Лабораторная работа № 1

ЯЗЫК РАЗМЕТКИ ГИПЕРТЕКСТОВ HTML

Цель работы: приобретение навыков создания HTML – документов.

Изучаемые вопросы

1. Базовый синтаксис языка, основные элементы HTML – документа.
2. Заголовочная часть документа <HEAD>.
3. Комментарии.
4. Тэги тела документа.
5. Ссылки в HTML-документе.
6. Графика внутри HTML-документа.
7. Карты сообщений.
8. HTML фреймы.

Постановка задачи

Создайте HTML - документ, состоящий из двух горизонтальных фрэймов. В правом фрэйме разместите изображение. По нажатию на отдельные элементы изображения во втором фрэйме должно появиться описание данного элемента.

Теоретические сведения

Базовый синтаксис языка, основные элементы HTML – документа

HTML (от англ. HyperText Markup Language — «язык разметки гипертекста») — стандартный язык разметки документов во Всемирной паутине. Большинство веб-страниц создаются при помощи языка HTML (или XHTML). Язык HTML интерпретируется браузерами и отображается в виде документа в удобной для пользователя форме. Когда документ создан с использованием HTML, WEB-браузер может интерпретировать HTML для выделения различных

элементов документа и первичной их обработки. Использование HTML позволяет форматировать документы для их представления с использованием шрифтов, линий и других графических элементов на любой системе, их просматривающей.

HTML-тэги могут быть условно разделены на две категории:

1. тэги, определяющие, как будет отображаться WEB-браузером тело документа в целом;
2. тэги, описывающие общие свойства документа, такие как заголовок или автор документа.

HTML-документы могут быть созданы при помощи любого текстового редактора или специализированных HTML-редакторов и конвертеров.

Все тэги HTML начинаются с "<" (левой угловой скобки) и заканчиваются символом ">" (правой угловой скобки). Как правило, существует стартовый тэг и завершающий тэг. Завершающий тэг отличается от стартового прямым слешем перед текстом внутри угловых скобок. В примере тэг <TITLE> говорит WEB-браузеру об использовании формата заголовка, а тэг </TITLE> - о завершении текста заголовка.

Документ в формате HTML состоит из трех частей:

1. строки, содержащей информацию о версии HTML;
2. раздела заголовков (определяемого элементом HEAD);
3. тела, которое включает собственно содержимое документа. Тело может вводиться элементом BODY или элементом FRAMESET.

Перед каждым элементом или после каждого элемента может находиться пустое пространство (пробелы, переход на новую строку, табуляции и комментарии). Разделы 2 и 3 должны отделяться элементом HTML.

Вот пример простого документа HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
<HTML>
   <HEAD>
      <TITLE>Документ HTML</TITLE>
   </HEAD>
   <BODY>
      <P>Первый HTML документ.
```

```
   </BODY>
</HTML>
```

Заголовочная часть документа <HEAD>

Тэг заголовочной части документа должен быть использован сразу после тэга <HTML> и более нигде в теле документа. Данный тэг представляет из себя общее описание документа. Стартовый тэг <HEAD> помещается непосредственно перед тэгом <TITLE> и другими тэгами, описывающими документ, а завершающий тэг </HEAD> размещается сразу после окончания описания документа. Например:

```
<HTML>
<HEAD>
<TITLE> Список учащихся </TITLE>
</HEAD>
...
```

Комментарии

HTML позволяет вставлять в тело документа комментарии, которые сохраняются при передаче документа по сети, но не отображаются браузером. Синтаксис комментария:

```
<!-- Это комментарий -->
```

Комментарии могут встречаться в документе где угодно и в любом количестве.

Тэги тела документа

Тело документа должно находиться между тэгами <BODY> и </BODY>. Это та часть документа, которая отображается как текстовая и графическая (смысловая) информация вашего документа.

Ссылки в HTML-документе

Для того, чтобы браузер отобразил ссылку на URL, необходимо выделить URL специальными тэгами в HTML-документе. Синтаксис HTML, позволяющий это сделать - следующий:

```
<A HREF="URL"> текст подсвечивается как ссылка </A>
```

Тэг <A HREF="URL">открывает описание ссылки, а тэг </A> - закрывает его. Любой текст, находящийся между данными двумя тэгами подсвечивается специальным образом Web-браузером.

Графика внутри HTML-документа

Существует возможность включения ссылок на графические и иные типы данных в HTML-документах. Делается это при помощи тэга <IMG...ISMAP>.

Синтаксис тэга:

```
<IMG SRC="URL" ALT="text" HEIGHT=n1 WIDTH=n2
ALIGN=top|middle|bottom|texttop ISMAP>
```

Опишем элементы синтаксиса тэга:

URL – Обязательный параметр, имеющий такой же синтаксис, как и стандартный URL. Данный URL указывает браузеру где находится рисунок. Рисунок должен храниться в графическом формате, поддерживаемом браузером.

ALT="text" – Данный необязательный элемент задает текст, который будет отображен браузером, не поддерживающим отображение графики или с отключенной подкачкой изображений.

HEIGTH=n1 – Данный необязательный параметр используется для указания высоты рисунка в пикселях. Если данный параметр не указан, то используется оригинальная высота рисунка.

WIDTH=n2 – Параметр также необязателен, как и предыдущий. Позволяет задать абсолютную ширину рисунка в пикселях.

ALIGN – Данный параметр используется, чтобы сообщить браузеру, куда поместить следующий блок текста. Это позволяет более строго задать расположение элементов на экране. Если данный па-

раметр не используется, то большинство браузеров располагает изображение в левой части экрана, а текст справа от него.

ISMAP – Этот параметр сообщает браузеру, что данное изображение позволяет пользователю выполнять какие-либо действия, щелкая мышью на определенном месте изображения.

Пример:

```
<IMG SRC="http://www.bntu.by/images/povt.jpg"
ALT="ПОВТиАС лого" ALIGN="top" ISMAP>
```

Карты сообщений

Создание карты изображения является одной из привлекательнейших возможностей HTML, позволяющей пользователю привязывать ссылки на другие документы к отдельным частям изображений. Щелкая мышью на отдельных частях изображения, пользователь может выполнять те или иные действия, переходить по той или иной ссылке на другие документы и т.п.

Чтобы включить поддержку карты для изображения, необходимо ввести дополнительный параметр в тэг IMG:

```
<IMG SRC="url" USEMAP="url#map_name" >
```

Синтаксис:

<MAP NAME="map_name"> - Данный тэг определяет начало описания карты с именем map_name.

<AREA> - Описывает участок изображения и ставит ему в соответствие URL.

Параметры:

SHAPE - Необязательный параметр, указывающий на форму определяемой области изображения. Может принимать значения:

default - по умолчанию (обычно прямоугольник)

rect - прямоугольник

circle - круг

poly - многоугольник произвольной формы

COORDS - Координаты в пикселях описываемой области. Для прямоугольника это четыре координаты левого верхнего и правого

нижнего углов, для круга - три координаты (две - центр круга, третья - радиус). Для многоугольника это описание каждого угла в двух координатах - соответственно число координат равно удвоенному количеству углов.

Координаты считаются с нуля, поэтому для описания области 100 на 100 используется описание:

<AREA COORDS="0,0,99,99" >

HREF="url" - Описание ссылки, действия по которой будут выполняться при щелчке мыши в заданной области.

NOHREF - Параметр, указывающий, что ссылка отсутствует для данного участка. По умолчанию, если не указан параметр HREF, то считается что действует параметр NOHREF. Также, для всех неописанных участков изображения считается, что используется параметр NOHREF.

Если две описанных области накладываются друг на друга, то используется ссылка, принадлежащая первой из описанных областей.

</MAP> - Завешающий тег

Пример:

```
<MAP NAME="map_name">
<AREA [SHAPE=" shape "] COORDS="x,y,..."
[HREF=" reference "] [NOHREF]>
</MAP>
```

HTML фреймы

Фреймы, позволяющие разбивать Web-страницы на множественные скроллируемые подокна, могут значительно улучшить внешний вид и функциональность Web-приложений. Каждое подокно, или фрейм, может иметь следующие свойства:

Каждый фрейм имеет свой URL, что позволяет загружать его независимо от других фреймов

Каждый фрейм имеет собственное имя (параметр NAME), позволяющее переходить к нему из другого фрейма

Размер фрейма может быть изменен пользователем прямо на экране при помощи мыши (если это не запрещено указанием специального параметра)

Формат документа, использующего фреймы, внешне очень напоминает формат обычного документа, только вместо тэга BODY используется контейнер FRAMESET, содержащий описание внутренних HTML-документов, содержащий собственно информацию, размещаемую во фреймах.

```
<FRAMESET COLS="value" | ROWS="value">
<FRAME SRC="url1">
<FRAME ...>
...
</FRAMESET>

FRAMESET
<FRAMESET [COLS="value" | ROWS="value"]>
```

Тэг <FRAMESET> имеет завершающий тэг </FRAMESET>. Все, что может находиться между этими двумя тэгами, это тэг <FRAME>, вложенные тэги <FRAMESET> и </FRAMESET>, а также контейнер из тэгов <NOFRAME> и </NOFRAME>, который позволяет строить двойные документы для браузеров, поддерживающих фреймы и не поддерживающих фреймы.

Данный тэг имеет два взаимоисключающих параметра: ROWS и COLS.

ROWS="список-определений-горизонтальных-подокон" - Данный тэг содержит описания некоторого количества подокон, разделенные запятыми. Каждое описание представляет собой числовое значение размера подокна в пикселях, процентах от всего размера окна или связанное масштабное значение. Отсутствие атрибута ROWS определяет один фрейм, величиной во все окно браузера.

Синтаксис используемых видов описания величин подокон:

value - Простое числовое значение определяет фиксированную высоту подокна в пикселях.

value% - Значение величины подокна в процентах от 1 до 100. Если общая сумма процентов описываемых подокон превышает 100, то размеры всех фреймов пропорционально уменьшаются до суммы 100%. Если, соответственно, сумма меньше 100, то размеры пропорционально увеличиваются.

value* - Вообще говоря, значение value в данном описании является необязательным. Символ "*" указывает на то, что все оставшееся место будет принадлежать данному фрейму.

COLS="список-определений-горизонтальных-подокон" - То же самое, что и ROWS, но делит окно по вертикали, а не по горизонтали.

Пример:

