МИНИСТЕРСТВО ВЫСШЕГО И СРЕДНЕГО СПЕЦИАЛЬНОГО ОБРАЗОВАНИЯ РЕСПУБЛИКИ УЗБЕКИСТАН

НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ УЗБЕКИСТАНА
имени Мирзо Улугбека

WEB –ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Для студентов магистратуры механико-математического факультета

УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС

Направление: **5А480103 Прикладная математика и информационные технологии**

Общее количество часов	64
Лекции	18
Практические занятия	24
Самостоятельная работа	22
Консультации	1,2
Рейтинг	10

Ташкент-2011

СОДЕРЖАНИЕ

Рабочий план и рейтинговые показатели утверждены на заседании кафедры Информатики и прикладного программирования от _28_ августа 2011г. протокол №1 .

Учебная программа разработана в соответствии с учебным планом направления Прикладная математика и информационные технологии.

Составитель: к.т.н. доцент Варламова Л.П**.** _____

Заведующий кафедрой: д.ф-м.н,проф.Абдурахимов Б.Ф**.** _____

Учебная программа рассмотрена и утверждена на заседании Ученого Совета Механико-математического факультета Национального Университета Узбекистана протокол № 1 от _28 августа 2011 г.

Председатель Ученого Совета
2011 г. "_28__" августа_. _____ проф. **Б А Шоимкулов**

Web-ориентированные технологии

Введение

На сегодняшний день технологии математического моделирования занимают одно из ведущих мест среди современных информационных технологий. Разработка и проектирование баз данных на основе веб-ориентированных технологий, как никогда становится все более актуальным.

Содержание лекций (18 ч).

ActiveX – это широкое понятие, описывающее целое семейство новых технологий: от новых языков (таких как VBScript, JavaScript и Java), новых возможностей операционной системы Windows, - и до полномасштабных технологий, ориентированных на Internet, т.е. элементов управления ActiveX, технологий активных сценариев (ActiveScripting) и активных серверных страниц (Active Server Pages).

WWW: основные технологии и термины. Язык HTML и его основы. Ввод и форматирование текста. Технологии ActiveX. Клиентские технологии. Серверные технологии. Элементы управления ActiveX. Активные документы. Создание активных сценариев с помощью VBScript, JavaScript. Создание элементов управления ActiveX с помощью Visual Basic. Информационный сервер Internet. Active Server Pages или создание серверных сценариев. Основы PHP технологии.

Содержание семинарских занятий (24 ч)

Язык HTML и его основы. Ввод и форматирование текста. Элементы и теги. Использование элементов заголовка. Задание характеристик шрифта. Элементы управления ActiveX. Создание активных документов. Создание активных сценариев с помощью VBScript, JavaScript. Типы данных. Объекты и их классы. Функции JavaScript. Операторы. Internet Explorer и объектная модель сценариев. Active Server Pages или создание серверных сценариев. Язык XML. Функции Cookie. Установка Web сервера Apache и модуля PHP.

Содержание самостоятельных работ(22 ч)

Этапы создания математических моделей. Разработка тематических Web - сайтов и страничек. Работа с базами данных.

Тематика аудиторных занятий

№	Тема	Лек.	Пр.	Лит-ра
1.	Правила построения HTML-документов. Структура HTML-документа Основные понятия языка HTML. Тэги, параметры тэгов.	2	2	1-3
2	Динамический HTML. Таблицы стилей. Форматирование текста, фон, графика, списки, таблицы. Гиперссылки.	2	2	1-4

3	Построение интерактивных WEB-страниц. Работа с фреймами и шаблонами. Создание динамических WEB-страниц.	2	2	4, 5
4	Элементы языка Java-script, Java, VBScript	2	4	4-6
5	Система Windows (ActiveScripting) и активные серверные страницы (Active Server Pages).	2	4	1-3, 5
6	MySQL. Серверные технологии	4	4	5-8
7	Информационный сервер Internet. Active Server Pages или создание серверных сценариев	2	4	6-8
8	Основы PHP.Установка модуля PHP Web сервера Apache. Функции Cookie. Язык XML.	2	2	6-8
	Всего	18	24	

Тематика самостоятельных работ

Тема	Самостоятельная работа	Часы
Правила построения HTML-документов.	Изучение тэгов и команд языка HTML	2
Динамический HTML. Таблицы стилей.	Вставка таблиц, изучение стилей. Каскадные таблицы	2
Форматирование текста, фон, графика, списки, таблицы. Гиперссылки.	Разработка Web- страниц, включающих различные фоны, графические изображения, списки, таблицы, гиперссылки	2
Построение интерактивных WEB-страниц.	Форма работа с фреймами и шаблонами. динамических WEB-страниц.	4
Элементы языка Java-script, Java, VBScript	Изучение скриптов языка	4
ActiveScripting, Active Server Pages	Разработка серверных страниц	4
MySQL	Изучение элементов языка	4
Всего		**22**

Литература:

1. И.Каримов. Мировой финансово-экономический кризис, пути и меры по его преодолению в условиях Узбекистана.: Т 2009.
2. Том Армстронг. ActiveX: Создание Web-приложений. BHV, Киев, 1998.
3. Д. Кирсанов. «Веб-дизайн» С.Петербург 2001.
4. И. Шапошников «Web-сайт своими руками» Москва 2000
5. Вахтуров В.В. JavaScript. Освой на примерах.-СПб.: БХВ-Петербург, 2007.-400с.: ил.
6. Х. Вильямсон. «Универсальный Dynamic HTML» М. «Питер» 2004.
7. Д. Лешев. «Создание интерактивного веб-сайта» М. «Питер» 2004.
8. Елена Бенкен. PHP, MySQL, XML программирование для интернета. Санк-Петербург, «БХВ- Петербург» 2007.
9. Кристина Пейтон, Андре Миллер. PHP 5& MySQL 5. Москва «Бином», 2007.
10. Максим Кузнецов, Игорь Симдянов. MySQL на примерах. Санк-Петербург, «БХВ- Петербург» 2007.

Дополнительная литература:

1. М.Арипов. «Internet ва электрон почта асослари». Тошкент, 2000.
2. Джорж М. «Компьютерная анимация персонажей» М. «Питер» 2004.
3. Рашевский Н. Модели и математические принципы в биологии. М.: Мир, 1968.-48с.
4. М. Арипов, А. Тиллаев «Информатика ва щисоблаш техникаси асослари» Олий ы=ув юртлари учун. Электрон Дарслик (CD) Тошкент, 2001.
5. М. Арипов, А. Тиллаев «Веб-сащифалар яратиш технологиялари» Тошкент, 2004.

Лекция 1: HTML. Основные понятия. Теги

План

История возникновения HTML

HTML (Hypertext markup language. С англ. — «язык гипертекстовой разметки») — это приложение SGML (Standard Generalized Markup Language [Стандартный обобщенный язык разметки]), соответствующее международному стандарту ISO 8879; оно считается стандартным языком публикации в World Wide Web.

Язык HTML был разработан британским учёным Тимом Бернерсом-Ли приблизительно в 1991—1992 годах в стенах Европейского совета по ядерным исследованиям в Женеве (Швейцария).

HTML создавался как язык для обмена научной и технической документацией, пригодный для использования людьми, не являющимися специалистами в области вёрстки. HTML успешно справлялся с проблемой сложности SGML путём определения небольшого набора структурных и семантических элементов, служащих для создания относительно простых, но красиво оформленных документов. Помимо упрощения структуры документа, в HTML внесена поддержка гипертекста. Мультимедийные возможности были добавлены позже. Изначально язык HTML был задуман и создан как средство структурирования и форматирования документов без их привязки к средствам воспроизведения. В идеале, текст с разметкой HTML должен был без стилистических и структурных искажений воспроизводиться на оборудовании с различной технической оснащенностью (цветной экран современного компьютера, монохромный экран органайзера, ограниченный по размерам экран мобильного телефона или устройства и программы голосового воспроизведения текстов).

Текстовые документы, содержащие код на языке HTML (такие документы традиционно имеют расширение «html» или «htm»), обрабатываются специальными приложениями, которые отображают документ в его форматированном виде. Наличие в телефоне HTML-браузера. С помощью такого веб-браузера можно просматривать обычные html-страницы в сети Интернет. Для сравнения, телефон с поддержкой WAP может просматривать только специально оптимизированные для мобильного телефона страницы формата WML или XHTML.

Создание документа в HTML

В HTML документы записываются в ASCII формате и поэтому могут быть созданы и отредактированы в любом текстовом редакторе (например, Emacs или vi на UNIX машинах, или любом редакторе на IBM PC).

Любой гипертекст похож на книгу и может быть разбит на отдельные структурные элементы: Собственно документ Главы, параграфы, пункты, подпункты Абзацы.

Для каждого из этих элементов в HTML существуют определенные стили, описывающие в каком виде пользователь увидит текст на экране. Пусть мы создали файл minihtml.html:

```
<BODY>
 <TITLE>Пример HTML-текста</TITLE>
```

<H1>Глава 1</H1>
<H2>Параграф 1.</H2>
Добро пожаловать в HTML! Здесь мы расскажем, как надо и как не надо писать гипертексты.
<H2>Параграф 2.</H2>
</BODY>

Заголовок документа начинается с <TITLE> и заканчивается </TITLE>. Заголовок первого уровня (Главы) выделится символами <H1> и </H1>) Заголовоки последующих уровней (параграфы, пункты, подпункты и т.п.) - символами <Hx> и </Hx>), где x - числа 2, 3, ... Абзац - символами <P>

NOTE: HTML не различает, какими буквами набраны символы форматирования: <title> равносильно <TITLE> или <TiTlE>. Не все стили поддержаны всеми WWW-browsers. Если browser не поддерживает стиль, то он его игнорирует. (Поэтому не страшно, если Вы уже сейчас начнете пользоваться при форматировании абзацев символом. Ниже мы более подробно остановимся на этом вопросе.)

HTML: понятие тега

HTML (HyperText Markup Language) — не является языком программирования в традиционном понимании. HTML — язык разметки документа. При разработке HTML-документа выполняется разметка текстового документа. Такие символы, которые управляют отображением текста и при этом сами не отображаются на экране, принято называть тегами.

Все теги языка HTML выделяются символами-ограничителями (< и >), между которыми записывается идентификатор (имя) тега и, возможно, его параметр.

Большинство тэгов используются попарно, т.е. для определенного тега, назовем его открывающим, в HTML-документе имеется соответствующий закрывающий тег. Закрывающий тег записывается так же, как и открывающий, но с символом / (прямой слэш) перед именем тега.

Теги, которые нуждаются в соответствующих завершающих тегах, будем называть тегами-контейнерами. Все, что записано между открывающим и завершающим тегом, будем называть содержанием тега-контейнера. Иногда завершающий тэг можно опускать.

HTML-документ — это один большой контейнер, который начинается с тега <HTML> и заканчивается тегом </HTML>:

<HTML>Содержание документа</HTML>

Контейнер HTML или гипертекстовый документ состоит из двух других вложенных контейнеров: заголовка документа (HEAD) и тела документа (BODY). Рассмотрим простейший пример классического документа.

```
<HTML>
<HEAD>
<TITLE>Простейший документ</TITLE>
</HEAD>
<BODY TEXT=#0000ff BGCOLOR=#f0f0f0>
<H1>Пример простого документа</H1>
<HR>
Формы HTML-документов
<UL>
<LI>Классическая
<LI>Фреймовая
</UL>
<HR>
```

</BODY>
</HTML>

Все теги по их назначению и области действий можно разделить на следующие основные группы:

- Определяющие структуру документа;
- Оформление блоков гипертекста (параграфы, списки, таблицы, картинки);
- Гипертекстовые ссылки и закладки;
- Формы для организации диалога;
- Вызов программ.

Компания Netscape Communication расширила классическую форму документа возможностью организации фреймов (кадров), позволяющих разделить рабочее окно программы просмотра на несколько независимых фреймов. В каждый фрейм можно загрузить свою страницу HTML. Приведем пример документа с фреймами.

<HTML>
<HEAD>
<TITLE>Документ с фреймами</TITLE>
</HEAD>
<FRAMESET COLS="30%,*">
<FRAME SRC=frame1.htm NAME=LEFT>
<FRAME SRC=frame2.htm NAME=RIGHT>
</FRAMESET>
</HTML>

Назначение заголовка

Заголовок HTML-документа является необязательным элементом разметки. В HTML 2.0 предлагалось вообще отказаться от элементов HEAD и BODY. В то время в HTML не было элементов, которые использовались одновременно и в заголовке, и в теле документа. Современная практика HTML-разметки такова, что почти в каждом документе есть HTML-заголовок.

Первоначально существование заголовка определялось необходимостью именования окна браузера. Это достигалось за счет элемента разметки TITLE:

```
<HTML>
 <HEAD>
  <TITLE>Это заголовок</TITLE>
 ...
 </HEAD>
 <BODY>
 ...
 </BODY>
</HTML>
```

Контрольные вопросы

1. Кто явился первым разработчиком HTML?
2. Что означает HTML?
3. Из каких частей состоит HTML документ?
4. Что такое теги?
5. Какие теги заголовка вы знаете?

Лекция 2: CSS- Каскадные таблицы стилей

План

Теги html предоставляют ограниченные возможности оформления страниц. Для их расширения, в тег может быть добавлен атрибут "style", указывающий на некоторое свойство объекта и устанавливающий значение этого свойства. При этом каждый тег может иметь несколько атрибутов "style", а каждый атрибут "style", указывать несколько свойств, заключенных в общие кавычки разделенных точкой с запятой. Например, два равносильных определения:

```
<P style='color: green' style='border: solid red'>
<P style='color: green; border: solid red'>
```

Действие указанных параметров оканчивается с окончанием действия заключающего их тега.

Наиболее часто атрибут "style" используется в элементах <BODY>, <DIV>, <P> и <SPAN>. Поскольку элемент SPAN не несет в себе перевода строки и сам по себе не отделяет каким-либо образом заключенный в него текст от остального документа, его применение имеет смысл только вместе с атрибутом "style". С другой стороны, если оформлению подлежит часть текста, не являющаяся отдельным блоком, выполнить его удобней с помощью этого элемента. Например:

```
<P> Это пример простого текста <span style=' color: green'>и зеленого текста
</span> в одной строке.</P>
```

Для сокращения размера страницы, вместо указания одного и того же оформления каждому абзацу, лучше указывать оформление DIV блокам или даже, всему BODY. Однако если текст имеет много чередующихся стилей оформления, их определение стоит выделить в отдельный блок: с одной стороны это сделает более удобной корректировку оформления, с другой, позволит дополнительно сократить размер файла. Загрузки и отображение файла ускорится не только за счет уменьшения размера, но и за счет того что, определения стиля уже будет находиться в оперативной памяти и не потребует дополнительной обработки.

Размещение CSS

Блок определения стилей помещается либо в заголовок страницы, внутри элемента head, либо в отдельный файл. При размещении в разделе заголовка, блок определения стилей объявляется и заканчивается тегом STYLE, а для исключения отображения его содержимого, оно заключается в знаки комментария (<!-- и -->):

```
<STYLE>
<!--
-->
</STYLE>
```

При размещении в отдельном файле, в разделе заголовка помещается ссылка типа "rel" с атрибутом Href, указывающим имя файла, содержащего блок определения стилей. Например:

```
<link rel="stylesheet" href="style.css">
```

Поскольку определение стиля возможно на различных языках, в объявлении блока STYLE и ссылке на файл с определениями стилей следует указать в атрибуте TYPE язык определения. Для языка **CSS** (каскадных таблиц стилей), в частности, следует указать TYPE="text/css". На сегодняшний день этот язык и тип является значением по умолчанию, но в будущем ситуация может измениться и, для предотвращения недоразумений, вышеприведенные объявление стилей и ссылку, следует уточнить:

```
<STYLE type="text/css">
<!--
-->
</STYLE>
```

и

```
<link rel="stylesheet" href="style.css" type="text/css">
```

Выбор места размещения определения стилей зависит от нескольких обстоятельств. Нет смысла выносить в отдельный файл определения стилей, если оно предназначено для единственного документа. Вынесение определения стилей в отдельный файл имеет смысл, если эти стили едины для большего числа документов, например, целого сайта – не придется вписывать определения в каждую страницу. Кроме того, ускорится загрузке: загрузив определение с первой страницей сайта, браузер не будет вновь обращаться за ним к серверу и тем сократит время загрузки следующих страниц. При этом, если на какой-то из страниц, определенный в этом файле стиль должен отображаться иначе, его можно переопределить, указав загрузку дополнительного файла определений, число которых не лимитируется или, указав новые параметры стиля в блоке STYLE после ссылки на файл определений. На худой конец, можно вставить <span style...> непосредственно в текст – иногда это целесообразней чем создание нового стиля, ради переопределения одного из множества свойств.

Однако существуют ситуации, когда вынесение определения стилей в отдельный файл даже для большого числа связанных документов оказывается нежелательным. Такая ситуация возникает, если документы могут использоваться в отрыве от остальных. Например, автор сайта приносит проект одной из своих страниц к приятелю, чтобы обсудить, в числе прочего, ее дизайн и, обнаруживает что, забыл скопировать на дискету определения стилей, в результате чего, страницы выглядит совершенно не так, как должна. Другой случай нежелательности отделения таблицы стилей обнаруживается, когда вместо всей страницы из интернета на диск сохраняется только текст. Для Internet Explorer это можно указать, для сокращения времени сохранения, а для Opera другой вариант вообще не предусмотрен (даже сохраняя рисунки, она не сохраняет файлы с таблицами определения стилей). В результате, увидев сохраненную страницу в Off-Line, посетитель испугается так, что больше никогда не зайдет на сайт, с которого загрузил такую глюченную страницу. Третья ситуация – недостаток квалификации пользователя. Не имея представления о каких-то там **CSS** и CSS-файлах, неискушенный пользователь может сам удалить, сохранившийся на жестком диске "мусор" в виде CSS-файлов, и потом долго пытаться понять, кто изуродовал закачанные им, такие красивые, изначально, страницы.

С учетом перечисленных факторов, следует сделать вывод что, если размер определения стилей не превышает 10 % от общего размера документа, то его вынос в отдельный файл следует трактовать как извращение. Впрочем, на пост советских частных сайтах, извращений хватает и без этого – повсеместным стандартом стали сайты, где в окружении огромного числа баннеров и прочей повторяющейся чепухи на каждой странице представлено всего один – два небольших абзаца уникального текста и ссылки на другие такие же бессодержательные страницы, в угоду баннерным сетям и увеличению показов баннеров, вынуждающих посетителя совершить десятки переходов для получения интересующей их информации.

Синтаксис CSS.

Структура определения стилей.

Определение оформления внутри тега с помощью атрибута STYLE имеет следующий вид: внутри угловых скобок тега, отделенный от имени тега пробелом, указывается атрибут STYLE, затем знак равенства и в кавычках перечень свойств и их значений, разделенных точкой с запятой. Свойства отделяются от значений двоеточием:

```
<P style='color: green; border: solid red'>
```

Определение стилей в заголовке страницы или отдельном файле происходит почти так же, только тег STYLE, указанный вначале определения не повторяется, а роль кавычек выполняют фигурные скобки:

```
P {color: green; border: solid red}
```

Поскольку для HTML-документа пробел и перенос строки равносильны, эту запись можно разнести на несколько строк:

```
P {
color: green;
border: solid red
}
```

Это обычно и делается многими авторами для удобства поиска отдельных свойств, хотя при увеличении таблицы стилей, усложняется поиск начала и конца стиля. Чтобы избежать этого, атрибуты часто пишутся с отступами от края:

```
P {
    color: green;
    border: solid red
}
```

Во внутрь определения стиля может быть помещен комментарий, заключенный между /* и */. Например:

```
P {/* Это пример определения стиля */
    color: green;
    border: solid red
}
/* комментарий может быть помещен в любом месте */
```

Поскольку определения стиля закомментировано тегами <!-- и -->, комментарий между /* и */ не будет отображаться в теле документа. К ошибкам интерпретации он тоже не приведет, поскольку на языке **CSS** это то же самое что <!-- и --> на языке HTML.

Для тегов имеющих одинаковые определения допускается группировка, при которой они записываются в одну строку через запятую, а затем указываются из свойства и значения свойств:

```
DT, DD {margin-top: 0; margin-bottom: 0}
```

В одном блоке STYLE может быть множество стилей. Допустимое количество самих блоков так же, неограниченно. При этом если определения стилей находятся в файлах, заданных атрибутом link, предпочтение отдается первому из них – следующий ищется только в случае невозможности загрузить или интерпретировать первый, если же одни и те же стили несколько раз определены в самом документе (что может быть вызвано только невнимательностью автора), предпочтение отдается последнему.

