий и закрывающий тег и все, что находится между ними. Так как элемент является объектом в объектной модели документа, то можете представлять себе, что объектом является сам тег. Тем самым каждому тегу можно приписать свойства, методы и события. Этим статический HTML превращается в *динамический*. Например, тег `INPUT`, очень важный для нас, имеет 42 свойства, 12 методов и поддерживает 22 события.

Теперь о событиях. События возникают в среде Windows, например при щелкании мышкой или нажатии клавиши. В зависимости от того, где Вы, например, щелкнули мышкой, Windows решает, какой программе направить это событие. Щелчок на окне браузера передает это событие браузеру. Браузер решает, обрабатывать ему это событие или нет. Если щелчок был на панели инструментов, то будет вызыван метод, соответствующий иконке. Если щелчок произошел внутри страницы HTML, браузер передает это событие объектной модели документа, которая определяет на каком элементе произошел щелчок. Событие будет передано этому элементу. Отличительной особенностью объектной модели документа (чего нет в других языках) является последующая передача события вверх по иерархии, пока событие не достигнет вершины – объекта `document`. Этот процесс называется "прохождением событий". Мы можем вмешаться в этот процесс и написать собственные программы обработки событий (на языке JavaScript), а также разрешить или запретить дальнейшее прохождение событий. Программы пишутся как некоторые функции, выполнение которых осуществляется при возникновении событий.

Синтаксис JavaScript

Литералами в JavaScript могут быть

- строки – последовательность символов заключенная в кавычки или в апострофы.
- Числа, такие же как в java
- Булевы константы true или false
- Значение null – отсутствие данных.

Разделителями являются фигурные скобки, обрамляющие блоки операторов и точка с запятой. *Идентификаторы* в JavaScrip могут иметь длину не более 255 символов. *Ключевые слова* нельзя употреблять в качестве идентификаторов.

Язык регистрово-зависимый – большие и маленькие буквы различаются.

Переменные в JavaScript не имеют выраженного типа. Переменная принимает тип значения, которое ей присваивается, например,

```
X=55; - целое
X="55"; - строка
X='55'; - тоже строка
Y = X + 1; если X – строка, то она будет автоматически преобразована в ближайшее целое, чтобы вып
```

Все это скорее похоже на Basic, чем на Java. Зато операторы управления имеют такой же вид, как и в Java. Имеются операторы условия

```
if {
...
} else {
....
}
```

и операторы цикла

```
while (...) {
  ....
}
```

и

```
for (...;...;) {
  ...
}
```

с синтаксисом аналогичным синтаксису Java.

Примеры

Все это может быть будет понятнее, если мы выполним маленький пример. Создадим HTML страницу с полем и кнопкой. При нажатии кнопки в поле напишем "Hello, java script!". Вот текст этой страницы:

```
<HTML>
<HEAD>
<SCRIPT>
function do_fill() {
  document.forms[0].elements[0].value = "Hello, java script!";
}
</SCRIPT>
</HEAD>
<BODY>
  <FORM ACTION="">
    <INPUT TYPE=TEXT></INPUT>
    <BR>
    <INPUT TYPE=BUTTON onclick="do_fill()"></INPUT>
  </FORM>
</BODY>
</HTML>
```

Под тегом `<FORM>` создается форма с полем (тип – текст) и кнопкой (тип – кнопка). Тег `<BR>` добавлен для того, чтобы кнопка была на новой строке. В теге кнопки указывается событие `onclick` (нажатие кнопки) и сценарий обработки этого события. *Сценарий* – это название функции, которая будет выполнена при наступлении события. Сам сценарий написан на языке JavaScript. Все сценарии помещаются под тегом `<SCRIPT>`. Мы поместили сценарий в заголовок (`HEAD`), чтобы он гарантированно загрузился перед телом документа. Тег `<SCRIPT>` имеет атрибут `LANGUAGE` (язык). По умолчанию подразумевается язык JavaScript. Если бы ми писали программу на VBScript, надо было бы указать язык явно.

Функция `do_fill` заполняет поле значением. В начале мы получаем поле как объект через объектную модель:

```
document.forms[0].elements[0]
```

а потом присваиваем свойству `value` этого объекта значение строки:

```
document.forms[0].elements[0].value = "Hello, java script!";
```

Текст этого примера сознательно сделан очень коротким. Можно было бы присвоить имена всем элементам и обращаться к элементам коллекций по именам.

Усложним наш пример следующим образом. Сделаем опять поле и кнопку. При нажатии кнопки значение поля должны выбираться из возрастающей последовательности. Существует множество способов решения этой задачи. Вот один из возможных:

```
<HTML>
<HEAD>
<SCRIPT>
function do_fill() {
  return document.forms[0].elements[1].value++;
}
</SCRIPT>
</HEAD>
<BODY>
  <FORM ACTION="">
    <INPUT TYPE=TEXT NAME="field1"></INPUT>
    <INPUT TYPE=HIDDEN VALUE="0"></INPUT>
```

```
    <BR>
    <INPUT TYPE=BUTTON onclick="{field1.value=do_fill();}"></INPUT>
  </FORM>
</BODY>
</HTML>
```

В этом примере мы ввели скрытое поле (`TYPE=HIDDEN`), которое содержит последний элемент последовательности. Этот элемент наращивается на единицу функцией `do_fill()`. Событие `onclick` на кнопке выполняет сценарий, написанный прямо в теге кнопки:

```
{field1.value=do_fill();}
```

Доступ к объекту поля производится по имени. Пример демонстрирует различные формы записи. Можно писать так

```
{document.forms[0].elements[0].value=do_fill();}
```

И так

```
{document.forms[0].elements["field1"].value=do_fill();}
```

а можно и просто имя поля, так нет противоречия имен. Скрипт присваивает значению поля значение функции `do_fill()`, обеспечивая тем самым увеличение значения последовательности.

Покажем еще, как можно контролировать значение поля перед отправкой его значения на сервер. Этот пример показывает также как отказаться от передачи формы при неверном значении поля. Возьмем опять поле с кнопкой. Пусть это будет кнопка `SUBMIT` – стандартная кнопка для отправки формы на сервер. Вот текст HTML страницы (файл `HtmlForm.html`):

```
<HTML>
<HEAD>
<SCRIPT>
function do_fill() {
  if (document.forms[0].elements[0].value == "") {
    alert("Пожалуйста введите значение");
    return false;
  } else {
    alert("Спасибо");
    return true;
  }
}
</SCRIPT>
</HEAD>
<BODY>
  <FORM ACTION="" ONSUBMIT="return do_fill()">
    <INPUT TYPE=TEXT NAME="field1"></INPUT>
    <BR>
    <INPUT TYPE=SUBMIT></INPUT>
  </FORM>
</BODY>
</HTML>
```

Тег `<FORM>` имеет событие `ONSUBMIT`, возникающее при отправке формы на сервер. Мы пишем сценарий, выполняемый при этом событии:

```
return do_fill();
```

Функция возвращает `true` или `false` в зависимости от результата проверки поля формы. Если она возвращает `false`, отправка формы не производится.

Обратите внимание, когда форма отправляется на сервер, в строке Address Интернет эксплорера появля-

ется значение отправленных полей. Например, если мы введем в поле `field1` значение `11`, в этой строке будет

```
file:///D:/Projects/Course/programs/HTML/Hello.html?field1=11
```

Такой метод передачи параметров (через URL) назывется методом `"GET"`. Тег `<FORM>` имеет атрибут `METHOD`. Значение этого атрибута `"GET"` по умолчанию. Методом `GET` передают небольшое число параметров, значения которых не являются секретными. Существует другой метод передачи – `"POST"`, отправляющий значения параметров в отдельном пакете. Использование второго метода предпочтительнее, но работает он несколько дольше.

Таким образом, для программирования на JavaScript Вам необходимо знать объектную модель документа, т.е. свойства, методы и события, поддерживаемы различными объектами модели.

Литература

1. Алекс Хоумер, Крис Улмен, “Dynamic HTML”, “Питер”, Санкт-Петербург, 2000.

2.Описание языка JavaScript от Netscape. [http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/index.html]

Глава 4. XML - расширенный язык разметки

XML - расширенный язык разметки

Ограниченность HTML состоит в его фиксированном наборе тегов. Нам приходится добавлять все новые и новые теги, чтобы удовлетворить растущим потребностям. Но в результате мы опять получаем некий фиксированный язык. XML раз и навсегда решает эту проблему. Основная идея XML состоит в предварительном описании синтаксиса языка, на котором потом будет написан документ. Синтаксис определяется через правила грамматики (регулярные выражения и пр.). В результате полный XML документ состоит из трех частей, как правило, лежащих в отдельных файлах:

- Описание грамматики языка документа. Этот файл называется DTD (Document Type Definition)
- Сам документ (XML-файл), составленный по правилам, описанным в DTD
- Правила отображения документа на экране (типа наших таблиц стилей). Эти файлы называются XLS (Extended Style Sheets).

Наличие DTD позволяет составить практически любой документ - ведь грамматику задаем мы сами! Так что же, все-таки, это нам дает?

- Во-первых, это дает стандарт составления документов. Например, файлы обмена данными между подразделениями предприятия (или между различными предприятиями) – это тоже некие документы. Вместо согласования "структуры" файлов обмена достаточно согласовать DTD этих файлов
- Во-вторых, поскольку DTD описывают грамматику, можно написать универсальные программы грамматического разбора, позволяющие контролировать правильность составления документов (в нашем примере – файлов данных), а также универсальные программы обработки документов, не требующие постоянных изменений при изменении "форматов" данных
- В третьих, многие DTD уже написаны, так же как и программы грамматического разбора. Поэтому мы не будем изобретать велосипед, а просто возьмем готовые Java классы.
- Ну, а, например, в нашем приложении, мы можем получать данные из базы данных не в виде наборов данных (Result Sets), каждый раз разных для разных таблиц и полей, а XML документы с известным DTD. Тогда мы можем значительно упростить уровень представления за счет универсальной обработки XML документов вместо разнородных наборов данных.

Замечание

Последнее я рассматриваю как введение некоторых "настроек", плодотворность которых доказана опытом. Фактически DTD мы можем рассматривать как систему "настроек" для программ обработки документов, удовлетворяющих данным DTD. Например, при получении данных из базы, в DTD могут входить не только названия полей, их обязательность и пр., но и такая дополнительная информация, как существующие ограничения (первичные и вторичные ключи, условия проверки и др.) Программа уровня представления может универсальным способом обработать эту информацию.

Мне кажется, перечисленного достаточно, чтобы заняться XML.

Правильные и хорошо оформленные документы

Рассмотрим структуру XML документа. В принципе, можно составить документ XML без файла DTD и файла стиля. Более того, любой документ HTML может считаться документом XML, если

- Пролог документа (т.е. то, что идет до тега <HTML>) заменить на

`<?xml version="1.0" encoding="windows-1251" ?>`

Указание `encoding` важно - без него русские тексты не показываются.

- Всем открывающим тегам поставить в соответствие закрывающие
- Закрывающие теги должны идти в обратном порядке к открывающим, т.е. должна соблюдаться вложенность тегов. Т.о. структура xml-документа является деревом, что значительно облегчает его разбор и интерпретацию.
- Все значения атрибутов заключить в кавычки
- Все пустые теги должны иметь в конце обратную косую, например <img />
- Учесть, что XML чувствителен к регистру

Документы, удовлетворяющие этим требованиям называются *хорошо оформленными*. Т.о. любой хорошо оформленный HTML документ автоматически является XML документом.

Вот пример хорошо оформленного xml-документа (файл Library.xml):

```
<?xml version="1.0" encoding="windows-1251" ?>
<Библиотека>
  <СписокЧитателей>
    <Читатель>
      <ФИО>Лепехин А.Ф.</ФИО>
    </Читатель>
  </СписокЧитателей>
  <СписокКниг>
    <Книга>
      <Автор>Romain Guy</Автор>
      <Название>Jext Editor</Название>
      <Издательство>www.jext.org</Издательство>
      <ФайлФото>gifs/jext.gif</ФайлФото>
      <ФИОЧитателя>Лепехин А.Ф.</ФИОЧитателя>
      </Книга>
      <Книга>
        <Автор>В.К.Мюллер</Автор>
        <Название>Англо-русский словарь</Название>
        <Издательство>1989</Издательство>
        <ФайлФото>gifs/muller.gif</ФайлФото>
        <ФИОЧитателя>Лепехин А.Ф.</ФИОЧитателя>
      </Книга>
      <Книга>
        <Автор>Elliotte Rusty Harold</Автор>
        <Название>Java I/O</Название>
        <Издательство>O'Reily</Издательство>
        <ФайлФото>gifs/javaio.s.gif</ФайлФото>
      </Книга>
    </СписокКниг>
 </Библиотека>
```

Полезно этот пример просмотреть в браузере, например IE5.

Замечание

Легко представить, что этот файл получен по запросу из базы данных.

Хорошо оформленный документ называется *правильным* (или состоятельным, в оригинале – valid), если он имеет ассоциированный с ним DTD. Такой документ называется правильным, потому что мы можем проверить его правильность, как соответствие документа грамматике, описанной в DTD.

Описание DTD

Рассмотрим структуру файла DTD на примере библиотеки, содержащей списки записей о книгах и читателях. Каждый читатель имеет фамилию, имя, отчество, каждая книга - автора, название, издательство, имя файла с картинкой обложки. Описание DTD, как все документы xml, начинается строкой

```
<?xml version="1.0" encoding="windows-1251" ?>
```

Определение начинается определением корневого элемента и его дочерних элементов:

```
<!ELEMENT Библиотека (СписокЧитателей?,СписокКниг)>
```

Частота использования элементов задается специальными символами

- + означает, что элемент должен быть использован один или более раз
- ? означает, что элемент может быть использован один раз или не использован совсем
- * означает, что элемент может быть использован один или более раз
- если ничего не указано, то элемент должен быть использован один раз

Т.о. библиотека состоит из двух элементов: `(СписокЧитателей?,СписокКниг)`, причем `СписокЧитателей` может отсутствовать.

Каждый элемент может в свою очередь содержать дочерние элементы. В частности каждый элемент `СписокЧитателей` содержит один и более элементов `Читатель` с данными о ФИО

```
<!ELEMENT СписокЧитателей (Читатель*)>
<!ELEMENT Читатель (ФИО)>
```

Затем для каждого элемента указывается, какого типа данные он может содержать. Обычно это - `#PCDATA` - обозначающий обычный текст:

```
<!ELEMENT ФИО (#PCDATA)>
<!ELEMENT Автор (#PCDATA)>
<!ELEMENT Название (#PCDATA)>
<!ELEMENT Издательство (#PCDATA)>
<!ELEMENT ФайлФото (#PCDATA)>
<!ELEMENT ФИОЧитателя (#PCDATA)>
```

Итак, окончательный вид файла DTD следующий:

```
<?xml version="1.0" encoding="windows-1251" ?>
<!ELEMENT Библиотека (СписокЧитателей?,СписокКниг)>
<!ELEMENT СписокЧитателей (Читатель*)>
<!ELEMENT Читатель (ФИО)>
<!ELEMENT СписокКниг (Книга*)>
<!ELEMENT Книга (Автор,Название,Издательство,ФайлФото,ФИОЧитателя?)>
<!ELEMENT ФИО (#PCDATA)>
<!ELEMENT Автор (#PCDATA)>
<!ELEMENT Название (#PCDATA)>
<!ELEMENT Издательство (#PCDATA)>
<!ELEMENT ФайлФото (#PCDATA)>
<!ELEMENT ФИОЧитателя (#PCDATA)>
```

Подключение DTD к документу обеспечивается командой

```
<!DOCTYPE Библиотека SYSTEM "Library.dtd" >
```

Ключевое слово `DOCTYPE` указывает на использование DTD, ключевое слово `SYSTEM` говорит, что DTD

находится во внешнем файле. Определение DTD может быть включено непосредственно в файл xml. В этом случе оно пишется внутри тега DOCTYPE в квадратных скобках:

```
<?xml version="1.0" encoding="windows-1251" ?>
<!DOCTYPE Библиотека [
<!ELEMENT Библиотека (СписокЧитателей?,СписокКниг)>
<!ELEMENT СписокЧитателей (Читатель*)>
<!ELEMENT Читатель (ФИО)>
<!ELEMENT СписокКниг (Книга*)>
<!ELEMENT Книга (Автор,Название,Издательство,ФайлФото,ФИОЧитателя?)>
<!ELEMENT ФИО (#PCDATA)>
<!ELEMENT Автор (#PCDATA)>
<!ELEMENT Название (#PCDATA)>
<!ELEMENT Издательство (#PCDATA)>
<!ELEMENT ФайлФото (#PCDATA)>
<!ELEMENT ФИОЧитателя (#PCDATA)>
]>
<Библиотека>
  <СписокЧитателей>
    <Читатель>
      <ФИО>Лепехин А.Ф.</ФИО>
    </Читатель>
  </СписокЧитателей>
  <СписокКниг>
    <Книга>
      <Автор>Romain Guy</Автор>
      <Название>Jext Editor</Название>
      <Издательство>www.jext.org</Издательство>
      <ФайлФото>gifs/jext.gif</ФайлФото>
      <ФИОЧитателя>Лепехин А.Ф.</ФИОЧитателя>
      </Книга>
      ..............
    </СписокКниг>
 </Библиотека>
```

Файлы xml могут быть просмотрены в Internet Explorere 5.0 без задания таблицы стилей. IE5.0 использует свою таблицу стилей, определенную по умолчанию. Все элементы дерева раскрыты и отмечены знаком "-". При желании можно свернуть некоторые элементы, щелкнув на этом знаке (тогда он будет преобразован в "+").

У каждого тега могут быть дополнительно заданы некоторые атрибуты. Например, тег <Книга> может содержать атрибут ID:

```
<Книга ID="ISN001">
```

Спецификация атрибутов в DTD дается тегом ATTLIST:

```
<!ATTLIST Книга ID #IMPLIED>
```

где `#IMPLIED` означает необязательность атрибута. Если наличие атрибута обязательно, пишется `#REQUIRED`.

Использование файлов стиля

Фалы стиля (xsl-файлы) задают правила отображения, а в общем случае - преобразования, xml-файла. Напишем файл стиля для данного xml-файла, чтобы отобразить его в виде таблицы, как это было сделано в примере на HTML. Файл стиля подключается к xml-файлу командой

```
<?xml-stylesheet type="text/xsl" href="LibraryBookList.xsl" ?>
```

Вот файл LibraryBookList.xsl:

```
<?xml version="1.0" encoding="windows-1251"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
  <xsl:apply-templates select="Библиотека"/>
</xsl:template>
<xsl:template match="Библиотека">
  <xsl:apply-templates select="СписокКниг"/>
</xsl:template>

<xsl:template match="СписокКниг">
<HTML>
<!--DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"-->
<LINK REL="STYLESHEET" TYPE="text/css" HREF="aqua.css"/>
<HEAD>
<META http-equiv="Content-Type" content="text/html" charset="windows-1251"/>
</HEAD>
<BODY  BGCOLOR="#FFFFFF">
<DIV ALIGN="CENTER">
<TABLE BORDER="1" WIDTH="100%" class="vrTable">
<THEAD ALIGN="CENTER" class="vrTableHeader">
<TR CLASS="vrTableHeaderRow">
  <TD>Автор</TD><TD>Название</TD><TD>Издание</TD>
</TR>
</THEAD>
<TBODY ALIGN="CENTER">
  <xsl:apply-templates/>
</TBODY>
</TABLE>
</DIV>
</BODY>
</HTML>
</xsl:template>


<xsl:template match="Книга">
<TR>
<TD><xsl:value-of select="Автор"/></TD>
<TD><xsl:value-of select="Название"/></TD>
<TD><xsl:value-of select="Издательство"/></TD>
</TR>

</xsl:template>

</xsl:stylesheet>
```

Файл стиля состоит из правил обработки тегов, выбираемых оператором

```
<xsl:template match="Имя тега">
```

Если правило содержит оператор

```
  <xsl:apply-templates />
```

то к содержимому элемента рекурсивно применяются все правила (или правила, отбираемые предложе-

нием `select`).

Включив использование этого файла в Library.xml и открыв этот файл в IE5, получим тот же вид, как и в примере на HTML.

