МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Автоматизированные системы управления»

ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ

Методические рекомендации к лабораторным работам для студентов специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» очной и заочной форм обучения

Могилев 2021

УДК 004.777: 004.432
ББК 32.973.202-04
П 78

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Автоматизированные системы управления»
«26» января 2020 г., протокол № 6

Составитель канд. техн. наук, доц. И. В. Акиншева;
Рецензент канд. техн. наук, доц. С. К. Крутолевич

Методические рекомендации предназначены к лабораторным работам по дисциплине «Программирование сетевых приложений». Приведены задания и список литературы для подготовки.

Учебно-методическое издание

ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ

Ответственный за выпуск И. В. Акиншева

Корректор Е. А. Галковская

Компьютерная верстка Н. П. Полевничая

Подписано в печать . Формат 60×84/16. Бумага офсетная. Гарнитура Таймс.
Печать трафаретная. Усл. печ. л. .Уч.-изд. л. . Тираж 26 экз. Заказ №

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

© Белорусско-Российский
университет, 2021

СОДЕРЖАНИЕ

Введение………………………………………………………………… 4
Лабораторная работа № 1. Язык разметки гипертекстов HTML………. 5
Лабораторная работа № 2. Каскадные таблицы сталей (CSS)…………. 11
Лабораторная работа № 3. Язык составления сценариев JavaScript…... 15
Лабораторная работа № 4. Сетевое программирование с сокетами и каналами……………………………………………………………………… 21
Лабораторная работа № 5. Сервлеты……………………………………. 28
Лабораторная работа № 6. Соединение с базой данных……………….. 33
Лабораторная работа № 7. Удаленный вызов метода (RMI)…………… 38
Список литературы……………………………………………………... 45

Введение

Подготовка современного специалиста требует уверенного владения возможностями, предоставляемыми компьютерными технологиями. Изучение настоящей учебной дисциплины обеспечивает подготовку специалиста, владеющего фундаментальными знаниями и практическими навыками в области основ программирования сетевых приложений при решении практических задач.

Цель учебной дисциплины состоит в подготовке специалистов, владеющих базовыми знаниями, умениями и практическими навыками в области языков и средств разработки сетевых приложений, ориентированных на клиент-серверную архитектуру, программирования элементов такой архитектуры.

Методические рекомендации предназначены для помощи студентам в выполнении лабораторных работ по дисциплине.

В методических рекомендациях для изучения представлены сведения о создании HTML-документов, работе с каскадными таблицами стилей (CSS), языке составления сценариев JavaScript, сетевом программировании с сокетами и каналами, установлении соединения с разработанной базой данных, удаленном вызове метода (RMI).

При выполнении лабораторных работ используется следующее программное обеспечение: операционная система Microsoft Windows; пакет jdk; среды разработки JCreatorPro или NetBeans IDE, или Eclipse, или IntelliJ IDEA; web-сервера Tom Cat или Glass Fish; СУБД MySQL, клиент для MySQL – HeidiSQL или инструмент для визуального проектирования баз данных MySQL Workbench.

1 Лабораторная работа № 1. Язык разметки гипертекстов HTML

Цель работы: приобретение навыков создания HTML-документов.

Порядок выполнения работы.

1 Изучить основные теоретические положения, сделав необходимые выписки в конспект.

2 Получить задание у преподавателя, выполнить задание.

3 Оформить отчет.

Требования к отчету.

1 Цель работы.

2 Полученные результаты.

3 Выводы.

Теоретические сведения

HTML (от англ. «HyperText Markup Language» – «язык разметки гипертекста») – стандартный язык разметки документов в Internet. 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. Например, 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 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.

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>
```

Тэг заголовочной части документа должен быть использован сразу после тэга <HTML>. 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>
...
```

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.

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

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.

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=nl 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=nl – 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 параметр 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>
```

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 – ккоординаты 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 COC>RDS="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> – завещающий тег.

Пример:

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

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="*,*">

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

Задание для самостоятельной работы.

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.

Вопросы для контроля знаний

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 фрейм NOFRAMES?

2 Лабораторная работа № 2. Каскадные таблицы стилей (CSS)

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

Порядок выполнения работы.

1 Изучить основные теоретические положения, сделав необходимые выписки в конспект.

2 Получить задание у преподавателя, выполнить задание.

3 Оформить отчет.

Требования к отчету.

1 Цель работы.

2 Результаты работы.

3 Выводы.

Теоретические сведения

CSS (от 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. Например:

```
<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 и потом его использования.