Пример определения стилей для нескольких тегов:

```
<STYLE type="text/css">
<!--
A:link  {color: #F30; background: transparent}
A:visited  {color: #363; background: transparent}
A:active   {color: #F30; background: transparent}
A:hover {background: #FFA}
PRE {margin-left: 2em}
P {
  margin-top: 0.6em;
  margin-bottom: 0.6em;
}
DT, DD {margin-top: 0; margin-bottom: 0} /* opera 3.50 */
-->
</STYLE>
```

В этом примере приведены специальные селекторы (теги) A:link, A:visited, A:active и A:hover, означающие: непосещенную гиперссылку, посещенную гиперссылку, выполняемую в данный момент гиперссылку и гиперссылку над которой находится указатель мыши. Эти теги не приходится указывать в теле документа (следует указать лишь <A> с необходимыми ему атрибутами). Браузер сам определит состояние ссылки и организует ее внешний вид соответствующим образом.

Разделение тегов на классы, понятие селектора

Вышеприведенный пример на гиперссылках показал что, **CSS** допускает наличие нескольких видов оформления одних и тех же тегов, таким образом, понятие тега становится слишком узким для применения в **CSS** и заменяется понятием селектора. Селектор это общее название стиля. Если стиль определяется для тега P, то тег P является селектором **CSS**, а если для A:Link, то селектором является не тег A, а именно A:Link, не являющийся тегом и требующий для своего обозначения отдельного термина. Таким образом, селектор – это имя, данное элементу страницы для которого определяется стиль отображения.

Оперирование селекторами имеет принципиальное значение для таблиц стилей, поскольку позволяет создавать различное оформления для одних и тех же тегов. Для того чтобы выделить тег в отдельный класс оформления, после его имени, через точку, записывается произвольное название класса:

```
<Style type="text/css"><!--
P.MyClass {margin-left:5pt; margin-top: 0.6em; margin-bottom: 0.6em}
--></STYLE>
```

При необходимости обращения к этому классу в объявлении тега указывается класс, на теги, имеющие то же имя, но принадлежащие к другому классу, параметры отображения указанного класса не повлияют. Например:

```
<P>Пример обычного абзаца
<P class='MyClass'>Пример абзаца с особыми параметрами
<P>Снова обычный абзац.
```

В определении классов допустима та же группировка, что и в общем определении оформления тега:

```
P.MyClass, Div.MyClass {margin-top: 0.6em; margin-bottom: 0.6em}
```

Если же произвольное имя класса является уникальным и не входит в состав других селекторов, внутри объявления стилей на него можно ссылаться, опуская имя тега, но обязательно с точки. Например:

```
Div.MyClass {margin-top: 0.6em; margin-bottom: 0.6em}
.MyClass {margin-left: 0.6em; margin-right: 0.6em}
```

Идентификаторы CSS

Идентификаторы пишутся не через точку, а через знак лестницы:

```
P#MyClass {margin-left:5pt; margin-top: 0.6em; margin-bottom: 0.6em}
```

Затем, внутри тегов указываются через атрибут ID:

```
<P>Пример обычного абзаца
<P ID='MyClass'>Пример абзаца с особыми параметрами
<P>Снова обычный абзац.
```

В отличие от селекторов они не привязаны к тегам и могут определяться отдельно от них, с тем, чтобы затем влиять на отображение любых элементов текста. Например:

```
<HTML>
 <HEAD>
  <TITLE>Title</TITLE>
  <STYLE TYPE="text/css">
    #pastoral { color: #00FF00 }
  </STYLE>
 </HEAD>
 <BODY>
  <H1 ID=pastoral>Зеленый заголовок, стиль которого не определялся</H1>
<P ID=pastoral>Зеленый текст простого абзаца стиль которого не определялся
<P>Текст с цветом по умолчанию для простого абзаца стиль которого не
определялся
</BODY>
</HTML>
```

Контекстные селекторы

Вместо создания отдельного стиля представления содержимого, **CSS** позволяет определять его условное оформление, зависящее от контекста, для чего в определении стилей указывается тег, в контексте которого элемент должен иметь особое оформление, а через пробел от него, тег оформляемого элемента с перечислением параметров отображения. Например:

```
H1 EM {color: red}
```

Элемент EM, являющийся тегом HTML, будет оформлен по тексту как обычный EM, но внутри заголовка первого уровня примет указанное оформление. Фактически, контекстные

селекторы мало отличаются от обычных. Единственное отличие заключается в отсутствии необходимости указывать класс элемента при его объявлении в теле документа. Вот пример оформления текста с помощью обычного селектора, идентификатора и контекстного селектора:

```
<HTML>
 <HEAD>
  <TITLE>Title</TITLE>
  <STYLE TYPE="text/css">
     P.Red { color: red }
     .Red small { background: blue }
     #red { color: red }
    H1 EM {color: red}
  </STYLE>
 </HEAD>
 <BODY>
  <H1>Заголовок, стиль которого не определялся и <EM>красный элемент в нем
</EM></H1>
<P>Текст простого абзаца стиль которого не определялся и <EM>элемент</EM>
стандартного оформления, рядом с которым <Span id=RED>красный
элемент</Span>,
определенный с помощью идентификатора.
<P class=RED>Полностью красный абзац и <Small>внутри него  элемент SMALL с
определенным
фоном</small>
</BODY>
</HTML>
```

Контекстные селекторы, практически не отличаясь от обычных, ставят одни теги в зависимость от других и позволяют менять оформление больших фрагментов документа или целых страниц (если находятся в отдельном файле), изменением только одного тега. Например:

```
H1, H2, H3 {text-align:center}
Div.Blue {color: Navy; background: Aqua; Padding:0 10 10; border: solid Navy
thin}
Div.Red { color: Maroon; background: Fuchsia; Padding:0 10 10 ; border:
solid Maroon thin}
Div.Grin { color: Green; background: Lime; Padding:0 10 10 ; border: solid
Green thin}
.Blue A {color: Blue}
.Red A {color: Red}
.Grin A {color: Olive}
.Blue H1, .Blue H2, .Blue H3 {color: Blue; border: solid Navy; margin:0 -10}
.Red H1, .Red H2, .Red H3 {color: Red; border: solid black; margin:0 -10}
.Grin H1, .Grin H2, .Grin H3 {color: Olive; border: solid red; margin:0 -10}
body {text-align:justify; margin:30; background:white}
```

Теперь, в зависимости от выбранного типа раздела, будет переопределяться вид гиперссылок и заголовков.

Псевдоклассы и псевдоэлементы CSS

Псевдоклассы CSS

Псевдоклассами являются селекторы, указывающие на несуществующие теги. Псевдоклассами являются уже упомянутые типы гиперссылок:

```
A:link – обычная гиперссылка
A:visited – посещенная гиперссылка
A:hover – гиперссылка над которой находится мышь.
A:active – активизированная гиперссылка
```

Поскольку других псевдоклассов не существует, **CSS** разрешает указывать их, опуская тег A, поэтому A:Link означает то же что и :Link. Для получения контекстного значения псевдокласса, его можно сгруппировать с любым селектором по общему правилу:

```
.Blue A:visited {color: Blue}
.Blue :active {color: Blue}
```

Либо по особому правилу псевдокласса, опирающемуся на то, что в данном классе могут быть представлены только элементы A, и ни какой путаницы не возникает, даже если перед порождающим классом убрать точку, а после, убрать пробел:

```
Blue:active {color: Blue}
```

В оформлении вида гиперссылок важна последовательность. Поскольку Link означает любой тип ссылок, то ее настройка перекрывает все остальные, если те не настроены после нее. Visited входит в состав Link, по-этому, чтобы они имели отличный от Link вид, их следует настраивать после. Hover входит в состав Link, но может входить и в visited, если мышь находится над посещенной ссылкой. Для выделения hover из других, ее определение следует указывать после определения первых двух. Аналогично с active, которая, обычно, так же и hover, поскольку, если она активизирована щелчком мыши, указатель находится над ней. Ее следует определять последней

В примере, приведенном в разделе "Структура определения стилей" и скопированном из английского руководства, последовательность иная, так как для CSS 1 не имела значение последовательность определения стилей сылок. Эта последовательность приобрела значение с появлением CSS 2, где стили стали наследоваться Поскольку CSS 2 давно стала стандартом, во избежание недоразумений за последовательностью и наследованием стилей стоит следить.

Псевдоэлементы CSS

Псевдоэлементы являются элементы оформления, дополняющие основной тег. Они имеют тот же синтаксис, что и псевдоклассы, но влияют лишь на отображение части содержимого класса.

first-line – оформляет первую строку абзаца, позволяя определить свойства шрифта (font), цвета и заднего плана (color и background), интервал между словами (word-spacing), интервал между символами (letter-spacing), декоративное оформление текста (text-decoration), вертикальное выравнивание (vertical-align), трансформацию текста (text-transform), высоту строки (line-height) и параметры обтекания (clear).

first-letter – оформляет первый символ абзаца, позволяя определить свойства шрифта (font), цвета и заднего плана (color и background), декоративное оформление текста (text-decoration), вертикальное выравнивание (vertical-align) – только если свойство float установлено в "none" (по умолчанию), трансформацию текста (text-transform), высоту строки (line-height), параметры обтекания (clear), отступы (margin), расстояние от обрамления (padding), параметры обрамления (border) и отделение от остального текста (float).

Например:

```
<HTML>
 <HEAD>
  <TITLE>Title</TITLE>
  <STYLE TYPE="text/css">
    P { color: red; font-size: 12pt }
    P:first-letter { color: green; font-size: 200% }
    P:first-line { color: navy }
  </STYLE>
 </HEAD>
 <BODY>
<P> Отобразит первую строку абзаца темно-синим цветом, все первые
буквы абзаца – зеленым и в два раза большего размера. А остальную часть
абзаца – красным.
<P>Как в этом примере.
 </BODY>
</HTML>
```

Каскадирование таблиц стилей

Таблицы стилей могут импортировать содержимое других таблиц стилей, для чего в начало таблицы вставляются инструкции @import url которых указывает на расположение импортируемой таблицы. Например:

```
@import url('http://www.Discoverer.h11.ru/style.css')
```

Если импортируемая таблица содержит стиль, определенных в дальнейшем коде таблицы стилей или следующей импортируемой таблице, оформление стиля задается его последним определением. Если же, какое-то определение имеет преимущество, оно отмечается инструкцией ! important. Например:

```
P  {font-size: 12pt ! important; font-style: italic}
```

И будет переопределено, только если такое же преимущество имеет следующее определение. В данном случае, font-size переопределится, только если новое определение так же отмечено преимуществом, а font-style переопределится в любом случае, если новое определение содержит его указание.

Форматирование содержимого

Свойства шрифта

font-style – начертание шрифта: 'italic' – курсив, 'oblique' – наклонный (это одно и то же, но в некоторых шрифтах может по-разному называться) и normal – обычный (он же roman или upright).

font-variant – регистр символов: normal – обычный, small-caps – малые прописные (поддерживается только Internet Explorer)

font-weight – толщина шрифта: normal – обычный, lighter – более тонкий, чем определен для стиля, bold – жирный и bolder – жирнее, чем определен для стиля. Так же возможны варианты 100, 200, 300, 400, 500, 600, 700, 800, 900, где 400 соответствует нормальной толщине. Вообще, определение толщины шрифта – самое хаотичное место в **CSS** и его лучше отобразить таблицей:

"Regular" и "Book"	00
"Medium"	00
"Bold"	00
"Heavy"	00

При этом возможность менять степень жирности шрифта зависит от самого шрифта и, если в нем предусмотрен только обычный и полужирный, то никакого жирного и среднего из него не сделать.

font-size – размер шрифта в пунктах (pt), миллиметрах (mm), пикселях (px), относительных величинах (em) или процентах (%) от стандартного размера. Так же допустимы значения: xx-small и x-small – самые маленькие, small – маленький, medium – средний, large – большой, x-large и xx-large – самые большие; smaller – меньше указанного и larger – больше указанного. Например:

```
P {font-size: 12pt;}
BLOCKQUOTE {font-size: larger}
EM {font-size: 150%}
EM {font-size: 1.5em}
```

font-family – определяет перечень предпочитаемых для данного элемента шрифтов, перечисленных через запятую. Указание перечня имеет смысл для корректного подбора заменяющего шрифта, если предпочитаемый не найден. Например:

```
BODY {font-family: gill, helvetica, sans-serif}
```

Вместо настоящего шрифта может указываться тип: serif – шрифт с засечками типа Times, sans-serif – шрифт без засечек типа Helvetica или Arial, cursive – шрифт, имитирующий рукописный ввод типа Zapf-Chancery или Kursiv 95, fantasy – тип Western, monospace – моноширинный шрифт типа Courier. Если имя шрифта содержит пробел, оно должно заключатся в кавычки. Например:

```
<BODY STYLE="font-family: 'My own font', fantasy">
```

font – позволяет установить название шрифта, его размер и величину междустрочного интервала (через дробь) и другие параметры. Применяется вместо указания отдельных атрибутов. Например:

```
P {font: bold italic large/120% Helvetica, serif}
```

Компактность такой записи указывает на предпочтительность ее применения во всех случаях, где не может возникнуть разночтения значения Normal.

Цвета и рисунки

color – цвет символов. Указываться тремя способами:

1. В 16-ти цветной палитре: Black (Черный), Gray (Серый), Silver (Серебро – светло-серый), White (Белый), Maroon (Темно-бордовый), Red (Красный), Purple (Фиолетовый), Fuchsia (Фуксия – розовый), Green (Зеленый), Lime (Известь – салатный), Olive (Оливковый), Yellow (Желтый), Navy (Темно-синий), Blue (Откровенно-голубой О), Teal (Чирок – темный сине-зеленый), и Aqua (Аква – светлый сине-зеленый).

2. Взятым в кавычки 16-ти битным кодом цвета, например "#FF0000". Запись кода может быть сокращена до трех символов, обозреватели расширят его, удваивая каждый из них, превращая, например, "#F00" в "#FF0000".

3. Указанием, с помощью атрибута RGB, что цвет определяется соотношением красного, зеленого и синего и, указанием этого соотношения в скобках, например: rgb(255,0,0). Вместо абсолютной величины, при этом, может задаваться величина в процентах, где за 100% принято значение 255.

background-color – цвет фона. Может быть transparent – прозрачный или указываться так же, как цвет символов.

background-image – фоновый рисунок. Либо none – нет, либо url(адрес рисунка). Например:

```
BODY {background-image: url(marble.gif)}
P {background-image: none}
```

background-repeat – повторение фонового рисунка для заполнения страницы: repeat – допускается повторение, repeat-x – заполнение только по горизонтали, repeat-y – заполнение только по вертикали, no-repeat – нет повторения. Допускается опускание имени свойства, с указанием только его значения. Например:

```
BODY {background-image: url(marble.gif) no-repeat}
```

background-attachment – прокрутка фонового рисунка. Либо scroll – прокрутка разрешена, и рисунок перемещается вместе с текстом, либо fixed – рисунок расположен неподвижно и текст перемещается относительно рисунка. Допускается опускание имени свойства, с указанием только его значения. Например:

```
BODY {background-image: url(marble.gif) fixed}
```

background-position – позиция фонового рисунка.

1. Либо определяется в процентах от размера окна, отсчитываясь с левого верхнего угла, например: {10%,20%} где первая цифра указывает отступ слева, а вторая отступ сверху, либо в сантиметрах: {10cm,20cm}.

2. Либо фиксируется по краям или центру: top – по верху, bottom – по нижнему краю, left – влево, center – по центру, right – вправо.

Допускается комбинация определений и опускание имени свойства, указывая только его значение:

```
BODY {background-image: url(marble.gif) center center}
BODY {background-image: url(marble.gif) center 50%}
```

Background – указывает либо на рисунок, либо на цвет фона, либо и на то, и на другое одновременно. Применяется вместо background-image и background-color.

Свойства текста

word-spacing – интервал между словами: либо normal, либо относительный интервал в em, например:

```
H1 {word-spacing: 1em}
```

Увеличит интервал между словами на единицу. Допустимы отрицательные значения. Значение по умолчанию – 0.

letter-spacing – межсимвольный интервал: либо normal, либо относительный интервал в em, например:

```
H1 {letter-spacing: 0.1em}
```

Увеличит интервал между символами на одну десятую. Допустимы отрицательные значения. Значение по умолчанию – 0.

text-decoration – оформление текста: none – нет, underline – подчеркиванием, overline – надчеркиванием, line-through – перечеркиванием, blink – мерцанием текста. Данное свойство работает не везде. IE, например, отказывается мерцать, Opera, если станица состоит из Frames (кадров, рамок, фреймов), мерцает через раз, то есть, она либо показывает текст, не мерцая, либо вообще не выводит его на экран – как придется. Поэтому, чтобы избежать ситуации, в которой потребитель вообще ничего не увидит, лучше избегать использования Blink.

vertical-align – определяет вертикальное выравнивание встроенных элементов: baseline – по опорной линии (нижнюю часть по нижней части родительского элемента), sub – нижний индекс, super – верхний индекс, top – выравнивание по верху, text-top – по верху текста, middle – посредине, bottom – понизу, text-bottom – по низу текста или в процентах от высоты строки.

text-transform – преобразование текста: capitalize – все слова с большой буквы, uppercase – все прописные, lowercase – все строчные, none – нет преобразования.

text-align – выравнивание текста: left – влево, right – вправо, center – по центру, justify – по обоим краям.

text-indent – отступ первой строки, указывается в пунктах, символах или процентах. При отрицательном значении означает выступ.

line-height – высота строки (междустрочный интервал): либо normal – обычный (от 1 до 1.2), либо пиксели, проценты, относительный размер (em).

Оформление абзацев

Отступы задаются в процентах, пунктах и относительных величинах (em). Значение AUTO, определяет отступ по умолчанию.

margin-top – верхний отступ.

margin-right – правый отступ.

margin-bottom – нижний отступ.

margin-left – левый отступ.

margin – определяет значение всех отступов в порядке верх, право, низ, лево (с верху, по часовой стрелке). Если указано только три значения, левый отступ равен правому, если только два, левый равен правому, а нижний верхнему; если один – все отступы одинаковы.

border-top-width – верхняя окантовка абзаца: thin – тонкая, medium – средняя, thick – толстая, либо точная величина в пикселях.

border-right-width – правая окантовка абзаца: thin – тонкая, medium – средняя, thick – толстая, либо точная величина в пикселях.

border-bottom-width – нижняя окантовка абзаца: thin – тонкая, medium – средняя, thick – толстая, либо точная величина в пикселях.

border-left-width – левая окантовка абзаца: thin – тонкая, medium – средняя, thick – толстая, либо точная величина в пикселях.

border-width – окантовка абзаца: thin – тонкая, medium – средняя, thick – толстая, либо точная величина в пикселях. Значения указываются в том же порядке: верх, право, низ, лево (с верху, по часовой стрелке). Если указано только три значения, левая рамка равна правой, если только два, левая равна правой, а нижняя верхней; если одно – все рамки одинаковы.

border-color – определяет цвет окантовки, по тому же принципу: с верху, по часовой стрелке.

border-style – стиль окантовки: none – нет окантовки (по умолчанию), dotted – точки, dashed – пунктир, solid – непрерывный, double – двойная линия, groove – канавка, ridge – выступ, inset – углубление блока, outset – возвышающаяся табличка. Стиль определяется для всех четырех сторон. Поскольку по умолчанию стиль не назначен, пока он не будет указан, рамка не отображается, несмотря на то, что может быть задан ее цвет или толщина.

border-top– позволяет в одной команде установить стиль, ширину и цвет верхней окантовки.

border-right – позволяет в одной команде установить стиль, ширину и цвет правой окантовки.

border-bottom – позволяет в одной команде установить стиль, ширину и цвет нижней окантовки.

border-left – позволяет в одной команде установить стиль, ширину и цвет левой окантовки.

Border – позволяет в одной команде установить стиль, ширину и цвет всей окантовки. Например:

```
border: solid 0.5em
```

или

```
border: solid thick blue
```

padding-top – верхний отступ окантовки от текста. Отрицательные значения не принимаются и уменьшение возможно лишь за счет уменьшения высоты строки (line-height).

padding-right – правый отступ окантовки от текста.

padding-bottom – нижний отступ окантовки от текста. Отрицательные значения не принимаются и уменьшение возможно лишь за счет уменьшения высоты строки (line-height).

padding-left – левый отступ окантовки от текста.

Padding – определяет значение всех отступов окантовки в порядке: верх, право, низ, лево (с верху, по часовой стрелке). Если указано только три значения, левый отступ равен

правому, если только два, левый равен правому, а нижний верхнему; если один – все отступы от окантовки одинаковы.

Атрибуты вставляемых элементов

width – ширина в процентах или пикселях, применяется в таблицах, разделителях (HR), а так же, для резервирования места рисункам.

height – высота.

Float – определяет положение нетекстовых элементов none – запрещает отделение от текста, left – выравнивает влево, right – выравнивает вправо.