Пространства имен

Совместное употребление разных XML документов может породить коллизии имен. Для разрешения этой проблемы введено понятие *пространства имен*. Пространство имен XML - это идентифицируемая с помощью ссылки (URI) коллекция имен, используемых в XML документах для обозначения типов элементов и именования атрибутов. Сама ссылка при этом не подразумевает наличия какого-либо документа по этой ссылке, более того, это может быть несуществующий адрес - это просто способ указать полное имя!.

Пространство имен вводится с помощью префикса `xmlns`

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl"/>
```

После такого определения, любой элемент, имеющий префикс `xsl` будет относиться к данному пространству имен `http://www.w3.org/TR/WD-xsl`. В частности, элемент `stylesheet` сам относится к данному пространству имен.

Префиксы пространств имен `xmlns` и `xml` зарезервированы. При этом префикс `xml` связан по определению с пространством имен `http://www.w3.org/XML/1998/namespace`, а `xmlns` используется только для связывания пространств имен и сам не связан ни с каким именем.

Если двоеточие и идентификатор пространства имен опущены, например

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/providers/java"/>
```

то такое пространство имен будет пространством имен по умолчанию В файле стиля присутствуют элементы без префикса. В этом случае они относятся к пространству имен по умолчанию, а если оно не задано, как в нашем примере, то к данному документу.

Для элементов HTML определено свое пространство имен:

```
"http://www.w3.org/TR/REC-html40">
```

т.о. предыдущий файл стиля можно было бы записать так:

```
<?xml version="1.0" encoding="windows-1251"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl"
    xmlns="http://www.w3.org/TR/REC-html40">
<xsl:template match="/">
  <xsl:apply-templates select="Библиотека"/>
</xsl:template>
<xsl:template match="Библиотека">
  <xsl:apply-templates select="СписокКниг"/>
</xsl:template>

<xsl:template match="СписокКниг">
<HTML>
<!--DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"-->
<LINK REL="STYLESHEET" TYPE="text/css" HREF="aqua.css"/>
<HEAD>

  .......
```

```
<TR>
<TD><xsl:value-of select="Автор"/></TD>
<TD><xsl:value-of select="Название"/></TD>
<TD><xsl:value-of select="Издательство"/></TD>
</TR>
</HEAD>
</HTML>
</xsl:template>
</xsl:stylesheet>
```

В последнем примере все имена без префикса относятся к пространству имен `http://www.w3.org/TR/REC-html40`, а имена с префиксом `xsl` относятся к пространству имен `http://www.w3.org/TR/WD-xsl`.

XML схема

DTD дает слишком простые правила проверки документа. Для введения более сложных правил была специфицирована *XML схема*, заменяющая DTD. С помощью XML схемы можно ввести практически любые ограничения на элементы и атрибуты. Например, описание в DTD

```
<!ELEMENT ФИО (#PCDATA)>
```

ничего не говорит об ограничениях на поле ФИО, в частности не запрещает писать в нем цифры. При описании этого поля в схеме можно указать не только то, что ФИО должно быть буквенным, но и обязать начинать ФИО с большой буквы:

```
<simpleType name="ФИО" base="string">
    <pattern value="[А-Я][а-я]*"/>
</simpleType>
```

Описание XML схемы само является XML документом и может использовать механизм пространств имен, рассмотренный выше. При этом, имена, определенные в XML схеме, принадлежат некоторому *целевому* пространству имен (target namespace). Каждая схема имеет одно целевое пространство имен и может использовать несколько других пространств для определения своих элементов. Например:

```
<xsd:schema targetNamespace="http://com.fors.library/schema"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    xmlns:lib="http://com.fors.library/schema">

    <xsd:element name="Название" type="xsd:string"/>
    <xsd:element name="ФИО" type="lib:person"/>

    <xsd:simpleType name="person" base="xsd:string">
            <xsd:pattern value="[А-Я][а-я]*"/>
    </xsd:simpleType>
</xsd:schema>
```

В этом примере имена "Название", "ФИО" и "person" принадлежат к target namespace.

Следует еще раз напомнить, что URI пространств имен являются только их именами, а не реальными ссылками. Чтобы указать на файл с описанием схемы, используют атрибут `schemaLocation`, например:

```
<xsd:schema targetNamespace="http://com.fors.library/schema"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
```

```
    xmlns:book="http://com.fors.library/book"
    xmlns:lib="http://com.fors.library/schema">

<xsd:import  namespace="http://com.fors.library/book"
    schemaLocation="http://com.fors.library/book/resorces/book.xsd"/>

    <xsd:element name="Название" type="xsd:string"/>
    <xsd:element name="ФИО" type="lib:person"/>

    <xsd:simpleType name="person" base="xsd:string">
            <xsd:pattern value="[А-Я][а-я]*"/>
    </xsd:simpleType>
</xsd:schema>
```

Для указания расположения файлов xsd в самом xml-документе используют атрибут `schemaLocation` из пространства имен "`http://www.w3.org/1999/XMLSchema-instance`":

```
<?xml version="1.0" encoding="windows-1251" ?>
<Библиотека xmlns="http://com.fors.library/schema"
    xmlns:book="http://com.fors.library/book"
    xmlns:lib="http://com.fors.library/schema"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xsi:schemaLocation="http://com.fors.library/book
                        http://com.fors.library/book/resorces/book.xsd
                        http://com.fors.library/schema
                        http://com.fors.library/schema/resorces/library.xsd">
  <СписокЧитателей>
    <Читатель>
      <ФИО>Лепехин А.Ф.</ФИО>
    </Читатель>
  </СписокЧитателей>
  <СписокКниг>
    <book:Книга>
      <book:Автор>Romain Guy</book:Автор>
      <book:Название>Jext Editor</book:Название>
      <book:Издательство>www.jext.org</book:Издательство>
      <book:ФайлФото>gifs/jext.gif</book:ФайлФото>
      <book:ФИОЧитателя>Лепехин А.Ф.</book:ФИОЧитателя>
      </book:Книга>
      ..............
    </СписокКниг>
 </Библиотека>
```

XML схема служит для описания типов элементов. Тип называется простым, если определяемый элемент не содержит атрибутов или других элементов. Другими словами, элемент простого типа может иметь только значение. Простой тип может быть либо предопределенным простым типом (`string, integer, decimal, time` и др.), либо произведенным простым типом, выведенным из предопределенного типа с наложенными ограничениями. Сложный тип определяет имена и типы атрибутов, а также вложенные элементы. Подробный пример XML схемы см. в [5].

Расширенный язык разметки гипертекста (XHTML)

Поскольку XML может полностью заменить HTML, развитие HTML прекратилось на версии 4.01. На смену HTML пришел "расширенный язык разметки гипертекста" - XHTML. XHTML фактически является переформулировкой HTML в терминах XML, тем самым позволяя [3]:

- отдельно описывать структуру данных без привязки к способу отображения;
- определять форму представления данных независимо от конкретного содержания;
- создавать метаданные и управлять доступом к данным;
- преобразовывать содержание и структуру данных;
- согласовывать практику применения данных с национальным законодательством;
- описывать стандартные коммуникативные форматы данных, принятые в различных предметных областях.

Основным направлением развития XHTML является его модуляризация с целью обслуживания не только пользователей персональных компьютеров, но и мобильных устройств, телевизоров и встроенных процессоров. Более полную и современную информацию можно найти в [4] в разделе "What's new...".

Литература

1.Сандра Э.Эдди "XML справочник" Питер, Санкт-Петербург, 1999

2. Elliotte Rusty Harold. Chapter 14 of the XML Bible: XSL Transformation [http://www.ibiblio.org/xml/books/bible/updates/14.html].

3. Е.Булах и др. "Развитие стека спецификаций W3C или гносеология XML"

4. The XML Cover Pages http://www.oasis-open.org/cover/sgml-xml.html

5. Неформальное описание XML схемы можно найти на http://www.w3.org/TR/xmlschema-0/.

6. Русские переводы спецификаций W3C можно найти на сайте http://croll.hotbox.ru/W3C/Consortium/Translation/russian.html

Глава 5. Web-сервер из J2EE

Java 2 Enterprise Edition (J2EE)

Итак, мы умеем писать HTML и XML документы. Раз так, установим их для всеобщего пользования под настоящим Web-сервером. В качестве сервера может быть взят любой Web-сервер, но мы выберем Web-сервер, поддерживающий сервлеты и Java Server Pages, т.е. имеющий встроенную виртуальную машину Java. Кроме того, мы должны иметь сервер приложений для выполнения Enterprise Java Beans (EJB) (см. урок "Архитектура Web-приложений"). Другими словами, мы должны обеспечить серверы уровня представления и бизнес-логики. Наличие открытых стандартов на интерфейсы данных серверов дает возможность их независимой разработки "третьими" фирмами. Поэтому мы можем выбрать между той или иной реализацией. Большинство этих реализаций имеет коммерческий характер, т.е. требует оплаты и лицензирования. Sun дает бесплатную "справочную" реализацию стандарта, известную как Java 2 Enterprise Edition (J2EE). Эта реализация *не является промышленной версией* и не предназначена для промышленной эксплуатации. Тем не менее для нас это - как раз то, что нужно. Мы можем использовать J2EE для обучения и демонстрации всех черт "enterprise" приложения. Для промышленной эксплуатации должна быть выбрана одна из коммерческих реализаций. Oracle поставляет серверы сервлетов, JSP и EJB, включеные в ядро с сервера баз данных Oracle 8i. Поэтому, если мы будем работать на Oracle, у нас все это будет.

Установка J2EE

В состав J2EE (j2ee.jar) входят: сервер приложений, реляционная СУБД, Web-сервер с возможностью работы по защищенным соединениям, инструментальные средства и Java API для разработки приложений. Чтобы пользоваться всем этим достаточно установить J2EE в некоторый каталог и установить переменные окружения JAVA_HOME и J2EE_HOME.

Установка приложения в J2EE

Обычно Web-приложения после изготовления передаются на установку в виде "Web-архива" - одного файла с расширением war. Установка такого файла на платформу J2EE производится инструментальным средством (программой) установки (deployment tool). Запуск этой программы в J2EE производится командным файлом deploytool.bat. Мы хотим просто посмотреть наши HTML страницы под Web-сервером, пока без использования сервлетов, JSP и EJB. Для этого достаточно их просто скопировать в корневой каталог Web-сервера. Перенесем туда всю папку course.

Запустив Интернет-эксплорер и набрав адрес

http://afl:8000/course/lessons/examples/html/Library.html

получим основные экранные формы нашей библиотеки. При выполнении этого примера опять же слово `afl` необходимо заменить на имя Вашего компьютера. Как видите, доступ к `HTML`-странице осуществляется по протоколу `http`, т.е. через Web-сервер. При обычном открытии файла Интернет-эксплорером доступ к файлу осуществляется по протоколу `file`. Теперь можно запустить Интернет-эксплорер с любого другого компьютера локальной сети и получить тот же результат.

Аналогично можно получить теперь доступ к содержанию данного курса, набрав адрес

http://afl:8000/course/lessons/lessons.html

Интересно также просто набрать адрес

http://afl:8000/course/lessons

Вы получите все содержимое данной папки.

Что дальше?

То, что мы отобразили в примере "Библиотека" - это статические тексты. Необходимо их сделать динамическими. Наши HTML-страницы должны взаимодействовать с Web-сервером, принимая и посылая ему данные. Динамическое формирование HTML-страниц и прием отсылаемых из HTML-страниц данных осуществляется *сервлетом* - Java-программой, работающей на Web-сервере. Чтобы писать сервлеты, необходимо знать язык Java. А перед изучением Java полезно знать общие принципы объектно-ориентированого программирования.

Глава 6. Сервлеты

Сервлеты

Сервлет – это Java программа, работающая на Web-сервере и реализующая уровень представления. Сервлет генерирует и отправляет браузеру клиента HTML страницы, а также обрабатывает параметры, пришедшие на сервер после отправки формы из браузера. Сейчас мы разберемся, как он это делает.

Во-первых, наш сервлет должен наследоваться из класса `javax.servlet.http.HttpServlet`. Этот класс делает всю черновую работу, обеспечивая взаимодействие сервлета и браузера по протоколу `http`.

Во-вторых, мы должны предоставить методы `doGet` или `doPost` – в зависимости от того, каким методом мы отправляем форму на сервер. В этих методах есть два параметра: `request` и `response` с типами `HttpServletRequest` и `HttpServletResponse` соответственно:

```
HttpServletRequest request, HttpServletResponse response
```

Эти параметры дают потоки ввода-вывода сервлета. Поскольку методы `doGet` и `doPost` вызываются откуда-то изнутри класса `HttpServlet`, значения этих параметров уже установлены и мы их просто можем использовать. Вначале разберемся с `response`. Когда сервлет запускается, вызывается метод `doGet`. Давайте оформим HTML страницу HtmlForm.html [JavaScript.html#HtmlForm] из урока по JavaScript в виде сервлета. Для этого напишем метод `doGet`, выводящий эту страницу.

Вот полный текст простейшего сервлета:

```
//-- Файл SimpleServlet.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = new PrintWriter (response.getOutputStream());
    String s1 = request.getParameter("field1");
    String s =
"<HTML>"
+"<HEAD>"
+"<SCRIPT>"
+"function do_fill() {"
+"  if (document.forms[0].elements[0].value == \"\") {"
+"    alert(\"Пожалуйста введите значение\");"
+"    return false;"
+"  } else {"
+"    alert(\"Спасибо\");"
+"    return true;"
+"  }"
+"}"
+"</SCRIPT>"
+"</HEAD>"
+"<BODY>"
+"<FORM ACTION=\"\" ONSUBMIT=\"return do_fill()\">"
+"<INPUT TYPE=TEXT NAME=\"field1\"></INPUT>"
+"<BR></BR>"
+"<INPUT TYPE=SUBMIT></INPUT>"
```

```
+"</FORM>"
+"<P>Получено: "+s1
+"</BODY>"
+"</HTML>";
    out.println(s);
    out.close();
}
}
//--
```

Мы видим, что `doGet` должен быть объявлен с возможными исключениями `ServletException`, `IOException`. Эти классы лежат в библиотеках `javax.servlet` и `java.io`, соответственно. Поэтому эти пакеты импортируются. Внутри метода вначале производится подготовка к выводу: оператор

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

устанавливает, что будет выводиться текст HTML страницы. Следующий оператор получает выходной поток:

```
PrintWriter out = new PrintWriter (response.getOutputStream());
```

Это - стандартное начало всех `HttpServlet`'ов. Потом мы получаем строку `s1`, содержащую значение переданного параметра и формируем строку `s`, содержащую текст HTML страницы. Понятно, раз мы ее формируем в Java программе, то в нее можно подставить значения любых Java переменных и вообще наворотить всякой логики на ее формирование. Поэтому текст страницы слегка изменен: внизу формы мы пишем "Получено:" и выводим значение переменной `s1`, полученной сервлетом из формы. Значение этой переменной получается из входного параметра `request`.

```
    String s1 = request.getParameter("field1");
    String s =
"<HTML>"
+"<HEAD>"
.......
+"</HTML>";
```

Потом строка `s` выводится в выходной поток, т.е. отправляется в браузер клиента, запросившего этот сервлет.

```
    out.println(s);
```

Потом выходной поток закрывается:

```
    out.close();
```

Вот и все.

Установка и выполнение сервлета

Вначале транслируем сервлет с помощью javac. Получим класс SimpleServlet.class. Чтобы установить сервлет в J2EE, запускаем `deploytool.bat` из каталога %J2EE_HOME%/bin. Создаем новое приложение SimpleAppl и в нем новую Web-компоненту, куда включаем класс SimpleServlet.class. Alias для сервлета дадим SimpleServlet и контекстный путь (при установке приложения) - course/example. Если в браузере задать путь

```
http://afl:8000/course/example/SimpleServlet
```

то мы получим требуемую HTML-страницу.

Глава 7. Шаблоны проектирования

Шаблоны проектирования

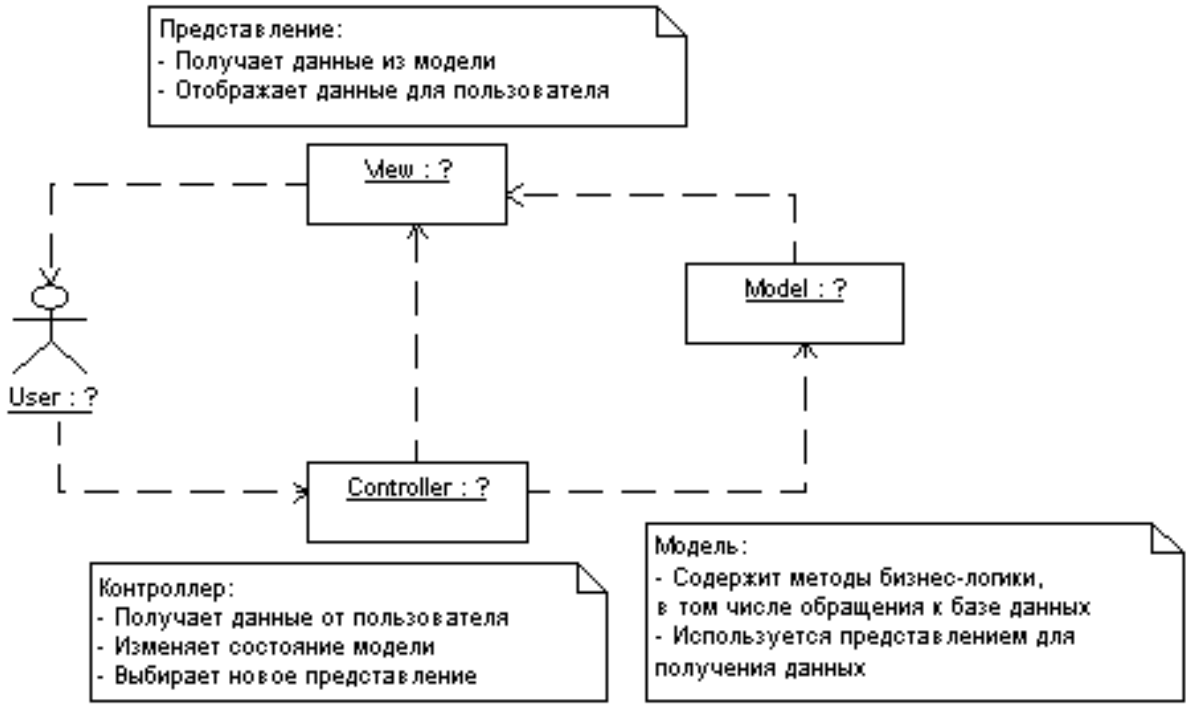
Прежде, чем начинать писать программы на Java, необходимо начать мыслить в объектах. Это достаточно трудно, особенно людям, привыкшим к процедурному мышлению. В объектно-ориентированном мире существует много типовых способов решения определенных задач. "Банда четырех" (Gamma, Helm, Johnson и Vlissides) в 1995 году опубликовала 23 шаблона решения типовых задач на C++ в книге "Design Patterns - Elements of Reusable Software" (книга "GoF"). С тех пор число шаблонов значительно выросло. В применении к Java эти шаблоны опубликованы в книге Джеймса Купера "Design Patterns Java Companion" [designjava.pdf]. Вы должны обязательно познакомиться с этой книгой. Как в ней сказано, "есть счастливые люди, для которых шаблоны проектирования являются очевидными. Для остальных из нас требуется долгий период осознания, за которым следует восклицание, типа "Ага!", когда становится ясно, как их применить в своей работе". Здесь мы рассмотрим только несколько наиболее употребительных шаблонов.

Шаблон "Model-View-Controller"

Шаблон "модель-представление-контроллер" известен с конца 80-х годов. Он разделяет проблему пользовательского интерфейса на три части:

- *представление*, занимающееся только отображением данных пользователю,
- *контроллер*, которому направляется весь ввод пользователя
- *модель* - часть, содержащая бизнес-логику задачи и, возможно, взаимодействующая с базой данных.