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.

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. 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>
```

Второй вариант, при котором описание стилей располагается в коде Web-страницы, 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/ess">
        .el_cl_l{display:inline; z-index:1};
        .el_lst{display:list-item; margin:-1%;
background:#ff0000 url("be.jpg") no-repeat};
</style>
<title></title>
<meta http-equiv="Content-Type" content="text/html;
charset=windows-1251">
</head>
```

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>
```

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

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

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

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 1.

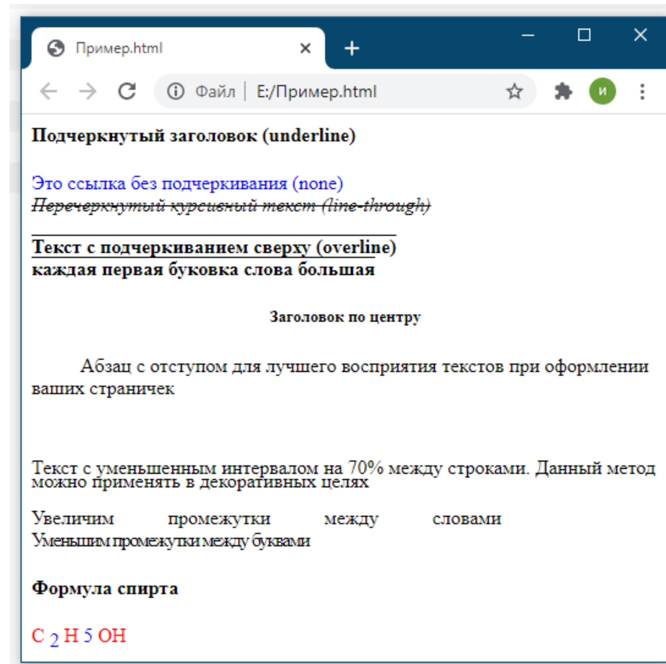

Рисунок 1 – Интерпретация свойств текста браузером

Ниже приведен пример кода.

```
<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 подчеркивания (none) </a><br>
<i>Перечеркнутый курсивный текст (line-through) </i><p>
<b>Текст с подчеркиванием сверху (overline) </b><br>
<b c1ass=cap>каждая первая буковка слова большая</b>
<h5>Заголовок по центру</h5>
<p class=otstup>Абзац с отступом для лучшего восприятия текстов при оформ-
лении ваших страничек</p>
<p class=interval>Текст с уменьшенным интервалом на 50% между строками.
Данный метод можно применять в декоративных целях</p>
<p><span style="word-spacing: 15pt">Увеличим промежутки между
словами</span>
<p><span style="letter-spacing: -2pt">уменьшим промежутки 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>
```

Задания для самостоятельной работы.

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

Вопросы для контроля знаний

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>?

3 Лабораторная работа № 3. Язык составления сценариев JavaScript.

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

Порядок выполнения работы.

1 Изучить основные теоретические положения, сделав необходимые выписки в конспект.

2 Получить задание у преподавателя, выполнить задание.

3 Оформить отчет.

Требования к отчету.

1 Цель работы.

2 Результаты работы.

3 Выводы.

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.

Скрипт размещается между двумя парными тегами <SCRIPT>...</SCRIPT>. Обычно запись скрипта выглядит так, как представлено в таблице 1.

Таблица 1 – Стандартная запись скрипта

Запись скрипта	Интерпретация записи
<SCRIPT language=JavaScript>	Начало скрипта
<!-- … Код на JavaScript … //-->	Скрипт представлен как HTML-комментарий, чтобы не «смущать» браузеры, которые о скриптах не знают.
</SCRIPT>	Конец скрипта
<NOSCRIPT> … Для браузеров, которые не поддерживают JavaScript ... </NOSCRIPT>	Эта команда специально для пользователей, у которых браузер не понимает скриптов.

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.

Тип переменной зависит от того, какой тип информации хранится. JavaScript не является жестко типизированным языком. Это означает, что программист может не определять тип 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..."
```

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 «приращ».