Clear – определяет отделение нетекстовых элементов от текста: none – нет отделения, текст располагается рядом с объектом, left – текст располагается под объектом, слева, right – текст под объектом, справа, both – текст под объектом с любой из сторон. Например:

```
P {Clear:left}
```

Классификационные реквизиты

Эти реквизиты позволяют переопределять тип тега.

Display – определяет тип отображения контейнера (в терминах HTML, многие теги являются блоковыми или текстовыми контейнерами): block – отображать как блоковый контейнер, inline – отображать как текстовый контейнер, list-item – отображать как элемент списка, none – не является контейнером (не отображается). Например, по умолчанию:

```
P {display: block}
EM {display: inline}
LI {display: list-item}
IMG {display: none}
```

Блоковые контейнеры, должны иметь закрывающий тег, либо закрываются автоматически, с объявлением следующего такого же контейнера. Блоковые элементы, кроме SPAN инициируют переход на новую строку. Текстовые (линейные, с точки зрения **CSS**), контейнеры не инициируют переход на новую строку и не закрываются автоматически при новом объявлении такого же контейнера. Элементы списка являются подчиненными контейнерами.

Переопределение типа контейнера должно изменить поведение элемента, но, судя по всему, не поддерживается браузерами.

white-space – определяет обработку пустого пространства (пробелы и табуляция): normal – стандартное сокращение всех пробелов и табуляторов, pre – сохранение пробелов и табуляторов, а так же переносов строк, как в PRE-элементах, nowrap – переносить строки внутри абзаца только при указании тега BR, но сокращать пробелы и табуляторы.

list-style-type – определяет нумерацию списков (элемент UL и OL) disc – жирными точками, circle – кружками, square – квадратиками, decimal – арабскими цифрами, lower-roman – маленькие римские цифры upper-roman – большие римские цифры, lower-alpha – маленькие английские буквы, upper-alpha – большие английские буквы, none – нет нумерации.

list-style-image – определяет рисунок для маркировки списка: none – список не маркируется, иначе url(), в скобках которого указывается адрес рисунка. Например:

```
UL {list-style-image: url("http://Disciverer.h11.ru/Smallogo.gif")}
```

list-style-position – размещение маркеров списковых элементов: inside – внутри текста, outside – выступающими из текста.

list-style – одной командой определяет list-style-type, list-style-image и list-style-position. Например:

```
UL {list-style: upper-roman inside}
```

Примечание

1em – значение по умолчанию (например, Font-size:2em – двойной размер шрифта)

***** аналогично em – относительный размер

inches – дюймы, 1in = 2,54 см.

millimeters – миллиметры, mm.

Points – пункты, 1pt = 1/72 дюйма

Picas – пикасы?, 1pc = 12pt

Pixels – пиксели, 1px зависит от размера экрана и его параметров.

Относительный размер может указываться как +число или -число, что означает изменение основного размера на эту величину.

Символ косой черты влево (\) отключает стандартное значение следующего за ним символа и может быть использован, например, для включения кавычки в текст **CSS**.

Атрибуту определяющие сразу несколько свойств в некоторых случаях могут не работать. Кроме того, одни и те же атрибуты могут поддерживаться одними и не поддерживаться другими обозревателями.

Контрольные вопросы

1. Что представляют из себя каскадные таблицы стилей?
2. Назначение идентификаторов.
3. Какую функцию выполняют селекторы?
4. Опишите псевдоклассы.
5. Какие особенности имеет каскадирование таблиц стилей?

Лекция 3: Размещение JavaScript на HTML-странице

Код скрипта JavaScript размещается непосредственно на HTML-странице. Чтобы увидеть, как делается рассмотрим следующий простой пример:

```
<html>
<body>
<br>
Это обычный HTML документ.
<br>
 <script language="JavaScript">
  document.write("А это JavaScript!")
 </script>
<br>
Вновь документ HTML.
</body>
</html>
```

С первого взгляда пример напоминает обычный файл HTML. Единственное новшество здесь - конструкция:

```
<script language="JavaScript">
 document.write("А это JavaScript!")
</script>
```

Это действительно код JavaScript. Чтобы видеть, как этот скрипт работает, запишите данный пример как обычный файл HTML и загрузите его в браузер, имеющий поддерку языка JavaScript. В результате Вы получите 3 строки текста:

Это обычный HTML документ.
А это JavaScript!
Вновь документ HTML.

Все, что стоит между тэгами *<script>* и *</script>,* интерпретируется как код на языке JavaScript. Здесь также виден пример использования инструкции document.write() - одной из наиболее важных команд, используемых при программировании на языке JavaScript. Команда document.write() используется, когда необходимо что-либо написать в текущем документе (в данном случае таком является наш HTML-документ). Так наша небольшая программа на JavaScript в HTML-документе пишет фразу "*А это JavaScript!"*.

Браузеры без поддержки JavaScript

А как будет выглядеть наша страница, если браузер не воспринимает JavaScript? Браузеры, не имеющие поддержки JavaScript, "не знают" и тэга <*script*>. Они игнорируют его и печатают все стоящие вслед за ним коды как обычный текст. Иными словами, читатель увидит, как код JavaScript, приведенный в нашей программе, окажется вписан открытым текстом прямо посреди HTML-документа. Разумеется, это не входило в наши намерения. На этот случай имеется специальный способ скрыть исходный код скрипта от старых версий браузеров - мы будем использовать для этого тэг комментария из HTML - <! - - >. В результате новый вариант нашего исходного кода будет выглядеть как:

```
<html>
<body>
<br>
Это обычный HTML документ.
<br>
 <script language="JavaScript">
 <!-- скрывает код от старых браузеров

   document.write("А это JavaScript!")

 // -->
 </script>
<br>
Вновь документ HTML.
</body>
</html>
```

В этом случае браузер без поддержки JavaScript будет печатать:

Это обычный HTML документ.
Вновь документ HTML.

А без HTML-тэга комментария браузер без поддержки JavaScript напечатал бы:

Это обычный HTML документ.
document.write("А это JavaScript!")
Вновь документ HTML.

Пожалуйста обратите внимание, что Вы не можете полностью скрыть исходный код JavaScript. То, что мы здесь делаете, имеет целью предотвратить распечатку кода скрипта на старых браузерах - однако тем не менее читатель сможет увидеть этом код посредством пункта меню *'View document source'*. Не существует также способа скрыть что-либо от просмотра в вашем исходном коде (и увидеть, как выполнен тот или иной трюк).

События

События и обработчики событий являются очень важной частью для программирования на языке JavaScript. События, главным образом, инициируются теми или иными действиями пользователя. Если он щелкает по некоторой кнопке, происходит событие "Click". Если указатель мыши пересекает какую-либо ссылку гипертекста - происходит событие MouseOver.

Существует несколько различных типов событий. Мы можем заставить нашу JavaScript-программу реагировать на некоторые из них. И это может быть выполнено с помощью специальных программ обработки событий. Так, в результате щелчка по кнопке может создаваться выпадающее окно. Это означает, что создание окна должно быть реакцией на событие щелка - Click. Программа - обработчик событий, которую мы должны использовать в данном случае, называется onClick. И она сообщает компьютеру, что нужно делать, если произойдет данное событие. Приведенный ниже код представляет простой пример программы обработки события onClick:

```
<form>
<input type="button" value="Click me" onClick="alert('Yo')">
</form>
```

Данный пример имеет несколько новых особенностей - рассмотрим их по порядку. Вы можете здесь видеть, что мы создаем некую форму с кнопкой (как это делать - проблема языка HTML, так что рассматривать это здесь я не буду). Первая новая особенность - *onClick="alert('Yo')"* в тэге *<input>*. Как мы уже говорили, этот атрибут определяет, что происходит, когда нажимают на кнопку. Таким образом, если имеет место событие Click, компьютере должен выполнить вызов *alert('Yo')*. Это и есть пример кода на языке JavaScript (Обратите внимание, что в этом случае мы даже не пользуемся тэгом *<script>*). Функция *alert()* позволяет Вам создавать выпадающие окна. При ее вызове Вы должны в скобках задать некую строку. В нашем случае это 'Yo'. И это как раз будет тот текст, что появится в выпадающем окне. Таким образом, когда читатель когда щелкает на кнопке, наш скрипт создает окно, содержащее текст 'Yo'.

Некоторое замешательство может вызвать еще одна особенность данного примера: в команде document.write () мы использовали двойные кавычки ("), а в конструкции *alert()* - только одинарные. Почему? В большинстве случаев Вы можете использовать оба типа кавычек. Однако в последнем примере мы написали *onClick="alert('Yo')"* - то есть мы использовали и двойные, и одинарные кавычки. Если бы мы написали *onClick="alert("Yo")"*, то компьютер не смог бы разобраться в нашем скрипте, поскольку становится неясно, к которой из частей конструкции имеет отношение функция обработки событий onClick, а к которой - нет. Поэтому Вы и вынуждены в данном случае перемежать оба типа кавычек. Не имеет значения, в каком порядке Вы использовали кавычки - сперва двойные, а затем одинарные или наоборот. То есть Вы можете точно так же написать и *onClick='alert("Yo")'*.

Вы можете использовать в скрипте множество различных типов функций обработки событий. Сведения о некоторых из них мы получим в данном описании, однако не о всех. Поэтому обращайтесь пожалуйста к соответствующему справочнику, если Вы хотите узнать, какие обработчики событий еще существуют.

Итак, если Вы используете браузер Netscape Navigator, то выпадающее окно содержит текст, что был передан функции JavaScript *alert*. Такое ограничение накладывается по соображениям безопасности. Такое же выпадающее окно Вы можете создать и с помощью метода *prompt()*. Однако в этом случае окно будет воспроизводить текст, введенный читателем. А потому, скрипт, написанный злоумышленником, может принять вид системного сообщения и попросить читателя ввести некий пароль. А если текст помещается в выпадающее окно, то тем самым читателю дается понять, что данное окно было создано web-браузером, а не вашей операционной системой. И поскольку данное ограничение наложено по соображениям безопасности, Вы не можете взять и просто так удалить появившееся сообщение.

Функции

В большинстве наших программ на языке JavaScript мы будем пользоваться функциями. Поэтому уже теперь мне необходимо рассказать об этом важном элементе языка. В большинстве случаев функции представляют собой лишь способ связать вместе нескольких команд. Давайте, к примеру, напишем скрипт, печатающий некий текст три раза подряд. Для начала рассмотрим простой подход:

```
<html>
<script language="JavaScript">
<!-- hide

document.write("Добро пожаловать на мою страницу!<br>");
document.write("Это JavaScript!<br>");

document.write("Добро пожаловать на мою страницу!<br>");
document.write(Это JavaScript!<br>");

document.write("Добро пожаловать на мою страницу!<br>");
document.write("Это JavaScript!<br>");

// -->
</script>
</html>
```

И такой скрипт напишет следующий текст
Добро пожаловать на мою страницу!
Это JavaScript!

три раза. Если посмотреть на исходный код скрипта, то видно, что для получения необходимого результата определенная часть его кода была повторена три раза. Разве это эффективно? Нет, мы можем решить ту же задачу еще лучше. Как насчет такого скрипта для решения той же самой задачи?:

```
<html>
<script language="JavaScript">
<!-- hide

function myFunction() {
  document.write("Добро пожаловать на мою страницу!<br>");
  document.write("Это JavaScript!<br>");
}

myFunction();
myFunction();
myFunction();

// -->
</script>
</html>
```

В этом скрипте мы определили некую функцию, состоящую из следующих строк:

```
function myFunction() {
  document.write("Добро пожаловать на мою страницу!<br>");
  document.write("Это JavaScript!<br>");
}
```

Все команды скрипта, что находятся внутри фигурных скобок - {} - принадлежат функции myFunction (). Это означает, что обе команды document.write() теперь связаны воедино и могут быть выполнены при вызове указанной функции. И действительно, нашем примере есть три вызова этой функции - можно увидеть, что мы написали строку myFunction() три раза сразу после того, как дали определение самой функции. То естькак раз и сделали три

вызова. В свою очередь, это означает, что содержимое этой функции (команды, указанные в фигурных скобках) было выполнено трижды. Поскольку это довольно простой пример использования функции, то у Вас мог возникнуть вопрос, а почему собственно эти функции столь важны в JavaScript. По прочтении данного описания Вы конечно же поймете их пользу. Именно возможность передачи переменных при вызове функции придает нашим скриптам подлинную гибкость - что это такое, мы увидим позже.

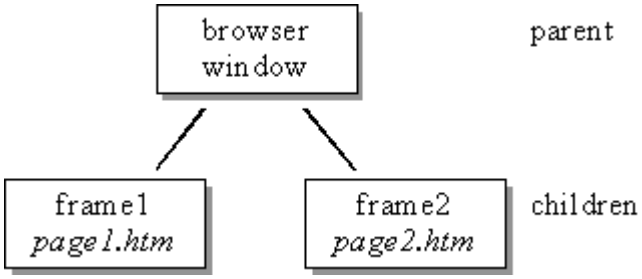
Функции могут также использоваться в совместно с процедурами обработки событий. Рассмотрим следующий пример:

```
<html>
<head>

<script language="JavaScript">
<!-- hide

function calculation() {
  var x= 12;
  var y= 5;

  var result= x + y;

  alert(result);
}

// -->
</script>

</head>
<body>

<form>
<input type="button" value="Calculate" onClick="calculation()">
</form>

</body>
</html>
```

Здесь при нажатии на кнопку осуществляется вызов функции *calculation()*. Как можно заметить, эта функция выполняет некие вычисления, пользуясь переменными x, y и result. Переменную мы можем определить с помощью ключевого слова *var*. Переменные могут использоваться для хранения различных величин - чисел, строк текста и т.д. Так строка скрипта *var result= x + y;* сообщает браузеру о том, что необходимо создать переменную result и поместить туда результат выполнения арифметической операции x + y (т.е. 5 + 12). После этого в переменный result будет размещено число 17. В данном случае команда *alert(result)* выполняет то же самое, что и *alert(17)*. Иными словами, мы получаем выпадающее окно, в котором написано число 17.

Лекция 4: Фреймы и JavaScript

Рассмотрим, как JavaScript "видит" фреймы, присутствующие в окне браузера. Для этой цели создадим два фрейма. Как мы уже видели, JavaScript организует все элементы, представленные на web-странице, в виде некой иерархической структуры. То же самое относится и к фреймам. На следующем рисунке показана иерархия объектов, представленных в первом примере:



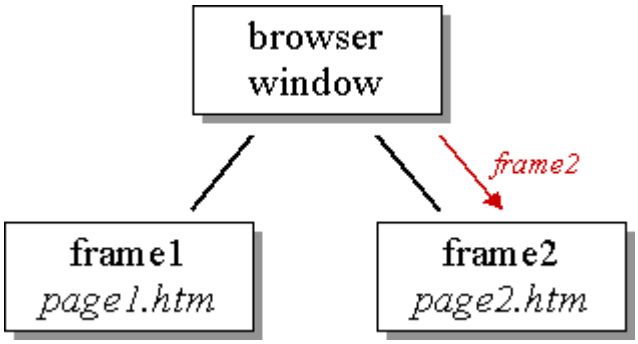
В вершине иерархии находится окно браузера (browser window). В данном случае он разбито на два фрейма. Таким образом, окно, как объект, является родоначальником, родителем данной иерархии (parent), а два фрейма - соответственно, его потомки (children). Мы присвоили этим двум фреймам уникальные имена - frame1 и frame2. И с помощью этих имен мы можем обмениваться информацией с двумя указанными фреймами.

С помощью скрипта можно решить следующую задачу: допустим посетитель активирует некую ссылку в первом фрейме, однако соответствующая страница должна загружаться не в этот же фрейм, а в другой. Примером такой задачи может служить составление меню (или навигационных панелей), где один фрейм всегда остается неизменным, но предлагает посетителю несколько различных ссылок для дальнейшего изучения данного сайта. Чтобы решить эту задачу, мы должны рассмотреть на три случая:

- *главное окно/фрейм получает доступ к фрейму-потомку*
- *фрейм-потомок получает доступ к родительскому окну/фрейму*
- *фрейм-потомок получает доступ к другому фрейму-потомку*

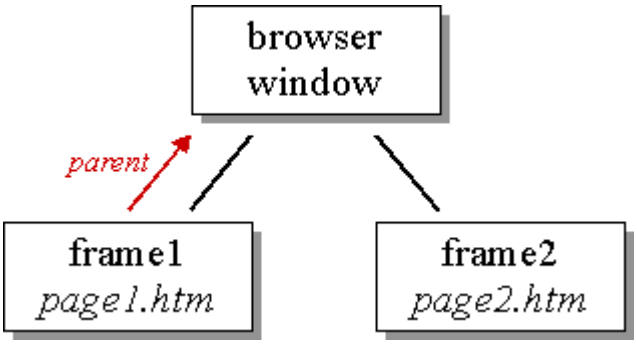
С точки зрения объекта “окно” (window) два указанных фрейма называются *frame1* и *frame2*. Как можно видеть на предыдущем рисунке, существует прямая взаимосвязь между родительским окном и каждым фреймом. Так образом, если Вы пишете скрипт для родительского окна - то есть для страницы, создающей эти фреймы - то можете обращаться к этим фреймам, просто называя их по имени. Например, можно написать:

frame2.document.write("Это сообщение передано от родительского окна.");



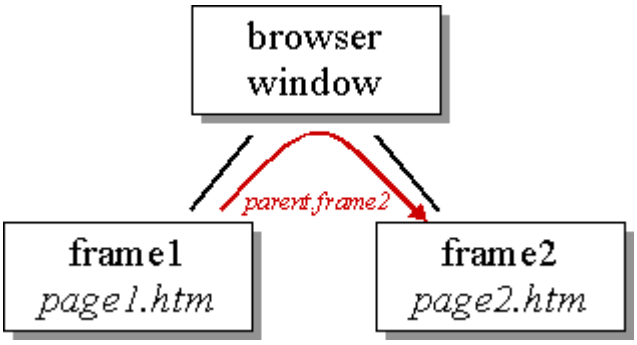
В некоторых случаях Вам понадобится, находясь во фрейме, получать доступу к родительскому окну. Например, это бывает необходимо, если Вы хотите при следующем переходе избавиться от фреймов. В таком случае удаление фреймов означает лишь загрузку новой страницы вместо содержавшей фреймы. В нашем случае это загрузка страницы в родительское окно. Сделать это нам поможет доступ к родительскому окну (или родительскому фрейму) из фреймов, являющихся его потомками. Чтобы загрузить новый документ, мы должны внести в *location.href* новый адрес URL. Поскольку мы хотим избавиться от фреймов, следует использовать объект *location* из родительского окна. (Напомним, что в каждый фрейм можно загрузить собственную страницу, то мы имеем для каждого фрейма собственный объект *location*). Итак, мы можем загрузить новую страницу в родительское окно с помощью команды:

parent.location.href= "http://...";



И наконец, очень часто Вам придется решать задачу обеспечения доступа с одного фрейма-потомка к другому такому же фрейму-потомку. Итак, как можно, находясь в первом фрейме, записать что-либо во второй - то есть, которой командой следует воспользоваться на HTML-странице page1.htm? Как можно увидеть на нашем рисунке, между двумя этими фреймами нет никакой прямой связи. И потому мы не можем просто так вызвать *frame2*, находясь в фрейме *frame1*, который просто ничего не знает о существовании второго фрейма. С точки же зрения родительского окна второй фрейм действительно существует и называется *frame2*, а к самому родительскому окну можно обратиться из первого фрейма по имени *parent*. Таким образом, чтобы получить доступ к объекту document, разместившемуся во втором фрейме, мы должны написать следующее,:

parent.frame2.document.write("Привет, это вызов из первого фрейма.");



Навигационные панели

Давайте рассмотрим, как создаются навигационные панели. В одном фрейме мы создаем несколько ссылок. Однако, если посетитель активирует какую-либо из них, соответствующая страница будет помещена не в тот же самый фрейм, а в соседний. Сперва нам необходимо написать скрипт, создающий указанные фреймы. Такой документ выглядит точно так же, как и тот, что мы рассматривали ранее в этой части описания:

frames3.htm

```
<html>
<frameset rows="80%,20%">
  <frame src="start.htm" name="main">
  <frame src="menu.htm" name="menu">
</frameset>
</html>
```

Здесь *start.htm* - это та страница, которая первоначально будет показана в главном фрейме (*main*). У нас нет никаких специальных требований к содержимому этой страницы. Следующая web-страница будет загружена во фрейм "*menu*":

menu.htm

```
<html>
<head>
<script language="JavaScript">
<!-- hide

function load(url) {
  parent.main.location.href= url;
}

// -->
</script>
</head>
<body>

<a href="javascript:load('first.htm')">first</a>
<a href="second.htm" target="main">second</a>
<a href="third.htm" target="_top">third</a>

</body>
</html>
```

Здесь Вы можете увидеть несколько способов загрузки новой страницы во фрейм *main*. В первой ссылке для этой цели используется функция *load()*. Давайте посмотрим, как это делается:

```
<a href="javascript:load('first.htm')">first</a>
```

Как Вы можете видеть, вместо явной загрузки новой страницы мы предлагаем браузеру выполнить некую команду на языка JavaScript - для этого мы всего лишь должны воспользоваться параметром javascript вместо обычного href. Далее, внутри скобок можно увидеть 'first.htm'. Эту строка передается в качестве аргумента функции *load()*. Сама же функция *load()* определяется следующим образом:

```
function load(url) {
  parent.main.location.href= url;
}
```

Здесь Вы можете увидеть, что внутри скобок написано url. Это означает, что в нашем примере строка 'first1.htm' при вызове функции заносится в переменную url. И эту новую переменную теперь можно использовать при работе внутри функций *load()*. Позже мы познакомимся с другими примерами использования важной концепции переменных.