Рисунок 7.1. Шаблон Model-View-Controller

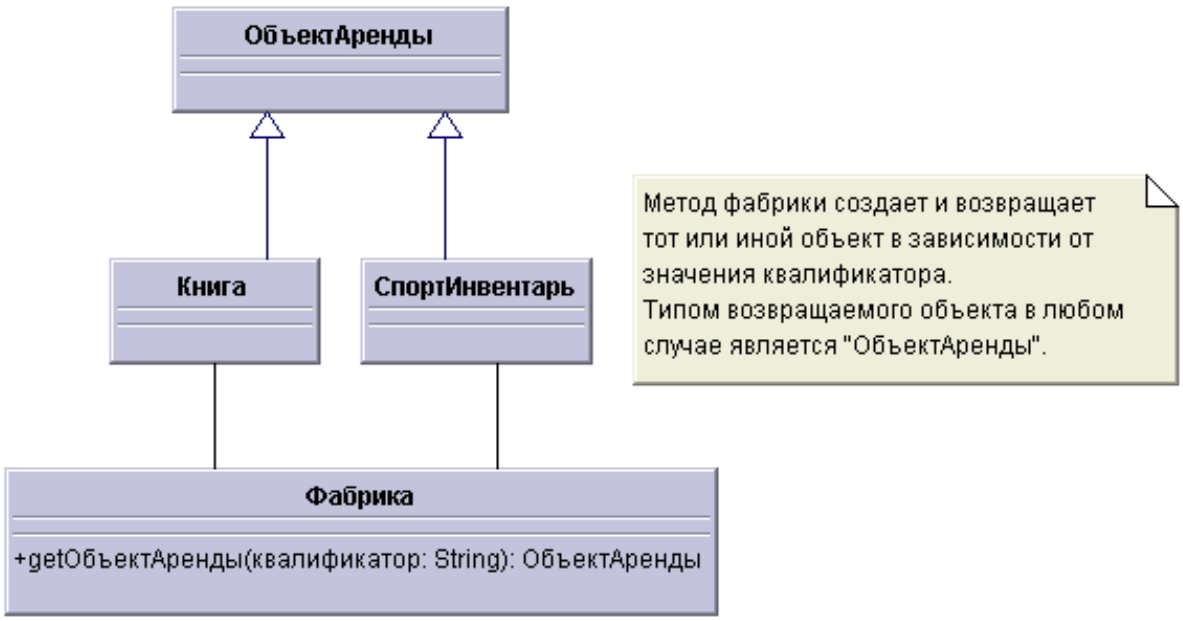


Представление и контроллер реализуются на уровне представления [architect.html#platforma] (на Web-сервере), а модель - на уровне сервера приложений [architect.html#platforma] (бизнес-логики). Этот шаблон является основным в проектировании приложений на платформе J2EE.

Шаблон фабрики

Шаблон фабрики относится к группе шаблонов создания объектов. Обычно фабрика служит для создания экземпляра одного из набора классов, имеющих общий родительский класс и одинаковые методы. Например, мы можем обобщить выдачу книг из библиотеки до аренды произвольных объектов. Приложение, манипулирующее объектами аренды, будет правильно работать с любыми объектами, имеющими интерфейс "объект аренды". При реальной работе мы должны выбрать все же, с какими объектами мы имеем дело. Создание реального объекта аренды может быть произведено фабрикой (рис 9_2).

Рисунок 7.2. Пример шаблона фабрики



Приложение, использующее фабрику, не зависит от классов - потомков суперкласса, поэтому оно не требует изменений при добавлении новых объектов аренды.

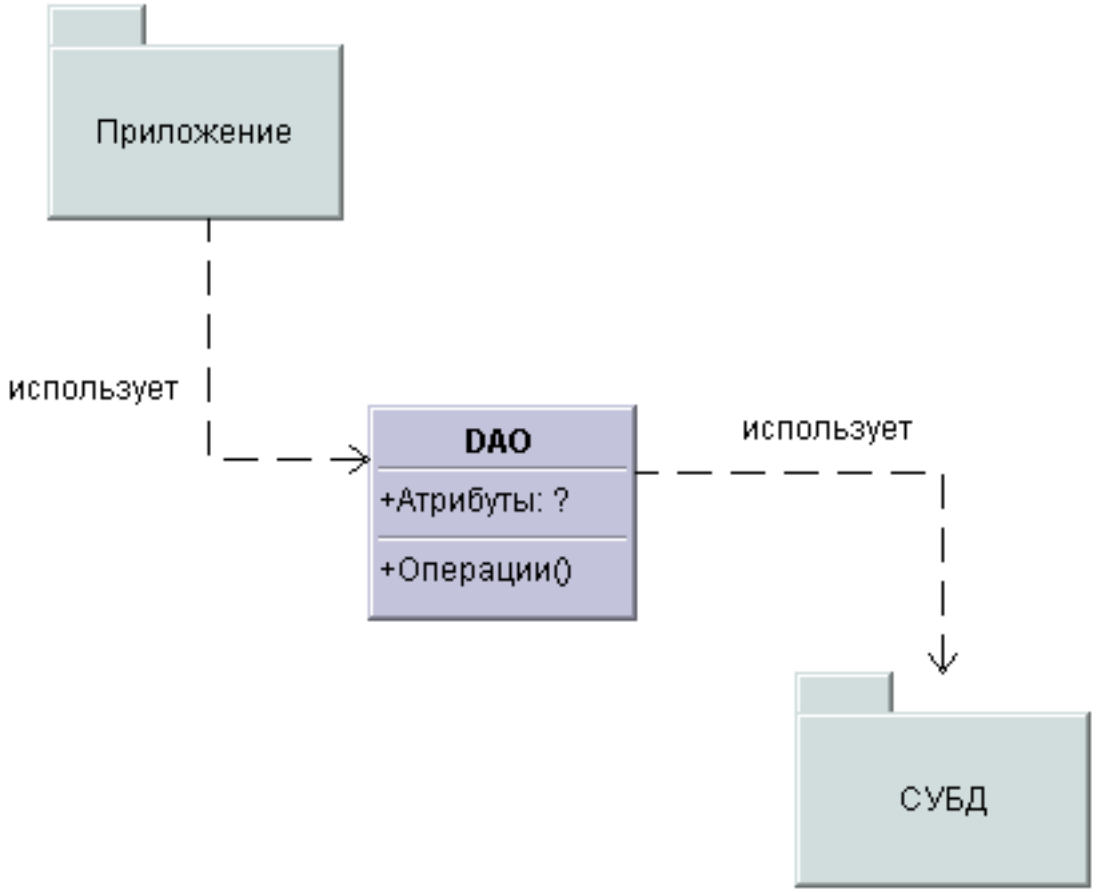
Как мы увидим далее, шаблон фабрики применяется при создании объектов типа Enterprise Java Beans. Часто метод получения объекта объявляется статическим, т.о. позволяя получить требуемый объект без создания объекта фабрики. Обычно шаблон фабрики употребляется в следующих случаях:

- Класс не может предвидеть, объекты какого класса он должен создать
- Есть желание локализовать решение о создании объекта того или иного класса
- Базовый класс является абстрактным, а шаблон должен вернуть полностью работающий класс
- Базовый класс содержит методы "по-умолчанию", а подклассы используются тогда, когда эти методы не достаточны
- Методы подклассов совпадают с базовым классом, но в подклассах выполняются по-разному

Шаблон объекта доступа к данным

Схожим с фабрикой является шаблон объекта доступа к данным (DAO - Data Access Object). Мотивом введения такого шаблона служит желание абстрагироваться от методов хранения данных, в частности от поставщика СУБД и типа доступа. DAO как правило содержит методы манипулирования данными (сохранения и восстановления из базы данных), а также связанные с ним методы бизнес логики. Остальная часть приложения "видит" данные только через DAO, и поэтому не содержит методов доступа к базе данных. Поскольку эти методы локализованы в DAO, они легко могут быть изменены при смене СУБД (рис 9_3).

Рисунок 7.3. Шаблон объекта доступа к данным



Шаблон может быть применен к данным, получаемым и другим путем, например из xml-файлов.

Замечание

Вообще, когда мы хотим сделать две подсистемы относительно независимыми, мы вводим третью подсистему между ними. Если интерфейсы взаимодействия с третьей подсистемой определены, то изменение одной подсистемы не окажет влияния на другую подсистему. DAO может служить примером такого подхода.

В отличии от фабрики, DAO переписывается всякий раз, когда меняется СУБД. Подразумевается, что это происходит редко.

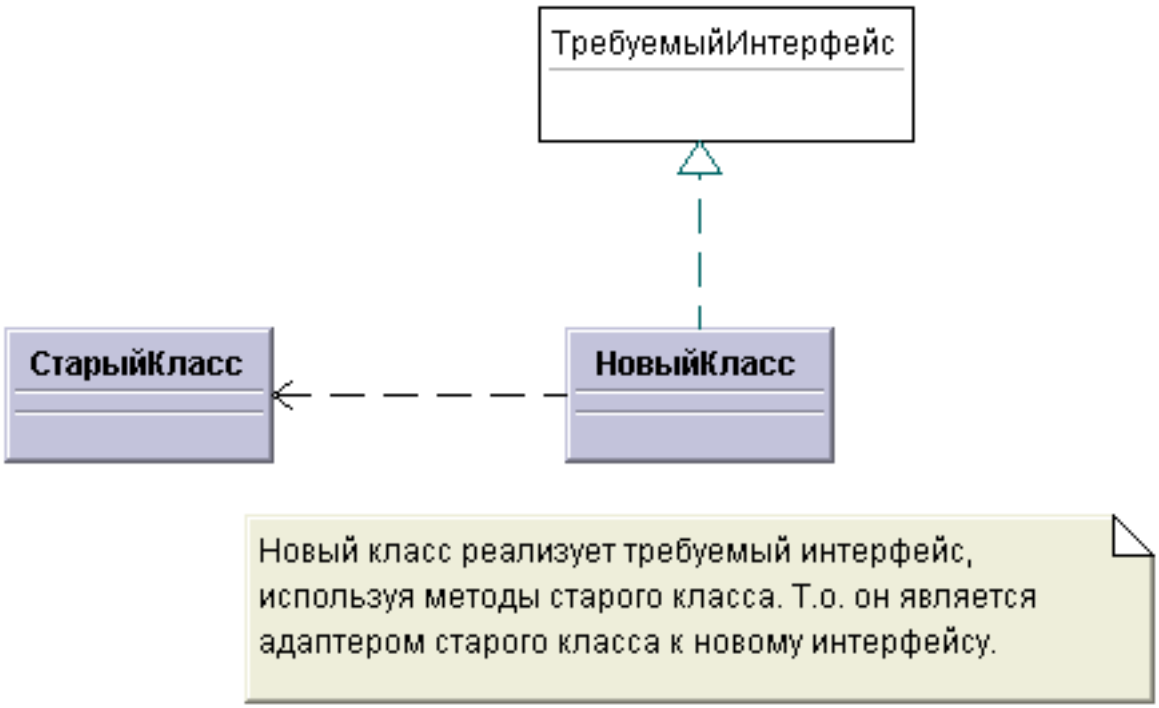
Шаблон адаптера

Адаптер используется для преобразования интерфейса одного класса в интерфейс другого класса. Старый класс без изменений может быть "подцеплен" к задаче, использующей другой интерфейс. Существует два способа реализации данного шаблона:

- Наследовать новый класс из старого и дописать недостающие методы
- Включить старый класс внутрь нового и создать методы для трансляции вызовов

Эти два подхода называются адаптером класса и адаптером объекта соответственно.

Рисунок 7.4. Шаблон адаптера объекта



Примером адаптера, реализованного в Java, является `WindowAdapter`, содержащий пустые реализации методов для событий окна. Если мы хотим, например, закрыть приложение при закрытии окна, достаточно перекрыть метод `windowClosing()`:

```
  //create window listener for window close click
  addWindowListener(
    new WindowAdapter() {
      public void windowClosing(WindowEvent e) {System.exit(0);}
    }
  );
```

Другими аналогичными адаптерами являются `ComponentAdapter`, `ContainerAdapter`, `FocusAdapter`, `KeyAdapter`, `MouseAdapter` и `MouseMotionAdapter`.

Шаблон моста

Мост служит для отделения интерфейса от его реализации. Клиентские программы, использующие данный интерфейс, в этом случае не изменяются при изменении реализации интерфейса. В шаблоне DAO [#dao] объект DAO с точки зрения приложения является мостом. Приложение использует интерфейс DAO, который не изменяется при изменении СУБД.

Шаблон фасада

Обычно системы состоят из подсистем. Каждая такая подсистема в свою очередь может быть сложной системой. Чтобы упростить представление подсистемы с точки зрения использующих ее подсистем служит шаблон фасада. В результате мы видим "фасад" подсистемы, скрывающий ее сложность и дающий нам только необходимый интерфейс. Например, для работы с базами данных в Java определено много сложных классов, обеспечивающих работу с любыми СУБД. Из контекста нашей задачи нам может быть не обязательно использовать все возможности, предоставляемые данными классами. Поэтому мы пишем "фасад", дающий только необходимые методы и упрощающий программирование доступа к базам данных. В качестве примера я приведу шаблон фасада, использованный мной в реальной работе. Определим класс `Database` с конструктором, принимающим тип используемого драйвера:

```
  public Database(String driver) throws DatabaseException;
```
Т.о., создавая объект Database с драйвером

```
  "oracle.jdbc.driver.OracleDriver"
```
мы получаем доступ к СУБД Oracle, а взяв драйвер

```
  "sun.jdbc.odbc.JdbcOdbcDriver"
```
мы получим доступ к источникам данных ODBC, например к dbf- файлам. Для получения связи с базой данных определим метод `open()`:

```
  public void open(String url, String user, String passwd) throws DatabaseException;
```
где url - строка связи с базой данных,

user - имя пользователя

passwd - пароль доступа к БД

После установления соединения можно выдавать запросы к БД. Такими запросами как правило являются:

- DML предложения (select, insert, update, delete)
- вызовы пакетных функций и процедур
- команды rollback и commit.

Для работы с запросами определим интерфейс типичного запроса:

```
public interface QueryI {

    public Statement execute() throws DatabaseException;
    public void setParameter(int index, Object value) throws DatabaseException;
    public void setOutParameterType(int index, int sqlType) throws DatabaseException;
    public void close() throws DatabaseException;
    public String toString();

}
```
Создание нового запроса производится методом класса Database:

```
  public Query newQuery(String s) throws DatabaseException;
```
где string - параметризованный или непараметризованный запрос к БД, вызов функции или процедуры.

Класс Database имеет также методы `rollback()`, `commit()` и `close()`. Вот и весь фасад. Полная его реализация находится в пакете com.fors.database.

Теперь, чтобы выполнить, например, такое предложение

```
  update emp set salary = 20000 where ename = 'SMITH'
```
мы пишем

```
    Database db = new Database("oracle.jdbc.driver.OracleDriver");
    db.open("jdbc:oracle:thin:@fors_pds:1521:805", "scott", "tiger");
    QueryI q = db.newQuery("update emp set salary = ? where ename = ?");
    q.setParameter(1,20000);
    q.setParameter(1,"SMITH");
    q.execute();
```

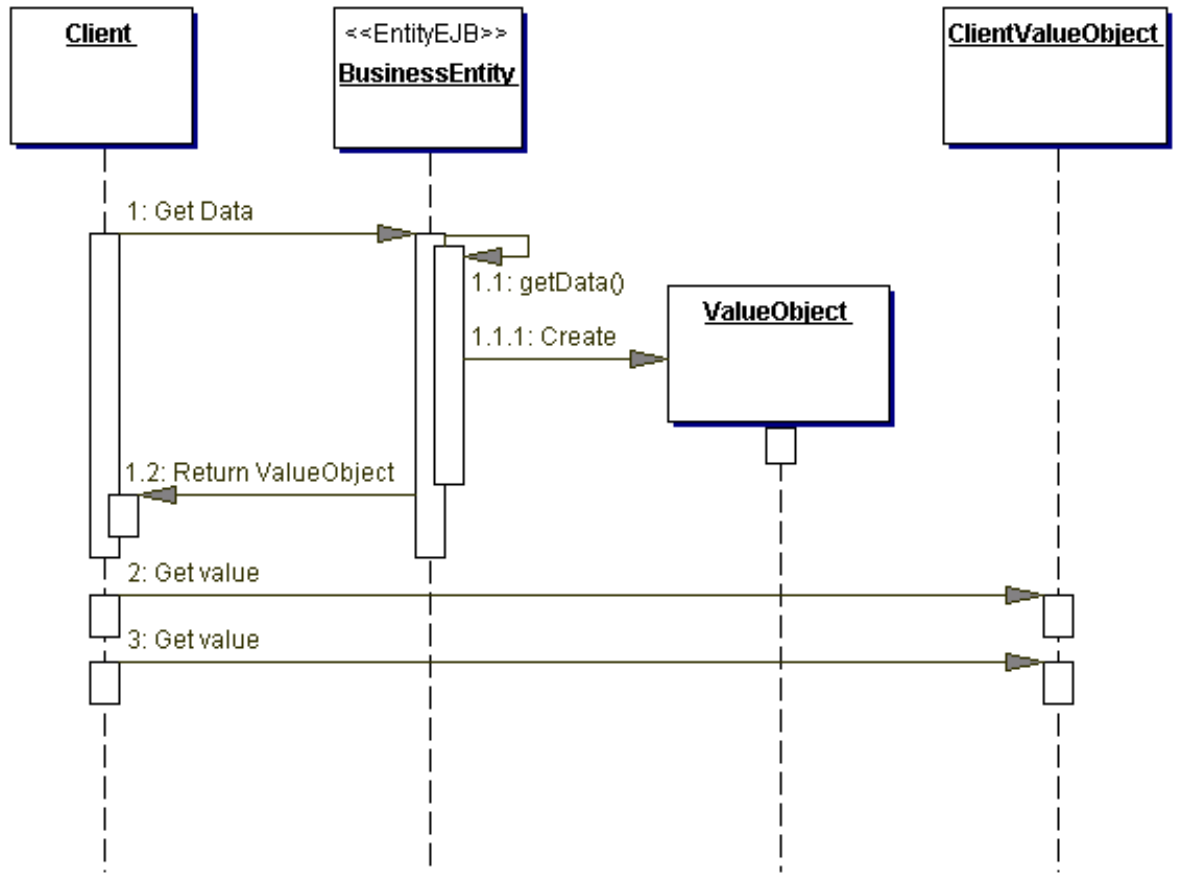
```
    q.close();
    db.close();
```

Заметьте, что запрос параметризован, поэтому он может использоваться многократно.

Value object

Как мы увидим далее, реализация бизнес объектов в виде Enterprise Java Bean (EJB), чревата увеличением сетевого трафика, т.к. каждый вызов метода EJB является потенциально сетевым. Чтобы оптимизировать доступ, обычно создают объект (Value Object), содержащий все свойства бизнес объекта, получаемые из базы данных одним запросом. Поскольку этот объект уже не является EJB и может быть целиком передан клиенту по сети, обращения к его локальной копии уже не будут сетевыми.

Рисунок 7.5. Диаграмма взаимодействий с Value Object



При реализации бизнес объектов удобно в качестве Value Object иметь так называемую "модель" объекта - совокупность состояния и поведения бизнес объекта, абстрагированную от его реализации как EJB или какой-то другой, в том числе от способа его хранения.

Глава 8. Java Database Connectivity

JDBC

Итак, в типичном Web-приложении можно выделить 4 уровня: хранения, приложения, представления и пользовательского интерфейса (см. стандартная платформа Интернет-приложений [architect.xml#platforma]). Обычно эти уровни реализуются на различных компьютерах. Уровень хранения реализуется в виде реляционной или объектной базы данных или просто в файловой системе. Этот уровень должен обеспечить постоянное хранение объектов с целью их последующего восстановления. Уровень приложения (бизнес логики) реализуется на сервере приложений или Web-сервере. Этот уровень имеет дело непосредственно с данными (объектами) и не связан с их визуальным представлением. На этом уровне данные, полученные от пользователя и из базы данных преобразуются и проверяются перед их дальнейшим прохождением в системе. Уровень представления готовит данные для отображения, составляя их в таблицы, списки, упорядочивает их определенным образом и пр. Это происходит на Web-сервере при подготовке HTML-страницы. Уровень пользовательского интерфейса реализован в браузере клиента средствами HTML. Его задача – проверить вводимые пользователем данные непосредственно на месте, без передачи на другие уровни, настолько, насколько это возможно.