Если условие ложно, цикл прекращается («команда» и «приращ» 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 команда 1, иначе команда 2. В сокращенной форме ветвь else отсутствует.

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

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.

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

Определение функции состоит из ключевого слова function, сопровождаемого следующими атрибутами:

1) именем функции;

2) списком аргументов функции, приложенной в круглых скобках и отделяемые запятыми:

3) JavaScript-утверждениями, которые определяют функцию, приведенную в фигурных скобках {...}.

Можно использовать любые функции, определенные в текущей странице браузера. 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 документе.

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.

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

```
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);
```

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, 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;
}
```

Необходимо использовать 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 определения метода. Например, пусть есть набор файлов изображений GIF, и необходимо определить метод, который показывает информацию для car, наряду с соответствующим 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.

Далее необходимо определить функцию методом из car, прибавляя утверждение определению объекта:

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

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:

```
carl.displayCar ()
car2.displayCar ()
```

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 объекта, 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)">
```

Метод this обращается к вызывающему объекту.

Задание для самостоятельной работы.

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

Вопросы для контроля знаний

1 Назначение JavaScript и область применения.
2 В какие документы внедряется JavaScript?
3 Чувствителен ли к регистру JavaScript?
4 Механизм определения объектов в JavaScript.
5 Использование ключевого слова «this» для обращения к объекту.

4 Лабораторная работа № 4. Сетевое программирование с сокетами и каналами

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

Порядок выполнения работы.

1 Изучить основные теоретические положения, сделав необходимые выписки в конспект.
2 Получить задание у преподавателя, выполнить задание.
3 Оформить отчет.

Требования к отчету.

1 Цель работы.
2 Результаты работы.
3 Выводы.

Теоретические сведения

Клиент-сервер (от 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 создается пользователем. Работа клиента состоит в попытке создать соединение с сервером, и это выполняется с помощью специального клиентского объекта. Как только соединение установлено, то и клиентская, и серверная сторона соединения превращаются в потоковый объект ввода/вывода, таким образом можно трактовать соединение, как будто пользователь читает и пишет файл. Таким образом, после установки соединения, используются команды ввода/вывода. Это одна из особенностей работы по сети в Java.

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 устанавливается клиент или сервер, тогда необходимо 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 запрашивается служба, ассоциированная 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.

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

В Java создается сокет, чтобы создать соединение с другой машиной, затем получается 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 используются 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 создается ServerSocket, то ему дается только номер порта и не дается ему IP адрес, поскольку он уже есть на той машине, на которой он установлен. Однако, когда создается Socket, то необходимо передать ему и IP адрес и номер порта, с которым необходимо соединиться. Socket, который возвращается из метода ServerSocket.accept( ) уже содержит всю эту информацию.

Представленный ниже 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 с соответствующей конвертацией и оберткой, чтобы «слушать», что «говорит» сервер, который, в данном случае, просто отсылает слова назад.

Пример сервера, 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 возникнет соединение:*/
       Socket socket = s.accept();
       try {
System.out.println("Connection accepted:
" + socket);
BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream())); /* Вывод
автоматически выталкивается из буфера PrintWriter'ом*/
       PrintWriter out = new PrintWriterlBufferedWriter(new
OutputStreamWriter(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 {
.println(
System.out.println("closing...");
socket.close();
}
finally {
s. close ();
} } }
```

Таким образом, для ServerSocket'а необходим только номер порта, а не IP адрес (так как он запускается на локальной машине!). Когда вызывается 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.

В описанном механизме тщательно обработана ответственность за очистку сокета. Если конструктор ServerSocket завершится неудачей, программа просто завершится. 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 необходимо 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.

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

```
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'а и OutputStream'а конвертируются в объекты Reader'а и Writer'а с помощью «классов-конвертеров» InputStreamReader и OutputStreamreader, соответственно. Можно также использовать классы InputStream и OutoutStream напрямую, но, с точки зрения вывода, есть явное преимущество в использовании этого подхода. Оно проявляется в PrintWriter'е, 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 при записи 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 представленном примере.