```
<FRAMESET COLS="50,*,50"> - описывает три фрейма,
два по 50 точек справа и слева, и один внутри этих
полосок.
FRAME
<FRAME SRC="url" [NAME="frame_name"]
[MARGINWIDTH="nw"] [MARGINHEIGHT="nh"]
[SCROLLING=yes|no|auto] [NORESIZE]>
```

Данный тэг определяет фрейм внутри контейнера FRAMESET.

SRC="url" - Описывает URL документа, который будет отображен внутри данного фрейма. Если он отсутствует, то будет отображен пустой фрейм.

NAME="frame_name" - Данный параметр описывает имя фрейма. Имя фрейма может быть использовано для определения действия с данным фреймом из другого HTML-документа или фрейма (как правило, из соседнего фрейма этого же документа). Имя обязательно должно начинаться с символа. Содержимое поименованных фреймов может быть задействовано из других документов при помощи специального атрибута TARGET, описываемого ниже.

MARGINWIDTH="value" - Это атрибут может быть использован, если автор документа хочет указать величину разделительных полос между фреймами сбоку. Значение value указывается в пикселях и не может быть меньше единицы.

MARGINHEIGHT="value" - То же самое, что и MARGINWIDTH, но для верхних и нижних величин разделительных полос.

SCROLLING="yes | no | auto" - Этот атрибут позволяет задавать наличие полос прокрутки у фрейма.

NORESIZE - Данный атрибут позволяет создавать фреймы без возможности изменения размеров.

NOFRAMES - Данный тэг используется в случае, если создается документ, который может просматриваться как браузерами, поддерживающими фреймы, так и браузерами, их не поддерживающими. Данный тэг размещается внутри контейнера FRAMESET, а все, что находится внутри тэгов <NOFRAMES> и </NOFRAMES> игнорируется браузерами, поддерживающими фреймы.

Примеры:

```
<FRAMESET ROWS="*,*">

<NOFRAMES>
<H1>Ваша версия WEB-браузера не поддерживает фрей-
мы!</H1>
</NOFRAMES>

<FRAMESET COLS="65%,35%">
<FRAME SRC="link1.php">
<FRAME SRC="link2.php">
</FRAMESET>

<FRAMESET COLS="*,40%,*">
<FRAME SRC="link3.php">
<FRAME SRC="link4.php">
<FRAME SRC="link5.php">
</FRAMESET>

</FRAMESET>
```

Контрольные вопросы:

1. На какие части разделяется HTML-документ?
2. При помощи какого тэга в HTML-документ добавляется графика?
3. Назовите основные тэги формы?
4. Для чего используется карта сообщений?
5. Для чего используется фрэйм NOFRAMES?

Лабораторная работа № 2

КАСКАДНЫЕ ТАБЛИЦЫ СТИЛЕЙ (CSS)

Цель работы: приобретение навыков создания HTML – документов с использованием CSS.

Изучаемые вопросы

1. Синтаксис и элементы CSS.
2. Добавление стилей CSS в HTML-документ.
3. Связывание.
4. Внедрение.
5. Встраивание в теги документа.
6. Импортирование.

Постановка задачи

Создать HTML-документ и оформить его при помощи CSS. Все стили необходимо вынести в отдельный файл.

Теоретические сведения

CSS (англ. Cascading Style Sheets - каскадные таблицы стилей) — формальный язык описания внешнего вида документа, написанного с использованием языка разметки. Преимущественно используется как средство описания, оформления внешнего вида веб-страниц, написанных с помощью языков разметки HTML и XHTML, но может также применяться к любым XML-документам, например, к SVG или XUL.

Основным понятием CSS является стиль – т. е. набор правил оформления и форматирования, который может быть применен к различным элементам документа. В стандартном HTML для присвоения какому-либо элементу определенных свойств (таких, как цвет, размер, положение на странице и т. п.) приходилось каждый раз описывать эти свойства, увеличивая размер файла и время загрузки на компьютер просматривающего ее пользователя.

CSS действует более удобным и экономичным способом. Для присвоения какому-либо элементу определенных характеристик необходимо один раз описать этот элемент и определить это описание как стиль, а в дальнейшем просто указывать, что элемент, который нужно оформить соответствующим образом, должен принять свойства указанного стиля.

Более того, можно сохранить описание стиля не в тексте кода документа, а в отдельном файле – это позволит использовать описание стиля на любом количестве Web-страниц, а также изменить оформление любого количества страниц, исправив лишь описание стиля в одном (отдельном) файле.

Кроме того, CSS позволяет работать со шрифтовым оформлением страниц на гораздо более высоком уровне, чем стандартный HTML, избегая излишнего утяжеления страниц графикой.

```
<html>
<head>
<style type="text/css">
    .newfont{font-size:24px; color:#CC9933}
</style>
<title>Классы для создания тэгов.</title>
<meta http-equiv="Content-Type" content="text/html;
charset=windows-1251">
</head>
<body>
<blockquote class=" newfont">Заголовок</blockquote>
</body>
</html>
```

Данный пример иллюстрирует вариант объявления нового стиля в документе и потом его использования.

Синтаксис и элементы CSS

Добавление стилей CSS в HTML-документ

Существует несколько способов связывания документа и таблицы стилей:

Связывание - позволяет использовать одну таблицу стилей для форматирования многих страниц HTML

Внедрение - позволяет задавать все правила таблицы стилей непосредственно в самом документе

Встраивание в теги документа - позволяет изменять форматирование конкретных элементов страницы

Импортирование - позволяет встраивать в документ таблицу стилей, расположенную на сервере.

Связывание

Напомним, что информация о стилях может располагаться либо в отдельном файле, либо непосредственно в коде документа. Расположение описания стилей в отдельном файле целесообразно при применении стилей при количестве страниц более одной. Для этого необходимо создать текстовый файл, описать необходимые стили и в коде документов, которые будут использовать эти стили необходимо создать ссылку на данный файл. Отметим, что данный файл может располагаться где угодно, необходимым условием является только то, чтобы браузер клиента мог его загрузить на свою сторону. Осуществляется это с помощью тега LINK, располагающегося внутри тега HEAD документов:

```
<LINK REL=STYLESHEET TYPE="text/css" HREF="URL">
```

Первые два параметра этого тега являются зарезервированными именами, требующимися для того, чтобы сообщить браузеру, что на этой страничке будет использоваться CSS. Третий параметр – HREF= «URL» – указывает на файл, который содержит описания стилей. Этот параметр должен содержать либо относительный путь к файлу – в случае, если он находится на том же сервере, что и документ, из которого к нему обращаются – или полный URL («http://...») в случае, если файл стилей находится на другом сервере.

```
<head>
<title></title>
<meta http-equiv="content-type" cotent=
```

```
   "text/html; charset=windows-1251">
   <link rel="stylesheet" href="css/default.css">
   </head>
```

Внедрение

Второй вариант, при котором описание стилей располагается в коде Web-странички, внутри тега HEAD, в теге <STYLE type="text/css">... </STYLE. В этом случае вы можете использовать эти стили для элементов, располагающихся в пределах странички. Параметр type="text/css" является обязательным и служит для указания браузеру использовать CSS.

```
   <head>
   <style type="text/css" >
       .el_cl_1{display:inline; z-index:1};
       .el_lst{display:list-item; margin:-1%;
   background:#ff0000 url("bc.jpg") no-repeat};
   </style>
   <title></title>
   <meta http-equiv="Content-Type" content="text/html;
charset=windows-1251">
   </head>
```

Встраивание в теги документа

Данный, третий по счету, метод позволяет располагать описания стилей непосредственно внутри тега элемента, который описывает. Это осуществляется при помощи параметра STYLE, используемого при применении CSS с большинством стандартных тегов HTML. Данный метод нежелателен, т.к. приводит к потере одного из основных преимуществ CSS – возможности разделения информации и описания оформления информации.

```
   <blockquote
style="color:#CCFF66">Внимание!</blockquote>
```

Импортирование

В теге <STYLE> можно импортировать внешнюю таблицу стилей с помощью свойства @import таблицы стилей:

```
@import: url(styles.css);
```

Его следует задавать в начале стилевого блока или связываемой таблицы стилей перед заданием остальных правил. Значение свойства @import является URL файла таблицы стилей.

Заметим, что импортирование от связывания отличается тем, что при импортировании можно не только поместить внешнюю таблицу стилей в документ, но и поместить одну внешнюю таблицу стилей в другую.

```
<head>
<title>Untitled </title>
<meta http-equiv="Content-Type" content="text/html;
charset=windows-1251" />
<style type="text/css">
@import url('css/default.css');
</style>
</head>
```

Приведем пример использования свойства текста на рисунке 2

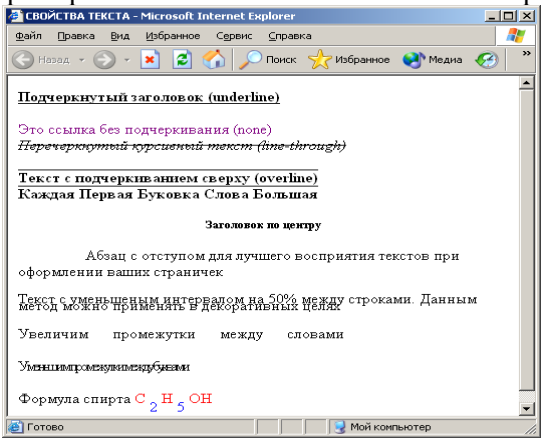

Рисунок 2

Ниже приведен код примера.

```
<STYLE type="text/css">
H4 {text-decoration: underline;}
A {text-decoration: none;}
i {text-decoration:line-through;}
b {text-decoration:overline;}
H5 {text-align: center}
b.cap {text-transform:capitalize;}
.otstup {text-indent: 50pt;}
.interval {line-height: 50 %}
</STYLE>
<h4>Подчеркнутый заголовок (underline)</h4>
<a href="/css/003/text.htm">Это ссылка без подчер-
кивания (none)</a><br>
<i>Перечеркнутый курсивный текст (line-
through)</i><p>
<b>Текст с подчеркиванием сверху (overline)</b><br>
<b class=cap>каждая первая буковка слова боль-
шая</b>
<h5>Заголовок по центру</h5>
<p class=otstup>Абзац с отступом для лучшего вос-
приятия текстов при оформлении ваших страничек</p>
<p class=interval>Текст с уменьшеным интервалом на
50% между строками.
Данным метод можно применять в декоративных це-
лях</p>
<p><span style="word-spacing: 15pt">Увеличим проме-
жутки между словами</span>
<p><span style="letter-spacing: -2pt">Уменьшим про-
межутки между буквами</span>
<p>Формула спирта
<span style=color:red>C</span>
<span style= vertical-
align:sub;color:blue;>2</span>
<span style=color:red>H</span>
<span style="color:blue; vertical-
align:sub;">5</span>
<span style=color:red>OH</span>
```

Контрольные вопросы:

1. Для чего используются каскадные таблицы стилей?
2. Какими способами таблицы стилей связываются с элементами документа?
3. Каковы основные отличия импортирования от связывания?
4. Каким образом сделать так, чтобы изменялся цвет ссылок только внутри тэга <b>?

Лабораторная работа № 3

ЯЗЫК СОСТАВЛЕНИЯ СЦЕНАРИЕВ JavaScript.

Цель работы: приобретение навыков создания HTML – документов с использованием JavaScript.

Изучаемые вопросы

1. Типы данных
2. Преобразование типов данных
3. Объявление переменных
4. Оператор цикла
5. Объектная модель JavaScript
6. Функции и методы

Постановка задачи

Создать HTML-документ, содержащий массив, в котором хранятся ссылки на объекты.

Теоретические сведения

JavaScript - это язык для составления сценариев, позволяющих выполнять разные действия непосредственно на машине пользователя. Располагаются данные сценарии внутри HTML документов.

JavaScript применяется для проверки правильности заполнения форм, создания удобной навигации и т.д.

Это язык программирования, который понятен браузеру. Это означает, что браузер умеет выполнять (интерпретировать) команды этого языка.

Программу на JavaScript можно помещать внутрь HTML-кода или держать в отдельном файле. Этот файл браузер прочитает (по специальной команде) во время интерпретации HTML-программы.

Программы на JavaScript (их называют скриптами) не работают самостоятельно. Коды JavaScript дополняют коды HTML и "живут" только вместе с ними. Даже если они расположены в отдельном файле

Размещение JavaScript на HTML-странице

Скрипт размещается между двумя парными тегами <SCRIPT>...</SCRIPT>. Обычно запись скрипта выглядит так:

```
<SCRIPT                    Начало скрипта
language=JavaScript>         Скрипт представлен как
<!--                         HTML-комментарий, чтобы
  ...                        не "смущать" браузеры,
  Код на                     которые о скриптах  не
JavaScript                   знают.
  ...
//-->
</SCRIPT>                  Конец скрипта

<NOSCRIPT>                 Эта команда -
  ...                      специально для
  Для браузеров,           пользователей,
которые                    у которых
  не поддержи-             браузер не понимает
вают JavaScript            скриптов.
  ...
</NOSCRIPT>
```

Типы данных

JavaScript распознает следующие типы величин:

- числа, типа 42 или 3.14159;
- логические (Булевы), значения true или false;
- строки, типа "Howdy!";
- пустой указатель, специальное ключевое слово, обозначающее нулевое значение.

Преобразование типов данных

Тип переменной зависит от того, какой тип информации в ней хранится. JavaScript не является жестко типизированным языком. Это означает, что программист может не определять тип данных переменной, в момент ее создания. Тип переменной присваивается переменной автоматически в течение выполнения. Таким образом можно определить переменную следующим способом:

```
var answer = 42
```

А позже, можно присвоить той же переменной, например следующее значение:

```
answer = "Thanks for all the fish..."
```

Объявления переменных

Переменная должна быть объявлена до ее использования.

Для объявления используется ключевое слово **var**:

```
var x;       // переменная с именем "x".
var y = 5;  // описание с присвоением числа.
var mes = "дядя Федор"; // описание с присвоением строки.
```

Оператор цикла

```
for (нач; усл; приращ) команда
```

Команда "нач" выполняется один раз перед входом в цикл. Цикл состоит в повторении следующих действий:

проверка условия "усл";

выполнение команды "команда";

выполнение команды "прираш".

Если условие ложно, цикл прекращается ("команда" и "прираш" после отрицательной проверки не работают).

```
// Произведение нечетных чисел массива
var set = new Array(1,2,3,4,5,6,7,8,9);
var p = 1;
for(var i=0; i<set.length; i++)
  if (set[i]%2) p *= set[i];
alert(p);
```

Условная команда

```
if (условие) команда1; else  команда2;
```

или

```
if (условие) команда1;
```

Если условие принимает значение true, выполняется команда1, иначе команда2. В сокращенной форме ветвь else отсутствует.

```
// Абсолютное значение числа
var x = -25.456;
if (x < 0) x = -x;
alert(x);
```

Объектная модель JavaScript

JavaScript основан на простом объектно-ориентированном примере. Объект - это конструкция со свойствами, которые являются переменными JavaScript. Свойства могут быть другими объектами. Функции, связанные с объектом известны как методы объекта.

В дополнение к объектам, которые сформированы в Navigator client и LiveWire server, вы можете определять ваши собственные объекты.

Объекты и Свойства

Объект JavaScript имеет свойства ассоциированные с ним. Обращаться к свойствам объекта необходимо следующей простой системой обозначений:

```
objectName.propertyName
```

И имя объекта и имя свойства чувствительны к регистру.

Например, пусть существует объект, с именем myCar. Можно задать свойства, именованные make и year следующим образом:

```
myCar.make = "Ford"
myCar.year = 69;
```

Также можно обратиться к этим свойствам, используя систему обозначений таблицы следующим образом:

```
myCar["make"] = "Ford"
myCar["year"] = 69;
```

Функции и Методы

Функции - один из фундаментальных встроенных блоков в JavaScript. Функция - JavaScript процедура - набор утверждений, которые выполняют определенную задачу.

Определение функции состоит из ключевого слова function , сопровождаемого

1. именем функции;
2. списком аргументов функции, приложенной в круглых скобках, и отделяемые запятыми;
3. JavaScript утверждениями, которые определяют функцию, приложенные в фигурных скобках, {...}.

Можно использовать любые функции, определенные в текущей странице. Лучше всего определять все функции в HEAD страницы. Когда пользователь загружает страницу, сначала загружаются функции.

Утверждения в функциях могут включать другие обращения к функции.

Например, есть функция с именем pretty_print:

```
function pretty_print(string)
{ document.write("" + string) }
```

Эта функция принимает строку как аргумент, прибавляет некоторые теги HTML, используя оператор суммы (+), затем показывает результат в текущем документу.

Определение функции не выполняет ее. Для этого необходимо вызвать функцию, чтобы выполнить ее. Например, можно вызывать функцию pretty_print следующим образом:

```
< SCRIPT>
pretty_print("This is some text to display")
</ SCRIPT>
```

Аргументы функции сохраняются в таблице. Внутри функции, вы можете адресовать параметры следующим образом:

```
functionName.arguments [i]
```

Где functionName - имя функции, и i - порядковое число аргумента, начинающегося с нуля. Так, первый аргумент в функции, с именем myfunc, будет myfunc.arguments [0]. Общее число аргументов обозначено переменным arguments.length.

Определение Методов

Метод - функция, связанная с объектом. Метод определяется таким же образом, как и стандартная функция. Затем, используется следующий синтаксис, чтобы связать функцию с существующим бъектом:

```
object.methodname = function_name
```

Где object - существующий объект, methodname - имя, которое присвоено методу, и function_name - имя функции.

Можно вызывать метод в контексте объекта следующим образом:

```
object.methodname (params);
```

Создание Новых Объектов

И клиент и сервер JavaScript имеют строки предопределенных объектов. Кроме того, можно создавать собственные объекты. Создание собственного объекта требует двух шагов:

1. Определить тип объекта, написанной функции.
2. Создать образец объекта с **new**.

Чтобы определять тип объекта, необходимо создать функцию для типа объекта, которая определяет его имя, и его свойства и методы. Например, пусть необходимо создавать тип объекта для автомобилей. Тип объектов будет назван car, и необходимо, чтобы он имел свойства для make, model, year, и color. Чтобы сделать это, необходимо написать следующую функцию:

```
function car(make, model, year) {
   this.make = make;
   this.model = model;
   this.year = year;
}
```

Замечание, используйте this, чтобы присвоить значения свойствам объекта, основанные на значениях функции.

Теперь можно создавать объект, с именем mycar следующим образом:

```
mycar = new car("Eagle", "Talon TSi", 1993);
```

Объект может иметь свойство, которое является самостоятельным другим объектом.

Можно определять методы для типа объекта включением определение метода на определении типа объекта. Например, пусть есть набор файлов изображений GIF, и необходимо определить метод, который показывает информацию для car, наряду с соответствую-

щим изображением. Для этого необходимо определить функцию типа:

```
function displayCar() {
   var result = "A Beautiful " + this.year
                + " " + this.make + " " +
this.model;
   pretty_print(result)
}
```

Где pretty_print - предопределенная функция, которая показывает строку. Используйте this, чтобы обратиться к объекту, который принадлежит методу.

Далее необходимо определить функцию методом из car, прибавляя утверждение

```
 This.displayCar = displayCar;
```

к определению объекта. Так, полное определение car теперь выглядит так:

```
function car(make, model, year, owner) {
   this.make = make;
   this.model = model;
   this.year = year;
   this.owner = owner;
   this.displayCar = displayCar;
}
```

Новый метод можно вызывать следующим образом:

```
car1.displayCar ()
car2.displayCar ()
```

Использование this для Ссылок Объекта

JavaScript имеет специальное ключевое слово, this, которое используется, чтобы обращаться к текущему объекту. Например, есть функция с именем validate , которая проверяет правильность свойства значения объекта, данного объект, и high и low значения:

```
function validate(obj, lowval, hival) {
   if ((obj.value < lowval) || (obj.value > hival))
      alert("Invalid Value!")
```

```
}
```

Вызывать validate можно в каждом элементе формы обработчика событий onChange, используя this, как показано в следующем примере:

```
< INPUT TYPE = "text"
        NAME = "age"
        SIZE = 3
        onChange="validate(this, 18, 99)">
```

Вообще, метод this обращается к вызывающему объекту.

Лабораторная работа № 4

СЕТЕВОЕ ПРОГРАММИРОВАНИЕ С СОКЕТАМИ И КАНАЛАМИ.

Цель работы: приобретение навыков проектирования и разработки приложений в архитектуре клиент-сервер.

Изучаемые вопросы

1. Тестирование программ без сети.
2. Порт
3. Сокеты
4. Простейший сервер и клиент

Постановка задачи

Создать на основе сокетов клиент/серверное приложение.

Теоретические сведения

Клиент-сервер (англ.Client-server) — вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг (сервисов), называемыми серверами, и заказчиками услуг, называемыми клиентами. Нередко клиенты и серверы взаимодействуют через компьютерную сеть и могут быть как различными физическими устройствами, так и программным обеспечением.

Таким образом, работа сервера состоит в прослушивании соединения, она выполняется с помощью специального объекта, который вы создаете. Работа клиента состоит в попытке создать соединение с сервером, и это выполняется с помощью специального клиентского объекта, который вы создаете. Как только соединение установлено, вы увидите, что и клиентская, и серверная сторона соединения превращаются в потоковый объект ввода/вывода, таким образом вы можете трактовать соединение, как будто вы читаете и пишете файл. Таким образом, после установки соединения, вы просто используете хорошо знакомые команды ввода/вывода. Это одна из особенностей работы по сети в Java.

Порт

IP адреса не достаточно для уникальной идентификации сервера, так как на одной машине может существовать несколько серверов. Каждая IP машина также содержит порты, и когда вы устанавливаете клиент или сервер, вы должны выбрать порт, через который и клиент, и сервер согласны соединиться.

Порт - это программная абстракция. Клиентская программа знает, как соединится с машиной через ее IP адрес, но она не может соединится с определенной службой на этой машине. Таким образом номер порта стал вторым уровнем адресации. Идея состоит в том, что при запросе определенного порта вы запрашиваете службу, ассоциированную с этим номером порта. Обычно каждая служба ассоциируется с уникальным номером порта на определенной серверной машине. Клиент должен предварительно знать, на каком порту запущена нужная ему служба.

Системные службы зарезервировали использование портов с номерам от 1 до 1024, так что вы не можете использовать этот или любой другой порт, про который вы знаете, что он задействован.

Сокеты

Сокет - это программный интерфейс, предназначенный для передачи данных между приложениями. Что же касается типов сокетов, то их два - потоковые и датаграммные.

В Java вы создаете сокет, чтобы создать соединение с другой машиной, затем вы получаете InputStream и OutputStream (или, с соответствующими конверторами, Reader и Writer) из сокета, чтобы получить возможность трактовать соединение, как объект потока ввода/вывода. Существует два класса сокетов, основанных на потоках: ServerSocket, который использует сервер для "прослушивания" входящих соединений, и Socket, который использует клиент для инициализации соединения. Как только клиент создаст сокетное соединение, ServerSocket возвратит (посредством метода accept( )) соответствующий Socket, через который может происходить коммуникация на стороне сервера. На этой стадии вы используете методы getInputStream( ) и getOutputStream( ) для получения соответствующих объектов InputStream'a и outputStream'a для каждого сокета. Они должны быть обернуты внутрь буферных и

форматирующих классов точно так же, как и другие объекты потоков.

Когда вы создаете ServerSocket, вы даете ему только номер порта. Вы не даете ему IP адрес, поскольку он уже есть на той машине, на которой он установлен. Однако когда вы создаете Socket, вы должны передать ему и IP адрес, и номер порта, с которым вы хотите соединиться. Socket, который возвращается из метода ServerSocket.accept( ) уже содержит всю эту информацию.

Простейший сервер и клиент

Этот пример покажет простейшее использование серверного и клиентского сокета. Все, что делает сервер, это ожидает соединения, затем использует сокет, полученный при соединении, для создания InputStream'a и OutputStream'a. Они конвертируются в Reader и Writer, которые оборачиваются в BufferedReader и PrintWriter. После этого все, что будет прочитано из BufferedReader'a будет переправлено в PrintWriter, пока не будет получена строка "END", означающая, что пришло время закрыть соединение.

Клиент создает соединение с сервером, затем создает OutputStream и создает некоторую обертку, как и в сервере. Строки текста посылаются через полученный PrintWriter. Клиент также создает InputStream (опять таки, с соответствующей конвертацией и оберткой), чтобы слушать, что говорит сервер (который, в данном случае, просто отсылает слова назад).

Сервер, который просто отсылает назад все, что посылает клиент:

```
   import java.io.*;
import java.net.*;
public class ExampleServer {
   // Выбираем порт вне пределов 1-1024:
   public static final int PORT = 8080;
    public static void main(String[] args) throws
IOException {
      ServerSocket s = new ServerSocket(PORT);
      System.out.println("Started: " + s);
      try { /* Блокирует до тех пор, пока не возник-
```

```
нет соединение:*/
         Socket socket = s.accept();
         try {
            System.out.println("Connection accepted:
" + socket);
            BufferedReader in = new BufferedRead-
er(new InputStreamReader(
                  socket.getInputStream())); /* Вывод
автоматически выталкивается из буфера
PrintWriter'ом*/
            PrintWriter out = new PrintWriter(new
BufferedWriter(
                  new OutputStreamWri-
ter(socket.getOutputStream())), true);
            while (true) {
               String str = in.readLine();
               if (str.equals("END"))
                  break;
               System.out.println("Echoing: " + str);
               out.println(str);
            }
            /* Всегда закрываем два сокета...*/
         }
         finally {
            System.out.println("closing...");
            socket.close();
         }
      }
      finally {
         s.close();
      }
   }
}
```

Вы можете видеть, что для ServerSocket'a необходим только номер порта, а не IP адрес (так как он запускается на локальной машине!). Когда вы вызываете accept( ), метод блокирует выполнение до тех пор, пока клиент не попробует подсоединится к серверу. То есть, сервер ожидает соединения, но другой процесс может выполняться. Когда соединение установлено, метод accept( ) возвращает объект Socket, представляющий это соединение.

Здесь тщательно обработана ответственность за очистку сокета. Если конструктор ServerSocket завершится неудачей, программа просто звершится (обратите внимание, что мы должны предположить, что конструктор ServerSocket не оставляет никаких открытых сокетов, если он зваершается неудачей). По этой причине main( ) выбрасывает IOException, так что в блоке try нет необходимости. Если конструктор ServerSocket завершится успешно, то все вызовы методов должны быть помещены в блок try-finally, чтобы убедиться, что блок не будет покинут ни при каких условиях и ServerSocket будет правильно закрыт.

Аналогичная логика используется для сокета, возвращаемого из метода accept( ). Если метод accept( ) завершится неудачей, то мы должны предположить, что сокет не существует и не удерживает никаких ресурсов, так что он не нуждается в очистке. Однако если он закончится успешно, то следующие выражения должны быть помещены в блок try-finally, чтобы при каких-либо ошибках все равно произошла очистка. Позаботится об этом необходимо, потому что сокеты используют важные ресурсы, не относящиеся к памяти, так что вы должны быть прилежны и очищать их (так как в Java нет деструкторов, чтобы сделать это за вас).

И ServerSocket и Socket, производимый методом accept( ), печатаются в System.out. Это означает, что автоматически вызывается их метод toString( ). Вот что он выдаст:

```
ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]
```

Следующая часть программы выглядит, как открытие файла для чтения и записи за исключением того, что InputStream и OutputStream создаются из объекта Socket. И объект InputStream'a и OutputStream'a конвертируются в объекты Reader'a и Writer'a с помощью "классов-конвертеров" InputStreamReader и OutputStreamreader, соответственно. Вы можете также использовать классы InputStream и OutoutStream напрямую, но, с точки зрения вывода, есть явное преимущество в использовании этого подхода. Оно проявляется в PrintWriter'e, который имеет перегруженный конструктор, принимающий в качестве второго аргумента флаг типа boolean, указывающий, нужно ли автоматическое выталкивание буфера вывода в конце каждого выражения println( ) (но не print( )). Каждый раз, когда вы записываете в вывод, буфер вывода должен

выталкиваться, чтобы информация проходила по сети. Выталкивание важно для этого конкретного примера, поскольку клиент и сервер ожидают строку от другой стороны, прежде, чем приступят к ее обработке. Если выталкивание буфера не произойдет, информация не будет помещена в сеть до тех пор, пока буфер не заполнится, что может привести к многочисленным проблемам в этом примере.

Когда пишите сетевую программу, вам необходимо быть осторожным при использовании автоматического выталкивания буфера. При каждом выталкивании буфера пакеты должны создаваться и отправляться. В данном случае это именно то, что нам надо, так как если пакет, содержащий строку, не будет отослан, то общение между сервером и клиентом остановится. Другими словами, конец строки является концом сообщения. Но во многих случаях, сообщения не ограничиваются строками, так что будет более эффективным использовать автоматическое выталкивание буфера, поэтому позвольте встроенному механизму буфферизации построить и отослать пакет. В таком случае могут быть посланы пакеты большего размера и процесс обработки пойдет быстрее.

Обратите внимание, что фактически все открытые вами потоки, буфферезированы.

В бесконечном цикле while происходит чтение строк из входного BufferedReader'a и запись информации в System.out и в выходной PrintWriter. Обратите внимание, что вход и выход могут быть любыми потоками, так случилось, что они связаны с сетью.

Когда клиент посылает строку, содержащую "END", программа прекращает цикл и закрывает сокет.

Клиент, который просто посылает строки на сервер и читает строки, посылаемые сервером:

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

public class ExampleClient {
   public static void main(String[] args) throws
IOException {
      // Передаем null в getByName(), получая
      // специальный IP адрес "локальной заглушки"
      // для тестирования на машине без сети:
```

```
      InetAddress addr = InetAddress.getByName(null);
      // Альтернативно, вы можете использовать
      // адрес или имя:
      // InetAddress addr =
      // InetAddress.getByName("127.0.0.1");
      // InetAddress addr =
      // InetAddress.getByName("localhost");
      System.out.println("addr = " + addr);
      Socket socket = new Socket(addr, JabberServ-
er.PORT);
      // Помещаем все в блок try-finally, чтобы
      // быть уверенным, что сокет закроется:
      try {
         System.out.println("socket = " + socket);
         BufferedReader in = new BufferedReader(new
InputStreamReader(socket
               .getInputStream()));
         // Вывод автоматически выталкивается
PrintWriter'ом.
         PrintWriter out = new PrintWriter(new Buffe-
redWriter(
               new OutputStreamWri-
ter(socket.getOutputStream())), true);
         for (int i = 0; i < 10; i++) {
            out.println("howdy " + i);
            String str = in.readLine();
            System.out.println(str);
         }
         out.println("END");
      }
      finally {
         System.out.println("closing...");
         socket.close();
      }
   }
}
```

В main( ) вы можете видеть все три способа получения IP адреса локальной заглушки: с помощью null, localhost или путем явного указания зарезервированного адреса 127.0.0.1, если вы хотите соединится с машиной по сети, вы замените его IP адресом машины.

Обратите внимание, что Socket создается при указании и InetAddress'a, и номера порта. Чтобы понять, что это значит, когда будете писать один из объектов Socket помните, что Интернет соединение уникально определяется четырьмя параметрами: клиентским хостом, клиентским номером порта, серверным хостом и серверным номером порта. Когда запускается сервер, он получает назначаемый порт (8080) на localhost (127.0.0.1). Когда запускается клиент, он располагается на следующем доступном порту на своей машине, 1077 - в данном случае порт, который так же оказался на той же самой машине (127.0.0.1), что и сервер. Теперь, чтобы передать данные между клиентом и сервером, каждая сторона знает, куда посылать их. Поэтому, в процессе соединения с "известным" сервером клиент посылает "обратный адрес", чтобы сервер знал, куда посылать данные. Вот что вы видите среди выводимого стороной сервера:

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

Это означает, что сервер просто принимает соединение с адреса 127.0.0.1 и порта 1077 во время прослушивания локального порта (8080). На клиентской стороне:

```
Socket[addr=localhost/127.0.0.1,PORT=8080,localport
=1077]
```

Это значит, что клиент установил соединение с адресом 127.0.0.1 по порту 8080, используя локальный порт 1077.