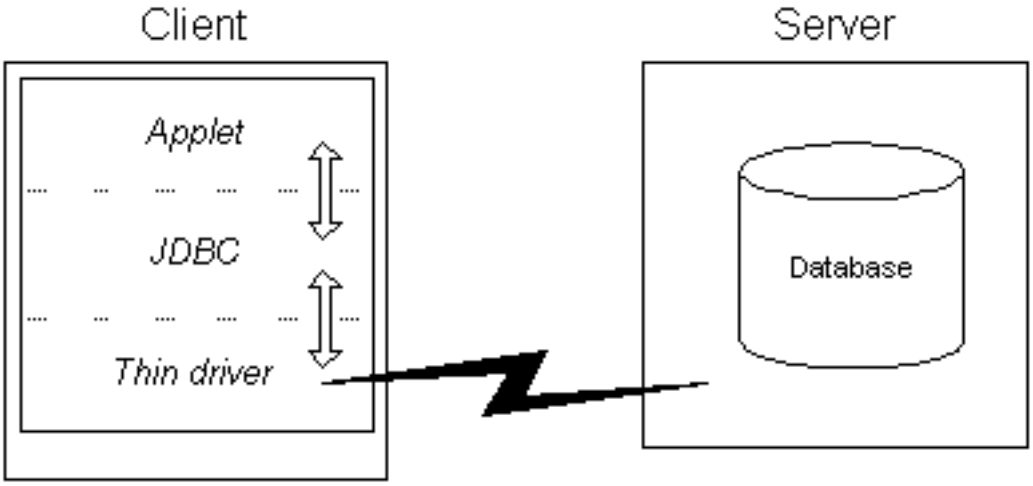
Взаимодействие между уровнем приложения и уровнем хранения осуществляется через интерфейсы jdbc – java database connectivity. "JDBC является стандартным API для выполнения SQL предложений. Он состоит из классов и интерфейсов для создания SQL предложений и получения результатов выполнения этих предложений в реляционных базах данных". Как мы уже не раз говорили, интерфейс – это полностью абстрактный класс. Следовательно, реализация интерфейса возлагается на кого-то другого. Тем не менее, мы можем использовать этот интерфейс в своих программах так, как будто он уже реализован – конечно, до тех пор, пока нам не надо выполнять свои программы – тогда интерфейс действительно *должен* быть реализован. Другими словами, мы опять используем полиморфизм – выполняем методы интерфейса, фактически - суперкласса, вместо выполнения методов конкретных классов. Такой подход имеет очевидные преимущества – нам не надо заботиться о конкретной реализации интерфейса, которая, собственно и определяет способ хранения информации. Разработчики различных СУБД поставляют свои реализации jdbc для обеспечения связи с их базами данных – Oracle, Informix или Sybase. Существуют версии (реализации) jdbc, работающие через известный драйвер odbc. Наши программы, использующие этот интерфейс, будут работать с любыми базами данных и с любыми драйверами, если последние реализуют указанный интерфейс. Конкретная реализация интерфейса загружается динамически во время выполнения java-программы. Интерфейс jdbc описан в пакете java.sql. Реализация драйвера для СУБД Oracle находится в файле `classes12.zip`, который можно найти в каталоге `%ORACLE_HOME%/jdbc/lib`.

Существует два вида Oracle JDBC драйверов:

- Драйверы, полностью написанные на java (драйверы "тонкого" клиента). Они
    - Написаны на java
    - Полностью переносимы
    - Могут быть загружены в браузер с Web-сервера
    - Соединяются с базой данных используя TCP/IP протокол
- Драйверы, использующие OCI (Oracle Call Interface)
    - Реализованы с использованием OCI DLL
    - Должны быть доступны как библиотеки на клиенте. Не могут быть загружены в браузер.
    - Соединяются с БД используя протокол SQL*Net
    - Высоко производительны

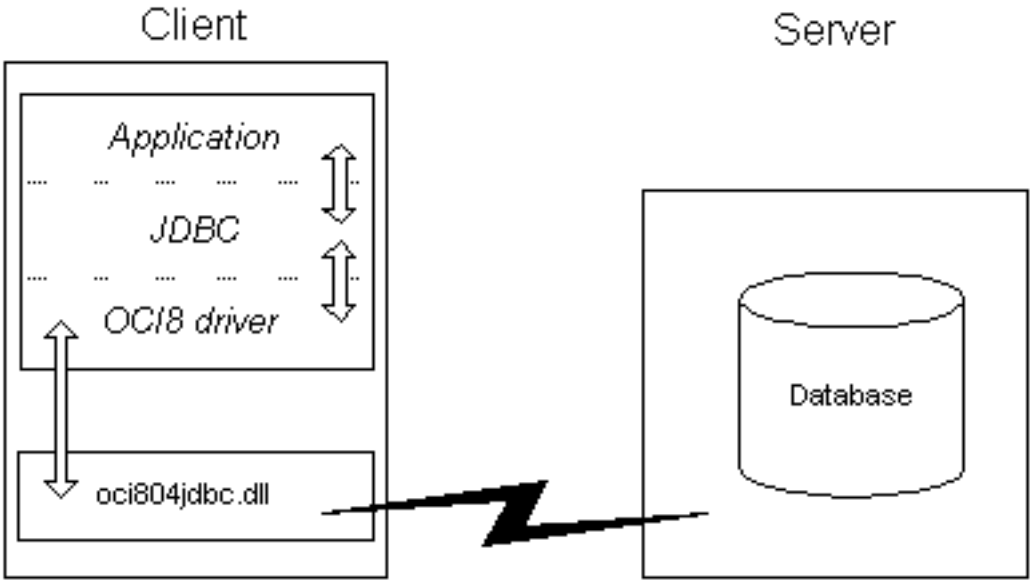
Следующие рисунки схематично пояснят эту разницу. Первый показывает взаимодействие "тонкого" драйвера с БД из апплета в браузере

Рисунок 8.1. Взаимодействие с "тонким" драйвером



Другой рисунок показывает, как работает "толстый" драйвер

Рисунок 8.2. Взаимодействие с "толстым" драйвером



Библиотека oci804jdbc.dll лежит в файловой системе на компьютере клиента.

Интерфейсы jdbc

Рассмотрим теперь некоторые наиболее употребительные интерфейсы и методы jdbc. Это:

- Driver – интерфейс, который должен реализовать любой драйвер БД
- DriverManager – базовый класс, оперирующий с набором драйверов
- Connection – интерфейс, обеспечивающий сессию с БД (connect)
- Statement – интерфейс, используемый для выполнения SQL предложения
- ResultSet - интерфейс, используемый для доступа к таблицам БД
- ResultSetMetadata - интерфейс, используемый для получения информации о типах и свойствах колонок в ResultSet

Прежде всего нам надо загрузить нужную реализацию jdbc драйвера. Динамическая загрузка драйвера осуществляется методом forName класса Class:

```
Driver d = (Driver)(Class.forName("oracle.jdbc.driver.OracleDriver").newInstance());
```
При этом будет искаться класс oracle.jdbc.driver.OracleDriver, он находится в библиотеке classes12.zip, поэтому нужная библиотека должна быть заранее указана в classpath. Тот же эффект достигается просто созданием нового объекта:

```
Driver d = new oracle.jdbc.driver.OracleDriver();
```
Затем драйвер регистрируется средствами jdbc. Для этого служит интерфейс DriverManager с методом

```
public static void  registerDriver(Driver d)
```
Поскольку этот метод статический, его можно вызвать сразу, например так:

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```
Связь с базой данных (connect) осуществляется методом getConnection интерфейса DriverManager, возвращающим объект Connection. Например:

```
Connection conn = DriverManager.getConnection ("jdbc:oracle:thin:@host:1521:ORCL", "scott", "tige
```
Здесь

- jdbc:oracle – присутствует всегда,
- thin – означает "тонкий" драйвер,
- host – имя сервера БД (компьютера в локальной сети),
- 1521 – порт, по которому БД Oracle прослушивает соединения по TCP/IP,
- ORCL – это SID БД, далее идет имя и пароль пользователя БД.

Получив объект Connection, мы создаем объект Statement, который потом используем для выполнения SQL предложений. Например:

```
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery ("select ENAME from EMP");
```
Полученный в результате запроса объект класса ResultSet – это фактически курсор, если Вы встречались с этим понятием в PL/SQL. По этому курсору можно пройти *один раз,* с тем, чтобы получить значения записей. ResultSet имеет метод next(), обеспечивающий переход к следующей записи и возвращающий true или false в зависимости от успеха этой операции. После прохождения по курсору назад вернуться нельзя. Вот пример выборки значений:

```
while (rset.next ())
  System.out.println (rset.getString (1));
}
```
Доступ к полям записи осуществляется по индексу. rset.getString(1) дает первое поле в виде строки, rset.getString(2) дает второе поле и т.д.

Новый объект Connection по умолчанию находится в режиме "авто-сохранения". Это означает, что commit выполняется после каждого выполнения предложения. Этот режим может быть отключен вызовом

```
Connection.setAutoCommit(false);
```
Даже когда авто-сохранение отключено, каждая сессия имеет неявную транзакцию, связанную с ней. Connection.commit() завершает транзакцию, Connection.rollback() – прерывает (абортирует) транзакцию. В любом случае начинается новая неявная транзакция.

HelloJDBC.java

Выполним простой пример. Однако, прежде, чем писать программы с jdbc, необходимо установить СУБД и получить реализацию jdbc для нее. Можно работать с Oracle, но это - большая СУБД, вообще говоря, для промышленных приложений. В рамках платформы J2EE мы имеем реляционную СУБД "Cloudscape", которая доступна сразу. Во-первых, запустим эту СУБД командой из каталога %J2EE_HOME%/bin:

```
cloudscape -start
```

Во-вторых создадим командный файл создания и заполнения таблиц cloudscape.sql:

```
drop table books;
drop table readers;

create table readers (
    reader_id  varchar(30);
    fio varchar(80) not null,
    constraint pk_readers primary key (reader_id)
);

create table books (
    book_id  varchar(30);
    author varchar(80) not null,
    title varchar(80) not null,
    published varchar(80) null,
    image_file varchar(80) null,
    reader varchar(80)  null,
    constraint pk_books primary key (book_id),
    constraint fk_books foreign key (reader) references readers (reader_id)
);

INSERT INTO readers VALUES('1','Лепехин А.Ф.');
INSERT INTO books VALUES('1','Romain Guy','Jext Editor','www.jext.org','images/jext.gif','Лепехин
INSERT INTO books VALUES('2','В.К.Мюллер','Англо-русский словарь','1989','images/Muller.gif','Леп
INSERT INTO books VALUES('3','Elliotte Rusty Harold','Java Network Programming','OReily','images/
INSERT INTO books VALUES('4','Elliotte Rusty Harold','Java Secrets','IDG Books','images/secretsco
INSERT INTO books VALUES('5','Elliotte Rusty Harold','Java Beans','IDG Books','images/beanscover.
INSERT INTO books VALUES('6','Elliotte Rusty Harold','XML Bible','IDG Books','images/mediumbiblec
```

В-третьих, создадим таблицы и загрузим данные:

```
populate.bat
```

командником, в котором одна строка:

```
java -Dcloudscape.system.home=%J2EE_HOME%\cloudscape -classpath .;populate.jar;
     %J2EE_HOME%\lib\cloudscape\client.jar;%J2EE_HOME%\lib\cloudscape\tools.jar;
     %J2EE_HOME%\lib\cloudscape\cloudscape.jar;%J2EE_HOME%\lib\cloudscape\RmiJdbc.jar;
     %J2EE_HOME%\lib\cloudscape\license.jar;
     %J2EE_HOME%\lib\j2ee.jar PopulateTables cloudscape.sql Cp1251 localhost
```

Замечание

здесь мы используем готовый класс `PopulateTables` для загрузки данных.

Откроем текст программы HelloJDBC.java. Приведем этот текст к виду:

```
import java.sql.*;
public class HelloJDBC {
```

```
  private static final String databaseDriver = "RmiJdbc.RJDriver";
  private static final String user= "estoreuser";
  private static final String password = "estore";
  private static final String host = "localhost";

  public static void main (String args []) throws SQLException  {
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
    String databaseName = "jdbc:rmi://" + host + ":1099/jdbc:cloudscape:CloudscapeDB;autocommit=t

    try {
        Class.forName(databaseDriver);
    } catch (ClassNotFoundException e) {
        System.out.println("HelloJDBC: caught getting connection: " + e);
    }
    Connection con = null;
    try {
        con = DriverManager.getConnection(databaseName, user, password);
    } catch (java.sql.SQLException se) {
        System.out.println("HelloJDBC: caught getting connection: " + se);
    }

    Statement stmt = con.createStatement ();
    ResultSet rset = stmt.executeQuery ("select author,title from books");
    while (rset.next ())
      System.out.println (toCp866(rset.getString (1)+" "+rset.getString (2)));
  }

  private static String toCp866(String s) {
    try {
      return new String(s.getBytes("Cp866"),"Cp1251");
    } catch (Exception e) {e.printStackTrace();}
    return null;
 }
}
```

Выполнив эту программу, получим в консольном окне авторов и названия книг.

```
D:\projects\Course\examples\jdbc>javac HelloJDBC.java

D:\projects\Course\examples\jdbc>java -classpath %classpath%;%J2EE_HOME%\lib\cloudscape\RmiJdbc.j

Romain Guy Jext Editor
В.К.Мюллер Англо-русский словарь
Elliotte Rusty Harold Java Network Programming
Elliotte Rusty Harold Java Secrets
Elliotte Rusty Harold Java Beans
Elliotte Rusty Harold XML Bible
```

Русские буквы в названиях при этом правильно отображаются (кроме буквы Ш, заметьте себе это!).

Покажем теперь на этом же примере применение шаблона фасада базы данных [patterns.html#facade]. Вот как выглядит та же приграмма с применением фасада:

```
/*
 * $Id: jdbc.xml,v 1.2 2002/10/04 13:23:01 afl Exp $
 */
import java.sql.ResultSet;
import com.fors.database.Database;
```

```
import com.fors.database.QueryI;
import com.fors.util.tracer.Debug;

public class HelloDatabase {

  private static final String databaseDriver = "RmiJdbc.RJDriver";
  private static final String user= "estoreuser";
  private static final String password = "estore";
  private static final String host = "localhost";

  public static void main (String args []) {
    String databaseName = "jdbc:rmi://" + host + ":1099/jdbc:cloudscape:CloudscapeDB;autocommit=t
    try {
      Database db = new Database(databaseDriver);
      db.open(databaseName, user, password);
      QueryI q = db.newQuery("select author,title from books");
      ResultSet rset = q.execute().getResultSet();
      while (rset.next ()) {
        Debug.println (rset.getString (1)+" "+rset.getString (2));
      }
      q.close();
      db.close();
    } catch (Exception e) {
      Debug.println(e.toString());
    }
  }
}
```

Запустив эту программу на выполнение командой

```
D:\projects\Course\examples\jdbc>java -classpath %classpath%;%J2EE_HOME%\lib\cloudscape\RmiJdbc.j
```

получим тот же результат, но текст программы стал гораздо прозрачней.

Выполнение этих маленьких примеров дает нам уверенность, что мы все правильно понимаем и что все это действительно работает. Рассматривая пример, мы видим, что в executeQuery подставляется просто строка. Это значит, что мы можем сформировать эту строку динамически, т.е. jdbc способно выполнять *динамические запросы*.

В JDBC 2.0 API появились возможности объявить скроллируемый и обновляемый ResultSet за счет введения нового метода createStatement:

```
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");
// rs will be scrollable, will not show changes made by others, and will be updatable
```

Кроме того, в JDBC 2.0 API появились следующие возможности:

- Возможность пакетной обработки
- Новые типы данных, удовлетворяющие стандарту SQL3
- Возможность работы с определяемым пользователем типами данных
- Символьные потоки
- Произвольная точность представления десятичных чисел
- Дополнительная защита (security)

Рассмотрим подробнее некотые аспекты jdbc.

Передача параметров и получение результатов

Результаты запроса

Как мы уже видели, результаты запроса возвращаются в виде набора строк через объект java.sql.ResultSet. Этот объект имеет методы доступа к различным колонкам текущей строки. Метод ResultSet.next может быть использован для передвижения по строкам полученного набора. В следующем примере SQL запрос возвращает строки, имеющие три колонки – первая типа int, вторая – String и третья – массив байт:

```
java.sql.Statement stmt = conn.createStatement();
ResultSet r = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (r.next()) {
  // print the values for the current row.
  int i = r.getInt("a");
  String s = r.getString("b");
  byte b[] = r.getBytes("c");
  System.out.println("ROW = " + i + " " + s + " " + b[0]);
}
```

Есть два альтернативных способа указания колонок. Их можно указывать или по индексу (для большей эффективности) или по имени (для большего удобства).

```
java.sql.Statement stmt = conn.createStatement();
ResultSet r = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (r.next()) {
  // print the values for the current row.
  int i = r.getInt(1);
  String s = r.getString(2);
  byte b[] = r.getBytes(3);
  System.out.println("ROW = " + i + " " + s + " " + b[0]);
}
```

Колонки читаются слева направо. Каждая колонка может быть прочитана только один раз.

Преобразование типов в результатах запроса

Методы ResultSet.getXXX пытаются преобразовать результат запроса в тип, указанный в ResultSet.getXXX независимо от типа в базе данных. Следующая таблица дает возможные преобразования типов. Например, тип базы данных VARCHAR может быть перобразован в целое с помощью getInt, но невозможно преобразовать тип БД FLOAT в java.sql.Date.

Рисунок 8.3. Преобразование типов

		SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	**X**	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	**X**	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	**X**	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	**X**	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	**X**	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	**X**	**X**	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	**X**	**X**	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	**X**	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	**X**	**X**	x	x	x	x	x	x	x
getBytes														**X**	**X**	x			
getDate											x	x	x				**X**		x
getTime											x	x	x					**X**	x
getTimestamp											x	x	x				x		**X**
getAsciiStream											x	x	**X**	x	x	x			
getUnicodeStream											x	x	**X**	x	x	x			
getBinaryStream														x	x	**X**			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Table 1: Use of ResultSet.getXXX methods to retrieve common SQL data types
An "x" means that the given getXXX method can be used to retrieve the given SQL type.
An "**X**" means that the given getXXX method is recommended for retrieving the given SQL type

При ошибке преобразования – когда оно не поддерживается или происходит с ошибкой (например, нельзя прочитать строку "foo" методом getInt) возникает исключение.

Значения null

Чтобы определить, что данный результат есть null, полученный из базы данных, необходимо вначале прочитать колонку, а затем использовать метод ResultSet.wasNull, чтобы определить, что именно null был получен. Чтение null из базы данных методами ResultSet.getXXX даст:

- Значение null для возвращаемых java объектов
- Ноль для примитивных типов, возвращаемых методами getByte, getShort, getInt, getLong, getFloat, and getDouble
- false для метода getBoolean

Возвращение очень больших значений

JDBC позволяет получить произвольно большие данные типа LONGVARBINARY или LONGVARCHAR с помощью методов getBytes и getString вплоть до размера, указанного значением

Statement.getMaxFieldSize. Однако, может быть удобно получать большие значения через небольшие буфера фиксированного размера. Эта возможность предоставляется классом ResultSet, который может возвратить java.io.Input stream – поток, из которого данные могут быть прочитаны в буфер. Этот поток должен быть прочитан немедленно, т.к. он автоматически закрывается при выполнении чтения следующей командой get. Java потоки возвращают просто байты и могут быть использованы для чтения, например ASCII и Unicode символов. Для удобства созданы несколько методов получения потоков: getBinaryStream дает "сырые" байты без какого-либо преобразования, getAsciiStream возвращает однобайтовые ascii-символы, getUnicodeStream – возвращает двубайтовые символы Unicode. Например:

```
java.sql.Statement stmt = conn.createStatement();
ResultSet r = stmt.executeQuery("SELECT x FROM Table2");
// Now retrieve the column 1 results in 4 K chunks:
byte[] buff = new byte[4096];
while (r.next()) {
  java.io.InputStream fin = r.getAsciiStream("x");
  for (;;) {
    int size = fin.read(buff);
    if (size == -1) {
      break;
    }
    // Send the newly filled buffer to some ASCII output stream:
    output.write(buff, 0, size);
  }
}
```

Множественные наборы данных

Обычно мы ожидаем, что SQL предложение будет выполнено или с помощью метода executeQuery, который вернет один набор данных (ResultSet) или с помощью executeUpdate, который вернет количество обновленных строк. Однако, при некоторых обстоятельствах приложение может не знать, что именно будет возвращено. Вдобавок, некоторые хранимые процедуры могут возвращать несколько наборов данных и/или счетчиков обновлений. Для этого случая предусмотрен механизм, основанный на общем методе "execute", поддерживаемый тремя другими методами getResultSet, getUpdateCount и getMoreResults. Эти методы позволяют определить, что возвращено в результате выполнения и узнать, есть ли еще наборы данных.