Когда пишется сетевая программа, необходимо быть осторожным при использовании автоматического выталкивания буфера. При каждом выталкива-

нии буфера пакеты должны создаваться и отправляться. Другими словами, конец строки является концом сообщения. Но во многих случаях, сообщения не ограничиваются строками, так что будет более эффективным использовать автоматическое выталкивание буфера, для этого используется встроенный механизм буфферизации для построения и отсылки пакета. В таком случае могут быть посланы пакеты большего размера и процесс обработки пойдет быстрее.

Все открытые в примере потоки являются буфферезированными.

В бесконечном цикле while происходит чтение строк из входного BufferedReader'а и запись информации в System.out и в выходной PrintWriter.

Вход и выход могут быть любыми потоками, связанными с сетью.

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

Пример клиента, 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, JabberServer.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 BufferedWriter(
                    new OutputStreamWriter(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( ) представлены все три способа получения IP адреса локальной заглушки: с помощью null, localhost или путем явного указания зарезервированного адреса 127.0.0.1. Чтобы соединится с машиной по сети, необходимо заменить зарезервированный адрес IP адресом машины.

Socket создается при указании и InetAddress'а, и номера порта. 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 оказался на той же 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 представлено среди выводимого стороной сервера:

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

Это означает, что сервер 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.

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 пользователь не перезапустит 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 было представлено в сервере (в обоих случаях создание соединения начинаете с Socket'а). В данном случае, клиент инициирует обмен путем посылки строки "howdy", за которой следует число. Буфер должен опять выталкиваться (это происходит автоматически из-за второго аргумента в конструкторе PrintWriter'а). Если буфер не будет выталкиваться, процесс обмена повиснет, поскольку начальное "howdy" никогда не будет послано (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.

Аналогичные меры приняты, чтобы сетевые ресурсы, представляемые сокетом, правильно очищены. Для этого используется блок «try-finally».

Задания для самостоятельной работы.

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 п1обеды 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.

Вариант назначается преподавателем.

Вопросы для контроля знаний

1 Что такое архитектура клиент-сервер?

2 Назначение порта, зарезервированные номера портов.

3 Что такое сокет, назначение сокета?

4 Опишите механизм создания входного и выходного потоков.

5 Опишите механизм работы метода accept().

5 Лабораторная работа № 5. Сервлеты

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

Порядок выполнения работы.

1 Изучить основные теоретические положения, сделав необходимые выписки в конспект.

2 Получить задание у преподавателя, выполнить задание.

3 Оформить отчет.

Требования к отчету.

1 Цель работы.

2 Результаты работы.

3 Выводы.

Теоретические сведения

Клиентский доступ из интернета или корпоративного интранета является легким способом 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. Стандартное предоставление Web сервером: 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 создать HTML страницу для описания проблемы; эта страница передавалась Web серверу (через стандартный вывод из CGI программы), который посылал ее назад пользователю. Пользователь должен был 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.

С развитием технологий возникла необходимость в переходе 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. Апплеты предоставляют технологию с вполне достаточной поддержкой, но являются проблематичными в использовании на Web, поскольку нет возможности рассчитывать на поддержку определенной версии Java на клиентском Web браузере, т. е. невозможно рассчитывать, что Web браузер поддерживает Java. В Интранете можно потребовать, чтобы определенная поддержка Java была доступна, что позволит реализовать намного большую гибкость в том, что можно 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 сайтом по причине того, что у него нет правильно установленного программного обеспечения.

Поскольку сервлеты предоставляют 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, но весь ваш код приобретает портируемость между платформами, получаемую от использования Java, и пользователь приобретает доступ ко всему Java API (за исключением, конечно, того, которое производит GUI, такого языка, как Swing).

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 ServletException;

public ServletConfig getServletConfig();

public void service(ServletRequest req, ServletResponse 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 Gener-

icServlet и специально предназначен для обработки HTTP протокола – HttpServelt является одним из тех классов, которые используется чаще всего.