Вы заметите, что при каждом повторном запуске клиента номер локального порта увеличивается. Он начинается с 1025 (первый после зарезервированного блока портов) и будет увеличиваться до тех пор, пока вы не перезапустите машину, в таком случае он снова начнется с 1025. (На машинах под управлением UNIX, как только будет достигнут верхний предел диапазона сокетов, номер будет возвращен снова к наименьшему доступному номеру.)

Как только объект Socket будет создан, процесс перейдет к BufferedReader и PrintWriter, как мы это уже видели в сервере (опять таки, в обоих случаях вы начинаете с Socket'a). В данном случае, клиент инициирует обмен путем посылки строки "howdy", за которой следует число. Обратите внимание, что буфер должен опять выталкиваться (что происходит автоматически из-за второго аргумента в конструкторе PrintWriter'a). Если буфер не будет выталкиваться, процесс обмена повиснет, поскольку начальное "howdy" ни-

когда не будет послана (буфер недостаточно заполнен, чтобы отсылка произошла автоматически). Каждая строка, посылаемая назад сервером, записывается в System.out, чтобы проверить, что все работает корректно. Для завершения обмена посылается ранее оговоренный "END". Если клиент просто разорвет соединение, то сервер выбросит исключение.

Вы можете видеть, что аналогичные меры приняты, чтобы быть уверенным в том, что сетевые ресурсы, представляемые сокетом, будут правильно очищены. Для этого используется блок try-finally.

Задание

Создать на основе сокетов клиент-серверное приложение:

1. Клиент посылает через сервер сообщение другому клиенту.
2. Клиент посылает через сервер сообщение другому клиенту, выбранному из списка.
3. Чат. Клиент посылает через сервер сообщение, которое получают все клиенты. Список клиентов хранится на сервере в файле.
4. Клиент при обращении к серверу получает случайно выбранный сонет Шекспира из файла.
5. Сервер рассылает сообщения выбранным из списка клиентам. Список хранится в файле.
6. Сервер рассылает сообщения в определенное время определенным клиентам.
7. Сервер рассылает сообщения только тем клиентам, которые в настоящий момент находятся в on-line.
8. Чат. Сервер рассылает всем клиентам информацию о клиентах, вошедших в чат и покинувших его.
9. Клиент выбирает изображение из списка и пересылает его другому клиенту через сервер.
10. Игра по сети в “Морской бой”.
11. Игра по сети “Го”. Крестики-нолики на безразмерном (большом) поле. Для победы необходимо выстроить пять в один ряд.
12. Написать программу, сканирующую сеть в указанном диапазоне IP адресов на наличие активных компьютеров.

13. Прокси. Программа должна принимать сообщения от любого клиента, работающего на протоколе TCP, и отсылать их на соответствующий сервер. При передаче изменять сообщение.
14. Телнет. Создать программу, которая соединяется с указанным сервером по указанному порту и производит обмен текстовой информацией.

Лабораторная работа № 5

СЕРВЛЕТЫ

Цель работы: Создать сервлет и взаимодействующие с ним пакеты Java-классов и JSP-страницы.

Изучаемые вопросы

1. Основа сервлета.

Постановка задачи

Создать сервлет и взаимодействующие с ним пакеты Java-классов и JSP-страницы, выполняющие действия, указанные в задании.

Теоретические сведения

Клиентский доступ из интернета или корпоративного интранета, конечно же, является легким способом позволить многим получить или предоставить доступ к данным и ресурсам. Этот тип доступа основывается на клиентах, использующих стандартный для World Wide Web Язык Гипертекстовой Разметки (Hypertext Markup Language - HTML) и Протокол Передачи Гипертекста (Hypertext Transfer Protocol - HTTP). Сервлетное API является набором абстрактных общих решений рабочей среды, соответствующей HTTP запросам.

Традиционно, способ решения таких проблем, как возможность для Интернет-клиента обновления базы данных, состояла в создании HTML страницы с текстовыми полями и кнопкой "Submit".

Пользователь впечатывал соответствующую информацию в текстовые поля и нажимал кнопку "Submit". Данные передавались вместе с URL, который говорил серверу, что делать с данными, указывая расположение программы Common Gateway Interface (CGI), которую запускал сервер, обеспечивая программу данными, которые она требовала. CGI программа обычно написана на Perl, Python, C, C++ или любом другом языке, который умеет читать стандартный ввод и писать в стандартный вывод. Это все, что предоставлял Web сервер: вызывалась CGI программа и использовались стандартные потоки (или, как возможный вариант для ввода, использовалась переменная окружения) для ввода и вывода. CGI программа отвечала за все остальное. Во-первых, она просматривала данные и решала, является ли их формат корректным. Если нет, CGI программа должна создпть HTML страницу для описания проблемы; эта страница передавалась Web серверу (через стандартный вывод из CGI программы), который посылал ее назад пользователю. Пользователь должен был, обычно, вернуться на страницу и попробовать снова. Если данные были корректными, CGI программа обрабатывала данные соответствующим способом, возможно, добавляла их в базу данных. Затем она должна была произвести соответствующую HTML страницу для Web сервера, чтобы он вернул ее пользователю.

Было бы идеально перейти к решению, полностью основанному на Java - апплет на клиентской стороне для проверки и пересылки данных и сервлет на серверной стороне для приема и обработки данных. К сожалению, хотя апплеты предоставляют технологию с вполне достаточной поддержкой, они проблематичны в использовании на Web, поскольку вы не можете рассчитывать на поддержку определенной версии Java на клиентском Web браузере; фактически, вы не можете рассчитывать, что Web браузер вообще поддерживает Java! В Интранете вы можете потребовать, чтобы определенная поддержка была доступна, что позволит реализовать намного большую гибкость в том, что вы можете сделать, но в Web более безопасный подход состоит в том, чтобы вся обработка была на стороне сервера, а клиенту доставлялся обычный HTML. Таким образом, ни один пользователь не будет отклонен вашим сайтом по причине того, что у него нет правильно установленного программного обеспечения.

Поскольку сервлеты предоставляют великолепное решение для программной поддержки на стороне сервера, они являются одной из наиболее популярных причин перехода на Java. Не только потому, что они предоставляют рабочую среду, которая заменяет CGI программирование (и снижает количество throny CGI проблем), но весь ваш код приобретает портируемость между платформами, получаемую от использования Java, и вы приобретаете доступ ко всему Java API (за исключением, конечно, того, которое производит GUI, такого, как Swing).

Основа сервлета

Архитектура API сервлета основывается на том, что классический провайдер сервиса использует метод service( ), через который все клиентские запросы будут посылаться программным обеспечением контейнера сервлетов, и методы жизненного цикла init( ) и destroy( ), которые вызываются только в то время, когда сервлет загружается и выгружается (это случается редко).

```
   public interface Servlet {
    public void init(ServletConfig config) throws Ser-
vletException;

    public ServletConfig getServletConfig();

    public void service(ServletRequest req, Servle-
tResponse res)
          throws ServletException, IOException;

    public String getServletInfo();

    public void destroy();
}
```

Основное назначение getServletConfig( ) состоит в возвращении объекта ServletConfig, который содержит параметры инициализации и запуска для этого сервлета. getServletInfo( ) возвращает строку, содержащую информацию о сервлете, такую, как автор, версия и авторские права.

Класс GenericServlet является оболочечной реализацией этого интерфейса и обычно не используется. Класс HttpServlet является

расширением GenericServlet и специально предназначен для обработки HTTP протокола - HttpServelt является одним из тех классов, которые вы будете использовать чаще всего.

Основное удобство атрибутов сервлетного API состоит во внешних объектах, которые вводятся вместе с классом HttpServlet для его поддержки. Если вы взглянене на метод service( ) в интерфейсе Servlet, вы увидите, что он имеет два параметра: ServeltRequest и ServletResponse. Вместе с классом HttpServlet, эти два объекта предназначены для HTTP: HttpServletRequest и HttpServletResponse. Вот простейший пример, который показывает использование HttpServelResponse:

```
   // {Depends: j2ee.jar}
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ServletsRule extends HttpServlet {
   int i = 0; // "постоянство" сервлета

   public void service(HttpServletRequest req,
HttpServletResponse res)
         throws IOException {
      res.setContentType("text/html");
      PrintWriter out = res.getWriter();
      out.print("<HEAD><TITLE>");
      out.print("A server-side strategy");
      out.print("</TITLE></HEAD><BODY>");
      out.print("<h1>Servlets Rule! " + i++);
      out.print("</h1></BODY>");
      out.close();
   }
}
```

Программа ServletsRule настолько проста, насколько может быть прост сервлет. Сервлет инициализируется только один раз путем вызова его метода init( ), при загрузке сервлета после того, как контейнер сервлетов будет загружен в первый раз. Когда клиент создает запрос к URL, который представлен сервлетом, контейнер сервлетов перехварывает этот запрос и совершает вызов метода service(

) после установки объектов HttpServletRequest и HttpServletResponse.

Основная ответственность метода service( ) состоит во взаимодействии с HTTP запросом, который посылает клиент, и в построении HTTP ответа, основываясь на атрибутах, содержащихся в запросе. ServletsRule манипулирует только объектом ответа, не обращая внимания на то, что посылает клиент.

После установки типа содержимого клиента (которое должно всегда выполнятся прежде, чем будет получен Writer или OutputStream), метод getWriter( ) объекта ответа производит объект PrintWriter, который используется для записи символьных данных ответа (другой вариант: getOutputStream( ) производит OutputStream, используемый для бинарного ответа, который применим для более специализированных решений).

Оставшаяся часть программы просто посылает HTML клиенту (предполагается, что вы понимаете HTML, так что эта часть не объясняется), как последовательность строк. Однако, обратите внимание на включение "счетчика показов", представленного переменной i. Он автоматически конвертируется в строку в инструкции print( ).

Когда вы запустите программу, вы увидите, что значение i сохраняется между запросами к сервлету. Это важное свойство сервлетов: так как только один сервлет определенного класса загружается в контейнер, и он никогда не выгружается (до тех пор, пока контейнер не завершит свою работу, что обычно случается только при перезагрузке серверного компьютера), любые поля сервлета этого класса действительно становятся постоянными объектами. Это значит, что вы можете без усилий сохранять значения между запросами к сервлету, в то время, как в CGI вы должны записывать значения на диск, чтобы сохранить их, что требует большого количества окружения для правильного их получения, а в результате получаем не кроссплатформенное решение.

Конечно, иногда Web сервер, а таким образом и контейнер сервлетов, должны перегружаться. Для предотвращения потери любой постоянной информации методы сервлета init( ) и destroy( ) автоматически вызываются во время загрузки или выгрузки клиента, что дает вам возможность сохранить важные данные во время остановки и восстановления после перезагрузки. Контейнер сервлетов вызывает метод destroy( ) когда он завершает свою работу, так что вы

всегда получаете возможность сохранить значимые данные до тех пор, пока серверная машина не будет сконфигурирована разумным способом.