Во второй ссылке присутствует параметр target. На самом деле это уже не относится к JavaScript. Это одна из конструкций языка HTML. Как видно, мы всего лишь указываем имя необходимого фрейма. Заметим, что в этом случае мы не обязаны ставить перед именем указанного фрейма слово *parent*, что, честно говоря, несколько смущает. Причина такого отступления от правил кроется в том, что параметр *target* - это функция языка HTML, а не JavaScript. И на примере третьей ссылки Вы можете видеть, как с помощью *target* можно избавиться от фреймов.

А если Вы хотите избавиться от фреймов с помощью функции *load()*, то Вам необходимо написать в ней лишь *parent.location.href= url*.

Итак, который способ Вам следует выбрать? Это зависит от вашего скрипта и от того, что собственно Вы хотите сделать. Параметр *target* использовать очень просто. Вы можете воспользоваться им, если хотите всего лишь загрузить новую страницу в другой фрейм. Решение на основе языка JavaScript (примером этого служит первая ссылка) обычно используется, если Вы хотите, чтобы при активизации ссылки произошло несколько вещей. Одна из наиболее часто возникающих проблем такого рода состоит в том, чтобы разом загрузить две страницы в два различных фрейма. И хотя Вы могли бы решить эту задачи с помощью параметра *target*, использование функции JavaScript в этом случае более предпочтительно. Предположим, мы имеем три фрейма с именами *frame1, frame2* и *frame3*. Допустим посетитель активирует ссылку в *frame1*. И мы хотим, чтобы при этом в два других фрейма загружались две различные web-страницы. В качестве решения этой задачи Вы можете, например, воспользоваться функцией:

```
function loadtwo() {
  parent.frame1.location.href= "first.htm";
  parent.frame2.location.href= "second.htm";
}
```

Если же Вы хотите сделать функцию более гибкой, то можете воспользоваться возможностью передачи переменной в качестве аргумента. Результат будет выглядеть как:

```
function loadtwo(url1, url2) {
  parent.frame1.location.href= url1;
  parent.frame2.location.href= url2;
}
```

После этого можно организовать вызов функции: *loadtwo("first.htm", "second.htm")* или *loadtwo("third.htm", "forth.htm")*. Очевидно, передача аргументов делает вашу функцию более гибкой. В результате Вы можете использовать ее многократно и в различных контекстах.

Контрольные вопросы

1. Допустимо ли следующее использование элемента FRAME?

```
<HTML>
<FRAME SRC="main.html">
<FRAMESET ROW="20%, *">
 <FRAME SRC="frame1.html">
 <FRAME SRC="frame2.html">
</FRAMESET>
</HTML>
```

- данный код соответствует стандартам HTML
- использовать данный код допустимо
- данный код не соответствует стандартам HTML

2. Какие атрибуты тэга FRAME вы можете привести?
3. В каком примере FRAMESET делит экран на два горизонтальных фрейма?
   - <FRAMESET rows="50%, 50%">
   - <FRAMESET cols="50%, 50%">
   - <FRAMESET cols="50%, *">
   - <FRAMESET rows="50%, *">
4. В каких примерах определены таблицы фреймов из 2 строк и 3 столбцов?
   - <FRAMESET rows="50%,50%" cols="50%,50%">
   - <FRAMESET rows="25%,65%,15%" cols="50%,50%">
   - <FRAMESET rows="25%,75%" cols="33%,34%,33%">
   - <FRAMESET rows="30%,70%" cols="33%,34%,33%">
5. В каком случае возможность прокрутки окна фрейма будет доступна всегда?
   - <FRAME scrolling="yes">

- <FRAME scrolling="no">
- <FRAME scrolling="auto">

6. При задании какого атрибута браузер будет рисовать разделитель между этим фреймом и каждым смежным фреймом?

- <FRAME frameborder="1">
- <FRAME frameborder="0">
- <FRAME border="1">

Лекция 5: Окна и динамически создаваемые документы

Создание окон

Открытие новых окон в браузере - грандиозная возможность языка JavaScript. Вы можете либо загружать в новое окно новые документы (например, те же документы HTML), либо (динамически) создавать новые материалы. Посмотрим сначала, как можно открыть новое окно, потом как загрузить в это окно HTML-страницу и, наконец, как его закрыть. Приводимый далее скрипт открывает новое окно браузера и загружает в него некую web-страничку:

```
<html>
<head>
<script language="JavaScript">
<!-- hide

function openWin() {
  myWin= open("bla.htm");
}

// -->
</script>
</head>
<body>

<form>
<input type="button" value="Открыть новое окно" onClick="openWin()">
</form>

</body>
</html>
```

В представленном примере в новое окно с помощью метода *open()* записывается страница *bla.htm*.

Заметим, что Вы имеете возможность управлять самим процессом создания окна. Например, Вы можете указать, должно ли новое окно иметь строку статуса, панель инструментов или меню. Кроме того Вы можете задать размер окна. Например, в следующем скрипте открывается новое окно размером 400x300 пикселов. Оно не имеет ни строки статуса, ни панели инструментов, ни меню.

```
<html>
```

```
<head>
<script language="JavaScript">
<!-- hide

function openWin2() {
  myWin= open("bla.htm", "displayWindow",
    "width=400,height=300,status=no,toolbar=no,menubar=no");
}

// -->
</script>
</head>
<body>

<form>
<input type="button" value=" Открыть новое окно" onClick="openWin2()">
</form>

</body>
</html>
```

(online-версия руководства позволит Вам проверить этот скрипт немедленно)

Как видите, свойства окна мы формулируем в строке "width=400,height=300,status=no,toolbar=no,menubar=no". Обратите внимание также и на то, что Вам не следует помещать в этой строке символы пробела!

Список свойств окна, которыми Вы можете управлять:

directories	yes|no
height	*количество пикселов*
location	yes|no
menubar	yes|no
resizable	yes|no
scrollbars	yes|no
status	yes|no
toolbar	yes|no
width	*количество пикселов*

В версии 1.2 языка JavaScript были добавлены некоторые новые свойства (то есть в Netscape Navigator 4.0). Вам не следует пользоваться этими свойствами, готовя материалы для Netscape 2.x, 3.x или Microsoft Internet Explorer 3.x, поскольку эти браузеры не понимают языка 1.2 JavaScript. Новые свойства окон:

alwaysLowered	yes|no
alwaysRaised	yes|no
dependent	yes|no
hotkeys	yes|no
innerWidth	*количество пикселов* (заменяет width)
innerHeight	*количество пикселов* (заменяет height)
outerWidth	*количество пикселов*

outerHeight	*количество пикселов*
screenX	*количество пикселов*
screenY	*количество пикселов*
titlebar	yes|no
z-lock	yes|no

Вы можете найти толкование этих свойств в описании языка JavaScript 1.2

Например, теперь с помощью этих свойств Вы можете определить, в каком месте экрана должно находиться вновь открываемое окно. Работая со старой версией языка JavaScript, Вы не смогли бы этого сделать.

Имя окна

Как видите, открывая окна, мы должны использовать три аргумента:

```
myWin= open("bla.htm", "displayWindow",
  "width=400,height=300,status=no,toolbar=no,menubar=no");
```

А для чего нужен второй аргумент? Это имя окна. Ранее мы видели, как оно использовалось в параметре target. Так, если Вы знаете имя окна, то можете загрузить туда новую страницу с помощью записи

```
<a href="bla.html" target="displayWindow">
```

При этом Вам необходимо указать имя соответствующего окна (если же такого окна не существует, то с этим именем будет создано новое). Обратите внимание, что *myWin* - это вовсе не имя окна. Но только с помощью этой переменной Вы можете получить доступ к окну. И поскольку это обычная переменная, то область ее действия - лишь тот скрипт, в котором она определена. А между тем, имя окна (в данном случае это *displayWindow*) - уникальный идентификатор, которым можно пользоваться с любого из окон браузера.

Создание окон

Вы можете также закрывать окна с помощью языка JavaScript. Чтобы сделать это, Вам понадобится метод close(). Давайте, как было показано ранее, откроем новое окно. И загрузим туда очередную страницу:

```
<html>
<script language="JavaScript">
<!-- hide

function closeIt() {
  close();
}

// -->
</script>

<center>
<form>
<input type=button value="Close it" onClick="closeIt()">
```

```
</form>
</center>

</html>
```

Если теперь в новом окне Вы нажмете кнопку, то оно будет закрыто. *open()* и *close()* - это методы объекта window. Мы должны помнить, что следует писать не просто *open()* и *close(),* а *window.open()* и *window.close()*. Однако в нашем случае объект window можно опустить - Вам нет необходимости писать префикс *window*, если Вы хотите всего лишь вызвать один из методов этого объекта (и такое возможно только для этого объекта).

Динамическое создание документов

Теперь мы готовы к рассмотрению такой замечательной возможности JavaScript , как динамическое создание документов. То есть Вы можете разрешить Вашему скрипту на языке JavaScript самому создавать новые HTML-страницы. Более того, Вы можете таким же образом создавать и другие документы Web, такие как VRML-сцены и т.д. Для удобства Вы можете размещать эти документы в отдельном окне или фрейме.

Для начала мы создадим простой HTML-документ, который покажем в новом окне. Рассмотрим следующий скрипт.

```
<html>
<head>
<script language="JavaScript">
<!-- hide

function openWin3() {
  myWin= open("", "displayWindow",
    "width=500,height=400,status=yes,toolbar=yes,menubar=yes");

  // открыть объект document для последующей печати
  myWin.document.open();

  // генерировать новый документ
  myWin.document.write("<html><head><title>On-the-fly");
  myWin.document.write("</title></head><body>");
  myWin.document.write("<center><font size=+3>");
  myWin.document.write("Данный документ HTML был создан ");
  myWin.document.write("с помощью JavaScript!");
  myWin.document.write("</font></center>");
  myWin.document.write("</body></html>");

  // закрыть документ - (но не окно!)
  myWin.document.close();
}

// -->
</script>
</head>
<body>

<form>
```

```
<input type=button value="On-the-fly" onClick="openWin3()">
</form>

</body>
</html>
```

Рассмотрим функцию *winOpen3 ()*. Очевидно, мы сначала открываем новое окно браузера. Поскольку первый аргумент функции *open()* - пустая строка (*""*), то это значит, что мы не желаем в данном случае указывать конкретный адрес URL. Браузер должен только не обработать имеющийся документ - JavaScript обязан создать дополнительно новый документ.

В скрипте мы определяем переменную myWin. И с ее помощью можем получать доступ к новому окну. Обратите пожалуйста внимание, что в данном случае мы не можем воспользоваться для этой цели именем окна (*displayWindow*). После того, как мы открыли окно, наступает очередь открыть для записи объект document. Делается это с помощью команды:

```
// открыть объект document для последующей печати   myWin.document.open();
```

Здесь мы обращаемся к *open()* - методу объекта *document.* Однако это совсем не то же самое, что метод *open()* объекта window! Эта команда не открывает нового окна - она лишь готовит *document* к предстоящей печати. Кроме того, мы должны поставить перед *document.open()* приставку *myWin*, чтобы получить возможность писать в новом окне.

В последующих строках скрипта с помощью вызова *document.write()* формируется текст нового документа:

```
// генерировать новый документ
myWin.document.write("<html><head><title>On-the-fly");
myWin.document.write("</title></head><body>");
myWin.document.write("<center><font size=+3>");
myWin.document.write("This HTML-document has been created ");
myWin.document.write("with the help of JavaScript!");
myWin.document.write("</font></center>");
myWin.document.write("</body></html>");
```

Как видно, здесь мы записываем в документ обычные тэги языка HTML. То есть мы фактически генерируем разметку HTML! При этом Вы можете использовать абсолютно любые тэги HTML.

По завершении этого мы обязаны вновь закрыть документ. Это делается следующей командой:

```
// закрыть документ - (но не окно!)
myWin.document.close();
```

можно не только динамически создавать документы, но и по своему выбору размещать их в в том или ином фрейме. Например, если Вы получили два фрейма с именами *frame1* и *frame2*, а теперь во *frame2* хотите сгенерировать новый документ, то для этого в *frame1* Вам достаточно будет написать следующее:

```
parent.frame2.document.open();

parent.frame2.document.write("Here goes your HTML-code");

parent.frame2.document.close();
```

Динамическое создание VRML-сцен

Чтобы продемонстрировать гибкость языка JavaScript, давайте теперь попытаемся динамически создать сцену на языке VRML. Напомним, что аббревиатура VRML расшифровывается как язык моделирования виртуальной деальности. То есть это язык для создания трехмерных сцен. можно, например, взять очки виртуальной реальности и наслаждиться прогулкой по таким сценам ... Возьмем самый простой пример - голубой куб. Тем не менее, чтобы рассмотреть его, понадобится программная приставка VRML к Вашему браузеру (plug-in). Предлагаемый Вашему вниманию скрипт не проверяет, а доступен ли браузеру plug-in VRML (впрочем сделать это - вовсе не проблема).

(online-версия руководства позволит Вам проверить этот скрипт немедленно)

Исходный код скрипта:

```
<html>
<head>
<script language="JavaScript">
<!-- hide

function vrmlScene() {
  vrml= open("", "displayWindow",
    "width=500,height=400,status=yes,toolbar=yes,menubar=yes");

  // открыть document для последующего вывода информации
  vrml.document.open("x-world/x-vrml");

  vr= vrml.document;

  // создать сцену VRML
  vr.writeln("#VRML V1.0 ascii");

  // Освещение
  vr.write("Separator { DirectionalLight { ");
  vr.write("direction 3 -1 -2.5 } ");

  // Камера
  vr.write("PerspectiveCamera { position -8.6 2.1 5.6 ");
  vr.write("orientation -0.1352 -0.9831 -0.1233 1.1417 ");
  vr.write("focalDistance 10.84 } ");

  // Куб
  vr.write("Separator { Material { diffuseColor 0 0 1 } ");
  vr.write("Transform { translation -2.4 .2 1 rotation 0 0.5 1 .9 } ");
  vr.write("Cube {} } } ");

  // Закрыть document - (но не окно!)
  vrml.document.close();
}

// -->
```

```
</script>
</head>
<body>

<form>
<input type=button value="VRML on-the-fly" onClick="vrmlScene()">
</form>

</body>
</html>
```

Как видно, текст скрипта совершенно такой же, как и в предыдущем примере. Сперва открывается новое окно. Затем мы открываем document для вывода него информации. Рассмотрим поподробнее соответствующую команду:

```
// открыть document для последующего вывода информации
vrml.document.open("x-world/x-vrml");
```

В предыдущих примерах мы не указывали в скобках ничего. Что же тогда означает новая запись "x-world/x-vrml"? На самом же деле, с помощью этой инструкции мы задаем тип MIME для документа, который хотим создать. То есть, тем самым мы сообщаем браузеру, какого типа данные будут ему сейчас переданы. Если же мы в этом месте не определили в скобках конкретный тип MIME, то по умолчанию для нового документа будет выбран тип *"text/html"* (а это как раз и есть тип MIME для файлов HTML).

(Есть несколько способов выяснить, что же означает тот или иной тип MIME - в самом же браузере содержится список распознаваемых MIME. Вы можете извлечь этот список из пунктов меню option или preference.)

Для создания трехмерной сцены мы должны составить инструкцию *vrml.document.write()*. Но поскольку это кажется слишком длинным, то мы просто определяем переменную *vr = vrml.document*. И затем вместо *vrml.document.write()* мы пишем просто *vr.write ()*.

Закончив это, мы можем писать обычные инструкции на языке VRML. Я не собираюсь описывать здесь элементы сцен VRML. А для желающих познакомиться с ними в Интернет имеется несколько хороших источников информации. Обычный же текст на языке VRML выглядит следующим образом:

```
#VRML V1.0 ascii

Separator {

 DirectionalLight { direction 3 -1 -2.5 }

 PerspectiveCamera {
  position -8.6 2.1 5.6
  orientation -0.1352 -0.9831 -0.1233 1.1417
  focalDistance 10.84
 }

 Separator {
  Material {
   diffuseColor 0 0 1
  }
  Transform {
   translation -2.4 .2 1
```

```
    rotation 0 0.5 1 .9
   }
   Cube {}
  }
}
```

А это как раз и есть тот код, который мы выводим на экран с помощью команды *document.write()*.

Впрочем, совершенно бессмысленно динамически создать сцену, которую с тем же успехом можно загрузить и как обычный VRML-файл.

Лекция 6: Строка состояния и таймеры

Строка состояния

Составленные программы на JavaScript могут выполнять запись в строку состояния - прямоугольник в нижней части окна Вашего браузера. Все, что Вам необходимо для этого сделать - всего лишь записать нужную строку в *window.status*. В следующем примере создаются две кнопки, которые можно использовать, чтобы записывать некий текст в строку состояния и, соответственно, затем его стирать.

```
<html>
<head>
<script language="JavaScript">
<!-- hide

function statbar(txt) {
  window.status = txt;
}

// -->
</script>
</head>
<body>

<form>
 <input type="button" name="look" value="Писать!"
  onClick="statbar('Привет! Это окно состо\яни\я!');">
 <input type="button" name="erase" value="Стереть!"
  onClick="statbar('');">
</form>

</body>
</html>
```

Итак, мы создаем форму с двумя кнопками. Обе эти кнопки вызывают функцию *statbar()*. Вызов от клавиши *Писать!* выглядит следующим образом:

```
statbar('Привет! Это окно состо\яни\я!');
```

В скобках мы написали строку: *'Привет! Это окно состо\яни\я!'* Это как раз и будет текст, передаваемый функции *statbar()*. В свою очередь, можно видеть, что функция *statbar()* определена следующим образом:

```
function statbar(txt) {
  window.status = txt;
}
```

В заголовке функции в скобках мы поместили слово *txt*. Это означает, что строка, которую мы передали этой функции, помещается в переменную *txt*. Передача функциям переменных - прием, часто применяемый для придания функциям большей гибкости. Вы можете передать функции несколько таких аргументов - необходимо лишь отделить их друг от друга запятыми. Строка *txt* заносится в строку состояния посредством команды *window.status = txt*. Соответственно, удаление текста из строки состояния выполняется как запись в *window.status* пустой строки.

Механизм вывода текста в строку состояния удобно использовать при работе со ссылками. Вместо того, чтобы выводить на экран URL данной ссылки, Вы можете просто на словах объяснять, о чем будет говориться на следующей странице. Следующая ссылка демонстрирует это - достаточно лишь поместить указатель вашей мыши над этой ссылкой:

```
<a href="dontclck.htm"
 onMouseOver="window.status= 'Don\'t click me!'; return true;"
 onMouseOut="window.status='';">link</a>
```

Здесь мы пользуемся процедурами *onMouseOver* и *onMouseOut*, чтобы отслеживать моменты, когда указатель мыши проходит над данной ссылкой. Вы можете спросить, а почему в *onMouseOver* мы обязаны возвращать результат *true*. На самом деле это означает, что браузер не должен вслед за этим выполнять свой собственный код обработки события *MouseOver*. Как правило, в строке состояния браузер показывает URL соответствующей ссылки. Если же мы не возвратим значение *true*, то сразу же после того, как наш код был выполнен, браузер перепишет строку состояния на свой лад - то есть наш текст будет тут же затерт и читатель не сможет его увидеть. В общем случае, мы всегда можем отменить дальнейшую обработку события браузером, возвращая true в своей собственной процедуре обработки события.

в JavaScript 1.0 процедура *OnMouseOut* еще не была представлена. И если Вы пользуетесь Netscape Navigator 2.x, то возможно на различных платформах Вы можете получить различные результаты. Например, на платформах Unix текст исчезает даже несмотря на то, что браузер не знает о существовании процедуры *onMouseOut*. В Windows текст не исчезает. И если Вы хотите, чтобы ваш скрипт был совместим с Netscape 2.x для Windows, то можете, к примеру, написать функцию, которая записывает текст в окно состояния, а потом стирает его через некоторый промежуток времени. Программируется это с помощью таймера timeout. Подробнее работу с таймерами мы рассмотрим в одном из следующих параграфов.