Основное удобство атрибутов сервлетного API состоит во внешних объектах, которые вводятся вместе с классом HttpServlet для его поддержки. Если взглянуть на метод service( ) в интерфейсе Servlet, то можно увидеть, что он имеет два параметра: ServeltRequest и ServletResponse. Вместе с классом HttpServlet, эти два объекта предназначены для HTTP: HttpServletRequest и HttpServletResponse.

Ниже придставлен простейший пример, который показывает использование HttpServletResponse:

```
// (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 ("</TITLEX/HEADXBODY>");
out.print("<hl>Servlets Rule! " + i++);
out.print("</hl></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 пользователь владеет HTML), как последовательность строк. Однако, необходимо обратить внимание на включение «счетчика показов», представленного переменной i. Он автоматически конвертируется в строку в инструкции print( ).

Когда пользователь запустит программу, то увидит, 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 пользователь может без усилий сохранять значения между запросами к сервлету, в то время, как в CGI пользователь должен записывать значения на диск, чтобы сохранить их, что требует большого количества окружения для правильного их получения, а в результате получается 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 пользователю возможность сохранить важные данные во время остановки и восстановления после перезагрузки. Контейнер сервлетов вызывает метод destroy( ) когда он завершает свою работу, так что пользователь всегда получает 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. В следующем примере используется метод service( ), который заботится о выборе GET или POST.

Когда бы форма не отсылалась сервлету, HttpServletRequest поступает со всеми предварительно загруженными данными формы, хранящимися как пары «ключ-значение». Если известно имя поля, то можно просто использовать его напрямую в методе getParameter( ), чтобы найти значение. Можно 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. Если Enumiration пустое, значит нет полей. Это означает, что форма не была отправлена. В данном случае производится форма, а кнопка подтверждения будет повторно вызывать тот же самый сервлет. Однако если поля существуют, они будут отображены, как представлено в примере:

```
// 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 fids = req.getParameterNames();
if (!fids.hasMoreElements()) {
        // Нет отосланной формы -- создаем:
out.print("<html>");
out.print("<form method="POST"" + "action="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("<hl>Your form contained:</hl>");
          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-страниц 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 выполнить замену любого вида для содержимого этой страницы, то Java плохо обработает строки. В этом случае, 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.

Задания для самостоятельной работы.

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, 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 0 °C; 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. Параметры 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.

Вариант назначается преподавателем.

Вопросы для контроля знаний

1 Дайте пояснение сервлетному API.

2 Назначение программы CGI.

3 В чем заключается недостаток апплетов?

4 В чем заключается достоинство сервлетов?

5 Перечислите основные методы жизненного цикла сервлетов.

6 Лабораторная работа № 6. Соединение с базой данных

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

Порядок выполнения работы.

1 Изучить основные теоретические положения, сделав необходимые выписки в конспект.

2 Получить задание у преподавателя, выполнить задание.

3 Оформить отчет.

Требования к отчету.

1 Цель работы.

2 Результаты работы.

3 Выводы.

Теоретические сведения

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 при работе с JDBC.