Есть другая особенность при использовании HttpServlet. Этот класс обеспечивает методы doGet( ) и doPost( ), которые приспособлены для CGI "GET" пересылки от клиента и CGI "POST". GET и POST отличаются только деталями в способе пересылки данных. Однако, большинство доступной информации, поддерживает создание раздельных методов doGet( ) и doPost( ), вместо единого общего метода service( ), который обрабатывает оба случая. Мы просто будем использовать метод service( ) в этом примере, и позволим этому методу заботится о выборе GET или POST.

Когда бы форма не отсылалась сервлету, HttpServletRequest посупает со всеми предварительно загруженными данными формы, хранящимися как пары ключ-значение. Если вы знаете имя поля, вы можете просто использовать его напрямую в методе getParameter( ), чтобы найти значение. Вы можете также получить Enumiraton (старая форма Итератора) из имен полей, как показано в следующем примере. Этот пример также демонстрирует как один сервлет может быть использован для воспроизведения страницы, содержащей форму, и для страницы ответа (более хорошее решение вы увидите позже, в разделе о JSP). Если Enumiration пустое, значит нет полей; это означает, что форма не была отправлена. В этом случае производится форма, а кнопка подтверждения будет повторно вызывать тот же самый сервлет. Однако если поля существуют, они будут отображены.

```
// Группа пар имя-значение любой HTML формы
// {Depends: j2ee.jar}
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class EchoForm extends HttpServlet {
   public void service(HttpServletRequest req,
HttpServletResponse res)
         throws IOException {
      res.setContentType("text/html");
```

```
      PrintWriter out = res.getWriter();
      Enumeration flds = req.getParameterNames();
      if (!flds.hasMoreElements()) {
         // Нет отосланной формы -- создаем:
         out.print("<html>");
         out.print("<form method="POST"" + "ac-
tion="EchoForm">");
         for (int i = 0; i < 10; i++)
            out.print("<b>Field" + i + "</b> " +
"<input type="text""
                  + " size="20" name="Field" + i + ""
value="Value"
                  + i + ""><br>");
         out.print("<INPUT TYPE=submit name=submit"
               + " Value="Submit"></form></html>");
      }
      else {
         out.print("<h1>Your form contained:</h1>");
         while (flds.hasMoreElements()) {
            String field = (String)
flds.nextElement();
            String value = req.getParameter(field);
            out.print(field + " = " + value +
"<br>");
         }
      }
      out.close();
   }
}
```

В Java не удобно обрабатывать строки в памяти - форматирование возвращаемой страницы достаточно тягостно из-за символов завершения строки, эскейп-символов и знаков "+", необходимых для построения объектов String. Для огромных HTML страницы становится неразумным помещение кода прямо в Java. Одно из решений состоит в хранении страницы, как отдельного текстового файла, который потом открывается и передается Web серверу. Если вы выполните замену любого вида для содержимого этой страницы, это будет нехорошо, так как Java плохо обрабатывает строки. В этом случае вам, вероятно, лучше использовать более подходящее решение (Python может быть таким решением; существует версия,

которая сама встраивается в Java и называется JPython) для генерации страницы ответа.

Задание

Создать сервлет и взаимодействующие с ним пакеты Java-классов и JSP-страницы, выполняющие следующие действия:

1. Генерация таблиц по переданным параметрам: заголовок, количество строк и столбцов, цвет фона.
2. Вычисление тригонометрических функций в градусах и радианах с указанной точностью. Выбор функций должен осуществляться через выпадающий список.
3. Поиск слова, введенного пользователем. Поиск и определение частоты встречаемости осуществляется в текстовом файле, расположенном на сервере.
4. Вычисление объемов тел (параллелепипед, куб, сфера, тетраэдр, тор, шар, эллипсоид и т.д.) с точностью и параметрами, указываемыми пользователем.
5. Поиск и (или) замена информации в коллекции по ключу (значению).
6. Выбор текстового файла из архива файлов по разделам (поэзия, проза, фантастика и т.д.) и его отображение.
7. Выбор изображения по тематике (природа, автомобили, дети и т.д.)
и его отображение.
8. Информация о среднесуточной температуре воздуха за месяц задана
в виде списка, хранящегося в файле. Определить:

а) среднемесячную температуру воздуха; б) количество дней, когда температура была выше среднемесячной; в) количество дней, когда температура опускалась ниже 0ºС; г) три самых теплых дня.

9. Определение значения полинома в заданной точке. Степень полинома
и его коэффициенты вводятся пользователем.
10. Вывод фрагментов текстов шрифтами различного размера. Размер шрифта и количество строк задаются на стороне клиента.
11. Информация о точках на плоскости хранится в файле. Выбрать все точки, наиболее приближенные к заданной прямой. Пара-

метры прямой
и максимальное расстояние от точки до прямой вводятся на стороне клиента.

12. Осуществить сортировку введенного пользователем массива целых чисел. Числа вводятся через запятую.
13. Реализовать игру с сервером в крестики-нолики.
14.

Лабораторная работа № 6

СОЕДИНЕНИЕ С БАЗОЙ ДАННЫХ

Цель работы: приобретение навыков проектирования и разработки приложений в архитектуре клиент-сервер.

Изучаемые вопросы

1. Драйверы, соединения и запросы.
2. Простое соединение и запрос
3. Сокеты
4. Простейший сервер и клиент
5. Обслуживание множества серверов
6. Дейтаграммы
7. Использование URL'ов из аплета
8. Чтение файла с сервера

Постановка задачи

Создать клиент/серверные приложения для базы данных.

Теоретические сведения

Драйверы, соединения и запросы

JDBC (Java DataBase Connectivity) – стандартный прикладной интерфейс (API) языка Java для организации взаимодействия между приложением и СУБД. Это взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов.

JDBC предоставляет интерфейс для разработчиков, использующих различные СУБД. С помощью JDBC отсылаются SQL-запросы

только к реляционным базам данных (БД), для которых существуют драйверы, знающие способ общения с реальным сервером базы данных.

Строго говоря, JDBC не имеет прямого отношения к J2EE, но так как взаимодействие с СУБД является неотъемлемой частью Web-приложений, то эта технология рассматривается в данном контексте.

Последовательность действий

1. Загрузка класса драйвера базы данных при отсутствии экземпляра этого класса.

Например:

```
String driverName = "org.gjt.mm.mysql.Driver";
для СУБД MySQL,
String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";
для СУБД MSAccess или
String driverName = "org.postgreesql.Driver";
для СУБД PostgreeSQL.
```

После этого выполняется собственно загрузка драйвера в память:

```
Class.forName(driverName);
```

и становится возможным соединение с СУБД.

Эти же действия можно выполнить, импортируя библиотеку и создавая объект явно. Например, для СУБД DB2 от IBM объект-драйвер можно создать следующим образом:

```
new com.ibm.db2.jdbc.net.DB2Driver();
```

2. Установка соединения с БД.

Для установки соединения с БД вызывается статический метод getConnection() класса DriverManager. В качестве параметров методу передаются URL базы данных, логин пользователя БД и пароль доступа. Метод возвращает объект Connection. URL базы данных, состоящий из типа и адреса физического расположения БД, может создаваться в виде отдельной строки или извлекаться из файла ресурсов. Например:

```
Connection cn = DriverManager.getConnection(
"jdbc:mysql://localhost/my_db", "root", "pass");
```

В результате будет возвращен объект Connection и будет одно установленное соединение с БД my_db. Класс DriverManager предоставляет средства для управления набором драйверов баз данных. С помощью метода registerDriver() драйверы регистрируются, а методом getDrivers() можно получить список всех драйверов.

3. Создание объекта для передачи запросов.

После создания объекта Connection и установки соединения можно начинать работу с БД с помощью операторов SQL. Для выполнения запросов применяется объект Statement, создаваемый вызовом метода createStatement() класса Connection.

```
Statement st = cn.createStatement();
```

Объект класса Statement используется для выполнения SQL-запроса без его предварительной подготовки. Могут применяться также объекты классов PreparedStatement и CallableStatement для выполнения подготовленных запросов и хранимых процедур. Созданные объекты можно использовать для выполнения запроса SQL, передавая его в один из методов executeQuery(String sql) или executeUpdate(String sql).

4. Выполнение запроса.

Результаты выполнения запроса помещаются в объект ResultSet:

```
ResultSet rs = st.executeQuery(
"SELECT * FROM my_table");//выборка всех данных таблицы my_table
```

Для добавления, удаления или изменения информации в таблице вместо метода executeQuery() запрос помещается в метод executeUpdate().

5. Обработка результатов выполнения запроса

Обработка результатов производится методами интерфейса ResultSet, где самыми распространенными являются next() и getString(int pos) а также аналогичные методы, начинающиеся с getТип(int pos) (getInt(int pos), getFloat(int pos) и др.) и updateТип(). Среди них следует выделить методы getClob(int pos) и getBlob(int pos), позволяющие извлекать из полей таблицы специфические объекты (Character Large Object, Binary Large Object), которые могут быть, например, графическими или архивными файлами. Эффективным способом извлечения значения поля из таблицы ответа является обращение к этому полю по его позиции в строке.

При первом вызове метода next() указатель перемещается на таблицу результатов выборки в позицию первой строки таблицы ответа. Когда строки закончатся, метод возвратит значение false.

6. Закрытие соединения

```
cn.close();
```

После того как база больше не нужна, соединение закрывается.

Для того чтобы правильно пользоваться приведенными методами, программисту требуется знать типы полей БД. В распределенных системах это знание предполагается изначально.

Простое соединение и простой запрос

Теперь следует воспользоваться всеми предыдущими инструкциями и создать пользовательскую БД с именем db2 и одной таблицей users. Таблица должна содержать два поля: символьное – name и числовое – phone и несколько занесенных записей. Сервлет, осуществляющий простейший запрос на выбор всей информации из таблицы, выглядит следующим образом.