Передача IN параметров

Чтобы обеспечить передачу параметров в SQL предложения, java.sql.PreparedStatement класс имеет набор методов типа setXXX. Они могут быть использованы перед каждым выполнением SQL предложения чтобы заполнить поля параметров. Если значение параметра определено для данного предложения, то оно может быть использовано многократно, пока не будет очищено методом PreparedStatement.clearParameters. Например,

```
java.sql.PreparedStatement stmt = conn.prepareStatement(
    "UPDATE table3 SET m = ? WHERE x = ?");
// We pass two parameters. One varies each time around the for loop,
// the other remains constant.
stmt.setString(1, "Hi");
for (int i = 0; i < 10; i++) {
  stmt.setInt(2, i);
  int rows = stmt.executeUpdate();
}
```

Метод PreparedStatement.setNull позволяет передать null в качестве входного параметра. Тип SQL параметра при этом должен быть указан. Для методов setXXX, в которых аргументом могут быть объекты, null может быть передан подстановкой null в качестве аргумента.

Передача очень больших параметров

Передача больших значений параметров может быть выполнена через поток, так же, как мы видели при чтении больших объемов данных, например

```
java.io.File file = new java.io.File("/tmp/foo");
int fileLength = file.length();
java.io.InputStream fin = new java.io.FileInputStream(file);
java.sql.PreparedStatement stmt = conn.prepareStatement(
   "UPDATE Table5 SET stuff = ? WHERE index = 4");
   stmt.setBinaryStream(1, fin, fileLength);
// When the statement executes, the "fin" object will get called
// repeatedly to deliver up its data.
stmt.executeUpdate();
```

Получение OUT параметров

Если требуется выполнить хранимую процедуру, то нужно использовать класс CallableStatement, являющийся подклассом PreparedStatement. Чтобы передать IN параметры в процедуру можно использовать методы setXXX, определенные в классе PreparedStatement. Однако, если хранимая процедура имеет OUT параметры, то каждый такой параметр должен зарегистрировать свой тип методом CallableStatememt.registerOutParameter до выполнения предложения. После выполнения предложения значение OUT параметра может быть получено соответствующим CallableStatement.getXXX методом. Например,

```
java.sql.CallableStatement stmt = conn.prepareCall("{call getTestData(?, ?)}");
stmt.registerOutParameter(1,java.sql.Types.TINYINT);
stmt.registerOutParameter(2,java.sql.types.DECIMAL, 2);
stmt.executeUpdate();
byte x = stmt.getByte(1);
BigDecimal n = stmt.getBigDecimal(2,2);
```

Вызов функций производится аналогично, но возвращаемое значение описывается, например, следующим образом:

```
java.sql.CallableStatement stmt = conn.prepareCall("{?= call getTestData()}");
stmt.registerOutParameter(1,java.sql.types.VARCHAR,);
stmt.executeUpdate();
String s = stmt.getString(1);
```

Метод CallableStatement.getXXX не производит преобразования типов. Вместо этого тип (SQL тип, т.е. тип в базе данных) указывается при вызове registerOutParameter, после чего программист должен вызвать метод getXXX, с java типом, соответствующим указанному SQL типу.

Следующая таблица дает соответствие между SQL и java типами

SQL type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte

SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Обратное соответствие указано в следующей таблице. String обычно отображается в VARCHAR, но будет отображено в LONGVARCHAR, если ее длина превосходит предел, установленный драйвером для данных типа VARCHAR. Аналогично для byte[] и VARBINARY и LONGVARBINARY.

Java Type	SQL type
String	VARCHAR or LONGVARCHAR
java.math.BigDecimal	NUMERIC
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[]	VARBINARY or LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

Глава 9. Java Server Pages

Java Server Pages

Километры HTML-текста, включаемые в сервлеты [servlet.html#servlet], производят ужасающее впечатление. Первое, что приходит в голову – нельзя ли написать нечто универсальное для включения текстов HTML страниц в Java программу? или может их (HTML страницы) читать во время выполнения? Приятной неожиданностью оказывается то, что это уже сделано. Идея Java Server Pages (JSP) состоит в том, чтобы не HTML страницы включать в текст Java, а наоборот – операторы Java включить в текст HTML добавив новые теги. В этом безусловно есть преимущества:

- Во-первых, можно вначале создать HTML страницы, посмотреть их, поправить и уже потом добавить в них Java-код. Конечно, это можно делать и с сервлетами, но исправлять их сложнее
- Во-вторых, Java-код можно сделать очень компактным, вынеся все требуемые вычисления в отдельно стоящие классы. Эти классы оформляются как обычные Java Beans (не Enterprise!), так что можно получить их свойства стандартными методами.
- Как следствие, мы приходим к удачному разделению логики уровня представления на части, которые легко обозримы и, следовательно, менее подвержены ошибкам.

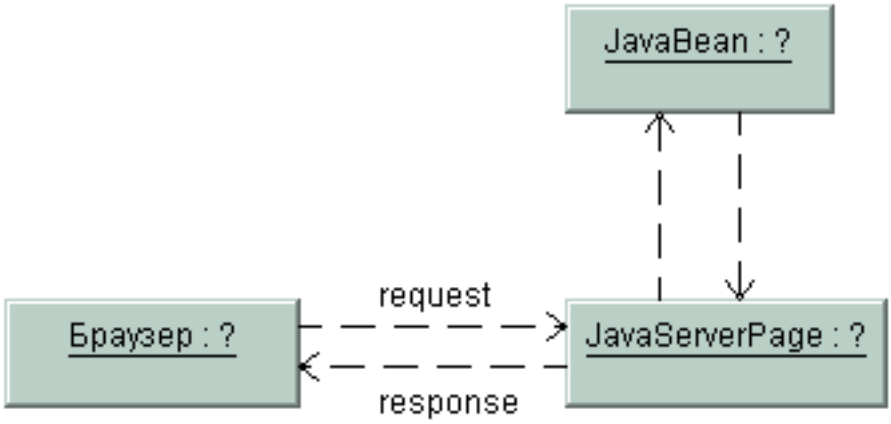
При запросе такой страницы она вначале преобразуется препроцессором сервера в программу на Java, которая потом компилируется и выполняется как сервлет. Таким образом наши HTML страницы будут содержать обычные теги HTM и сценарии JavaScript, выполняемые в браузере, и Java-код, выполняемый на сервере. Очень удобно.

Методы доступа

SunSoft описывает два метода доступа к JSP:

- непосредственный вызов JSP из браузера как обычной HTML страницы:

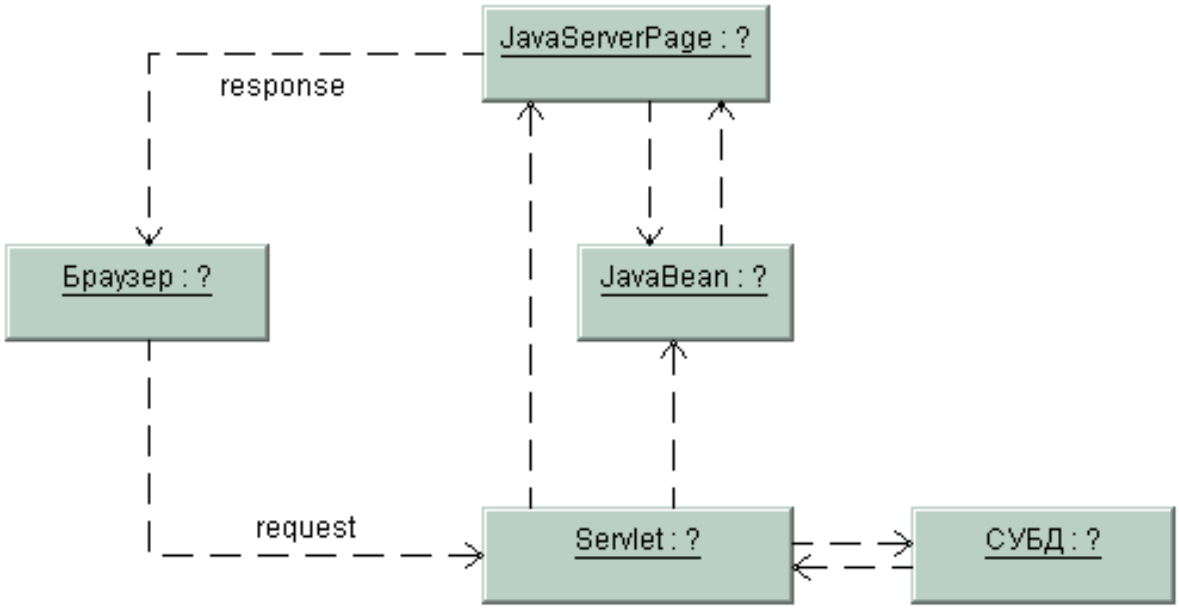
Рисунок 9.1. Первая модель



Здесь JavaBean – это обычный Java Bean, который может обеспечивать, например, доступ к базе данных и иметь результирующий набор данных как одно из своих свойств.

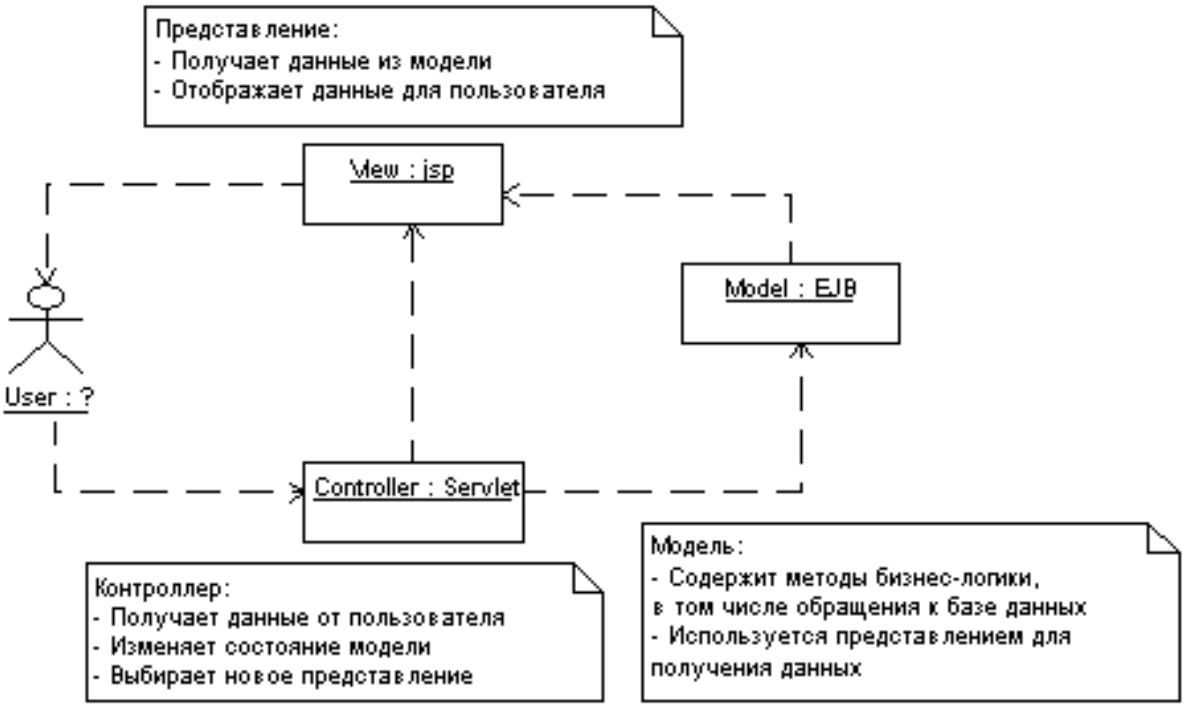
- Второй метод доступа – через сервлет:

Рисунок 9.2. Вторая модель



Здесь сервлет генерирует динамическое содержание. Сервлет обращается к базе данных и создает Java Bean для хранения результирующего набора данных. Затем сервлет вызывает JSP, которая использует JavaBean для представления данных также, как и в первом случае. Отделяя в этой модели бизнес-логику от уровня представления, получаем шаблон "Модель-Представление-Контроллер" [patterns.html#mvc], в котором роль "Модели" играют Enterprise Java Beans, роль контроллера - сервлет, а роль представления - Java Server Pages:

Рисунок 9.3. Вторая модель с EJB



Вызов JSP из сервлета производится командой:

```
getContext().getRequestDispatcher("/path/page.jsp").forward(request,response)
```

Чтобы вызвать сервлет из JSP, поместите в его конец JSP команду

```
<jsp:forward page="mymainjsp.jsp"/>
```

В обоих случаях JSP может иметь свой Java-код. Существует два API (Application Program Interface) для обеспечения той или иной модели доступа. Когда использовать ту или иную модель? Sun дает следующие рекомендации:

- Если для сбора данных от пользователя необходим графический интерфейс – используйте JSP
- Если запрос и его параметры так или иначе доступны через сервлет, а результат запроса должен быть представлен в графической форме – используйте JSP
- Если требования к представлению минимальны (требуют небольшого числа строк println в сервлете), и Вы не хотите раскрывать логику представления пользователю или Web-мастеру – используйте сервлеты.

Три примера

Все это будет намного понятнее, если мы, как всегда, выполним несколько небольших примеров. Заодно научимся размещать JSP в J2EE.

Первый пример

В качестве JSP может выступать обычный HTML файл. Надо просто сменить расширение на jsp. Мы имели пример простого HTML файла – Hello.html. Скопируем этот файл в Hello.jsp. Для выполнения JSP необходим Web-сервер с поддержкой JSP. Воспользуемся J2EE. Как мы знаем, HTML и JSP страницы надо класть в корневой каталог. Он у нас уже есть – при копировании курса мы создали каталог course. Положим туда файл hello.jsp со следующим содержимым:

```
<HTML>
<HEAD>
<DIV ALIGN="CENTER">
JSP example
</DIV>
</HEAD>
<BODY>
<P>
Hello, world!
</P>
</BODY>
</HTML>
```

Это – обычный текст HTML, который будет интерпретироваться как JSP. Запустим J2EE и откроем этот файл (адрес – мой, Вам надо поменять на свой): http://afl:8000/Course/hello.jsp Увидим прежний текст, как это было при отображении HTML страницы. Значит процессор JSP работает.

Замечание

Если нет - то надо честно установить Hello.jsp с помощью deploytool. Как это сделать, описано ниже.

Теперь включим в эту страницу Java код. Он будет выполняться на сервере.

Второй пример

Изменим текст файла Hello.jsp. Мы добавляем вывод текущей даты и времени:

```
<HTML>
<HEAD>
<DIV ALIGN="CENTER">
JSP example
```

```
</DIV>
</HEAD>
<BODY>
<P>
Hello, world!
</P>
<P>
Now <%out.println(new java.util.Date());%>
</P>
</BODY>
</HTML>
```

Выполнив этот файл, мы увидим, что к предыдущему выводу добавилсь строка:

Now Tue Feb 06 11:40:47 GMT 2001

Это - результат выполнения Java кода. Обновив страницу получим новый момент времени. Т.о самое простое – включить требуемый Java код в тег `<%...%>`.

JSP имеют несколько предопределенных переменных, одна из них - `out`, дающая возможность писать в выходной поток. Ее мы использовали в данном примере. Другой возможностью является использование "выражений":

```
Now <%=(new java.util.Date())%>
```

Тегом выражения является `<%=` после которого должно стоять выражение, приводимое к виду строки.

Третий пример

Но еще проще использовать Java Beans. Это – классы Java, которые имеют публичные методы типа `get` для чтения своих свойств, публичные методы типа `set` для записи своих свойств и публичный конструктор без параметров. JSP сконструированы так, чтобы мы могли легко вставлять свойства Java Bean'а в текст. Для этого надо вначале указать, какой бин (или бины) мы будем использовать. Создадим текст бина в каталоге dates:

```
//-- Файл JspDate.java
package jsp.dates;

import java.util.Date;

/**
Класс, оформленный в виде Java Bean
для получения даты.
Имеет конструктор по умолчанию.
*/

public class JspDate {

  /**
  Метод чтения свойства date.
  */
  public Date getDate() { return  new Date(); }

}
//--
```

Создадим текст dates.jsp:

```
<html>
<%@ page session="false"%>
```

```
<body bgcolor="white">
<jsp:useBean id="clock" scope="page" class="jsp.dates.JspDate" type="jsp.dates.JspDate" />
<font size=4>
<ul>
<li> Current time is  <jsp:getProperty name="clock" property="date"/>
</ul>
</font>
</body>
</html>
```

Как видите, объявление бина производится в теге <jsp: useBean= ... />, а использование – в теге <jsp: getProperty .../>.

Транслируем JspDate.java и установим приложение. Для установки JSP вызовем инструментальное средство установки командой deploytool.bat из %J2EE_HOME%/bin. Выполним следующую последовательность команд:

- Создадим новое приложение SimpleJsp
- Создадим новый Web-архив SimpleJsp
- Добавим в него папку jsp с папкой dates, где лежит JspCalendar.class
- Нажмем <Next>
- Добавим в него dates.jsp из папки jsp
- Выберем пункт "Describe a JSP"
- Установим Display Name=SimpleJsp и будем нажимать Next, пока не дойдем до конца.
- War-файлу SimpleJsp установим контекстный путь Course/example
- В меню Tools выберем Deploy Application и установим приложение.

Открыв JSP в браузере по адресу `http://afl:8000/Course/example/dates.jsp`, получим:

```
Current time is Wed Feb 07 12:09:21 GMT 2001
```

Скрипты и бины можно использовать одновременно на одной странице. Использование бинов предпочтительнее, т.к. исключает логику представления из HTML страницы, оставляя ей только функции отображения.

Русификация вывода

В отношении русификации можно сказать, что в мире Java такого понятия нет. Есть понятие *интернационализации* - создания продукта, который может быть использован на любом языке и в любой стране. Русификации в нашем понимании соответствует *локализация* интернационализированного продукта. При этом меняется не только язык (отображаемые символы), но и форматы дат, чисел, денежных единиц и т.п. Важно то, что интернационализированный продукт, созданный на Java, может быть затем локализован без его изменения. Сама Java интернационализирована. Основными классами, связанными с интернационализацией являются `Locale, TimeZone, ResourceBundle` и классы из пакета `java.text`. В качестве примера, выведем те же дату и время в формате текущей установки Locale (как правило - русской). Для этого изменим бин:

```
//-- Файл JspDate.java
package jsp.dates;

import java.util.Date;
import java.text.DateFormat;

/**
Класс, оформленный в виде Java Bean
для получения даты.
Имеет конструктор по умолчанию.
*/
```

```
public class JspDate {

  /**
  Метод чтения свойства date.
  */
  public String getDate() { return  DateFormat.getDateInstance(DateFormat.FULL).format(new Date()

}
//--
```

Установив заново приложение с помощью deploytool и открыв страницу в браузере, получим:

```
Current time is 7 Февраль 2001 г.
```

Попытка заменить "Current time is" на "Текущая дата" приводит к обескураживающему результату - русские буквы не получаются! Если немного подумать, то так и должно быть. Web-страница должна выглядеть по-разному (и, скорее всего, не по-русски), когда она открывается пользователями в разных странах. Поэтому текст на ней должен настраиваться автоматически в соответствии с текущей локализацией пользователя. Это достигается путем интернационализации страницы с помощью механизма ResourceBundle, а не просто созданием "русской" страницы.