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);
```

Тогда 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.j dbc.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 registerDriverO драйверы регистрируются, а методом getDriversQ 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) (getlnt(int pos), getFloat(int pos) и др.) и updateТип ().Среди них следует выделить методы getClob(int pos) и getBlob(intpos), 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.

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; charset=Cpl251");
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 =
    DriverManager .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 users");
          out.print("From DataBase:");
    while (rs.next()) {
          out.print("<br>Name:–> " + rs.getString(1) + " Phone:-> " + rs.getlnt(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 созда-
ния 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-гоблока 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 инициализирован 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, т. к. незакрытое («провисшее») соединение снижает быстродействие системы.

Задания для самостоятельной работы.

В соответствии с заданием, выданным преподавателем, необходимо выполнить следующие действия.

1 Организацию соединения с базой данных вынести в отдельный класс, метод которого возвращает соединение.

2 Создать БД. Привести таблицы к одной из нормированных форм.

3 Создать класс для выполнения запросов на извлечение информации из БД с использованием компилированных запросов.

4 Создать класс на добавление информации.

5 Создать HTML-документ с полями для формирования запроса.

6 Результаты выполнения запроса передать клиенту в виде HTML-документа.

Вопросы для контроля знаний

1 Дайте определение JDBC.

2 Опишите последовательность действий при работе с JDBC.

3 Поясните работу сервлета, осуществляющего простейший запрос на выбор всей информации из созданной таблицы.

7 Лабораторная работа № 7. Удаленный вызов метода (RMI)

Цель работы: реализовать приложение, используя технологию RMI.

Порядок выполнения работы.

1 Изучить основные теоретические положения, сделав необходимые выписки в конспект.

2 Получить задание у преподавателя, выполнить задание.
3 Оформить отчет.

Требования к отчету.
1 Цель работы.
2 Результаты работы.
3 Выводы.

Теоретические сведения

Удаленный вызов метода (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.

RMI делает тяжелым использование интерфейсов. Когда пользователь желает создать удаленный объект, он маскирует 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 пользователь не задумывается об этом, он просто посылает сообщения через ссылку на интерфейс.

Когда пользователь создает удаленный интерфейс, он следует 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 PerfectTimel extends Remote {
long getPerfectTime() throws RemoteException;
}
```

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

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.

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

Реализация удаленного интерфейса PerfectTimel представлена в следующем примере:

```
// Реализация удаленного
// объекта PerfectTime.
// {Broken}
package c15.rmi;
import java.rmi.*;
import ]ava.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;
public class PerfectTime extends UnicastRemoteObject
implements PerfectTimel {
   // 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 Exception {
         System.setSecurityManager(new RMISecurityManager());
         PerfectTime pt = new PerfectTime();
         Naming.bind("//peppy:2005/PerfectTime", pt)
         System.out.println("Ready to do time");
      }
}
```

Здесь main( ) обрабатывает все детали установки сервера. При обслуживании RMI объектов, в некотором месте программы необходимо выполнить следующее.

1 Создать и установить менеджер безопасности, который поддерживает RMI. Для RMI доступен только один менеджер, как часть пакета Java, он называется RMISecurityManager.

2 Создать один или несколько экземпляров удаленного объекта. В приведенном примере создан объект 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 пользователю сделать так, чтобы клиент обращался к репозиторию только один раз, чтобы получить первый удаленный объект.

В этом примере осуществлен вызов статического метода Naming.bind( ). Однако, этот вызов требует, чтобы репозиторий был запущен как отдельный процесс на компьютере. Имя сервера репозитория rmiregistry, в системе Windows используется команда 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 вызвать rmiregestry как это показано выше, без аргументов, для репозитория будет использован порт по умолчанию 1099. Если необходимо, чтобы репозиторий был на другом порту, то нужно добавить аргумент в командную строку, который указывает порт. Например, интересующий порт 2005, так что rmiregestry должна быть запущена следующей командой под 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.

Существуют некоторые особенности работы с RMI.

1 localhost не работает с RMI. То есть, экспериментируя с RMI на одной единственной машине, необходимо предоставить имя машины. Чтобы найти имя используемой машины в среде Windows, нужно запустить контрольную панель и выберать «Network». При выборе закладки «Identification», можно увидеть имя используемого компьютера.

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

Учитывая все вышесказанное, команда bind( ) принимает вид:

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

Если используется порт по умолчанию 1099, то не нужно указывать порт, поэтому можно написать следующее:

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

Пользователь должен быть способен 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 также возможно использовать любое название. 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, то ответ будет следующий: AlreadyBoundException. Чтобы предотвратить это, можно всегда использовать rebind( ) вместе bind( ), так как rebind( ) либо добавляет новую запись, либо заменяет уже существующую.

Даже после того, как main( ) завершит работу, объект будет создан и зарегистрирован, он останется «жить» в репозитории, ожидая прихода клиентов и запросов от них. До тех пор, пока работает nniregistry и не вызван метод Naming.unbind( ) с присвоенным именем, объект будет существовать. По этой причине, когда разрабатывается код, необходимо выгружать rmiregistry и перезапускать его, когда скомпилирована новая версия удаленного объекта.