В этом скрипте Вы можете видеть еще одну вещь - в некоторых случаях Вам понадобится печатать символы кавычек. Например, мы хотим напечатать текст *Don't click me* - однако поскольку мы передаем эту строку в процедуру обработки события *onMouseOver*, то мы используем для этого одинарные кавычки. Между тем, как слово *Don't* тоже содержит символ одинарной кавычки! И в результате если Вы просто впишете *''Don't ...'*, браузер запутается в этих символах '. Чтобы разрешить эту проблему, Вам достаточно лишь поставить обратный слэш \ перед символом кавычки - это означает, что данный символ предназначен именно для печати. (То же самое Вы можете делать и с двойными кавычками - ").

Таймеры

С помощью функции Timeout (или таймера) Вы можете запрограммировать компьютер на выполнение некоторых команд по истечении некоторого времени. В следующем скрипте демострируется кнопка, которая открывает выпадающее окно не сразу, а по истечении 3 секунд.

Скрипт выглядит следующим образом:

```
<script language="JavaScript">
<!-- hide

function timer() {
  setTimeout("alert('Врем\я истекло!')", 3000);
}

// -->
</script>

...

<form>
<input type="button" value="Timer" onClick="timer()">
</form>
```

Здесь *setTimeout()* - это метод объекта window. Он устанавливает интервал времени - я полагаю, Вы догадываетесь, как это происходит. Первый аргумент при вызове - это код JavaScript, который следует выполнить по истечении указанного времени. В нашем случае это вызов - *alert('Врем\я истекло!')*. Обратите пожалуйста внимание, что код на JavaScript должен быть заключен в кавычки. Во втором аргументе компьютеру сообщается, когда этот код следует выполнять. При этом время Вы должны указывать в миллисекундах (3000 миллисекунд = 3 секунда).

Прокрутка

Теперь, когда Вы знаете, как делать записи в строке состояния и как работать с таймерами, мы можем перейти к управлению прокруткой. Вы уже могли видеть, как текст перемещается строке состояния. В Интернет этим приемом пользуются повсеместно. Теперь же мы рассмотрим, как можно запрограммировать прокрутку в основной линейке. Рассмотрим также и всевозможные усовершенствования этой линейки. Создать бегущую строку довольно просто. Для начала давайте задумаемся, как вообще можно создать в строке состояния перемещающийся текст - бегущую строку. Очевидно, сперва мы должны записать в строку состояния некий текст. Затем по истечении короткого интервала времени мы должны записать туда тот же самый текст, но при этом немного переместив его влево. Если мы это сделаем несколько раз, то у пользователя создастся впечатление, что он имеет дело с бегущей строкой. Однако при этом мы должны помнить еще и о том, что обязаны каждый раз вычеслять, какую часть текста следует показывать в строке состояния (как правило, объем текстового материала превышает размер строки состояния).

Итак, исходный код скрипта и к нему еще некоторые комментарии:

```
<html>
```

```
<head>
<script language="JavaScript">
<!-- hide


// выбор текста дл\я прокрутки
var scrtxt = "Это JavaScript! " +
    "Это JavaScript! " +
    "Это JavaScript!";
var len = scrtxt.length;
var width = 100;
var pos = -(width + 2);

function scroll() {

  // напечатать заданный текст справа и установить таймер

  // перейти на исходную позицию для следующего шага
  pos++;

  // вычленить видимую часть текста
  var scroller = "";
  if (pos == len) {
    pos = -(width + 2);
  }

  // если текст еще не дошел до левой границы, то мы должны
  // добавить перед ним несколько пробелов. В противном случае мы должны
  // вырезать начало текста (ту часть, что уже ушла за левую границу
  if (pos < 0) {
    for (var i = 1; i <= Math.abs(pos); i++) {
      scroller = scroller + " ";}
    scroller = scroller + scrtxt.substring(0, width - i + 1);
  }
  else {
    scroller = scroller + scrtxt.substring(pos, width + pos);
  }

  // разместить текст в строке состо\яни\я
  window.status = scroller;

  // вызвать эту функцию вновь через 100 миллисекунд
  setTimeout("scroll()", 100);
}

// -->
</script>
</head>

<body onLoad="scroll()">
Наша HTML-страница.
</body>
</html>
```

Большая часть функции *scroll()* нужна для вычленения той части текста, которая будет показана пользователю. Я не буду объяснять этот код подробно - Вам необходимо лишь понять, как вообще осуществляется эта прокрутка. Чтобы запустить этот процесс, мы используемся процедурой обработки события onLoad, описанной в тэге <*body*>. То есть функция *scroll()* будет вызвана сразу же после загрузки HTML-страницы. Через посредство процедуры *onLoad* мы вызываем функцию *scroll()*. Первым делом в функции *scroll()* мы устанавливаем таймер. Этим гарантируется, что функция *scroll()* будет повторно вызвана через 100 миллисекунд. При этом текст будет перемещен еще на один шаг и запущен другой таймер. Так будет продолжаться без конца.

(В Netscape Navigator 2. x с таким типом скроллинга были некоторые проблемы - его выполнение иногда приводило к появлению ошибки 'Out of memory'. Это возникает вследствие рекурсивного вызова функции scroll(), что в конце концов приводит к выходу за пределы памяти. Но это не так. Данный вызов функции не является рекурсивным! Рекурсию мы получим, если будем вызывать функцию scroll() непосредственно внутри самой же функции scroll(). А этого здесь мы как раз и не делаем. Прежняя функция, установившая таймер, заканчивается еще до того, как начинается выполнение новой функции. Проблема же состояла в том, что в действительности мы не могли в JavaScript выполнять коррекцию строк. И если Вы пробуете сделать это, то JavaScript просто-напросто создавал новый объект - но при этом не удалял старый. Именно таким образом происходило переполнение памяти.)

Скроллинг используется в Интернет довольно широко. И есть риск, что быстро он станет непопулярным. Я должен признаться, что и сам не очень его люблю. В большинстве страниц, где он применяется, особенно раздражает то, что из-за непрерывного скроллинга становится невозможным прочесть в строке состояния адрес URL. Эту проблему можно было бы решить, позаботившись о приостановке скроллига, если происходит событие MouseOver - и, соответственно, продолжении, когда финсируется onMouseOut. Если Вы хотите попытаться создать скроллинг, то пожалуйста не используйте стандартный его вариант - пробуйте привнести в него некоторые приятные особенности. Возможен вариант, когда одна часть текста приходит слева, а другая - справа. И когда они встречаются посередине, то в течение некоторых секунд текст остается неизменным. Воспользовавшись небольшой долей фантазии, Вы конечно же сможете найти еще несколько хороших альтернатив.

Контрольные вопросы

1. Какой тег используется для индексирования документов в поисковых системах?
   a. HTML
   b. BODY
   c. HEAD
   d. META
2. Укажите неверные варианты описания синтаксиса тега SCRIPT.
   a. <SCRIPT TYPE=тип_языка_программирования>текст программы</SCRIPT>
   b. +<SCRIPT TYPE=тип_документа >текст программы</SCRIPT>
   c. +<SCRIPT NAME=язык_программирования>текст программы</SCRIPT>
3. Что определяет атрибут CELLSPACING у элемента разметки TABLE?
   a. расстояние между ячейками
   b. расстояние от содержания до границы ячейки
   c. ширину ячейки
   d. ширину границы
4. В каких случаях атрибут valign имеет более высокий приоритет?
   a. <TH valign="top">
   b. <COL valign="top">
   c. <TABLE valign="top">
5. 15.Какие из приведенных тегов неверно описывают активное изображение?
   a. <img src="image.jpg" width=100 height=100>

b. <img src="image.jpg" width=100 height=100 usemap="#map1">
c. <+img src="image.jpg" width=100 height=100 usemap>

6. Укажите, что является интерпретатором языка HTML?
7. +функции интерпретатора разделены между Web-сервером и интерфейсом пользователя
8. только web-сервер гипертекстовой базы данных
9. только интерфейс пользователя
10. В чем разница между понятиями "элемент разметки" и "контейнер"?
11. разницы нет
12. "контейнер" - это реализация "элемента разметки"
13. это два разных понятия
14. Какие из приведенных тегов HTML начинают вывод текста с новой строки на странице?
    a. NOBR
    b. P
    c. BR
    d. H1
15. Какой из приведенных тегов позволяет создавать нумерованные списки?
    a. DT
    b. DL
    c. UL
    d. OL

Лекция 7: Предопределенные объекты

Объект Date

В JavaScript Вам разрешено пользоваться некоторыми заранее заданными объектами. Примерами таких объектов могут служить Date, Array или Math. Есть еще несколько таких же объектов - полное описание см. в документации, предоставляемой фирмой Netscape.

Для начала давайте рассмотрим объект Date. Судя по названию, он позволяет Вам работать как со временем, так и с датой. Например, Вы можете легко определить, сколько дней еще остается до следующего рождества. Или можете внести в Ваш HTML-документ запись текущего времени.

Так что давайте начнем с примера, который высвечивает на экран текущее время. Сперва мы должны создать новый объект Date. Для этого мы пользуемся оператором new:

today= new Date()

Здесь создается новый объект Date, с именем today. Если при создании этого нового объекта Date Вы не указали какой-либо определенной даты и времени, то будут предоставлены текущие дата и время. То есть, после выполнения команды *today= new Date()* вновь созданный объект *today* будет указывать именно те дату и время, когда данная команда была выполнена.

Объект Date предоставляет нам кое-какие методы, которые теперь могут применяться к нашему объекту today. Например, это методы - *getHours(), setHours(), getMinutes(), setMinutes(), getMonth(), setMonth()* и так далее. Полное описание объекта Date и его методов Вы сможете найти в документации по JavaScript, предоставляемой фирмой Netscapes.

Обратите пожалуйста внимание, что объект Date лишь содержит определенную запись о дате и времени. Он не уподобляется часам, автоматически отслеживающим время каждую секунду, либо миллисекунду.

Чтобы зафиксировать какое-либо другие дату и время, мы можем воспользоваться видоизмененным конструктором (это будет метод *Date()*, который при создании нового объекта Date вызывается через оператор *new*):

```
today= new Date(1997, 0, 1, 17, 35, 23)
```

При этом будет создан объект Date, в котором будет зафиксировано первое января 1997 года 17:35 и 23 секунд. Таким образом, Вы выбираете дату и время по следующему шаблону:

```
Date(year, month, day, hours, minutes, seconds)
```

Заметьте, что для обозначения января Вы должны использовать число 0, а не 1, как Вы вероятно думали. Число 1 будет обозначать февраль, ну и так далее.

Теперь мы напишем скрипт, печатающий текущие дату и время. Результат будет выглядеть следующим образом:

```
Time: 17:53
Date: 4/3/2010
```

Сам же код выглядит следующим образом:

```
<script language="JavaScript">
<!-- hide

now= new Date();

document.write("Time: " + now.getHours() + ":" + now.getMinutes() + "<br>");
document.write("Date: " + (now.getMonth() + 1) + "/" + now.getDate() + "/" +
        (1900 + now.getYear()));

// -->
</script>
```

Здесь мы пользуемся такими методами, как *getHours()*, чтобы вывести на экран время и дату, указанные в объекте Date с именем now. можно видеть, что мы добавляем к записи года еще число 1900. Дело в том, что метод *getYear()* указывает количество лет, прошедших после 1900 года. А стало быть, если сейчас 1997 год, то будет выдано значение 97, а если 2010 год - то 110, а не 10! Если мы так и будем всякий раз добавлять 1900, то у нас не будет проблемы 2000 года. Помните также, что мы обязаны увеличивать на единицу значение, получаемое от метода *getMonth()*.

В данном скрипте не выполняется проверки на тот случай, если количество минут окажется меньше, чем 10. Это значит, что Вы можете получить запись времени примерно в следующем виде: 14:3, что на самом деле должно было бы означать 14:03. Решение этой проблемы мы рассмотрим в следующем примере.

Рассмотрим теперь скрипт, создающий на экране изображение работающих часов:

```
<html>
<head>
```

```
<script Language="JavaScript">
<!-- hide

var timeStr, dateStr;

function clock() {
        now= new Date();

        // врем\я
    hours= now.getHours();
    minutes= now.getMinutes();
    seconds= now.getSeconds();
    timeStr= "" + hours;
    timeStr+= ((minutes < 10) ? ":0" : ":") + minutes;
    timeStr+= ((seconds < 10) ? ":0" : ":") + seconds;
    document.clock.time.value = timeStr;

    // дата
    date= now.getDate();
    month= now.getMonth()+1;
    year= now.getYear();
    dateStr= "" + month;
    dateStr+= ((date < 10) ? "/0" : "/") + date;
    dateStr+= "/" + year;
    document.clock.date.value = dateStr;

    Timer= setTimeout("clock()",1000);
}

// -->
</script>
</head>

<body onLoad="clock()">

<form name="clock">
 Время:
 <input type="text" name="time" size="8"  value=""><br>
 Дата:
 <input type="text" name="date" size="8" value="">
</form>

</body>
</html>
```

Здесь для ежесекундной коррекции времени и даты мы пользуемся методом *setTimeout()*. Фактически это сводится к кому, что мы каждую секунду создаем новый объект Date, занося туда текущее время. можно видеть, что функции *clock()* вызываются программой обработки события onLoad, помещенной в тэг <*body*>. В разделе body нашей HTML-страницы имеется два элемента формы для ввода текста. Функция *clock()* записывает в оба эти элемента в корректном формате текущие время и дату. Для этой цели используются две строки *timeStr* и *dateStr*.

Как мы уже упомянули ранее, существует проблема с индикацией, когда количество минут меньше 10 - в данном скрипте эта проблема решается с помощью следующей строки:

timeStr+= ((minutes < 10) ? ":0" : ":") + minutes;

Как видим, количество минут заносится в строку *timeStr*. Если у нас менее 10 минут, то мы еще должны приписать спереди 0. Для Вас эта строка в скрипте может показаться немного странной, и ее можно было бы переписать в более знакомом Вам виде:

```
if (minutes < 10) timeStr+= ":0" + minutes
  else timeStr+= ":" + minutes;
```

Объект Array

Массивы играют в программировании очень важную роль. Подумайте только, что бы Вы делали, если бы Вам понадобилось хранить 100 различных имен. Как бы Вы могли это сделать с помощью JavaScript? Хорошо, Вы могли бы явным образом задать 100 переменных и присвоить им различные имена. Но согласитесь, это будет весьма утомительно.

Массив может быть полезен там, где имеется много взаимосвязанных переменных. При этом к каждой из них Вы можете получить доступ, воспользовавшись общим названием и неким номером.

Допустим, есть массив в именем *names*. В этом случае мы можем получить доступ к первой переменной с именем name, написав *names[0]*. Вторая переменная носит *name[1]* и так далее.

Начиная с версии 1.1 языка JavaScript (Netscape Навигатор 3.0), Вы можете использовать объект Array. Вы можете создать новый массив, записав *myArray= new Array()*. После этого можно начать заносить в массив значения:

```
myArray[0]= 17;
myArray[1]= "Stefan";
myArray[2]= "Koch";
```

Массивы JavaScript обладают большой гибкостью. Например, Вам нет нужды беспокоиться о размере массива - он устанавливается динамически. Если Вы напишете *myArray[99]= "xyz"*, размер массива будет установлен 100 элементов. (В языке JavaScript размер массива может только увеличиваться - массив не может “сжиматься”. Поэтому старайтесь делать Ваши массивы как можно копактнее.) Не имеет значения, заносите ли Вы в массив числа, строки, либо другие объекты. Я не останавливаюсь на каждой такой подробности структуры массивов, но надеюсь, Вы поймете, что массивы - очень важный элемент языка.

Конечно же многое станет понятнее, если рассматривать примеры. Следующий скрипт печатает следующий текст:

```
first element
second element
third element
```

Исходный код:

```
<script language="JavaScript">
<!-- hide
```

```
myArray= new Array();

myArray[0]= "first element";
myArray[1]= "second element";
myArray[2]= "third element";

for (var i= 0; i< 3; i++) {
  document.write(myArray[i] + "<br>");
}

// -->
</script>
```

Первым делом мы создаем здесь новый массив с именем *myArray*. Затем мы заносим в него три различных значения. После этого мы запускаем цикл, который трижды выполняет команду *document.write(myArray[i] + "
");*. В переменной *i* ведется отсчет циклов от 0 до 2. Заметим, что в цикле мы пользуемся конструкцией *myArray[i]*. И поскольку *i* меняет значения от 0 до 2, то в итоге мы получаем три различных вызова *document.write()*. Иными словами, мы могли бы расписать этот цикл как:

```
document.write(myArray[0] + "<br>");
document.write(myArray[1] + "<br>");
document.write(myArray[2] + "<br>");
```

Массивы в JavaScript 1.0

Поскольку в JavaScript 1.0 (Netscape Navigator 2.x, и Microsoft Internet Explorer 3.x) объекта Array еще не существовало, то мы должны думать и об его альтернативе. Следующий фрагмент кода можно найти в документации фирмы Netscape:

```
function initArray() {
  this.length = initArray.arguments.length
  for (var i = 0; i < this.length; i++)
    this[i+1] = initArray.arguments[i]
}
```

После этого Вы можете создавать массив одной строкой:

```
myArray= new initArray(17, 3, 5);
```

Числа в скобках - значения, которыми инициализируется массив (это можно также делать и с объектом Array из JavaScript 1.1). Обратите внимание, что данный тип массива не может включать все элементы, которые являются частью в объекта Array от JavaScript 1.1 (например, там имеется метод sort(), который позволяет сортировать все элементы в определенном порядке).

Объект Math

Если необходимо в скрипте выполнить математические расчеты, то некоторые полезные методы для этого можно найти в объекте Math. Например, имеется метод синуса *sin()*. Полную

информацию об этом объекте можно найти в документации фирмы Netscape. Продемонстрируем работу метода *random()*. В начале у нас были некоторые проблемы с методом *random()*. Тогда мы написали функцию, позволяющую генерировать случайные числа. Теперь, чтобы работать на всех без исключения платформах, нам не нужно ничего, кроме метода *random()*.
Если Вы вызовете функцию *Math.random(),* то получите случайное число, лежащее в диапазоне между 0 и 1. Один из возможных результатов вызова *document.write(Math.random())*:

.7184317731538611

Лекция 8: Формы

Проверка информации, введенной в форму

Формы широко используются на Интернет. Информация, введенная в форму, часто посылается обратно на сервер или отправляется по электронной почте на некоторый адрес. Проблема состоит в том, чтобы убедиться, что введеная пользователем в форму информация корректна. Легко проверить ее перед пересылкой в Интернет можно с помощью языка JavaScript. Продемонстрируем, как можно выполнить проверку формы, рассмотрим, какие есть возможности для пересылки информации по Интернет.

Сперва нам необходимо создать простой скрипт. Допустим, HTML-страница содержит два элемента для ввода текста. В первый из них пользователь должен вписать свое имя, во второй элемент - адрес для электронной почты. Если пользователь ввел свое имя (например, 'Stefan') в первый элемент, то скрипт создает выпадающее окно с сообщением '*Hi Stefan!*'.

Что касается информации, введенной в первый элемент, то Вы будете получать сообщение об ошибке, если туда ничего не было введено. Любая представленная в элементе информация будет рассматриваться на предмет корректности. Конечно, это не гарантирует, что пользователь введет не то имя. Браузер даже не будет возражать против чисел. Например, если Вы ввведете '17', то получите приглашение '*Hi 17!*'. Так что эта проверка не может быть идеальна. Второй элемент формы несколько более сложнее. Попробуйте ввести простую строку - например Ваше имя. Сделать это не удастся до тех пор, пока Вы не укажите @ в Вашем имени... Признаком того, что пользователь правильно ввел адрес электронной почты служит наличие символа @. Этому условию будет отвечать и одиночный символ @, даже несмотря на то, что это бессмысленно. В Интернет каждый адрес электронной почты содержит символ @, так что проверка на этот символ здесь уместна.