Вначале познакомимся с ResourceBundle. Для этого создадим в папке dates два файла:

файл Dates_Resources.properties с текстом

```
dates.prompt=Current date is
```

и файл Dates_Resources_ru.txt с текстом

```
dates.prompt=Текущая дата
```

Из последнего файла создадим файл в кодировке Unicode командой

```
native2ascii Dates_Resources_ru.txt Dates_Resources_ru.properties
```

Чтение того или иного файла определяется текущей установкой локализации в операционной системе: если локализация - Россия, то будет читаться файл Dates_Resources_ru.properties, иначе - файл Dates_Resources.properties. Теперь создадим класс, который будет читать эти файлы и возвращать значение свойства по ключу:

```
//-- Файл ResourceBean.java
package jsp.dates;

import java.util.*;

/**
Читает данные из файла Dates_Resource*.properties
*/
public class ResourceBean {

  private ResourceBundle myResources = null;

  //**
  Возвращает значение свойства по ключу.
  */
  public String getText(String key) {
    return myResources.getString(key);
  }

  //**
  Создает объект типа ResourceBundle чтением файла с именем bundle.
```

```
  */
  public void setBundle(String bundle) {
    myResources = ResourceBundle.getBundle(bundle);
  }

  //**
  Метод для тестирования данного класса.
  */
  public static void main(String[] args) {
    ResourceBean r = new ResourceBean();
    r.setBundle("Dates_Resources");
    System.out.println(r.getText("dates.prompt"));
  }
}
//--
```

Транслируем и выполним этот класс. Получим:

```
Тхъє•р  фрЄр
```

Это - строка "Текущая дата" как она отобразилась в кодировке Cp866. Если теперь изменить локализацию в операционной системе с России на какую-либо другую и снова выполнить этот класс, получим:

```
Current date is
```

т.е. прочитался файл `Dates_Resources.properties`, что и требовалось показать.

Подключим теперь чтение этих файлов в текст JSP-страницы:

```
<html>
<%@ page session="false" contentType="text/html; charset=windows-1251" %>
<body bgcolor="white">
<jsp:useBean id="clock" scope="page" class="jsp.dates.JspDate" type="jsp.dates.JspDate" />
<jsp:useBean id="messages" scope="page" class="jsp.dates.ResourceBean" type="jsp.dates.ResourceBe
<jsp:setProperty name="messages" property="bundle" value="Dates_Resources" />
<font size=4>
<ul>
<li>
<%=messages.getText("dates.prompt")%>
 <jsp:getProperty name="clock" property="date"/>
</li>
</ul>
</font>
</body>
</html>
```

Заново установим dates.jsp в j2ee с помощью deploytool. При выборе классов, выберем папку dates и файлы *.properties, поскольку они должны быть доступны через classpath. При выборе содержимого выберем только dates.jsp. Опять назначим контекстный путь Course/example. Снова открыв страницу http://afl:8000/Course/example/dates.jsp, получим в браузере то, что мы хотели, предварительно вернув обратно локализацию "России":

```
Текущая дата 14 Февраль 2001 г.
```

Замечание

Если это не получилось или браузер ругается, то проверьте следующее:

- В каталоге public_html под %J2EE_HOME% должен быть каталог Course и под ним - каталог

example.
- В example должен быть файл dates.jsp
- Под каталогом example должен найтись каталог Web-inf. Под ним должен быть каталог classes.
- В каталоге classes должны быть файлы Dates_resourses.properties, Dates_resourses_ru.properties и taglib.tld
- Под каталогом classes должен быть каталог jsp/dates с файлами JspDate.class, ResourceBean.class и TextTag.class.
- Если каких-либо файлов нет или они не на месте, надо переустановить приложение с помощью deploytool. Если все на месте - приложение должно работать.

С первого раза обычно не выходит.

Библиотеки тегов

Полный синтаксис JSP версии 1.1 приведен в спецификации Sun [jsp1_1-spec.pdf]. Эта спецификация описывает также *библиотеку тегов* - коллекцию действий, обеспечивающих некоторую функциональность, используемую внутри страницы. Библиотеки тегов уменьшают (в идеале до нуля) использование скриплетов. В результате текст страницы становится более читабельным. Кроме того, разработчику страниц не надо знать Java, т.к. весь код вынесен в библиотеку. В идеале текст страницы может быть доведен до синтаксиса, совместимого с XML.

В приведенном выше листинге dates.jsp хорошо бы заменить не стандартный тег

```
<%=messages.getText("dates.prompt")%>
```

на что-нибудь вроде

```
<message:text key="dates.prompt" />
```

Тогда мы могли бы получить любые локализованные сообщения из файла `Dates_Resources.properties` стандартным образом. Для этого определим новый тег `text` в специальном xml-файле - *определении библиотеки тегов*, определящим тег и Java-класс, его выполняющий. В jsp укажем путь на это определение в теге `taglib`.

Основным механизмом для определения семантики тега является tag handler - Java класс, реализующий интерфейс javax.servlet.jsp.tagext.Tag или javax.servlet.jsp.tagext.TagBody. Первый служит для обработки тегов без тел, второй - с телами. Существует два готовых класса - заготовки: TagSupport и BodyTagSupport. Их можно расширять для настройки. Наш пример - простейший: вызов функциональности без тела. Вначале создадим определение библиотеки тегов в файле taglib.tld:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
        PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
        "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- a tab library descriptor -->

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>messages</shortname>
  <info>
        Dates demo custom tags
  </info>

  <tag>
    <name>text</name>
```

```
    <tagclass>dates.TextTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>
      Provide messages from Resource Bundle.
    </info>

    <attribute>
      <name>key</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>

  </tag>

</taglib>
```

Я это определение помещаю в каталог dates вместе с текстами программ, при установке приложения в j2ee он окажется в каталоге Web-inf/classes. Это надо иметь в виду при формировании uri в теге taglib.

Теперь создадим Java-класс, выполняющий работу тега:

```
//-- Файл TextTag
package dates;

import java.io.IOException;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.TagSupport;

public class TextTag extends TagSupport {
  private String key = null;

  public String getKey() { return key; }
  public void setKey(String key) { this.key = key; }

  public int doStartTag() throws JspException {
    ResourceBean messages = (ResourceBean)pageContext.getAttribute("messages", PageContext.APPLIC
    String text = messages.getText(getKey());
    JspWriter writer = pageContext.getOut();
    try {
            writer.print(text);
        } catch (IOException e) {
            throw new JspException(e.toString());
        }

    return SKIP_BODY;
  }
}//--
```

Как видите, атрибуты тега являются свойствами этого класса. В программе предполагается, что `messages` уже установлена на странице и является ссылкой на бин `ResourceBean`. Теперь изменим нашу страницу:

```
<html>
<%@ page session="false" contentType="text/html; charset=windows-1251" %>
<%@ taglib uri="/WEB-INF/classes/taglib.tld" prefix="messages" %>
<body bgcolor="white">
<jsp:useBean id="clock" scope="page" class="dates.JspDate" type="dates.JspDate" />
<jsp:useBean id="messages" scope="application" class="dates.ResourceBean" type="dates.ResourceBea
```

```
<jsp:setProperty name="messages" property="bundle" value="Dates_Resources" />
<font size=4>
<ul>
<li>
<messages:text key="dates.prompt" />
 <jsp:getProperty name="clock" property="date"/>
</li>
</ul>
</font>
</body>
</html>
```

Вот и все. Приложение можно устанавливать и выполнять. Это приложение берет локализацию сервера. Чтобы взять локализацию пользователя, ее необходимо получить из Web-страницы и подставить в getBundle() вторым параметром.

Глава 10. CORBA и RMI

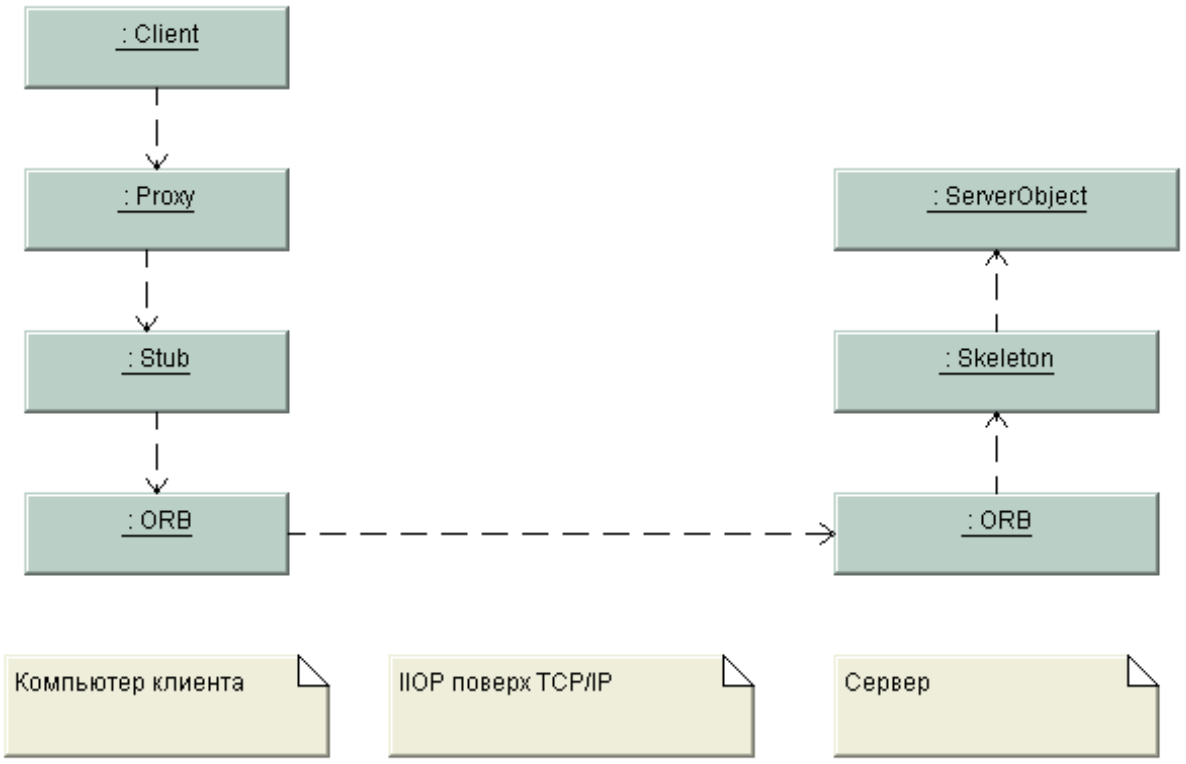
Введение в CORBA

CORBA (Common Object Request Broker Architecture) – общая архитектура брокера объектных запросов разработана косорциумом Object Management Group (OMG), объединяющим более 800 компаний – разработчиков компьютеров и программного обеспечения. Начиная с 1989 г. задачей OMG является разработка спецификаций на архитектуру шины открытого программного обеспечения (брокера объектных запросов), с помощью которой объекты, написанные различными разработчиками смогут взаимодействовать друг с другом, находясь на различных компьютерах сети с различными операционными системами. Этот стандарт позволяет CORBA-объектам вызывать друг друга без знания о том, где объект находится и на каком языке написан. OMG специфицировала язык определения интерфейсов (IDL - Interface Definition Language), который используется для определения интерфейсов всех CORBA-объектов. CORBA-объекты отличаются от обычных объектов, определяемых различными программными языками, следующим:

- CORBA-объекты могут быть расположены в любом месте сети
- CORBA-объекты могут взаимодействовать с другими объектами на других платформах
- CORBA-объекты могут быть написаны на любом языке, который может быть отображен на IDL. Это отбражение в настоящее время определено для Java, C++, C, Smalltalk, COBOL и Ada.

Следующая диаграмма показывает, как запрос на выполнение какого-либо метода пересылается от клиента к CORBA-объекту на сервере. Клиентом может быть любой код, возможно тоже CORBA-объект. "ServerObject" обозначает действительную реализацию объекта – действительный код и данные, расположенные на сервере.

Рисунок 10.1. Запрос к удаленному объекту



Клиент имеет ссылку на удаленный объект, называемую "прокси". Клиент использует эту ссылку, чтобы вызвать метод удаленного объекта. Ссылка указывает на функцию стаба, которая используется механиз-

мом ORB для передачи запроса серверному объекту. Код стаба использует ORB для идентификации машины, на которой выполняется удаленный объект и запрашивает у ORB'а этой машины связь с объектом. Получив связь, стаб посылает объектную ссылку и параметры коду скелетона, связанному с реализацией удаленного объекта. Код скелетона преобразует вызов и параметры в формат, зависящий от языка реализации объекта (это называется "маршаллинг"), и вызывает метод объекта. Результаты и исключения возвращаются по этому же пути.

Клиент ничего не знает ни о расположении CORBA-объекта, ни о деталях его реализации, ни даже о том, какой ORB будет использован для вызова объекта. Различные ORB от различных производителей взаимодействуют друг с другом по специфицированному OMG Интернет меж-брокерному протоколу IIOP (Internet InterORB Protocol). Клиент может вызвать только те методы объекта, которые описаны в его публичном интерфейсе.

Интерфейс CORBA –объектов описывается на IDL. Этот язык очень похож на Java и C++. На нем описываются типы (классы) объектов, методы, параметры методов, а также типы возвращаемых значений и исключений. Компиляторы IDL, такие, как `idltojava`, транслируют определения CORBA-объектов на конкретный язык программирования в соответствии с правилами отображения, определенными OMG.

Стабы и скелетоны генерируются IDL компилятором для каждого типа объектов. Файлы стабов предоставляют клиенту возможность доступа к методам, описанным на IDL, на языке программирования клиента. Серверные скелетоны являются "клеем" между реализацией объекта и ORB.

Все CORBA-объекты должны поддерживать IDL интерфейс, определяющий их тип. Интерфейс может наследоваться от *одного или многих интерфейсов*. В качестве примера приведем интерфейс классической программы "Hello World":

```
module HelloApp {
  interface Hello {
    string sayHello();
  };
};
```

После того, как IDL интерфейсы написаны и скомпилированы транслятором `idltojava`, должны быть написаны Java файлы, содержащие реализацию интерфейсов. Эти файлы затем компилируются обычным Java транслятором (javac) вместе с интерфейсными файлами и библиотеками ORB, в результате чего получается объект сервера. Т.к. только серверы могут создать новые объекты, то должен быть определен интерфейс *объекта фабрики,* для каждого типа объектов. Фабрика – это объект, имеющий метод для создания экземпляров объектов того или иного класса. Например, если Document есть тип объекта, то реализация DocumentFactory может иметь метод create (имя метода может быть любым), создающий новые экземпляры объектов документов

```
Document document  = DocumentFactory.create();
orb.connect(document);
```

Java IDL ORB поддерживает только временные объекты, чье время жизни ограничено временем жизни серверного процесса. По завершению серверного процесса исчезают все связанные с ним серверное объекты. Свойство постоянства объекта может быть реализовано в самом объекте записью состояния объекта в файл или БД так, что объект сможет восстановить свое состояние при создании экземпляра.

Клиентский код компилируется вместе с файлами, полученными через `idltojava`, и библиотеками ORB. Клиенты могут создать CORBA-объект только через опубликованный фабричный интерфейс, обеспечиваемый сервером. Аналогично, клиент может удалить CORBA-объект только через опубликованный метод деструкции. Т.к. CORBA-объект может разделятся многими клиентами в сети, только сервер знает, когда его действительно можно удалить сборщиком мусора. Клиентский код обращается к CORBA-объекту через его объектную ссылку. Она представляет собой структуру, идентифицирующую компьютер CORBA-объекта, номер порта, который слушает листнер сервера для получения запросов, и указатель на данный объект в процессе. Т.к. поддерживаются только временные объекты, то ссылка становится неверной, если серверный процесс остановлен или рестартован. Клиенты получают ссылки сле-

дующими путями:

- Через объект фабрики. Например, клиент может обратиться к методу create фабрики DocumentFactory для создания нового документа. Метод create возвратит ссылку на объект нового документа
- Через службу имен. Например, ссылка на фабрику DocumentFactory может быть получена соответствующим запросом к службе имен
- Из строки, создаваемой специальным образом из объектной ссылки

После того, как ссылка получена, клиент должен сузить ее (*narrow*) до соответствующего типа. IDL поддерживает наследование. Корнем иерархии с IDL является `Object`, соответствующий классу `org.omg.CORBA.Object` в java. (Естественно, `org.omg.CORBA.Object` является подклассом `Object` в java). Многие операции, такие как поиск имени (lookup) и получение ссылки из строки (unstringifying), возвращают `org.omg.CORBA.Object`, который должен быть сужен с помощью helper-класса, генерируемого idltojava, до типа требуемого объекта. CORBA-объекты должны быть явно приведены к требуемому типу, поскольку их тип не известен исполняющей системе java во время выполнения.

Временная служба имен (Java IDL Transient Nameservice) – это серверный объект, поставляемый с Java IDL. Чтобы запустить ее, выполните команду `tnameserv` из командной строки с указанием порта прослушивания. Служба имен хранит объектные ссылки по имена в структуре дерева, аналогичной файловой системе. Клиент может обратиться к этой службе для поиска (lookup) или разрешения (resolve) объектной ссылки по ее имени. Т.к. эта служба временная, то дерево имен теряется при остановке службы.

Как всегда, чтобы быть уверенным и для того, чтобы лучше понять, как это все работает (и убедиться, что это вообще работает), выполним маленький пример. Мне, лично, эти примеры дают некоторую уверенность. Я знаю, что если пример проходит, я смогу сделать его как угодно сложным, дописав бесчисленные if-then-else и while-do. Для работы этого примера необходим компилятор idltojava из jdk1.3.

Создание IDL интерфейса

Откроем проводник и перейдем в каталог, где мы писали тестовые программы Hello.java. Создадим там файл HelloCORBA.idl с текстом

```
module HelloApp {
    interface HelloCORBA {
        string sayHello();
    };
};
```

Это - описание интерфейса на языке IDL. Вызовем интерпретатор командной строки и перейдем в этот каталог. Теперь скомпилируем IDL интерфейс. Программа idltojava в jdk1.3 называется idlj:

```
idlj –v –fall HelloCORBA.idl
```

-v означает verbose – с выводом промежуточных сообщений

-f специфицирует назначение – клиент, сервер или оба. all означает оба.

В проводнике посмотрим на этот каталог. В нем создался подкаталог HelloApp, в котором лежит шесть файлов:

- `_HelloCORBAImplBase.java`

  Этот абстрактный класс является серверным скелетоном, обеспечивающим CORBA-функциональность сервера. Он реализует интерфейс Hello.java. Серверный класс HelloServant расширяет класс _HelloCORBAImplBase.
- _HelloCORBAStub.java

  Этот класс является клиентским стабом, обеспечивающим CORBA-функциональность клиента. Он имплементирует HelloCORBA.java интерфейс.

- `HelloCORBA.java`

  Этот интерфейс содержит Java версию IDL интерфейса. В нем находится единственный метод `sayHello()`. `HelloCORBA.java` расширяет `org.omg.CORBA.Object`, также обеспечивая стандартную функциональность CORBA.
- `HelloCORBAHelper.java`

  Класс, используемый методом `narrow()` для приведения объектной ссылки CORBA к правильному типу.
- `HelloCORBAHolder.java`

  This final class holds a public instance member of type `Hello`. It provides operations for `out` and `inout` arguments, which CORBA has but which do not map easily to Java's semantics.
- `HelloCORBAOperations.java`

Все эти файлы сгенерированы idltojava транслятором. Посмотрите тексты этих файлов. Все они начинаются оператором

```
package HelloApp;
```

Это соответствует их размещению в данном подкаталоге. Скомпилируем их командой

```
javac HelloApp/*.java
```

Создадим теперь в нашем исходном каталоге файлы классов сервера и клиента. Класс сервера должен расширять класс HelloApp._HelloCORBAImplBase. Назовем этот файл HelloServer.java. Вот его текст:

```
//-- Файл HelloServer.java
package corba;

import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

class HelloServant extends _HelloCORBAImplBase {
   public String sayHello() {
     return "\nHello from CORBA object!!\n";
   }
}


public class HelloServer {

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

      // создаем объект сервера и регистрируем его в ORB
      HelloServant helloRef = new HelloServant();
      orb.connect(helloRef);

      // получаем корень контекста имен
      org.omg.CORBA.Object objRef =
      orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      // связываем ссылку на объект с именем
      NameComponent nc = new NameComponent("HelloCORBA", "");
      NameComponent path[] = {nc};
```

```
      ncRef.rebind(path, helloRef);

      // ждем вызовов от клиентов
      java.lang.Object sync = new java.lang.Object();
      synchronized (sync) {
        sync.wait();
      }

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

Этот класс реализует методы удаленного объекта. У нас есть только один метод `sayHello()`. Скомпилируем этот файл:

```
javac HelloServer.java
```