Нет необходимости принудительно запускать rmiregistry, как внешний процесс. Если известно, что разрабатываемое приложение является единственным, использующим репозиторий, то можно запускать его внутри программы с помощью следующей строки:

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

Как и прежде, 2005 – это номер порта, который использовался в примере. Это эквивалентно запуску rmiregistry2005 из командной строки, но такой запуск часто будет более последовательным, при разрабатке RMI кода, так как при этом пропускаются дополнительные шаги запуска и остановки репозитория. Как только выполнится этот код, можно вызвать bind( ), используя Naming.

Если скомпилировать и запустить PerfectTime.java, программа не будет работать, даже если репозиторий запущен правильно. Это происходит потому, что рабочая среда для RMI еще не до конца создана. Необходимо сначала создать заглушки и скелеты, которые обеспечат операцию сетевого соединения и позволят считать, что удаленный объект – это просто другой локальный объект на рабочей машине.

Любые объекты, которые передаются при вызове или получении от удаленного объекта должны реализовывать Serializable (если необходимо передавать удаленную ссылку вместо самого объекта, объектные аргументы могут реализовывать Remote), так что можно представить, что заглушки и скелеты автоматически выполняют сериализацию и десериализацию, как будто они «руководят» всеми аргументами через сеть и возвращают результат. Поэтому необходимо создать заглушки и скелеты. Это простой процесс: нужно вызвать инструмент rmic для скомпилированного кода и создать необходимые файлы. Таким образом, требуется просто добавить еще один шаг в процесс компиляции.

Однако, инструмент rmic на распознает пакеты и classpaths. PerfectTime.java расположен в пакете c15.rmi, и если вызвать rmic в том же директории, в котором расположен PerfectTime.class, rmic не найдет файл, так как он ищет в переменной classpath. Таким образом, нужно указать расположение в переменной classpath, например: rmic c15.rmi.PerfectTime.

Нет необходимости находиться в директории, содержащем PerfectTime.class, когда запускается эта команда, а результат будет помещен в текущий каталог.

Когда запуск rmic завершится успешно, будут получены два новых класс в директории:

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

Соответственно это заглушка и скелет. После их создания можно получить сервер и клиент для общения между собой.

Главная цель RMI состоит в упрощении использования удаленных объектов. Есть только одна дополнительная вещь, которую необходимо выполнить в клиентской программе, это найти и получить удаленный интерфейс с сервера. Все остальное – это обычное Java программирование: посылка сообщений объекту. Ниже приведен пример программы, 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 Exception {
    System.setSecurityManager(new RMISecurityManager());
```

```
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 используется URL, то можно 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 будет использоваться класс, то получится исключение. Можно видеть вызов метода t.getPerfectTime(), так как то, что есть в наличии – это ссылка на удаленный объект, программирование с которой не отличается от программирования с локальным объектом (с одним отличием: удаленные методы выбрасывают RemoteException).

Задание для самостоятельной работы.

Создадать систему, функционально состоящую из двух компонентов – сервера (процессингового центра) и клиента (касса). Предполагается, 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).

Вопросы для контроля знаний

1 Поясните назначение RMI.

2 Перечислите особенности создания удаленного интерфейса.

3 Приведите пример реализации удаленного интерфейса.

4 Перечислите действия, которые необходимо выполнить для обслуживания RMI объектов.

5 Особенности работы с RMI.

Список литературы

1 Программирование сетевых приложений : лабораторные работы (практикум) для студентов специальностей 1-40 01 01 «Программное обеспечение информационных технологий» и 1-40 01 02 «Информационные системы и технологии» / сост. **Н. М. Прибыльская**. – Минск : БНТУ, 2013. – 67 с.

2 **Флэнаган, Д.** JavaScript : карманный справочник / Д. Флэнаган ; пер. с англ. – 3-е изд. – Москва: Вильямс, 2015. – 314 с.

3 **Шилдт, Г.** Java 8. Полное руководство / Г. Шилдт ; пер. с англ. – 9-е изд. – Москва: Вильямс, 2015. – 1375 с.

4 **Курняван, Б.** Программирование WEB-приложений на языке Java / Б. Курняван ; пер. с англ. – Москва: Лори, 2014. – 880 с.