Как скрипт работает с этими двумя элементами формы и как выглядит проверка?

```
<html>
<head>
<script language="JavaScript">
<!-- Скрыть

function test1(form) {
  if (form.text1.value == "")
    alert("Пожалуйста, введите строку!")
  else {
   alert("Hi "+form.text1.value+"! Форма заполнена корректно!");
  }
}
```

```
function test2(form) {
  if (form.text2.value == "" ||
     form.text2.value.indexOf('@', 0) == -1)
       alert("Неверно введен адрес e-mail!");
  else alert("OK!");
}
// -->
</script>
</head>

<body>
<form name="first">
Введите Ваше имя:<br>
<input type="text" name="text1">
<input type="button" name="button1" value="Проверка" onClick="test1(this.form)">
<P>
Введите Ваш адрес e-mail:<br>
<input type="text" name="text2">
<input type="button" name="button2" value="Проверка" onClick="test2(this.form)">
</body>
</html>
```

Рассмотрим сначала HTML-код в разделе body. Здесь мы создаем лишь два элемента для ввода текста и две кнопки. Кнопки вызывают функции *test1(...)* или *test2(...)*, в зависимости от того, которая из них была нажата. В качестве аргумента к этим функциям мы передаем комбинацию this.form, что позже позволит нам адресоваться в самой функции именно к тем элементам, которые нам нужны. Функция test1(form) проверяет, является ли данная строка пустой. Это делается посредством *if (form.text1.value == "")...* . Здесь '*form*' - это переменная, куда заносится значение, полученное при вызове функции от '*this.form*'. Мы можем извлечь строку, введенную в рассматриваемый элемент, если к *form.text1* припишем '*value*'. Чтобы убедиться, что строка не является пустой, мы сравниваем ее с "". Если же окажется, что введенная строка соответствует "", то это значит, что на самом деле ничего введено не было. И наш пользователь получит сообщение об ошибке. Если же что-то было введено верно, пользователь получит подтверждение - ok.

Следующая проблема заключается в том, что пользователь может вписать в поле формы одни пробелы. И это будет принято, как корректно введенная информация! Если есть желание, то Вы конечно можете добавить проверку такой возможности и исключить ее. Я полагаю, что это будет сделать легко, опираясь лишь на представленную здесь информацию. Рассмотрим теперь функцию *test2(form)*. Здесь вновь сравнивается введенная строка с пустой - "" (чтобы удостовериться, что что-то действительно было введено читателем). Однако к команде if мы добавили еще кое-чего. Комбинация символов || называется оператором OR (ИЛИ). С ним Вы уже знакомились в шестой части Введения. Команда if проверяет, чем заканчивается первое или второе сравнения. Если хотя бы одно из них выполняется, то и в целом команда if имеет результатом *true*, а стало быть будет выполняться следующая команда скрипта. Словом, Вы получите сообщение об ошибке, если либо предоставленная Вами строка пуста, либо в ней отсутствует символ @. (Второй оператор в команде if следит за тем, чтобы введенная строка содержала @.)

Проверка на присутствие определенных символов

В некоторых случаях Вам понадобится ограничивать информацию, вводимую в форму, лишь некоторым набором символов или чисел. Достаточно вспомнить о телефонных номерах - представленная информация должна содержать лишь цифры (предполагается, что номер телефона, как таковой, не содержит никаких символов). Нам необходимо проверять, являются ли введенные данные числом. Сложность ситуации состоит в том, что большинство людей вставляют в номер телефона еще и разные символы - например: *01234-56789*, *01234/56789* or *01234 56789* (с символом пробела внутри). Не следует принуждать пользователя отказываться от таких символов в телефонном номере. А потому мы должны дополнить наш скрипт процедурой проверки цифр и некоторых символов. Решение задачи продемонстрировано в следующем примере, взятом из моей книги о JavaScript:

(online-версия руководства позволит Вам проверить этот скрипт немедленно)

```
<html>
<head>
<script language="JavaScript">
<!-- hide

// ****************************************************
// Script from Stefan Koch - Voodoo's Intro to JavaScript
//     http://rummelplatz.uni-mannheim.de/~skoch/js/
//       JS-book: http://www.dpunkt.de/javascript
//    You can use this code if you leave this message
// ****************************************************

function check(input) {
  var ok = true;

  for (var i = 0; i < input.length; i++) {
    var chr = input.charAt(i);
    var found = false;
    for (var j = 1; j < check.length; j++) {
      if (chr == check[j]) found = true;
    }
    if (!found) ok = false;
  }

  return ok;
}

function test(input) {

  if (!check(input, "1", "2", "3", "4",
        "5", "6", "7", "8", "9", "0", "/", "-", " ")) {

    alert("Input not ok.");
  }
  else {
    alert("Input ok!");
  }
}

// -->
```

```
</script>
</head>

<body>
<form>
Telephone:
<input type="text" name="telephone" value="">
<input type="button" value="Check"
  onClick="test(this.form.telephone.value)">
</form>
</body>
</html>
```

Функция *test()* определяет, какие из введенных символов признаются корректными.

Предоставление информации, введенной в форму

Какие существуют возможности для передачи информации, внесенной в форму? Самый простой способ состоит в передаче данных формы по электронной почте (этот метод мы рассмотрим поподробнее). Если Вы хотите, чтобы за вносимыми в форму данными следил сервер, то Вы должны использовать интерфейс CGI (Common Gateway Interface). Последнее позволяет Вам автоматически обрабатывать данные. Например, сервер мог бы создавать базу данных со сведениями, доступную для некоторых из клиентов. Другой пример - поисковые страницы, такие как Yahoo. Обычно в них представлена форма, позволяющая создавать запрос для поиска в собственной базе данных. В результате пользователь получает ответ вскоре после того, как нажимает на соответствующую кнопку. Ему не приходится ждать, пока люди, отвечающие за поддержание данного сервера, прочтут указанные им данные и отыщут требуемую информацию. Все это автоматически выполняет сам сервер. JavaScript не позволяет делать таких вещей. С помощью JavaScript Вы не сможете создать книгу читательских отзывов, поскольку JavaScript лишен возможности записывать данные в какой-либо файл на сервере. Делать это Вы можете только через интерфейс CGI. Конечно, Вы можете создать книгу отзывов, для которой пользователи присылали сведения по электронной почте. Однако в этом случае Вы должны заносить отзывы вручную. Так можно делать, если Вы не предполагаете получать ежедневно по 1000 отзывов. Соответствующий скрипт будет простым текстом HTML. И никакого программирования на JavaScript здесь вовсе не нужно! Конечно за исключением того случая, если Вам понадобится перед пересылкой проверить данные, занесенные в форму - и здесь уже Вам действительно понадобится JavaScript. Добавим, что команда mailto работает не повсюду - например, поддержка для ее отсутствует в Microsoft Internet Explorer 3.0.

```
<form method=post action="mailto:your.address@goes.here" enctype="text/plain">
Нравится ли Вам эта страница?
  <input name="choice" type="radio" value="1">Вовсе нет.<br>
  <input name="choice" type="radio" value="2" CHECKED>Напрасная трата времени.<br>
  <input name="choice" type="radio" value="3">Самый плохой сайт в Сети.<br>
  <input name="submit" type="submit" value="Send">
</form>
```

Параметр *enctype="text/plain"* используется для того, чтобы пересылать именно простой текст без каких-либо кодируемых частей. Это значительно упрощает чтение такой почты.

Если Вы хотите проверить форму прежде, чем она будет передана в сеть, то для этого можете воспользоваться программой обработки событий onSubmit. Вы должны поместить вызов этой программы в тэг <*form*>. Например:

```
      function validate() {
        // check if input ok
        // ...

  if (inputOK) return true
    else return false;
}

...

<form ... onSubmit="return validate()">

...
```

Форма, составленная таким образом, не будет послана в Интернет, если в нее внесены некорректные данные.

Выделение определенного элемента формы

С помощью метода *focus()* Вы можете сделать вашу форму более дружественной. Так, Вы можете выбрать, который элемент будет выделен в первую очередь. Либо Вы можете приказать браузеру выделить ту форму, куда были введены неверные данные. То есть, что браузер сам установит курсор на указанный Вами элемент формы, так что пользователю не придется щелкать по форме, прежде чем что-либо занести туда. Сделать это Вы можете с помощью следующего фрагмента скрипта:

```
function setfocus() {
  document.first.text1.focus();
}
```

Эта запись могла бы выделить первый элемент для ввода текста в скрипте, который я уже показывал. Вы должны указать имя для всей формы - в данном случае она называется first - и имя одного элемента формы - text1. Если Вы хотите, чтобы при загрузке страницы данный элемент выделялся, то для этого Вы можете дополнить Ваш тэг <*body*> атрибутом onLoad. Это будет выглядеть как:

```
 <body onLoad="setfocus()">
```

Остается еще дополнить пример следующим образом:

```
function setfocus() {
  document.first.text1.focus();
  document.first.text1.select();
}
```

При этом не только будет выделен элемент, но и находящийся в нем текст.

Контрольные вопросы

1. Какие методы можно применять для отправки формы?
   - GET
   - POST
   - HEAD
   - MAILTO

2. Какой из перечисленных элементов формы является необходимым для передачи формы на сервер?
   - <INPUT TYPE=submit NAME=a1>
   - <TEXTAREA NAME=a1>
   - <INPUT TYPE=text NAME=a1>
   - <INPUT TYPE=reset>

3. Какие теги используются для создания текстовых полей ввода в форме?
   - <INPUT TYPE=text>
   - <TEXTAREA>
   - <SELECT>
   - <OPTION>

4. В каких примерах данные формы будут переданы обработчику как часть URL?
   - <FORM method="get" action="http://www.intuit.ru/cgi">
   - <FORM method="post" action="http://www.intuit.ru/help/first.pl">
   - <FORM method="post" action="mailto:info@intuit.ru">
   - <FORM method="get" action="http://www.intuit.ru/">

5. Укажите варианты, в которых правильно определён обработчик формы:
   - <FORM method="get" action="/cgi-bin/">
   - <FORM method="post" action="http://www.intuit.ru/sp.pl">
   - <FORM method="put" action="http://www.intuit.ru/shop.pl">

6. С помощью какого атрибута элемента FORM указывается адрес, куда отправлять данные формы?
   - HREF
   - LOCATION
   - ACTION
   - TARGET

7. Какие контейнеры используются для задания элементов формы?
   - SELECT
   - TEXTAREA
   - SUBMIT
   - FORM

8. Какие теги используются для создания текстовых полей ввода в форме?
   - <INPUT TYPE=text>
   - <TEXTAREA>
   - <SELECT>
   - <OPTION>

9. Какой из приведенных тегов создает неотображаемый элемент в форме?
   - <HIDDEN NAME=a1 VALUE=1>
   - <INPUT TYPE=HIDDEN NAME=a1 VALUE=1>
   - такой элемент создать нельзя

10. Какой из приведенных фрагментов кода создает переключатель?
    - <input type=radio name=a1 value=1><input type=radio name=a1 value=2>
    - <input type=checkbox name=a1 value=1><input type=checkbox name=a1 value=2><input type=text name=a1 value=2>
    - <input type=radiobutton name=a1 value=1><input type=radiobutton name=a1 value=2>

Лекция 9: Объект Image

Изображения на web-странице

Рассмотрим теперь объект Image, который стал доступен, начиная с версии с 1.1 языка JavaScript (то есть с Netscape Navigator 3.0). С помощью объекта Image Вы можете вносить изменения в графические образы, присутствующие на web-странице. В частности, это позволяет нам создавать мультипликацию.

Заметим, что пользователи браузеров более старых версий (таких как Netscape Navigator 2.0 или Microsoft Internet Explorer 3.0 - т.е. использующих версию 1.0 языка JavaScript) не смогут запускать скрипты, приведенные в этой части описания. Или, в лучшем случае, на них нельзя будет получить польный эффект. Давайте сначала рассмотрим, как из JavaScript можно адресоваться к изображениям, представленным на web-странице. В рассматриваемом языке все изображения предстают в виде массива. Массив этот называется images и является свойством объекта document. Каждое изображение на web-странице получает порядковый номер: первое изображение получает номер 0, второе - номер 1 и т.д. Таким образом, к первому изображению мы можем адресоваться записав document.images[0].

Каждое изображение в HTML-документе рассматривается в качестве объекта Image. Объект Image имеет определенные свойства, к которым и можно обращаться из языка JavaScript. Например, Вы можете определить, который размер имеет изображение, обратившись к его свойствам *width* и *height*. То есть по записи *document.images[0].width* Вы можете определить ширину первого изображения на web-странице (в пикселах).

К сожалению, отслеживать индекс всех изображений может оказаться затруднительным, особенно если на одной странице у Вас их довольно много. Эта проблема решается назначением изображениям своих собственных имен. Так, если Вы заводите изображение с помощью тэга

<img src="img.gif" name="myImage" width=100 height=100>

то Вы сможете обращаться к нему, написав *document.myImage* или *document.images["myImage"]*.

Загрузка новых изображений

Необходимо осуществлять смену изображений на web-странице и для этого понадобится атрибут *src*. Как и в случае тэга *<img>*, атрибут *src* содержит адрес представленного изображения. Теперь - в языке JavaScript 1.1 - имеется возможность назначать новый адрес изображению, уже загруженному в web-страницу. И в результате, изображение будет загружено с этого нового адреса, заменив на web-странице старое. Рассмотрим к примеру запись:

<img src="img1.gif" name="myImage" width=100 height=100>

Здесь загружается изображение *img1.gif* и получает имя *myImage*. В следующей строке прежнее изображение *img1.gif* заменяется уже на новое - *img2.gif*:

document.myImage.src= "img2.src";

При этом новое изображение всегда получает тот же размер, что был у старого. И Вы уже не можете изменить размер поля, в котором это изображение размещается.

(online-версия руководства позволит Вам проверить этот скрипт немедленно)

Упреждающая загрузка изображения

Один из недостатков такого подхода может заключаться в том, что после записи в src нового адреса начинает процесс загрузки соответствующего изображения. И поскольку этого не было сделано заранее, то еще пройдет некоторое время, прежде чем новое изображение будет передано через Интернет и встанет на свое место. В некоторых ситуациях это допустимо, однако часто подобные задержки неприемлемы. И что же мы можем сделать с этим? Конечно, решением проблемы была бы упреждающая загрузка изображения. Для этого мы должны создать новый объект Image. Рассмотрим следующие строки:

```
hiddenImg= new Image();
hiddenImg.src= "img3.gif";
```

В первой строке создается новый объект Image. Во второй строке указывается адрес изображения, которое в дальнейшем будет представлено с помощью объекта hiddenImg. Как мы уже видели, запись нового адреса в атрибуте src заставляет браузер загружать изображение с указанного адреса. Поэтому, когда выполняется вторая строка нашего примера, начинает загружаться изображение img2.gif. Но как подразумевается самим названием hiddenImg (“скрытая картинка”), после того, как браузер закончит загрузку, изображение на экране не появится. Оно будет лишь будет сохранено в памяти компьютера (или точнее в кэше) для последующего использования. Чтобы вызвать изображение на экран, мы можем воспользоваться строкой:

```
document.myImage.src= hiddenImg.src;
```

Но теперь изображение уже немедленно извлекается из кэша и показывается на экране. Таким образом, сейчас мы управляли упреждающей загрузкой изображения. Конечно браузер должен был к моменту запроса закончить упреждающую загрузку, чтобы необходимое изображение было показано без задержки. Поэтому, если Вы должны предварительно загрузить большое количество изображений, то может иметь место задержка, поскольку браузер будет занят загрузкой всех картинок. Вы всегда должны учитывать скорость связи с Интернет - загрузка изображений не станет быстрее, если пользоваться только что показанными командами. Мы лишь пытаемся чуть раньше загрузить изображение - поэтому и пользователь может увидеть их раньше. В результате и весь процесс пройдет более гладко.

Если у Вас есть быстрая связь с Интернет, то Вы можете не понять, к чему весь этот разговор. О какой задержке все время говорит этот парень? Прекрасно, но еще остаются люди, имеющие более медленный модем, чем 14.4 (Нет, это не я. Я только что заменил свой на 33.6, да ...).

Изменение изображений в соответствии с событиями, инициируемыми самим читателем

Вы можете создать красивые эффекты, используя смену изображений в качестве реакции на определенные события. Например, Вы можете изменять изображения в тот момент, когда курсор мыши попадает на определеннуючасть страницы.

(online-версия руководства позволит Вам проверить этот скрипт немедленно)

Исходный код этого примера выглядит следующим образом:

```
<a href="#"
 onMouseOver="document.myImage2.src='img2.gif'"
 onMouseOut="document.myImage2.src='img1.gif'">
<img src="img1.gif" name="myImage2" width=160 height=50 border=0></a>
```

При этом могут возникнуть следующие проблемы:

- Читатель пользуется браузером, не имеющим поддержки JavaScript 1.1.
- Второе изображение не было загружено.
- Для этого мы должны писать новые команды для каждого изображения на web-странице.
- Мы хотели бы иметь такой скрипт, который можно было бы использовать во многих web-страницах вновь и вновь, и без больших переделок.

Теперь мы рассмотрим полный вариант скрипта, который мог бы решить эти проблемы. Хотя скрипт и стал намного длиннее - но написав его один раз, Вы не больше будете беспокоиться об этих проблемах. Чтобы этот скрипт сохранял свою гибкость, следует соблюдать два условия:

- Не оговаривается количество изображений - не должно иметь значения, сколько их используется, 10 или 100
- Не оговаривается порядок следования изображений - должна существовать возможность изменять этот порядок без изменения самого кода

(online-версия руководства позволит Вам проверить этот скрипт немедленно)

Рассмотрим скрипт (я внес туда некоторые комментарии):

```
<html>
<head>

<script language="JavaScript">
<!-- hide

// ****************************************************
// Script from Stefan Koch - Voodoo's Intro to JavaScript
//     http://rummelplatz.uni-mannheim.de/~skoch/js/
//      JS-book: http://www.dpunkt.de/javascript
//    You can use this code if you leave this message
// ****************************************************

 // ok, у нас браузер с поддержкой JavaScript
 var browserOK = false;
 var pics;

// -->
</script>

<script language="JavaScript1.1">
<!-- hide

 // браузер с поддержкой JavaScript 1.1!
 browserOK = true;
```

```
  pics = new Array();

// -->
</script>



<script language="JavaScript">
<!-- hide

var objCount = 0; // количество изображений на  web-странице

function preload(name, first, second) {

  // предварительна\я загрузка изображений и размещение их в массиве

  if (browserOK) {
    pics[objCount] = new Array(3);
    pics[objCount][0] = new Image();
    pics[objCount][0].src = first;
    pics[objCount][1] = new Image();
    pics[objCount][1].src = second;
    pics[objCount][2] = name;
    objCount++;
  }
}

function on(name){
  if (browserOK) {
    for (i = 0; i < objCount; i++) {
      if (document.images[pics[i][2]] != null)
        if (name != pics[i][2]) {
  // вернуть в исходное систо\яние все другие изображени\я
          document.images[pics[i][2]].src = pics[i][0].src;
} else {
  // показывать вторую картинку, поскольку курсор пересекает данное изображение
            document.images[pics[i][2]].src = pics[i][1].src;
        }
    }
  }
}

function off(){
  if (browserOK) {
    for (i = 0; i < objCount; i++) {
      // вернуть в исходное систо\яние все изображени\я
      if (document.images[pics[i][2]] != null)
          document.images[pics[i][2]].src = pics[i][0].src;
    }
  }
}

// заранее загружаемые изображени\я - Вы должны здесь указать
```

```
// изображени\я, которые нужно загрузить заранее, а также объект Image,
// к которому они относ\ятс\я (первый аргумент). Именно эту часть
// нужно корректировать, если Вы хотите использовать скрипт
// применительно к другим изображени\ям (конечно это не освобождает
// Вас от об\язанности подредактировать в документе также и раздел body)

preload("link1", "img1f.gif", "img1t.gif");
preload("link2", "img2f.gif", "img2t.gif");
preload("link3", "img3f.gif", "img3t.gif");

// -->
</script>
</head>


<body>
<a href="link1.htm" onMouseOver="on('link1')"
 onMouseOut="off()">
<img name="link1" src="img1f.gif"
 width="140" height="50" border="0"></a>

<a href="link2.htm" onMouseOver="on('link2')"
 onMouseOut="off()">
<img name="link2" src="img2f.gif"
 width="140" height="50" border="0"></a>

<a href="link3.htm" onMouseOver="on('link3')"
 onMouseOut="off()">
<img name="link3" src="img3f.gif"
 width="140" height="50" border="0"></a>
</body>
</html>
```

Данный скрипт помещает все изображения в массив pics. Создает этот массив функция *preload()*, которая вызвается в самом начале. Вызов функции *preload()* выглядит просто как:

```
preload("link1", "img1f.gif", "img1t.gif");
```

Это означает, что скрипт должен загрузить с сервера два изображения: img1f.gif и img1t.gif. Первое из них - это та картинка, которая будет представлена, пока курсор мыши не попадает в область изображение. Когда же пользователь помещает курсор мыши на изображение, то появляется вторая картинка. При вызове функции *preload()* в качестве первого аргумента мы указываем слово *"link1"* и тем самым задаем на web-странице объект Image, которому и будут принадлежать оба предварительно загруженных изображения. Если Вы посмотрите в нашем примере в раздел<*body*>, то обнаружите изображение с тем же именем *link1*. Мы пользуем не порядковый номер, а именно имя изображения для того, чтобы иметь возможность переставлять изображения на web-странице, не переписывая при этом сам скрипт.