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

public class ServletToBase extends HttpServlet {
    public void doGet(HttpServletRequest req,
HttpServletResponse resp)
                        throws ServletException {
          performTask(req, resp);
    }
    public void doPost(HttpServletRequest req,
 HttpServletResponse resp)
                         throws ServletException {
          performTask(req, resp);
    }

public void showInfo(PrintWriter out, ResultSet rs)
throws SQLException {
          out.print("From DataBase:");
      while (rs.next()) {
          out.print("<br>Name:-> " + rs.getString(1)
+ " Phone:-> " + rs.getInt(2));
          }
```

```
      }
      public void performTask(HttpServletRequest req,
  HttpServletResponse resp) {
      resp.setContentType("text/html; char-
set=Cp1251");
      PrintWriter out = null;
            try {//1
                  out = resp.getWriter();
                  try {//2

      Class.forName("org.gjt.mm.mysql.Driver");
  // для MSAccess
            /* return "sun.jdbc.odbc.JdbcOdbcDriver"
*/
            // для PostgreeSQL
            /* return " org.postgreesql.Driver " */
      Connection cn = null;
                        try {//3
      cn =
  DriverManag-
er.getConnection("jdbc:mysql://localhost/db2",
   "root", "pass");
  // для MSAccess
            /* return "jdbc:odbc:db2"; */
            // для PostgreeSQL
            /* return
"jdbc:postgreesql://localhost/db2"; */

            Statement st = null;
                              try {//4
            st = cn.createStatement();
                        ResultSet rs = null;
                                    try {//5
            rs = st.executeQuery("SELECT * FROM us-
ers");
      out.print("From DataBase:");
        while (rs.next()) {
            out.print("<br>Name:-> " + rs.getString(1)
  + " Phone:-> " + rs.getInt(2));
                              }
  } finally {// для 5-го блока try
```

```
      /*     закрыть ResultSet, если он был открыт и
ошибка
             произошла во время чтения из него данных
*/
                  // проверка успел ли создаться
ResultSet
                  if (rs != null)   rs.close();
                  else
      out.print("ошибка во время чтения данных из
БД");
                                        }
                                  } finally {// для 4-го
блока try
      /*     закрыть Statement, если он был открыт и
ошибка
             произошла во время создания ResultSet
*/
                  // проверка успел ли создаться
Statement
                  if (st != null)   st.close();
                  else  out.print("Statement не
создан");
                                  }
                            } finally {// для 3-го блока
try
      /*    закрыть Connection, если он был открыт и
ошибка произошла
             во время создания ResultSet или создания
и использования
             Statement  */
            // проверка - успел ли создаться
Connection
            if (cn != null) cn.close();
            else  out.print("Connection не создан");
                        }
      } catch (ClassNotFoundException e) {// для 2-го
блока try
      out.print("ошибка во время загрузки драйвера
БД");
                  }
            }
```

```
      /* вывод сообщения о всех SQLException и
IOException в блоках finally, */
            /* поэтому следующие блоки catch оставлены
пустыми */
            catch (SQLException e) {
            }// для 1-го блока try
            catch (IOException e) {
            }// для 1-го блока try
            finally {// для 1-го блока try
      /*    закрыть PrintWriter, если он был инициали-
зирован и ошибка
            произошла во время работы с БД  */
            // проверка, успел ли инициализироваться
PrintWriter
            if (out != null)  out.close();
            else
  out.print("PrintWriter не проинициализирован");
            }
      }
  }
```

В несложном приложении достаточно контролировать закрытие соединения, так как незакрытое (“провисшее”) соединение снижает быстродействие системы.

Задание

В каждом из заданий необходимо выполнить следующие действия:

- Организацию соединения с базой данных вынести в отдельный класс, метод которого возвращает соединение.
- Создать БД. Привести таблицы к одной из нормированных форм.
- Создать класс для выполнения запросов на извлечение информации из БД с использованием компилированных запросов.
- Создать класс на добавление информации.
- Создать HTML-документ с полями для формирования запроса.
- Результаты выполнения запроса передать клиенту в виде HTML-документа.

1. **Файловая система.** В БД хранится информация о дереве каталогов файловой системы – каталоги, подкаталоги, файлы.

- Определить полный путь заданного файла (каталога).
- Подсчитать количество файлов в заданном каталоге, включая вложенные файлы и каталоги.
- Подсчитать место, занимаемое на диске содержимым заданного каталога.
- Найти в базе файлы по заданной маске с выдачей полного пути.
- Переместить файлы и подкаталоги из одного каталога в другой.
- Удалить файлы и каталоги заданного каталога.

2. **Видеотека.** В БД хранится информация о домашней видеотеке – фильмы, актеры, режиссеры.

- Найти все фильмы, вышедшие на экран в текущем и прошлом году.
- Вывести информацию об актерах, снимавшихся в заданном фильме.
- Вывести информацию об актерах, снимавшихся как минимум в N фильмах.
- Вывести информацию об актерах, которые были режиссерами хотя бы одного из фильмов.
- Удалить все фильмы, дата выхода которых была более заданного числа лет назад.

3. **Расписание занятий.** В БД хранится информация о преподавателях и проводимых ими занятиях.

- Вывести информацию о преподавателях, работающих в заданный день недели в заданной аудитории.
- Вывести информацию о преподавателях, которые не ведут занятия
в заданный день недели.
- Вывести дни недели, в которых проводится заданное количество занятий.
- Вывести дни недели, в которых занято заданное количество аудиторий.
- Перенести первые занятия заданных дней недели на последнее место.

4. **Письма.** В БД хранится информация о письмах и отправляющих их людях.

- Найти пользователя, длина писем которого наименьшая.
- Вывести информацию о пользователях, а также количестве полученных и отправленных ими письмах.
- Вывести информацию о пользователях, которые получили хотя бы одно сообщение с заданной темой.
- Вывести информацию о пользователях, которые не получали сообщения с заданной темой.
- Направить письмо заданного человека с заданной темой всем адресатам.

5. **Сувениры.** В БД хранится информация о сувенирах и их производителях.

- Вывести информацию о сувенирах заданного производителя.
- Вывести информацию о сувенирах, произведенных в заданной стране.
- Вывести информацию о производителях, чьи цены на сувениры меньше заданной.
- Вывести информацию о производителях заданного сувенира, произведенного в заданном году.
- Удалить заданного производителя и его сувениры.

6. **Заказ.** В БД хранится информация о заказах магазина и товарах в них.

- Вывести полную информацию о заданном заказе.
- Вывести номера заказов, сумма которых не превосходит заданную,

и количество различных товаров равно заданному.

- Вывести номера заказов, содержащих заданный товар.
- Вывести номера заказов, не содержащих заданный товар и поступивших в течение текущего дня.
- Сформировать новый заказ, состоящий из товаров, заказанных в текущий день.
- Удалить все заказы, в которых присутствует заданное количество заданного товара.

7. **Продукция.** В БД хранится информация о продукции компании.

- Вывести перечень параметров для заданной группы продукции.
- Вывести перечень продукции, не содержащий заданного параметра.
- Вывести информацию о продукции для заданной группы.
- Вывести информацию о продукции и всех ее параметрах со значениями.
- Удалить из базы продукцию, содержащую заданные параметры.
- Переместить группу параметров из одной группы товаров в другую.

8. **Погода.** В БД хранится информация о погоде в различных регионах.

- Вывести сведения о погоде в заданном регионе.
- Вывести даты, когда в заданном регионе шел снег и температура была ниже заданной отрицательной.
- Вывести информацию о погоде за прошедшую неделю в регионах, жители которых общаются на заданном языке.
- Вывести среднюю температуру за прошедшую неделю в регионах
с площадью больше заданной.

9. **Магазин часов.** В БД хранится информация о часах, продающихся в магазина.

- Вывести марки заданного типа часов.
- Вывести информацию о механических часах, цена на которые не превышает заданную.
- Вывести марки часов, изготовленных в заданной стране.
- Вывести производителей, общая сумма часов которых в магазине
не превышает заданную.

10. **Города.** В БД хранится информация о городах и их жителях.

- Вывести информацию обо всех жителях заданного города, разговаривающих на заданном языке.
- Вывести информацию обо всех городах, в которых проживают жители выбранного типа.

- Вывести информацию о городе с заданным количеством населения

и всех типах жителей, в нем проживающих.

- Вывести информацию о самом древнем типе жителей.

11. **Планеты.** В БД хранится информация о планетах, их спутниках и галактиках.

- Вывести информацию обо всех планетах, на которых присутствует жизнь, и их спутниках в заданной галактике.
- Вывести информацию о планетах и их спутниках, имеющих наименьший радиус и наибольшее количество спутников.
- Вывести информацию о планете, галактике, в которой она находится,

и ее спутниках, имеющей максимальное количество спутников, но с наименьшим общим объемом этих спутников.

- Найти галактику, сумма ядерных температур планет которой наибольшая.

12. **Треугольники.** В БД хранятся треугольники и координаты их точек на плоскости.

- Вывести треугольник, площадь которого наиболее приближена к заданной.
- Вывести треугольники, сумма площадей которых наиболее приближена к заданной.
- Вывести треугольники, которые помещаются в окружность заданного радиуса.

13. **Словарь.** В БД хранится англо-русский словарь, в котором для одного английского слова может быть указано несколько его значений и наоборот. Со стороны клиента вводятся последовательно английские (русские) слова. Для каждого из них вывести на консоль все русские (английские) значения слова.

14. **Словари.** В двух различных базах данных хранятся два словаря: русско-белорусский и белорусско-русский. Клиент вводит слово и выбирает язык. Вывести перевод этого слова.

Лабораторная работа № 7

УДАЛЕННЫЙ ВЫЗОВ МЕТОДА (RMI)

Цель работы: Реализовать приложение, используя технологию RMI

Изучаемые вопросы

1. Удаленный интерфейс
2. Реализация удаленного интерфейса
3. Настройка репозитория
4. Создание заглушек и скелетов
5. Использование удаленного объекта

Постановка задачи

Создадать систему, функционально состоящую из двух компонентов - сервера и клиента. При этом реализовать удаленный вызов метода.

Теоретические сведения

RMI дает возможность выполнять объекты Java на различных компьютерах или в отдельных процессах путем взаимодействия их друг с другом посредством удаленных вызовов методов. Технология RMI основана на более ранней подобной технологии удаленного вызова процедур (RPC) для процедурного программирования, разработанной в 80-х годах. RPC позволяет процедуре вызывать функцию на другом компьютере столь же легко, как если бы эта функция была частью программы, выполняющейся на том же компьютере. RPC выполняет всю работу по организации сетевых взаимодействий и маршалинга данных (т.е. пакетирования параметров функций и возврата значений для передачи их через сеть). Но RPC не подходит для передачи и возврата объектов Java, потому что она поддерживает ограниченный набор простых типов данных. Есть и другой недостаток у RPC - программисту необходимо знать специальный язык определения интерфейса (IDL) для описания функций, которые допускают удаленный вызов. Для устранения этих недостатков и была разработана технология RMI.

Удаленный интерфейс

RMI делает тяжелым использование интерфейсов. Когда вы хотите создать удаленный объект, вы маскируете лежащую в основе реализацию, общаясь по интерфейсу. То есть, когда клиент получает ссылку на удаленный объект, реально он получает ссылку на интерфейс, который соединен с некоторым локальным кодом заглушки, которая общается по сети. Но вы не думаете об этом, вы просто посылаете сообщения через вашу ссылку на интерфейс.

Когда вы создаете удаленный интерфейс, вы должны следовать следующему руководству:

1. Удаленный интерфейс должен быть публичным (он не может иметь "пакетный уровень доступа", таким образом он не может быть "дружественным"). В противном случае, клиент получит ошибку при попытке загрузить удаленный объект, который реализует удаленный интерфейс.
2. Удаленный интерфейс должен наследоваться от интерфейса java.rmi.Remote.
3. Каждый метод удаленного интерфейса должен декларировать java.rmi.RemoteException в своем разделе throws в дополнение ко всем остальным специфичным для приложения исключениям.
4. Удаленный объект, передающийся как аргумент или возвращаемое значение (либо напрямую, либо как встроенная часть локального объекта), должен быть декларирован как удаленный интерфейс, а не класс реализации.

Вот простейший удаленный интерфейс, который представляет службу точного времени:

```
   // The PerfectTime remote interface.
package c15.rmi;

import java.rmi.*;

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

Он выглядит как и любой другой интерфейс, за исключением того, что он является наследником Remote и все его методы выбрасывают RemoteException. Помните, что все методы интерфейса автоматически являются публичными.