Теперь создадим класс клиента. Назовем этот файл `HelloClient.java`. Вот его текст:

```
//-- Файл HelloClient.java
package corba;

import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class HelloClient {
    public static void main(String args[])  {
      try{
        // create and initialize the ORB
        ORB orb = ORB.init(args, null);

        // получаем корень контекста имен
        org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);

        // находим ссылку на объект по имени в контексте
        NameComponent nc = new NameComponent("HelloCORBA", "");
        NameComponent path[] = {nc};
        HelloCORBA helloRef = HelloCORBAHelper.narrow(ncRef.resolve(path));

       // вызываем Hello серверный объект и выводим результат
       String hello = helloRef.sayHello();
       System.out.println(hello);

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

Скомпилируем этот файл:

```
javac HelloClient.java
```

Ну, вот, собственно и все. Теперь надо запустить сервер имен и опубликовать (зарегистрировать в нем) наш серверный объект. Для этого откроем еще два интерпретатора командной строки. В первом выпол-

ним команду:

```
tnameserv -ORBInitialPort 1051
```
запускающую сервер имен. Во втором окне выполним команду в каталоге, где лежит `HelloServer.java`

```
java corba.HelloServer -ORBInitialPort 1051
```
для опубликования серверного объекта. Выполним программу клиента в третьем окне:

```
java corba.HelloClient -ORBInitialPort 1051
```
Получим текст

```
Hello from CORBA object!!
```
Все работает. Это радует. Я выполнял этот пример по Window NT 4.0. Думаю, что он будет работать и под Windows 98 – все-таки это – многозадачная система. Хотя мы выполняем все это на одной машине – это действительно распределенная система, т.к. объекты взаимодействуют по протоколу IIOP поверх TCP/IP так, как будто они находятся на разных компьютерах.

Введение в RMI

Альтернативой CORBA является RMI (Remote Method Invocation) – вызов удаленных методов – стандарт, поддерживаемый SunSoft. RMI гораздо проще, чем CORBA, но до последного времени через RMI можно было работать только с объектами java. Последнее достижжение SunSoft – это "RMI поверх IIOP" - дающий возможность обращаться через RMI к CORBA объектам. Поэтому про CORBA мы забудем. Я рассказывал об этой архитектуре только для Вашего общего развития. Реально мы будем использовать RMI. RMI имеет следущие преимущества по сравнению с CORBA:

- Интерфейсы пишутся на java, поэтому не надо изучать IDL
- RMI гораздо легче для изучения и использования. В частности утилита rmic (аналог idltojava) производит только два файла. Эти файлы являются классами (имеют расширение class) и сразу готовы к использованию.
- RMI-IIOP обеспечивает связь с CORBA объектами (написанными на любом языке)

PerfectTime.java

Для того, чтобы по-быстрому ознакомиться с RMI, выполним маленький пример. Он взят из книги "Thinking in Java" Брюса Экела и изменен мной в соответствии с указаниями SunSoft для реализации последней версии RMI-IIOP. Пример состоит из серверной и клиентской части. Клиент запрашивает у сервера точное время (т.к. у разных клиентов может быть разное местное время). Сервер возвращает клиенту значение своего времени в миллисекундах.

Создадим в каталоге C:\src подкаталог rmi, где создадим три файла:

- PerfectTimeI.java

  Интерфейс серверной части
- PerfectTime.java

  Реализация интерфейса – серверная программа
- DisplayPerfectTime.java

  Клиентский код, запрашивающий и отображающий точное время

Вот их тексты:

```
//-- Файл PerfectTimeI.java
// The PerfectTime remote interface.
package rmi;
```

```
import java.rmi.*;

interface PerfectTimeI extends Remote {
  long getPerfectTime() throws RemoteException;
}
//--

//-- Файл PerfectTime.java
// The implementation of the PerfectTime remote object.
package rmi;

import java.rmi.*;
import java.net.*;

import javax.naming.*;
import javax.rmi.PortableRemoteObject;

public class PerfectTime
  extends PortableRemoteObject
  implements PerfectTimeI {
  // Implementation of the interface:
  public long getPerfectTime()
      throws RemoteException {
    return System.currentTimeMillis();
  }
  // Must implement constructor to throw RemoteException:
  public PerfectTime() throws RemoteException {
    // super(); // Called automatically
  }
  // Registration for RMI serving:
  public static void main(String[] args) {
    try {
      Context initialNamingContext = new InitialContext();
      PerfectTime pt = new PerfectTime();
      initialNamingContext.rebind(
        "PerfectTime", pt);
      System.out.println("Ready to do time");
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
//--


//-- Файл DisplayPerfectTime.java
// Uses remote object PerfectTime.
package rmi;

import java.rmi.*;

import javax.naming.*;
import javax.rmi.PortableRemoteObject;

public class DisplayPerfectTime {
  public static void main(String[] args) {
    try {
      Context initialNamingContext = new InitialContext();
```

```
      PerfectTimeI t =
        (PerfectTimeI)PortableRemoteObject.narrow(
         initialNamingContext.lookup("PerfectTime"),
                        PerfectTimeI.class);
      for(int i = 0; i < 10; i++)
        System.out.println("Perfect time = " +
          t.getPerfectTime());
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
//--
```

Встанем на каталог выше (чтобы папка rmi была в текущем каталоге). Поскольку наши программы создаются в пакете rmi, то требуется установить classpath так, чтобы классы пакета (папки) rmi были видны. В начале создадим файлы стаба и скелетона. Для этого служит программа rmic, которая в данном случае запускается из командной строки так:

```
rmic -classpath . –iiop rmi.PerfectTimeI
rmic -classpath . –iiop rmi.PerfectTime
```

rmic создаст два файла (стаб и скелетон), готовых к употреблению:

- `_PerfectTimeI_Stub.class`
- `_PerfectTime_Tie.class`

Потом скомпилируем наши файлы java (встав в каталог rmi):

```
javac PerfectTimeI.java
javac PerfectTime.java
javac DisplayPerfectTime.java
```

Вот, собственно и все. Запускаем временный сервер имен, как мы это делали в примере с CORBA:

```
start tnameserv
```

По умолчанию сервер будет слушать порт 900. Потом регистрируем серверный объект:

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory -Djava.naming.provider.url
```

Вместо "afl" должно быть подставлено имя компьютера, на котором это все делается. Потом выполняем клиента:

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory -Djava.naming.provider.url
```

И клиент показывает нам точное время в миллисекундах, прошедшее с полночи 01.01.1970 !

Глава 11. Enterprise Java Beans

EJB

При проектировании приложения производится декомпозиция бизнес-функций, описанных в спецификации требований, в набор компонентов, называемых *бизнес-объектами*. Бизнес-объекты, как и все объекты в объектно-ориентированном программировании, имеют состояние и поведение. Бизнес-специфичные правила, которые помогают определить структуру и поведение бизнес-объектов, вместе с пре- и пост-условиями, накладываемыми на поведение объекта, называются *бизнес-логикой*.

Этот урок является ключевым в разделе разработки компонент бизнес-логики. Предыдущее рассмотрение CORBA и RMI должно служить фундаментом для понимания этого урока. Используя jdbc, мы можем писать программы, работающие с базой данных. Компоненты этих программ мы можем распределить по сети и заставить взаимодействовать друг с другом через RMI. Казалось бы все? Однако, при построении большой промышленной системы встают другие побочные вопросы:

- как обеспечить безопасность системы, авторизованный доступ, аудит и т.п.?
- как обеспечить транзакции? Как обеспечить транзакции в распределенной базе данных?
- как обеспечить масштабирование системы?
- как обеспечить надежность системы, чтобы она не падала при выходе одного компьютера из строя?
- как повысить производительность системы – построить кеш, систему буферов и очередей, обеспечить жизненный цикл объектов?
- как обеспечить многопроцессорную обработку и как распределить ресурсы между процессами?
- как обеспечить повторное использование компонент при смене марки компьютера или операционной системы?
- как включать в систему новые компоненты, компоненты, приобретенные от других производителей и как продать свои компоненты?

Прямое решение этих вопросов через jdbc и RMI потребует от нас неимоверных усилий. Кроме того, эти вопросы встают перед каждым разработчиком. Поэтому Sun разработала общую схему решения: компоненты бизнес-логики должны выполняться в некоем контейнере. Этот контейнер и будет ответственен за решение всех общих вопросов. Обращение к компонентам будет выполняться только через контейнер, так, что он будет следить - разрешить доступ или нет, выделить дополнительные ресурсы или нет и т.п. Эти компоненты в контейнере и есть Enterprise Java Beans.

При разработке EJB, ни разработчик компонент (bean-ов), ни разработчик клиентской части приложения (уровня представления), не заботятся о таких деталях, как поддержка транзакций, безопасность, доступ к удаленным объектам и многих других сложных и трудно программируемых аспектах задачи. Они прозрачно для разработчика поддерживаются EJB сервером и контейнером. Кроме того, EJB разрабатываются полностью на java. Поэтому нам не надо изучать IDL. Из-за этой простоты, мы можем быстро разрабатывать приложения, использующие EJB. Эти приложения будут переносимыми, т.к. пишутся на java. Кроме того, EJB, работающий на одном EJB-сервере, будет работать и на другом, от другого поставщика, если эти серверы удовлетворяют спецификации на EJB. И хотя эта совместимость для большинства серверов еще не проверена, она обещается в будущем. Oracle8i JServer реализует спецификацию 1.0 EJB. Взаимодействие между компонентами реализовано через RMI "поверх IIOP". EJB бывают двух видов:

- Session beans – создаются для каждой сессии, т.е. вызова бина. Они манипулируют постоянными данными (т.е. данными БД), но сами не являются постоянными. Бин исчезает, если исчезает контейнер.
- Entity beans – являются постоянными – при крахе контейнера могут возобновить работу, когда контейнер будет восстановлен. (Конечно, без контейнера никакой бин работать не может!). Один такой бин может использоваться многими клиентами, в этом их отличие от session beans. Entity бины используются для представления разделяемых ресурсов, таких как записи базы данных. При этом задача синхронизации доступа решается сервером EJB.

Session beans в свою очередь тоже бывают двух видов:

- Stateless beans – бины без состояния – не сохраняют своего состояния между последовательными вызовами методов. Эти бины используются в основном для обработки частых, но коротких транзакций, типа имеющихся в системах OLTP Сервер EJB может использовать один и тот же объект (инстанс) stateless бина для нескольких сессий, поэтому если такой бин и имеет состояние, например, коннект к базе данных, это состояние не должно связываться с сессиями. Другими словами, stateless session бин предназначен чисто для реализации серверной логики.
- Statefull beans – бины с состоянием – сохраняют состояние между последовательными вызовами методов. Такие бины могут иметь состояние, связанной с сессией. В частности, такое состояние можно хранить в "куки" браузера клиента, но, если такая возможность отключена пользователем, то это состояние можно хранить на сервере в таких бинах.

Архитектура EJB

Наконец, мы дошли до самого интересного – как же писать и устанавливать EJB? Чтобы разработать законченный EJB, необходимо создать четыре компонеты:

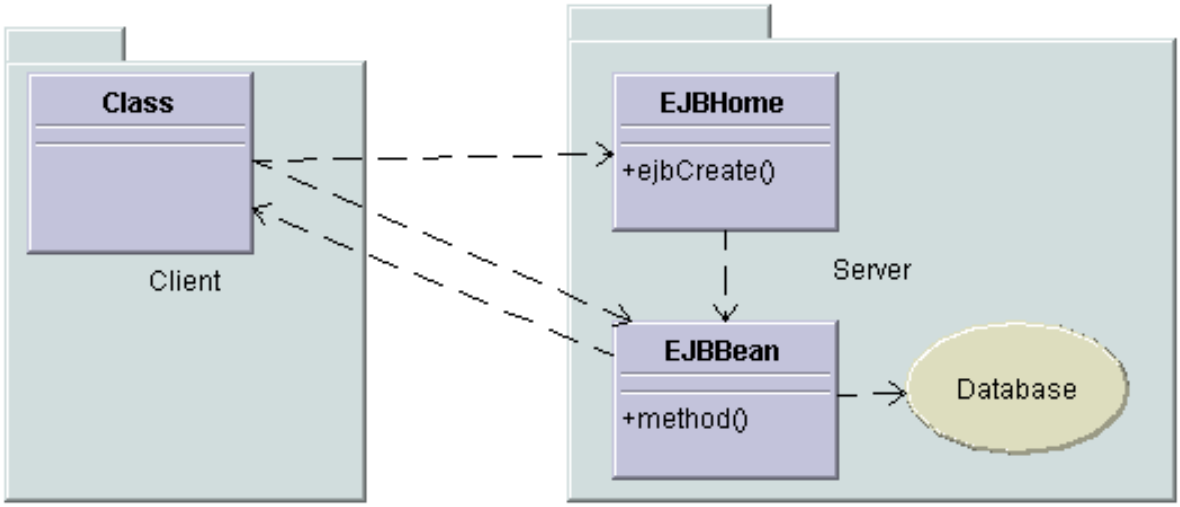
- домашний интерфейс (the home interface)
- удаленный интерфейс (the remote interface)
- реализацию методов интерфейсов – собственно класс разрабатываемой компонеты (EJB)
- дескриптор размещения (deployment descriptor) для каждого EJB

Дескриптор размещения, вообще говоря, должен создаваться некоторым средством размещения EJB. Так сделано в реализации J2EE. Поэтому нам надо озаботиться только первыми тремя компонентами.

Домашний интерфейс – это интерфейс, который реализуется в контейнере. Домашний интерфейс модет иметь метод `create()` для создания бина как объекта и `find()` для восстановления объекта из базы данных. Домашний интерфейс вместе с домашним объектом в действительности являются фабрикой EJB.

Удаленный интерфейс определяет методы, которые бин предоставляет клиенту. Эти методы составляют бизнес-логику приложения. Бин должен также предоставить дополнительные методы, которые вызываются контейнером на различных стадиях жизненного цикла бина. Клиентское приложение никогда не вызывает бин прямо. Вместо этого контейнер генерирует серверный объект, называемый EJBObject, который служит локальным заместителем серверного объекта (прокси) для клиента. Объект EJBObject получает сообщения от клиента, и затем контейнер обрабатывает эти сообщения прежде, чем они будут переданы бину. Зачем нужны эти сложности? Дело в том, что контейнер предоставляет свои сервисы прозрачно (незаметно) для бина. Например, если бин установлен с транзакционным аттрибутом, объявляющим, что бин должен выполняться в своем собственном транзакционном контексте, то контейнер может начать транзакцию до передачи сообщения бину и завершить или откатить транзакцию до передачи обратного сообщения от бина клиенту. Следующий рисунок показывает основные связи между компонентами бина:

Рисунок 11.1. Вызов методов EJB



Java-код бина должен содержать реализацию удаленного интерфейса и дополнительные методы, требуемые контейнером. Прежде, чем переходить к деталям архитектуры EJB, еще раз повторим основные концепции.

Итак, EJB выполняется в контейнере, который является частью EJB сервера. Контейнер обеспечивает EJB службами, причем так, что сам бин об этом не знает. Такими службами являются служба транзакций, служба синхронизации и служба безопасности.

Чтобы обеспечить эти службы, контейнер должен иметь возможность перехватывать вызовы методов бина. Например, если контейнер должен сделать бин транзакционным, то он должен создать новую транзакцию до того, как бин начнет выполнять какой-либо свой метод, и должен попытаться завершить транзакцию до того, как бин вернет результат клиенту. По этой и по другим причинам, клиентское приложение никогда не вызывает непосредственно методы бина. Вместо этого существует двухступенчатый процесс, в котором участвуют ORB и контейнер.

На первом шаге клиент вызывает локальный прокси-стаб удаленного метода. Стаб производит преобразование значений параметров (маршаллинг) так, что они могут быть переданы по сети. Не забудьте, что параметрами могут быть объекты, а не просто числа или строки, так что их не так просто передать. Затем стаб вызывает серверный скелетон. Последний декодирует значения параметров (операция обратная маршаллингу) и вызывает контейнер бина. Этот шаг требуется только из-за удаленной природы вызова. Заметьте, что этот шаг полностью прозрачен как для разработчика клиентского приложения, так и для разработчика бина. Эти детали Вам не обязательно знать, когда Вы пишете код приложения для сервера или клиента. Тем не менее, знать полезно знать, что происходит.

На втором шаге контейнер бина получает вызов от скелетона и выполняет службы, требуемые контекстом. Этими службами могут быть

- идентификация клиента при первом вызове
- выполнение управления транзакциями
- вызов синхронизирующего метода бина

Затем контейнер передает (делегирует) вызов бину. Бин выполняет затребованный метод. По завершению метода управление опять передается контейнеру бина, который опять вмешивается со своими службами. Например, он может попытаться закоммитить транзакцию, если бин выполняется в транзакционном контексте. Последнее зависит от значения атрибута транзакций в дескрипторе бина (а не от кода самого бина).

Затем контейнер вызывает скелетон, который производит маршаллинг возвращаемых данных, и передает их клиентскому стабу.

Эти шаги совершенно невидимы разработчикам как клиентской части приложения, так и бина. Одним из основных преимуществ модели EJB является скрытие всех этих сложных моментов от разработчиков.

Домашний интерфейс

Когда клиенту необходимо создать объект бина, он делает это через домашний интерфейс. Домашний интерфейс расширяет класс java.ejb.EJBHome. Этот интерфйес имеет один или более методов `create()` для создания объектов. Метод `create()`может принимать параметры, передаваемые от клиента. Для каждого метода `create()`, должен быть соответствующий метод `ejbCreate()`, указанный в удаленном интерфейсе с той же самой сигнатурой. Когда клиент вызывает метод `create()`, контейнер вмешивается (как всегда), чтобы предоставить службы, требуемые в этой точке, а затем вызывает соответствующий метод `ejbCreate()` самого бина. Ссылка на домашний интерфейс опубликовывается в базе данных утилитой `deployejb`. Эта ссылка – тот объект, который клиент ищет методом lookup, чтобы создать объект бина.

Удаленный интерфейс

Разработчик бина пишет удаленный интерфейс для каждого EJB в приложении. Удаленный интерфейс описывает бизнес-методы, которыми располагает данный бин. Каждый метод, к которому клиент должен получить доступ, должен быть описан в удаленном интерфейсе. Приватные методы бина не указываются в удаленном интерфейсе.

Сигнатура каждого метода в удаленном интерфейсе должна совпадать с сигнатурой реализации метода в бине.

Замечание

(Те, кто писал на PL/SQL могут узнать в удаленном интерфейсе спецификацию пакета, в то время, как реализация бина сродни телу пакета. Однако, удаленный интерфейс не объявляет публичных переменных. Он объявляет только методы, реализованные в бине.)

Удаленный интерфейс должен быть публичным и должен быть подклассом `javax.ejb.EJBObject`. Например, Вы можете написать удаленный интерфейс для управления персоналом как:

```
public interface employeeManagement extends javax.ejb.EJBObject {

  public void hire(int empNumber, String startDate, double salary)
        throws java.rmi.RemoteException; // Нанять на работу
  public double getCommission(int empNumber) throws java.rmi.RemoteException;
        // Установить комиссионные
  ...   // и т.д.
}
```

Все методы удаленного интерфейса должны бросать исключение `RemoteException`. Это обычный механизм уведомления клиента об ошибках выполнения бина. Однако, контейнер бина может возбудить и другие исключения, например, `SQLException`. Так как исключения передаются по сети, все они сериализуемы.

Бин

Сам бин – это Java класс, реализующий методы удаленного интерфейса, т.е. бизнес-логику. Он практически не отличается от тех компонент бизнес-логики, которые мы писали раньше. Отличия состоят в том, что бин должен реализовать интерфейс SessionBean (для session бинов) или EntityBean (для entity бинов) и каждый его публичный метод, описанный в удаленном интерфейсе, должен быть обявлен с возможностью исключения RemoteException. Кроме того, бин может получить переменные окружения (контекст) типа SessionContext. Например,

```
public class AcmeTitleBean implements SessionBean {
  SessionContext ctx;
  public String getTitle (int title_id)
```