Обе функции *on()* и *off()* вызываются посредством программ обработки событий *onMouseOver* и *onMouseOut*. Поскольку сам элемент image не может отслеживать события *MouseOver* и *MouseOut*, то мы обязаны сделать на этих изображениях еще и ссылки. можно видеть, что функция *on()* возвращает все изображения, кроме указанного, в исходное состояние. Делать это необходимо потому, что в противном случае выделеными могут

оказаться сразу несколько изображений (дело в том, что событие *MouseOut* не будет зарегистрировано, если пользователь переместит курсор с изображения сразу за пределы окна).

Изображения - без сомнения могучее средство уличшения Вашей web-страницы. Объект Image дает Вам возможность создавать действительно сложные эффекты. Однако заметим, что не каждое изображение или программа JavaScript способно улучшить Вашу страницу. Если Вы пройдетесь по Сети, то сможете увидеть множество примеров, где изображения использованы самым ужасным способом. Не количество изображений делает Вашу web-страницу привлекательной, а их качество. Сама загрузка 50 килобайт плохой графики способна вызвать раздражение.

Лекция 10: Слои I

Что такое слои?

Слои - это одна из замечательных новых возможностей браузера Netscape Navigator 4.0. Она позволяет выполнять точное позиционирование таких объектов web-страницы, как изображения. Кроме того, теперь Вы можете перемещать объекты по вашей HTML-странице. Вы можете также делать объекты невидимыми.

Управлять слоями можно легко с помощью языка JavaScript. Я надеюсь, что Вы, как и я, проникнитесь энтузиазмом в отношении возможностей этих слоев.

Что такое в действительности слои? Объяснять это очень легко на простом примере: если взять несколько листов бумаги. На одном листе пишете текст. На другом - рисуете картинку. На третьем листе берете картинку и вписываете вокруг нее некий текст. И так далее. Теперь кладете эти листы на стол. Допустим, каждый лист - это некий слой. В этом смысле слой представляет собой в некотором роде контейнер. То есть он может включать в себя (содержать) некие объекты - в данном случае это будет текст и изображения.

Теперь берем бумагу с изображением и двигаем его по столу. Внимательно следите за тем, как это изображение движется вслед за бумагой. Если Вы сместим лист бумаги вправо, то и изображение тоже сдвинется! И что же собственно мы должны извлечь из этого увлекательного опыта? Слои, способные содержать различные объекты, например изображения, формы, текст, могут быть наложены на Вашу HTML-страницу и даже перемещаться по ней. Если Вы сдвигаете какой-либо слой, то и все содержащиеся в этом слое объекты тоже будут повторять это движение. Слои могут накладываться друг на друга подобно простым листам бумаги на столе. В каждом слое могут присутствовать прозрачные области. Сделайте в листе бумаги дырку. Теперь положите этот лист на другой. Такое отверстие - это 'прозрачная область' на первом листе - и через нее как раз видны нижележащие листы.

Создание слоев

Чтобы создать слой, мы должны использовать либо тэг <*layer*> либо <*ilayer*>. Вы можете воспользоваться следующими параметрами:

Параметр	Описание
name="*layerName*"	Название слоя
left=*xPosition*	Абсцисса левого верхнего угла
top=*yPosition*	Ордината левого верхнего угла
z-index=*layerIndex*	Номер индекса для слоя
width=*layerWidth*	Ширина слоя в пикселах
clip="*x1_offset,y1_offset,x2_offset,y2_offset*"	Задает видимую область слоя
above="*layerName*"	Определяет, какой слой окажется

	под нашим
below="*layerName*"	Определяется, какой слой окажется над нашим
Visibility=show|hide|inherit	Видимость этого слоя
bgcolor="*rgbColor*"	Цвет фона - либо название стандартного цвета, либо rgb-запись
background="*imageURL*"	Фоновая картинка

Тэг <*layer*> используется для тех слоев, которые Вы можете точно позиционировать. Если же Вы не указываете положение слоя (с помощью параметров *left* и *top*), то по умолчанию он помещается в верхний левый угол окна.

Тэг <*ilayer*> создает слой, положение которого определяется при формировании документа.

Давайте теперь начнем с простого примера. Мы хотим создать два слоя. В первом из них мы помещаем изображение, а во втором - текст. Все, что мы хотим сделать - показать этот текст поверх данного изображения.

Текст поверх изображения

Исходный код:

```
<html>

<layer name=pic z-index=0 left=200 top=100>
<img src="img.gif" width=160 height=120>
</layer>

<layer name=txt z-index=1 left=200 top=100>
<font size=+4> <i> Layers-Demo </i> </font>
</layer>

</html>
```

Как видим, с помощью тэга <*layer*> мы формируем два слоя. Оба слоя позиционируются как 200/100 (через параметры *left* и *top*). Все, что находится между тэгами <*layer*> и </*layer*> (или тэгами <*ilayer*> и </*ilayer*>) принадлежит описываемому слою.

Кроме того, мы используем параметр *z-index*, определяя тем самым порядок появления указанных слоев - то есть, в нашем случае, Вы тем самым сообщаете браузеру, что текст будет написан поверх изображения. В общем случае, именно слой с самым высоким номером z-index будет показан поверх всех остальных. Вы не ограничены в выборе z-index лишь значениями 0 и 1 - можно выбирать вообще любое положительное число.

Так, если в первом тэге <*layer*> Вы напишете *z-index=100*, то текст окажется под изображением - его слой номер Z-индекса (*z-index=1*). Вы сможете увидеть текст сквозь изображение, поскольку я использовал в нем прозрачный фон (формат gif89a).

Тект под изображением

Слои и JavaScript

Рассмотрим теперь, как можно получить доступ к слоям через JavaScript. Начнем же мы с примера, где пользователь получает возможность, нажимая кнопку, прятать или показать некий слой.

Для начала мы должны знать, каким образом слои представлены в JavaScript. Как обычно, для этого имеются несколько способов. Самое лучшее - дать каждому слою свое имя. Так, если мы задаем слой

```
<layer ... name=myLayer>
...
</layer>
```

то в дальнейшем можем получить доступ к нему с помощью конструкции *document.layers["myLayer"]*. Согласно документации, предоставляемой фирмой Netscape, мы можем также использовать запись document.myLayer - однако в моем браузере это приводит к сбою. Конечно, это всего лишь проблема предварительной версии и в заключительном варианте будет успешно решена (сейчас я пользуюсь Netscape Navigator 4.0 PR3 на WinNT). Однако, по-видимому, нет никаких проблем с конструкцией *document.layers["myLayer"]* - поэтому мы и будем пользоваться именно такой альтернативой из всех возможных.

Доступ к этим слоям можно также получить через целочисленный индекс. Так, чтобы получить доступ к самому нижнему слою, Вы можете написать *document.layers[0]*. Обратите внимание, что индекс - это **не** то же самое, что параметр *z-index*. Если, например, Вы имеете два слоя, называемые layer1 и layer2 с номерами z-index 17 и 100, то Вы можете получить доступ к этим слоям через *document.layers[0]* и *document.layers[1]*, а **не** через *document.layers[17]* и *document.layers[100]*.

Слои имеют несколько свойств, которые можно изменять с помощью скрипта на JavaScript. В следующем примере представлена кнопка, которая позволяет Вам скрывать или, наоборот, предоставлять один слой (требуется Netscape Navigator версии 4.0 или выше).

(online-версия руководства позволит Вам проверить этот скрипт немедленно)

Исходный код скрипта выглядит следующим образом:

```
<html>
<head>
<script language="JavaScript">
<!-- hide

function showHide() {
  if (document.layers["myLayer"].visibility == "show")
    document.layers["myLayer"].visibility= "hide"
  else document.layers["myLayer"].visibility= "show";
```

```
}

// -->
</script>
</head>
<body>

<ilayer name=myLayer visibility=show>
<font size=+1 color="#0000ff"><i>This text is inside a layer</i></font>
</ilayer>

<form>
<input type="button" value="Show/Hide layer" onClick="showHide()">
</form>

</body>
</html>
```

Данная кнопка вызывает функцию *showHide()*. можно видеть, что в этих функциях реализуется доступ к такому свойству объекта layer (myLayer), как видимость. Присвоивая параметру *document.layers["myLayer"]*.visibility значения *"show"* или *"hide"*, Вы можете показать или скрыть наш слой. Заметим, что *"show"* и *"hide"* - это строки, а не зарезервированные ключевые слова, то есть Вы **не можете** написать *document.layers["myLayer"].visibility= show*. Вместо тэга *<layer>* я также пользовался тэгом *<ilayer>*, поскольку хотел поместить этот слой в "информационный поток" документа.

Перемещение слоев

Свойства left и top определяют задают положение данного слоя. Вы можете менять его, записывая в эти атрибуты новые значения. Например, в следующей строке задается горизонтальное положение слоя в 200 пикселов:

```
document.layers["myLayer2"].left= 200;
```

Перейдем теперь к программе перемещения слоев - она создает нечто вроде линейки прокрутки внутри окна браузера.

(online-версия руководства позволит Вам проверить этот скрипт немедленно)

Сам скрипт выглядит следующим образом:

```
<html>
<head>
<script language="JavaScript">
<!-- hide

function move() {
  if (pos < 0) direction= true;
  if (pos > 200) direction= false;

  if (direction) pos++
    else pos--;

  document.layers["myLayer2"].left= pos;
}
```

```
// -->
</script>
</head>
<body onLoad="setInterval('move()', 20)">

<ilayer name=myLayer2 left=0>
<font size=+1 color="#0000ff"><i>This text is inside a layer</i></font>
</ilayer>

</body>
</html>
```

Мы создаем слой с именем *myLayer2*. можно видеть, что в тэге *<body>* мы пользуемся процедурой onLoad. Нам необходимо начать прокручивание слоя, как только страница будет загружена. В процедуре обработки события onLoad мы пользуемся функцией *setInterval()*. Это один из новых методов версии 1.2 языка JavaScript (то есть версии JavaScript, реализованной в Netscape Navigator 4.0). Им можно пользоваться, чтобы вызывать некую функцию мвновь и вновь через определенные интервалы времени. В прошлом для этого мы пользовались функцией *setTimeout()*. Функция *setInterval()* работает почти так же, однако Вам нужно вызвать ее всего лишь один раз.

С помощью *setInterval()* мы вызываем функцию *move()* каждые 20 миллисекунд. А функция *move()*, в свою очередь, всякий раз смещает слой на новую позицию. И поскольку мы вызываем эту функцию вновь и вновь, то мы получаем быстрый скроллинг нашего текста. Все, что мы нужно сделать в функции *move()* - это вычислить новую координату для слоя и записать ее: *document.layers["myLayer2"].left= pos*.

Если посмотреть исходный код этой части в онлайновом описании, то можно увидеть, что в действительности код выглядит несколько иначе - был добавлен некий фрагмент кода с тем, чтобы люди, работающие со старыми версиями JavaScript-браузеров, не получали из-за этого никаких сообщений об ошибках. Как этого можно достичь? Следующий фрагмент кода будет выполняться только на тех браузерах, которые воспонимают язык JavaScript 1.2:

```
<script language="JavaScript1.2">
<!-- hide
document.write("You are using a JavaScript 1.2 capable browser.");
// -->
</script>
```

Та же самая проблема возникает, когда мы работаем с объектом Image. Мы можем аналогичным способом переписать кодекс. Установка переменной *browserOK* решает эту проблему.

Лекция 11: Слои II

Мы уже обсудили основные понятия новой технологии слоев. В этой же части будут рассмотрены следующие темы:

- Вырезка из слоя
- Вложенные слои
- Различные эффекты с прозрачными слоями

Вырезка из слоя

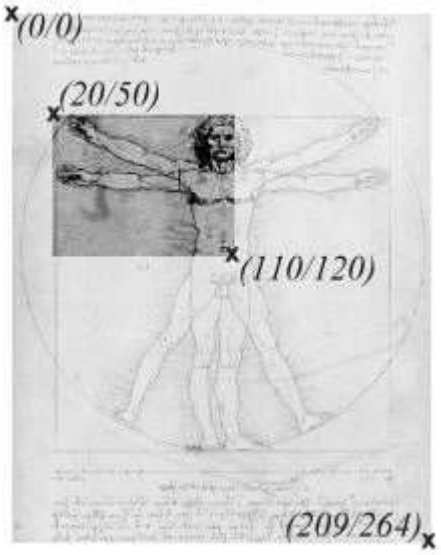
можно постулировать, что какая-то (прямоугольная) часть слоя будет нам видима. Все же, что лежит за ее пределами, показано на экране не будет. Такой прием называется вырезанием. Например, в разметке HTML можно задать следующую функцию вырезания:

```
<ilayer left=0 top=0 clip="20,50,110,120">
<img src="davinci.jpg" width=209 height=264>
</ilayer>
```

(Здесь приписаны параметры *left=0* и *top=0*, поскольку в противном случае, если этого не сделать, то с версией Netscape (PR3 on WinNT) возникают некоторые проблемы)

Хотя само изображение и имеет размеры 209x264 пикселов, мы можем видеть лишь его малую часть:

Данный фрагмент изображения имеет размер 90x70 (пикселов). Первые два значения, указанные в атрибуте clip (атрибуте HTML-тэга <layer> или <ilayer>), указывают верхний левый угол вырезаемой части. Следующие два значения указывают нижний правый угол. Сказанное можно проиллюстрировать следующим рисунком:



Еще более интересных результатов можно добиться, управляя вырезанной частью с помощью языка JavaScript. Точнее, Вы можете изменять значения свойств clip.left, clip.top, clip.right и clip.bottom объекта Layer. Достаточно всего лишь занести в одно из этих свойств новое значение, как фрагмент тут же будет кадрирован соответствующим образом . В следующем примере параметры вырезанной части изображения меняются динамически, и в результате у пользователя создается впечатление, будто изображение медленно "растет":

Код соответсвующего скрипта:

```
<html>
<head>

<script language="JavaScript">
<!-- hide

var middleX, middleY, pos;

function start() {
  // получить размер изображени\я
```

```
  var width= document.layers["imgLayer"].document.davinci.width;
  var height= document.layers["imgLayer"].document.davinci.height;

  // определить, какой пиксел находитс\я в центре изображени\я
  middleX= Math.round(width/2);
  middleY= Math.round(height/2);

  // начальная позици\я
  pos= 0;

  // запуск!
  show();
}

function show() {

  // увеличить размер вырезаемой области
  pos+= 2; // величина  шага
  document.layers["imgLayer"].clip.left= middleX- pos;
  document.layers["imgLayer"].clip.top= middleY- pos;
  document.layers["imgLayer"].clip.right= middleX+ pos;
  document.layers["imgLayer"].clip.bottom= middleY+ pos;

  // проверить, не высвечено ли все изображение
    if (!((pos > middleX) && (pos > middleY)))
    setTimeout("show()", 20);

}

// -->
</script>
</head>

<body>

<ilayer name="imgLayer" clip="0,0,0,0">
<img name=davinci src="davinci.jpg" width=209 height=264>
</ilayer>

<form>
<input type=button value="Start" onClick="start()">
</form>

</body>
</html>
```

Кнопка, представленная в разделе *<body>*, вызывает функцию *start()*. Сначала мы должны определить точку, с которой нам следует начать работу - фактически это будет некий пиксел в центре нашего изображения. Значения координат x и y этого пиксела мы помещаем в переменные *middleX* и *middleY*. После этого мы вызываем функцию *show()*, которая задает размеры вырезаемой части изображения в зависимости от значений переменных *middleX*, *middleY* и параметра *pos*. При этом значение переменной *pos* автоматически увеличивается при каждом вызове функции *show()*. То есть размер вырезаемой части изображения с каждым разом становится все больше и больше. В самом конце процедуры *show()* мы устанавливаем таймер с помощью вызова setTimeout () - и благодаря этому функция *show()* вызывается вновь и вновь. И этот процесс остановится только тогда, когда изображение будет показано целиком.

Заметим, что размер изображения мы получаем в самом начале функции *start()*:

```
var width= document.layers["imgLayer"].document.davinci.width;
var height= document.layers["imgLayer"].document.davinci.height;
```

С помощью конструкции *document.layers["imgLayer"]* мы можем обратиться к слою с именем *imgLayer*. Однако почему после *document.layers["imgLayer"]* мы ставим *document*? Дело в том, что каждый слой имеет свою собственную HTML-страницу - то есть, **каждый слой получает имеет объект document**. Чтобы получить доступ к изображению внутри слоя *imgLayer,* нам необходимо получить доступ к этому объекту document. В приведенном выше примере такое изображение носило название *davinci*. Все остальное поле листа должно быть чистым.

Вложенные слои

Как мы уже видели, слой может содержать несколько различных объектов. Он могут даже включать в себя другие слои. Конечно, может возникнуть вопрос, для чего это нужно. На самом деле есть несколько причин, чтобы пользоваться вложенными слоями. Рассмотрим несколько примеров, демонстрирующих применение вложенных слоев.

В первом примере используется слой (называемый *parentLayer*), в который вложено еще два других слоя (*layer1* и *layer2*).

После открытия мы видим три кнопки. Эти кнопки могут запускать и останавливать движение слоев. Также можно видеть, что перемещение слоя parentLayer сопровождается перемещением и двух других слоев, тогда как перемещение слоя layer1 (или layer2) ни на что другое не влияет. Этот пример демонстрирует возможность объединения группы объектов с помощью механизма вложенных слоев.

Рассмотрим теперь исходный код скрипта:

```
<html>
<head>

<script language="JavaScript">
<!-- hide

// начальна\я позици\я
var pos0= 0;
var pos1= -10;
var pos2= -10;

// движение?
var move0= true;
var move1= false;
var move2= false;

// направление?
var dir0= false;
var dir1= false;
var dir2= true;

function startStop(which) {
  if (which == 0) move0= !move0;
  if (which == 1) move1= !move1;
  if (which == 2) move2= !move2;
}

function move() {

  if (move0) {
    // move parentLayer
```

```
    if (dir0) pos0--
      else pos0++;

    if (pos0 < -100) dir0= false;

    if (pos0 > 100) dir0= true;

    document.layers["parentLayer"].left= 100 + pos0;
   }

   if (move1) {
    // перемещение parentLayer
    if (dir1) pos1--
      else pos1++;

    if (pos1 < -20) dir1= false;

    if (pos1 > 20) dir1= true;

    document.layers["parentLayer"].layers["layer1"].top= 10 + pos1;
   }

   if (move2) {
    // перемещение parentLayer
    if (dir2) pos2--
      else pos2++;

    if (pos2 < -20) dir2= false;

    if (pos2 > 20) dir2= true;

    document.layers["parentLayer"].layers["layer2"].top= 10 + pos2;
   }

  }

  // -->
  </script>
  </head>

  <body onLoad="setInterval('move()', 20)">

  <ilayer name=parentLayer left=100 top=0>
   <layer name=layer1 z-index=10 left=0 top=-10>
    Это первый слой
   </layer>

   <layer name=layer2 z-index=20 left=200 top=-10>
Это второй слой
   </layer>

   <br><br>
   Это главный (родительский) слой
  </ilayer>

  <form>
```

```
<input type="button" value="Move/Stop parentLayer" onClick="startStop(0);">
<input type="button" value="Move/Stop layer1" onClick="startStop(1);">
<input type="button" value="Move/Stop layer2" onClick="startStop(2);">
</form>

</body>
</html>
```

можно видеть, что внутри parentLayer мы определили два слоя. Это как раз и есть вложенные слои. Как получить к этим слоям доступ в языке JavaScript? Как это делается, можно посмотреть в функции *move()*:

```
document.layers["parentLayer"].left= 100 + pos0;
...
document.layers["parentLayer"].layers["layer1"].top= 10 + pos1;
...
document.layers["parentLayer"].layers["layer2"].top= 10 + pos2;
```

Чтобы получить доступ к вложенным слоям, Вам недостаточно будет просто написать *document.layers["layer1"]* или *document.layers["layer2"],* поскольку слои *layer1* и *layer2* лежат внутри *parentLayer*.

Посмотрим теперь, как можно задать выделяемую область. В следующем примере используется механизм вырезания и перемещение изображения. Чего этим мы хотим достичь - чтобы вырезаемая часть была зафиксирована, т.е. чтобы при перемещении всего изображения не происходила смена видимого на экране фрагмента.