Реализация удаленного интерфейса

Сервер должен содержать класс, который наследуется от UnicastRemoteObject и реализует удаленный интерфейс. Этот класс может также иметь дополнительные методы, но для клиента, конечно, будут доступны только методы удаленного интерфейса, так как клиент будет получать только ссылку на интерфейс, а не на класс, реализующий его.

Вы должны явно определить конструктор для удаленного объекта, даже если вы определяете только конструктор по умолчанию, который вызывает конструктор базового класса. Вы должны написать его, так как он должен выбрасывать RemoteException.

Вот реализация удаленного интерфейса PerfectTimeI:

```
   // Реализация удаленного
// объекта PerfectTime.
// {Broken}
package c15.rmi;

import java.rmi.*;

import java.rmi.server.*;

import java.rmi.registry.*;

import java.net.*;

public class PerfectTime extends UnicastRemoteObject
implements PerfectTimeI {
   // Реализация интерфейса:
   public long getPerfectTime() throws
RemoteException {
      return System.currentTimeMillis();
   }

   // Необходимо реализовывать конструктор,
   // выбрасывающий RemoteException:
   public PerfectTime() throws RemoteException {
      // super(); // Вызывается автоматически
   }

   // Регистрация для обслуживания RMI. Выбрасывает
```

```
    // исключения на консоль.
    public static void main(String[] args) throws Ex-
ception {
        System.setSecurityManager(new RMISecurityManag-
er());
        PerfectTime pt = new PerfectTime();
        Naming.bind("//peppy:2005/PerfectTime", pt);
        System.out.println("Ready to do time");
    }
} //
```

Здесь main( ) обрабатывает все детали установки сервера. Когда вы обслуживаете RMI объекты, в некотором месте вашей программы вы должны:

1. Создать и установить менеджер безопасности, который поддерживает RMI. Для RMI досупен только один такой менеджер, как часть пакета Java, он называется RMISecurityManager.
2. Создать один или несколько экземпляров удаленного объекта. Здесы вы можете видеть создание объекта PerfectTime.
3. Зарегистрировать как минимум один удаленный объект в RMI репозитории удаленных объектов в целях загрузки. Один удаленный объект может иметь методы, которые производят ссылки на другие удаленные объекты. Это позволит вам сделать так, чтобы клиент обращался к репозиторию только один раз, чтобы получить первый удаленный объект.

Настройка репозитория

В этом примере вы видите вызов статического метода Naming.bind( ). Однако, этот вызов требует, чтобы репозиторий был запущен как отдельный процесс на компьютере. Имя сервера репозитория rmiregistry, в 32-х системе Windows вы используете команду:

```
start rmiregistry
```

для запуска его в фоновом режиме. Для системы Unix используется команда:

```
rmiregistry &
```

Как и многие сетевые программы, rmiregestry получает IP адрес той машины, на которой она запущена, но она также должна слушать порт. Если вы вызовите rmiregestry как это показано выше, без аргументов, для репозитория будет использован порт по умолчанию 1099. Если вы хотите, чтобы репозиторий был на другом порту, вы

добавляете аргумент в командную строку, который указывает порт. Например, интересующий порт 2005, так что rmiregestry должна быть запущена следующей командой под 32-х битной Windows:

```
start rmiregistry 2005
```

или для Unix:

```
rmiregistry 2005 &
```

Информация о номере порта также должна быть передана в команду bind( ), точно так же, как и IP адрес машины, где расположен репозиторий.

1. localhost не работает с RMI. То есть, экспериментируя с RMI на одной единственной машине, вы должны предоставить имя машины. Чтобы найти имя вашей машины в 32-х битной среде Windows, запустите контрольную панель и выберите "Network". Выберите закладку "Identification", и вы увидите имя вашего компьютера.

2. RMI не будет работать до тех пор, пока ваш компьютер не будет иметь активного TCP/IP соединения, даже если все компоненты просто общаются друг с другом на вашей локальной машине. Это означает, что вы должны соединить ваш компьютер с вашим провайдером Internet прежде, чем пробовать запускать программу или вы получите непонятное сообщение об исключении.

Учитывая все вышесказанное, команда bind( ) принимает вид:

```
Naming.bind("//peppy:2005/PerfectTime", pt);
```

Если вы используете порт по умолчанию 1099, вам не нужно указывать порт, так что вы можете написать:

```
Naming.bind("//peppy/PerfectTime", pt);
```

Вы должны быть способны выполнить локальное тестирование оставив в покое IP адрес и используя только идентификатор:

```
Naming.bind("PerfectTime", pt);
```

Имя службы произвольно, в данном случае это PerfectTime, что совпадает с именем класса, но вы можете использовать любое название, какое хотите. Важно то, что оно должно быть уникальным в известном клиенту репозитории, к которому он обращается для производства удаленного объекта. Если имя уже есть в репозитории, вы получите AlreadyBoundException. Чтобы предотвратить это, вы можете всегда использовать rebind( ) вместе bind( ), так как rebind( ) либо добавляет новую запись, либо заменяет уже существующую.

Даже после того, как main( ) завершит работу, ваш объект будет создан и зарегистрирован, так что он останентся жить в репозитории, ожидая прихода клиентов и запросов от них. До тех пор, пока работает rmiregistry и вы не вызвали метод Naming.unbind( ) с вашим именем, объект будет существовать. По этой причине, когда вы разрабатываете ваш код, вам необходимо выгружать rmiregistry и перезапускать его, когда вы скомпилировали новую версию вашего удаленного объекта.

Вам не нужно принудительно запускать rmiregistry, как внешний процесс. Если вы знаете, что ваше приложение является единственным, использующим репозиторий, вы можете запускать его внутри программы с помощью строки:

```
LocateRegistry.createRegistry(2005);
```

Как и прежде, 2005 - это номер порта, который мы будем использовать в этом примере. Это эквивалентно запуску rmiregistry 2005 из командной строки, но такой запуск часто будет более последовательным, когда вы разрабатываете RMI код, так как при этом пропускаются дополнительные шаги запуска и остановки репозитория. Как только вы выполните этот код, вы можете вызвать bind( ), используя Naming, как и прежде.

Создание заглушек и скелетов

Если вы скомпилируете и запустите PerfectTime.java, программа не будет работать, даже если у вас будет правильно запущенный репозиторий. Это происходит потому, что рабочая среда для RMI еще не до конца создана. Вы должны сначала создать заглушки и скелеты, которые обеспечат операцию сетевого соединения и позволят вам считать, что удаленный объект - это просто другой локальный объект на вашей машине.

То, что происходит за сценой достаточно сложно. Любые объекты, котоые вы передаете при вызове или получаете от удаленного объекта должны реализовывать Serializable (если вы хотите передавать удаленную ссылку вместо самого объекта, объектные аргументы могут реализовывать Remote), так что вы можете представить, что заглушки и скелеты автоматически выполняют сериализацию и десериализацию, как будтно они "руководят" всеми аргументами через сеть и возвращают результат. К счастью, вам не нужно знать

ничего этого, но вы должны создать заглушки и скелеты. Это простой процесс: вы вызываете инструмент rmic для вашего скомпилированного кода и создаете необходимые файлы. Таким образом требуется просто добавить еще один шаг в процесс компиляции.

Однако, инструмент rmic привередлив к пакетам и classpaths. PerfectTime.java расположен в пакете c15.rmi, и если вы вызовите rmic в том же директории, в котором расположен PerfectTime.class, rmic не найдет файл, так как он ищет в переменной classpath. Таким образом вы должны указать расположение в переменной classpath, например так:

```
rmic c15.rmi.PerfectTime
```

Вам не нужно находиться в директории, содержащем PerfectTime.class, когда вы запускаете эту команду, а результат будет помещено в текущий каталог.

Когда запуск rmic завершится успешно, вы получите два новых класс в директории:

```
PerfectTime_Stub.class
PerfectTime_Skel.class
```

соответственно это заглушка и скелет. Теперь вы готовы получить сервер и клиент для общения между собой.

Использование удаленного объекта

Главная цель RMI состоит в упрощении использования удаленных объектов. Есть только одна дополнительная вещь, которую вы должны выполнить в клиентской программе, это найти и получить удаленный интерфейс с сервера. Все остальное - это обычное Java программирование: посылка сообщений объекту. Вот программа, которая использует PerfectTime:

```
// Использование удаленного объекта PerfectTime.
// {Broken}
package c15.rmi;
import java.rmi.*;
import java.rmi.registry.*;
public class DisplayPerfectTime {
   public static void main(String[] args) throws Ex-
ception {
      System.setSecurityManager(new RMISecurityManag-
```

```
er());
      PerfectTimeI t = (PerfectTimeI) Naming
            .lookup("//peppy:2005/PerfectTime");
      for (int i = 0; i < 10; i++)
         System.out.println("Perfect time = " +
t.getPerfectTime());
   }
} //
```

Строка идентификатора точно такая же, как и используемая для регистрации объекта с помощью Naming, а первая часть представляет URL и номер порта. Так как вы используете URL, вы можете также указать машину в Internet'e.

То, что приходит от Naming.lookup( ) должно быть приведено к удаленному интерфейсу, а не к классу. Если вместо этого вы будете использовать класс, вы получите исключение.

Вы можете видеть вызов метода

```
t.getPerfectTime()
```

так как то, что вы имеете - это ссылка на удаленный объект, программирование с которой не отличается от программирования с локальным объектом (с одним отличием: удаленные методы выбрасывают RemoteException).

Задание

Создадать систему, функционально состоящую из двух компонентов - сервера (процессингового центра) и клиента (касса). Будем предполагать, что сервер в системе один (именно он обладает всей информацией о зарегистрированных картах и их балансах), а касс много и на них проходят операции регистрации новых карт, а также операции изменения баланса карт (соответственно - оплаты и занесения наличных).