```
        throws SQLException, RemoteException {
    ....
    }
    ....
}
```

Кроме методов бизнес-логики, EJB должен как минимум реализовать следующие методы, указанные в спецификации `javax.ejb.SessionBean` интерфейса:

- ejbActivate()

  Реализуйте его как пустой метод, т.к. он никогда не вызывается в данной реализации EJB сервера.
- EjbPassivate()

  Реализуйте его как пустой метод, т.к. он никогда не вызывается в данной реализации EJB сервера.
- ejbRemove()

  Контейнер вызывает этот метод в коце жизненного цикла объекта сессии. Этот метод должен выполнить завершающую подчистку, например закрыть открытые файлы.
- SetSessionContext(SessionContext ctx)

  Устанавливает контекст бина.

Дескриптор размещения

Во время создания бина мы ничего не знаем о контейнере, где он будет выполняться. Мы можем продать свой бин на сторону или наоборот приобрести бин от какого-либо поставщика. Поэтому роли разработчика бина и его "установщика" и различаются в спецификации. Установщик бина должен знать свойства контейнера, где он будет размещать бин. Эти свойства описываются в дескрипторе размещения. Поэтому основная задача установщика – написать дескриптор размещения бина. Для этого, конечно, надо знать и свойства самого бина – чтобы знать, что делает бин, а что возложить на контейнер.

Дескриптор размещения должен быть представлен в сериализованной форме, т.е. он должен быть преобразован в последовательность байт.

Пример сессионного бина без состояния

Создадим приложение, выполняющее умножение двух чисел. Оно будет состоять из сервлета, генерирующего HTML-страницы вывода, и session stateless ejb, выполняющего вычисления ("бизнес-логику"). Ввод данных будем производить из обычной HTML страницы. Вот она - файл `code/src/sbean/presentation/content/index.html`:

```
<HTML>
<BODY BGCOLOR = "WHITE">
<BLOCKQUOTE>
<title>Умножение</title>
<FORM METHOD="GET" ACTION="BonusAlias">
<P>
Первый сомножитель:
<P>
<INPUT TYPE="TEXT" NAME="SOCSEC"></INPUT>
<P>
Второй сомножитель:
<P>
<INPUT TYPE="TEXT" NAME="MULTIPLIER"></INPUT>
<P>
<INPUT TYPE="SUBMIT" VALUE="Submit">
<INPUT TYPE="RESET">
```

```
</FORM>
</BLOCKQUOTE>
</BODY>
</HTML>
```

Форма на этой странице посылает данные в BonusAlias - псевдониму (alias), под которым известен сервлет, принимающий данные. Создадим этот сервлет:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import Beans.*;

public class BonusServlet extends HttpServlet {
  CalcHome homecalc;
  public void init(ServletConfig config)
  throws ServletException{
    //Look up home interface
    try{
      InitialContext ctx = new InitialContext();
      Object objref = ctx.lookup("calcs");
      homecalc = (CalcHome)PortableRemoteObject.narrow(objref, CalcHome.class);
    } catch (Exception NamingException) {
      NamingException.printStackTrace();
    }
  }

  public void doGet (HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {
    String socsec = null;
    int multiplier = 0;
    double calc = 0.0;
    PrintWriter out;
    response.setContentType("text/html");
    String title = "EJB Example";
    out = response.getWriter();
    out.println("<HTML><HEAD><TITLE>");
    out.println(title);
    out.println("</TITLE></HEAD><BODY>");
    try{
      Calc theCalculation;
      //Get Multiplier and Social Security Information
      String strMult =
      request.getParameter("MULTIPLIER");
      Double integerMult = new Double(strMult);
      multiplier = integerMult.doubleValue();
      socsec = request.getParameter("SOCSEC");
      //Calculate bonus
      double bonus = (new Double(socsec)).doubleValue();
      theCalculation = homecalc.create();
      calc =
      theCalculation.calcBonus(multiplier, bonus);
    } catch(Exception CreateException){
      CreateException.printStackTrace();
    }
    //Display Data
    out.println("<H1>Результат умножения</H1>");
```

```
    out.println("<P>Множимое: " + socsec + "<P>");
    out.println("<P>Множитель: " +
    multiplier + "<P>");
    out.println("<P>Произведение: " + calc + "<P>");
    out.println("</BODY></HTML>");
    out.close();
  }

  public void destroy() {
    System.out.println("Destroy");
  }
}
```

Теперь создадим EJB, выполняющий вычисления. Для этого нам придется создать три файла:

- `CalcHome.java` - домашний интерфейс
- `Calc.java` - удаленный интерфейс
- `CalcBean.java` - тело бина

Вот их тексты:

```
//-- Файл CalcHome.java
package sbean;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
public interface CalcHome extends EJBHome {
  Calc create() throws CreateException,RemoteException;
}
```

```
//-- Файл Calc.java
package sbean;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface Calc extends EJBObject {
  public double calcBonus(double multiplier, double bonus) throws RemoteException;
}
```

```
//-- Файл CalcBean.java
package sbean;
import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
public class CalcBean implements SessionBean {
  public double calcBonus(double multiplier, double bonus) {
    double calc = (multiplier*bonus);
    return calc;
  }
  //Остальные методы пока реализуем как пустые
  public void ejbCreate() { }
  public void setSessionContext(SessionContext ctx) { }
  public void ejbRemove() { }
  public void ejbActivate() { }
  public void ejbPassivate() { }
  public void ejbLoad() { }
  public void ejbStore() { }
}
```

Трансляция JAVA файлов производится как обычно:

```
set JAVA_HOME=D:\java\sun\jdk1.3
set J2EE_HOME=D:\java\sun\j2sdkee1.2
set CPATH=%J2EE_HOME%\lib\j2ee.jar
cd ..
javac -d . -classpath %CPATH% sbean\*.java
```

После этого будут созданы классы бина в каталоге sbean и класс сервлета в текущем каталоге.

Установка приложения производится с помощью `deploytool`:

- Создается новое приложение
- В нем создается новый EJB, в Contents которого добавляются файлы EJB (весь каталог sbean)
- На следующем экране устанавливаются свойства бина: session stateless и указываются классы бина, домашнего и удаленного интерфейса. Будьте внимательны: по умолчанию в классах стоят не верные значения.
- Потом создается новая Web-компонента (сервлет) - вначале добавляется класс сервлета, потом HTML-страница
- При описании сервлета в URL-mapping прописывается сервлет-алиас (BonusAlias)
- В описание приложения следует указать имя JNDI для EJB - у нас это calcs.
- Затем надо установить приложение и посмотреть его браузером (начиная со страницы bonus.html).

Аналогично пишутся и устанавливаются entity beans и session statefull beans.

Простой entity бин

Entity бин реализует постоянные (хранимые) данные, представленные одной строкой таблицы базы данных. Строки в базе данных создаются и обновляются синхронно с изменением состояния entity бина. При этом нам не надо явно писать никаких SQL-предложений - все делается сервером EJB автоматически. Такие бины сохраняют свое состояние (потому что оно пишется в базу данных) после краха системы. Если транзакции обеспечиваются контейнером EJB, то после восстановления системы состояние записи базы данных будет соответствовать последней выполненной транзакции (commit'у). Создадим простой entity бин, соответствующий данным о читателе библиотеки. Для простоты будем считать, что читатель описывается одной строкой, содержащей ФИО. Взаимодействие с бином будем по-прежнему осуществлять через сервлет. Чтобы показать постоянство данных, создадим несколько читателей, вырубим сервер, а после его восстановления посмотрим список читателей. Таким образом, у нас должно быть три HTML-страницы: на ввод нового читателя, на подтверждение ввода и на просмотр списка читателей. Оформим эти страницы в виде одного сервлета:

```
//-- Файл ReaderServlet.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import ebean.*;

public class ReaderServlet extends HttpServlet {

  ReaderHome home;
  private static int counter = 1;

  public void init(ServletConfig config)
  throws ServletException{
    //Look up home interface
    try{
      InitialContext ctx = new InitialContext();
      Object objref = ctx.lookup("test/reader");
      home = (ReaderHome)PortableRemoteObject.narrow(objref, ReaderHome.class);
```

```
  } catch (Exception NamingException) {
    NamingException.printStackTrace();
  }
}

public void doGet (HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
  PrintWriter out;
  response.setContentType("text/html");
  out = response.getWriter();
  out.println("<HTML>");
  out.println("<LINK REL=STYLESHEET TYPE=\"text/css\" HREF=\"aqua.css\">");
  out.println("<META charset=\"windows-1251\" />");
  out.println("<HEAD>");
  out.println("</HEAD>");
  out.println("<BODY>");
  // Прием параметра
  String fio =  request.getParameter("FIO");
  ebean.Reader reader = null;
  try{
    if ( fio == null || fio.equals("") ) {
      // Выводим начальную страницу
      out.println("<DIV ALIGN=\"CENTER\">");
      out.println("<FORM NAME=\"login\" ACTION=\"ReaderServlet\">");
      out.println("<BR><BR><BR><BR>");
      out.println("Введите ФИО для регистрации или list для получения списка");
      out.println("<INPUT TYPE=\"TEXT\" NAME=\"FIO\">");
      out.println("<P>");
      out.println("<INPUT TYPE=\"SUBMIT\" VALUE=\"Отослать\">");
      out.println("<INPUT TYPE=\"RESET\" VALUE=\"Очистить\">");
      out.println("</FORM>");
    } else if ( !fio.equals("list") ) {
      // Перекодируем из ISO8859_1 в которой работает сервлет!
      fio = new String(fio.getBytes("ISO8859_1"), "Cp1251");
      // Создаем строку в базе данных и выводим подтверждение записи
      for (int j = 1; j < 100; j++ ) {
        try {
          reader = home.findByPrimaryKey(""+j);
          counter = j;
        } catch (Exception notFound) {}
      }
      reader = home.create("" + counter++, fio);
      out.println(fio + " записан в БД с id="+counter);
    } else {
      // Выводим список
      out.println("<DIV ALIGN=\"CENTER\">");
      out.println("<TABLE BORDER=\"1\" WIDTH=100% class=vrTable>");
      out.println("<THEAD ALIGN=CENTER class=vrTableHeader>");
      out.println("<TR CLASS=vrTableHeaderRow>");
      out.println("<TD>ФИО</TD>");
      out.println("</TR>");
      out.println("</THEAD>");
      out.println("<TBODY ALIGN=CENTER>");
      for (int j = 1; j < 100; j++ ) {
        try {
          reader = home.findByPrimaryKey(""+j);
          out.println("<TR><TD>"+reader.getFio()+"</TD></TR>");
        } catch (Exception notFound) {}
      }
```

```
        out.println("<TBODY>");
        out.println("</TABLE>");
      }
    } catch(Exception e){
      e.printStackTrace();
    }
    out.println("</BODY></HTML>");
    out.close();
  }

  public void destroy() {
    System.out.println("Destroy");
  }
}
//--
```

Теперь создадим entity bean. Он состоит из трех классов (в каталоге `ebean`):

Домашний интерфейс:

```
package ebean;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import javax.ejb.EJBHome;

public interface ReaderHome extends EJBHome {

  public Reader create(String id, String fio)
  throws CreateException, RemoteException;

  public Reader findByPrimaryKey(String id)
  throws FinderException, RemoteException;

}
```

Удаленный интерфейс:

```
package ebean;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Reader extends EJBObject {

public String getFio() throws RemoteException;

public String getId() throws RemoteException;

}
```

Сам бин:

```
package ebean;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
```

```
public class ReaderBean implements EntityBean {
  public String fio;
  public String id;

  private EntityContext ctx;

  public String getFio() {
    return this.fio;
  }

  public String getId() {
    return this.id;
  }

  public String ejbCreate(String id, String fio)
  throws CreateException{
    //Called by container after setEntityContext
    this.id=id;
    this.fio=fio;
    return null;
  }

  public void ejbPostCreate(String id, String fio) {
    //Called by container after ejbCreate
  }

  //These next methods are callback methods that
  //are called by the container to notify the
  //Bean some event is about to occur

  public void ejbActivate() {
    //Called by container before Bean
    //swapped into memory
  }

  public void ejbPassivate() {
    //Called by container before
    //Bean swapped into storage
  }

  public void ejbRemove() throws RemoteException {
      //Called by container before
  //data removed from database
  }

  public void ejbLoad() {
    //Called by container to
    //refresh entity Bean's state
  }

  public void ejbStore() {
    //Called by container to save
    //Bean's state to database
  }

  public void setEntityContext(EntityContext ctx){
    //Called by container to set Bean context
  }
```

```
  public void unsetEntityContext(){
    //Called by container to unset Bean context
  }
}
```

Компиляция этих файлов производится обычным образом. Для установки приложения необходимо запустить `j2ee, deploytool` и cloudscape. Последняя запускается командой

```
cloudscape -start
```

из каталога %J2EE_HOME%\bin.

При установке бина выбираем "entity" бин, container-managed persistance, resorce factory под произвольным CodedName с типом javax.sql.DataSource и Authentication=Container. После установки бина в его закладке "Entity" выбираем "Deployment Setting" и назначаем базу данных для хранения состояния бина. Имя этого дата сорса берется из строки

```
jdbc.datasources=jdbc/Cloudscape|jdbc:cloudscape:rmi:CloudscapeDB;
   create=true|jdbc/EstoreDB|jdbc:cloudscape:rmi:CloudscapeDB;
   create=true|jdbc/InventoryDB|jdbc:cloudscape:rmi:CloudscapeDB;
   create=true
```

файла конфигурации default.properties каталога %J2EE_HOME%\conf и равняется `jdbd/Cloudscape`.

В закладке приложения "JNDI" связываем имя из CodeName с именем jdbc/Cloudscape. Последнее - это имя в JNDI, под которым опубликована база данных. JNDI Name для бина устанавливаем как test/reader. Это - то имя, которое употреблено в lookup() метода init() сервлета.

При установке сервлета даем ему alias ReaderServlet и context root = test. Не забудьте включить в контекст файл aqua.css.

Глава 12. Web-сервисы

Web-сервисы

В объектно-ориентированном мире взаимодействие между объектами осуществляется посылкой им "сообщения". В частности сообщением может быть просто вызов того или иного метода объекта. Если объекты находятся на разных компьютерах, то такое сообщение является "вызовом удаленной процедуры" (RPC - remote procedure call). Попыткой объединить разнородные объекты является CORBA [rmi.xml] (common object request broker architecture). Эта архитектура основана на однообразном описании интерфейсов объектов с помощью IDL [rmi.xml#idl] (interface definition language) вне зависимости от языка, на котором реализованы объекты. Ни CORBA, ни RMI не являются действительно все-платформенными: CORBA требует присутствия ORB (object request broker) на машине клиента, а RMI работает только для Java-программ. Между тем, обмен сообщениями в Интернет достаточно легок - достаточно использовать уже готовые решения - протокол HTTP, request и response. Первым эта мысль пришла в голову компании HP, затем была подхвачена Microsoft и IBM. Окончательно стандарт на интеграцию приложений через Интернет был разработан w3-консорциумом (W3C). Последний дал следующее определение web-сервисам:

Web-сервисом называется программное приложение, идентифицируемое по URI, чьи интерфейсы и связывания могут быть определены, описаны и найдены как XML-артефакты. Web-сервисы поддерживают прямое взаимодействие с другими программными агентами используя обмен XML-сообщениями через протоколы интернета [1].

Это определение фиксирует только XML как язык сообщений, не уточняя протоколы Интернета, с помощью которых передаются сообщения. В настоящее время приняты следующие протоколы:

- SOAP (simple object access protocol) - обеспечивает взаимодействие между объектами, представляя собой "сообщение"
- WSDL (web service description language) - служит описанием интерфейса (вместо IDL)
- UDDI (universal description, discovery and integration) - помогает найти сервис в сети

Все эти протоколы реализуются поверх протокола HTTP (т.е. они - "легковесные"). Протокол SOAP по-видимому будет заменен протоколом XMLP, над которым в настоящее время работает W3C. Реализация протоколов поверх HTTP позволяет обойти проблему с Firewall, который часто не пропускает новый протокол (как, например, это было поначалу с IIOP). Поскольку в основе всех протоколов лежит XML, необходимо достаточно хорошее знание XML, включая XML-схему.

Механизм использования web-сервиса прост: Вы отсылаете запрос в виде xml-документа по некоторому адресу в Интернете и получаете оттуда ответ также в виде xml-файла. И-за этой простоты web-сервисы называют очередной революцией в информационной индустрии, наряду с появлением Java и XML.

Рассмотрим структуру запроса и ответа в протоколе SOAP на следующем примере. Пусть мы хотим получить список книг, имеющийся в библиотеке. Вот перехваченное с помощью tcp монитора (запуск tcp монитора осуществляется командой

`java -cp axis.jar org.apache.axis.utils.tcpmon`) сообщение, посланное серверу по протоколу HTTP:

```
POST /axis/services/BookList HTTP/1.0
Content-Length: 415
Host: localhost
Content-Type: text/xml; charset=utf-8
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
```

```
<SOAP-ENV:Envelope
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
  <getAllBooks/>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOPA-сообщение состооит из трех частей: конверта (Envelope), заголовка (в примере - отсутствует) и тела. Тело сообщения содержит название вызываемого метода и его параметры (у нас параметров нет). Сообщение отправлено по адресу `/axis/services/BookList` узлу `localhost`. Веб-сервис, находящийся по этому адресу, разбирает запрос, выполняет метод `getAllBooks` и отсылает ответное сообщение, так же перехваченное tcp монитором:

```
HTTP/1.1 200 OK
Date: Thu, 19 Sep 2002 08:33:14 GMT
Server: Jetty/4.0.1 (Windows 98 4.10 x86)
Servlet-Engine: Jetty/1.1 (Servlet 2.3; JSP 1.2; java 1.4.0-rc)
Content-Type: text/xml; charset=utf-8
Content-Length: 692

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <SOAP-ENV:Body>
  <getAllBooksResponse
          SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <getAllBooksResult
                  xsi:type="SOAP-ENC:Array"
                SOAP-ENC:arrayType="xsd:string[2]"
                xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
                        <item xsi:type="xsd:string">www.w3c.org, SOAP 1.1 Spec </item>
                        <item xsi:type="xsd:string">A.F.Lepekhine, Web Programmer </item>
        </getAllBooksResult>
  </getAllBooksResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

В теле этого сообщения находится массив из двух строк, элементы которого дают найденные книги.

Для того, чтобы узнать методы и аргументы веб-сервиса необходимо получить их описание. Для этого служит язык WSDL. Чтобы получить описание сервиса `BookList` наберите в браузере строку

```
        http://localhost:8080/axis/services/BookList?wsdl
```

Это делается после установки веб-сервиса примера нашего приложения. В результате будет показано описание интерфейса веб-сервиса в формате WSDL:

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://localhost:8080/axis/services/BookList"
```

```
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:intf="http://localhost:8080/axis/services/BookList"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:impl="http://localhost:8080/axis/services/BookList-impl"
xmlns="http://schemas.xmlsoap.org/wsdl/">
        <types>
                <schema xmlns="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://localhost:8080/axis/services/BookList">
                        <complexType name="ArrayOf_xsd_string">
                                <complexContent>
                                        <restriction base="SOAP-ENC:Array">
                                                <attribute ref="SOAP-ENC:arrayType" wsdl:arrayTyp
                                        </restriction>
                                </complexContent>
                        </complexType>
                        <element name="ArrayOf_xsd_string" nillable="true" type="intf:ArrayOf_xsd
                </schema>
        </types>
        <wsdl:message name="getAllBooksRequest" />
        <wsdl:message name="getAllBooksResponse">
                <wsdl:part name="return" type="intf:ArrayOf_xsd_string" />
        </wsdl:message>
        <wsdl:portType name="WebServiceBean">
                <wsdl:operation name="getAllBooks">
                        <wsdl:input message="intf:getAllBooksRequest" />
                        <wsdl:output message="intf:getAllBooksResponse" />
                </wsdl:operation>
        </wsdl:portType>
        <wsdl:binding name="BookListSoapBinding" type="intf:WebServiceBean">
                <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
                <wsdl:operation name="getAllBooks">
                        <wsdlsoap:operation soapAction="" />
                        <wsdl:input>
                                <wsdlsoap:body use="encoded"
                                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                                namespace="http://localhost:8080/axis/services/BookList" />
                        </wsdl:input>
                        <wsdl:output>
                                <wsdlsoap:body use="encoded"
                                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                                namespace="http://localhost:8080/axis/services/BookList" />
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>
        <wsdl:service name="WebServiceBeanService">
                <wsdl:port name="BookList" binding="intf:BookListSoapBinding">
                        <wsdlsoap:address location="http://localhost:8080/axis/services/BookList"
                </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

Литература

1. Архитектура Web-сервисов. Спецификация требований http://www.w3.org/TR/2002/WD-wsa-reqs-20020819