(online-версия руководства позволит Вам проверить этот скрипт немедленно)

Исходный код скрипта:

```
<html>
<head>

<script language="JavaScript">
<!-- hide

var pos= 0; // начальное положение
var direction= false;

function moveNclip() {

  if (pos<-180) direction= true;
  if (pos>40) direction= false;

  if (direction) pos+= 2
    else pos-= 2;

  document.layers["clippingLayer"].layers["imgLayer"].top= 100 + pos;

}

// -->
</script>

</head>
```

```
<body onLoad="setInterval('moveNclip()', 20);">

<ilayer name="clippingLayer" z-index=0 clip="20,100,200,160" top=0 left=0>
<ilayer name="imgLayer" top=0 left=0>
<img name=davinci src="davinci.jpg" width=209 height=264>
</ilayer>
</ilayer>

</body>
</html>
```

И снова, можно видеть пример обращения к вложенному слою:

```
document.layers["clippingLayer"].layers["imgLayer"].top= 100 + pos;
```

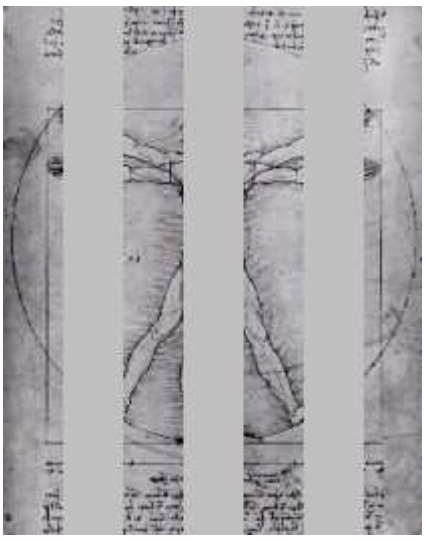
С остальными элементами этого скрипта Вы уже должны быть знакомы.

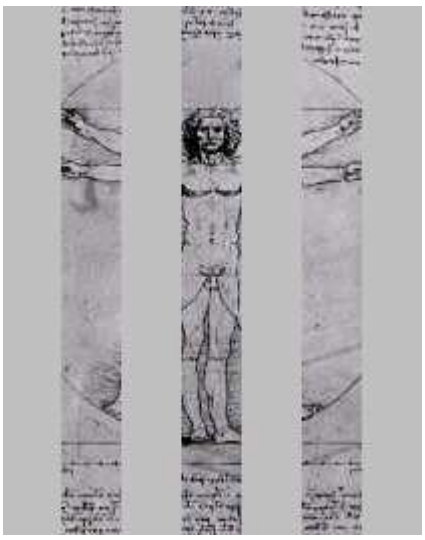
Различные эффекты с вложенными слоями

Интересные эффекты могут быть созданы с помощью (частично) прозрачных слоев. Сочетание специально подобранных изображений с прозрачными областями может создавать совершенно потрясающий результат. Не все форматы изображений поддерживают работу с прозрачными частями. В настоящее время лучший из отвечающих этому условию форматов - gif89a. Большинство новых графических программ поддерживает этот формат. Помимо этого, в Internet доступны некоторые свободно распространяемые инструменты для работы с графикой.

Новый формат изображений PNG также поддерживает эффект прозрачных частей изображения. Я полагаю, что в ближайшем будущем мы увидим множество страниц, использующих этот формат (точнее, как только большинство браузеров смогут его поддерживать). По сравнению с gif этот формат имеет множество преимуществ.

В данном примере используются два изображения (сплошные серые зоны здесь на самом деле являются прозрачными):





Сам скрипт не сильно отличается от других примеров - так что я не буду здесь его распечатывать (впрочем, конечно можно увидеть его, выбрав в меню браузера пункт *'View document source*').

Лекция 12: Модель событий в JavaScript 1.2

Новые события

Наступило время, рассмотреть одну из новых особенностей Netscape Navigator 4.x - модель событий JavaScript 1.2. Приведенные здесь примеры будут работать только в Netscape Navigator 4.x (хотя большинство из них работают также и в предварительных версиях этого браузера).

В JavaScript 1.2 поддерживается обработка следующих событий (если Вы хотите узнать побольше об этих событиях, обратитесь к документации JS 1.2 от фирмы Netscape -

http://developer.netscape.com/library/documentation/communicator/jsguide/js1_2.htm):

Abort	Focus	MouseOut	Submit
Blur	KeyDown	MouseOver	Unload
Click	KeyPress	MouseUp	
Change	KeyUp	Move	
DblClick	Load	Reset	
DragDrop	MouseDown	Resize	
Error	MouseMove	Select	

Изучая таблицу, можете увидеть, что была реализована обработка некоторых новых событий. В этой лекции рассмотрим некоторые из них.

Рассмотрим событие Resize. С помощью этого события можно определить, был бы размер окна изменен читателем. Следующий скрипт демонстрирует, как это делается:

```
<html>
<head>
<script language="JavaScript">

window.onresize= message;

function message() {
  alert("The window has been resized!");
}

</script>
</head>
<body>
Пожалуйста, измените размер этого окна.
</body>
</html>
```

В строке

```
window.onresize= message;
```

мы задаем процедуру обработки такого события. Точнее, функция message() будет вызываться всякий раз, как только пользователь изменит размер окна. Возможно, Вы не знакомы с таким способом назначения программ, обрабатывающих события. Однако JavaScript 1.2 ничего нового здесь не привносит. Например, если у Вас есть объект button, то можно определить процедуру обработки события следующим образом:

```
<form name="myForm">
<input type="button" name="myButton" onClick="alert('Click event occured!')">
</form>
```

Однако можно написать это и по другому:

```
<form name="myForm">
<input type="button" name="myButton">
</form>

...

<script language="JavaScript>

document.myForm.myButton.onclick= message;

function message() {
  alert('Click event occured!');
}

</script>
```

можно подумать, что вторая альтернатива немного сложнее. Однако почему тогда именно ее мы используем в первом скрипте? Причина состоит в том, что объект window нельзя определить через какой-либо определенный тэг - поэтому нам и приходится использовать второй вариант.

Два важных замечания: Во-первых, Вам не следует писать *window.onResize* - я имею в виду, что Вы должны писать все прописными буквами. Во-вторых, Вы не должны ставить после сообщения никаких скобок. Если Вы напишете *window.onresize= message(),* то браузер интерпретирует *message()* как вызов функции. Однако в нашем случае мы не хотим напрямую вызывать эту функцию - мы лишь хотим определить обработчик события.

Объект Event

В язык JavaScript 1.2 добавлен новый объект Event. Он содержит свойства, описывающие некое событие. Каждый раз, когда происходит какое-либо событие, объект Event передается соответствующей программе обработки.

В следующем примере на экран выводится некое изображение. можно щелкнуть где-нибудь над ним клавишей мыши. В результате появится окошко сообщений, где будут показаны координаты той точки, где в этот момент находилась мышь.

Код скрипта:

```
        <layer>
        <a href="#" onClick="alert('x: ' + event.x + ' y: ' + event.y); return false;">
<img src="davinci.jpg" width=209 height=264 border=0></a>
        </layer>
```

Как видите, в тэг <a> мы поместили программу обработки событий onClick, как это мы уже делали в предшествующих версиях JavaScript. Новое здесь заключается в том, что для создания окошка с сообщением мы используем event.x и event.y. А это как раз и есть объект Event, который здесь нам нужен, чтобы узнать координаты мыши. К тому же я поместил все

команды в тэг <*layer*>. Благодаря этому мы получаем в сообщении координаты относительно данного слоя, т.е. в нашем случае относительно самого изображения. В противном же случае мы получили бы координаты относительно окна браузера. (инструкция *return false;* используется здесь для того, чтобы браузер обрабатывал далее данную ссылку)

Объект Event получил следующие свойства (их мы рассмотрим в следующих примерах):

Property	**Description**
Data	Массив адресов URL оставленных объектов, когда происходит событие *DragDrop*.
LayerX	Горизонтальное положение курсора (в пикселах) относительно слоя. В комбинации с событием Resize это свойство представляет ширину окна браузера.
LayerY	Вертикальное положение курсора (в пикселах) относительно слоя. В комбинации с событием Resize это свойство представляет высоту окна браузера.
modifiers	Строка, задающая ключи модификатора - *ALT_MASK*, *CONTROL_MASK*, *META_MASK* or *SHIFT_MASK*
pageX	Горизонтальное положение курсора (в пикселах) относительно окна браузера.
pageY	Вертикальное положение курсора (в пикселах) относительно окна браузера.
screenX	Горизонтальное положение курсора (в пикселах) относительно экрана.
screenY	Вертикальное положение курсора (в пикселах) относительно экрана.
target	Строка, представляющая объект, которому исходно было послано событие.
type	Строка, указывающая тип события.
which	ASCII-значение нажатой клавиши или номер клавиши мыши.
x	Синоним *layerX*.
y	Синоним *layerY*.

Перехват события

Одна из важных особенностей языка - перехват события. Если кто-то, к примеру, щелкает на кнопке, то вызывается программа обработки события onClick, соответствующая этой кнопке. С помощью обработки событий можно добиться того, чтобы объект, соответсвующий вашему окну, документу или слою, перехватывал и обрабатывал событие еще до того, как для этой цели объектом указанной кнопки будет вызван обработчик событий. Точно так же объект вашего окна, документа или слоя может обрабатывать сигнал о событии еще до того, как он достигает своего обычного адресата.
Чтобы увидеть, для чего это может пригодиться, давайте рассмотрим следующий пример:

```
<html>
<head>
<script language="JavaScript">

window.captureEvents(Event.CLICK);

window.onclick= handle;

function handle(e) {
```

```
  alert("Объект window перехватывает это событие!");
  return true; // т.е. проследить ссылку
}

</script>
</head>
<body>
<a href="test.htm">Click on this link</a>
</body>
</html>
```

(online-версия позволит Вам проверить этот скрипт немедленно)

Как видно, мы не указываем программы обработки событий в тэге <a>. Вместо этого мы пишем

```
window.captureEvents(Event.CLICK);
```

с тем, чтобы перехватить событие *Click* объектом window. Обычно объект window не работает с событием *Click*. Однако, перехватив, мы затем его переадресуем в объект window. Заметим, что в *Event.CLICK* фрагмент *CLICK* должен писаться заглавными буквами. Если же Вы хотите перехватывать несколько событий, то Вам следует отделить их друг от друга символами |. Например:

```
window.captureEvents(Event.CLICK | Event.MOVE);
```

Помимо этого в функции *handle()*, назначенной нами на роль обработчика событий, мы пользуемся инструкцией *return true;*. В действительности это означает, что браузер должен обработать и саму ссылку, после того, как завершится выполнение функции *handle()*. Если же Вы напишете вместо этого *return false;*, то на этом все и закончится.

Если теперь в тэге *<a>* Вы зададите программу обработки события *onClick*, то поймете, что данная программа при возникновении данного события вызвана уже не будет. И это не удивительно, поскольку объект window перехватывает сигнал о событии еще до того, как он достигает объекта link. Если же Вы определите функцию *handle()* как

```
function handle(e) {
  alert("The window object captured this event!");
  window.routeEvent(e);
  return true;
}
```

то компьютер будет проверять, определены ли другие программы обработки событий для данного объекта. Переменная e - это наш объект Event, передаваемый функции обработки событий в виде аргумента.

Кроме того, можно непосредственно послать сигнал о событии какому-либо объекту. Для этого можно воспользоваться методом *handleEvent()*. Это выглядит следующим образом:

```
<html>
<script language="JavaScript">

window.captureEvents(Event.CLICK);
```

```
window.onclick= handle;

function handle(e) {
  document.links[1].handleEvent(e);
}

</script>
<a href="test.htm">"Кликните" по этой ссылке</a><br>
<a href="test.htm"
  onClick="alert('Обработчик событий для второй ссылки!');">Вторая ссылка</a>
</html>
```

Все сигналы о событиях Click, посылаются на обработку по второй ссылке - даже если Вы вовсе и не щелкнули ни по одной из ссылок!

Следующий скрипт демонстрирует, как скрипт может реагировать на сигналы о нажатии клавиш. Нажмите на какую-либо клавишу и посмотрите, как работает этот скрипт.

```
<html>
<script language="JavaScript">

window.captureEvents(Event.KEYPRESS);

window.onkeypress= pressed;

function pressed(e) {
  alert("Key pressed! ASCII-value: " + e.which);
}

</script>
</html>
```

Лекция 13: Drag & Drop

Что такое drag & drop?

С помощью новой модели событий в языке JavaScript, 1.2 и механизма слоев мы можем реализовать на нашей web-странице схему drag & drop ("перетащил и оставил"). Для этого Вам понадобится по крайней мере Netscape Navigator 4.0, поскольку мы будем пользоваться особенностями языка JavaScript 1.2.

Что такое drag & drop? Например, некоторые операционные системы (такие как Win95/NT или MacOS) позволяют стирать файлы, просто перетаскивая их в мусорную корзину. Иными словами, Вы щелкаете клавишей мыши над изображением файла, перетаскиваете его (то есть держите клавишу нажатой и просто двигаете мышь) - drag - в мусорную корзину, а затем отпускаете - drop - его там.

Механизм drag & drop, который мы хотим здесь реализовать, ограничивается web-страницей. Поэтому Вы не можете использовать представленный здесь код, чтобы переносить объекты с HTML-страницы на жесткий диск вашего компьютера или другие подобные действия. (Начиная с версии 4.0 браузера Netscape Navigator ваш скрипт может реагировать на

событие с названием DragDrop, событие, когда кто-либо перетаскивает файл на окно вашего браузера. Но это не совсем то, о чем мы здесь хотим поговорить)

Язык JavaScript не поддерживает напрямую механизм drag & drop. Это значит, что у нас нет возможности назначить объекту image свойство dragable (перемещаемый) или что-либо в этом роде. Поэтому мы должны сами писать необходимый для этого код. Впрочем, Вы увидите, что это не так сложно.

Итак, то же нам нужно? Нам нужны две вещи. Во-первых, мы должны регистрировать определенные события, связанные с работой мышью, то есть нужно понять, каким образом, мы сможем узнать, какой объект необходимо переместить и на какую позицию? Затем нам нужно подумать, каким именно образом мы сможем показывать перемещение объектов по экрану. Конечно же, мы будем пользоваться такой новой возможностью языка, как слои, при создании объектов и перемещении их по экрану. Каждый объект представлен собственным слоем.

События при работе с мышью в JavaScript 1.2

Какие события, происходящие при работе с мышью, нам следует использовать? У нас нет такого события, как MouseDrag, однако того же самого мы можем достичь, отслеживая события *MouseDown*, *MouseMove* и *MouseUp*. В версии 1.2 языка JavaScript используется новая модель событий. И без нее мы не смогли бы решить нашу задачу. Я уже говорил об этой новой модели на предыдущем уроке. Однако давайте взглянем на некоторые важные ее части еще раз.

Пользователь нажал клавишу мыши в каком-либо месте на окне браузера. Наш скрипт должен зафиксировать это событие и вычислить, с каким объектом (то есть слоем) это было связано. Нам необходимо знать координаты точки, где произошло это событие. В JavaScript 1.2 реализован новый объект Event, который сохраняет координаты этой точки (а также еще и другую информацию о событии).

Другой важной момент заключается в перехвате событий. Если пользователь, например, щелкает по клавише мыши, то сигнал о соответствующем событии посылается непосредственно объекту button. Однако в нашем примере необходимо, чтобы событие обрабатывалось объектом window (окно). Поэтому мы позволяем объекту окна перехватывать сигнал о событии, связанном с мышью, т.е. чтобы именно объект window фиксировал это событие и имел возможность на него реагировать. Это демонстрируется в следующем примере (на примере события *Click*). можно щелкнуть в любом месте окна браузера. При этом возникнет окно сообщения, где будут показаны координаты точки, где это событие имело место.

Код этого примера:

```
<html>

<script language="JavaScript">
<!--

 window.captureEvents(Event.CLICK);

 window.onclick= displayCoords;


 function displayCoords(e) {
   alert("x: " + e.pageX + " y: " + e.pageY);
 }

// -->
</script>

Щелкните клавишей мыши где-нибудь в окне браузера.

</html>
```

Сперва мы сообщаем, что объект window перехватывает сигнал о событии *Click*. Для этого мы пользуемся методом *captureEvent()*. Строка

```
window.onclick= displayCoords;
```

говорит о том, что должно происходить, когда случается событие *Click*. Конкретнее, здесь сообщается, что в качестве реакции на событие *Click* браузер должен вызвать процедуру *displayCoords()* (Заметим, что Вам при этом нельзя ставить скобки после слова *displayCoords*). В свою очередь, *displayCoords()* - это функция, которая определяется следующим образом:

```
function displayCoords(e) {
  alert("x: " + e.pageX + " y: " + e.pageY);
}
```

Как видно, эта функция имеет аргумент (мы назвали его e). На самом деле это объект Event, который передается на обработку функции *displayCoords()*. Объект Event имеет свойства *pageX* и *pageY* (наряду с другими), из которых моно получить координаты точки, где произошло событие. Окно с сообщением лишь показывает эти значения.

MouseDown, MouseMove и MouseUp

В языке JavaScript нет события *MouseDrag*. Поэтому мы должны пользоваться событиями *MouseDown*, *MouseMove* и *MouseUp*, реализуя механизм drag & drop. В следующем примере демонстрируется применение MouseMove - текущие координаты курсора мыши отображаются в окне состояния.
можно видеть, что код скрипта почти такой же, как и в предыдущем примере:

```
<html>

<script language="JavaScript">
<!--

 window.captureEvents(Event.MOUSEMOVE);

 window.onmousemove= displayCoords;


 function displayCoords(e) {
   status= "x: " + e.pageX + " y: " + e.pageY;
 }

// -->
</script>
```

Координаты мыши показаны в строке состояния.

```
</html>
```

Заметим, что необходимо написать именно *Event.MOUSEMOVE*, где слово *MOUSEMOVE* обязательно должно быть написано заглавными буквами. А указывая, какая функция должна быть вызвана, когда произойдет событие *MouseMove*, Необходимо писать ее строчными буквами: *window.onmousemove=...*

Теперь мы можем объединить оба последних примера. Мы хотим, чтобы были представлены координаты указателя мыши, когда пользователь перемещает мышь, нажав на клавишу.
Код этого примера выглядит следующим образом:

```
<html>

<script language="JavaScript">
<!--

window.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP);

window.onmousedown= startDrag;
window.onmouseup= endDrag;
window.onmousemove= moveIt;

function startDrag(e) {
  window.captureEvents(Event.MOUSEMOVE);
}

function moveIt(e) {
  // показывать координаты
  status= "x: " + e.pageX + " y: " + e.pageY;
}

function endDrag(e) {
  window.releaseEvents(Event.MOUSEMOVE);
}


// -->
</script>

Двигайте мышь, удерживая ее клавишу нажатой. В окне состояния при этом
отображаются координаты курсора.

</html>
```

Во-первых, мы заставляем объект window перехватывать сигналы о событиях *MouseDown* and *MouseUp*:

```
window.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP);
```

Как видно, мы пользуемся символом | (или), чтобы сказать, что объект window должен перехватывать несколько указанных событий. Следующие две строки описывают, что именно должно происходить, когда указанные события имеют место:

```
window.onmousedown= startDrag;
window.onmouseup= endDrag;
```

В следующей строке кода определяется, что происходит, когда объект window получает сигнал о событии *MouseMove.*

```
window.onmousemove= moveIt;
```

Однако постойте, мы же не определили *Event.MOUSEMOVE* в *window.captureEvents()*! Это означает, что данное событие не будет перехватываться объектом window. Тогда почему мы указываем объекту window вызывать *moveIt()*, раз сигнал об этом событии никогда не достигает объекта window? Ответ на этот вопрос можно найти в функции *startDrag()*, которая вызывается сразу после того, как произойдет событие *MouseDown*:

```
function startDrag(e) {
  window.captureEvents(Event.MOUSEMOVE);
}
```

Это означает, что объект window начнет перехватывать событие *MouseMove*, как только будет нажата клавиша кнопка мыши. И мы должны прекратить перехватывать событие *MouseMove*, если произойдет событие *MouseUp*. Это делается в функции *endDrag()* с помощью метода *releaseEvents()*:

```
function endDrag(e) {
  window.releaseEvents(Event.MOUSEMOVE);
}
```

Функция moveIt() записывает координаты мыши в окно состояния.

Теперь у нс есть все элементы скрипта, необходимые для регистрации событий, связанных с реализацией механизма drag & drop. И мы можем приступить к рисованию на экране наших объектов.

Показ движущихся объектов

На предыдущих уроках мы видели, как с помощью слоев можно создать перемещающиеся объекты. Все, что мы должны теперь сделать - это определить, по какому именно слою пользователь щелкнул клавишей мыши. И затем этот объект должен двигаться вслед за мышью. Итак, код примера, показанного в начале этого урока:

```
<html>
<head>

<script language="JavaScript">
<!--

var dragObj= new Array();
var dx, dy;

window.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP);

window.onmousedown= startDrag;
window.onmouseup= endDrag;
window.onmousemove= moveIt;

function startDrag(e) {
  currentObj= whichObj(e);
  window.captureEvents(Event.MOUSEMOVE);
}
```

```
function moveIt(e) {
  if (currentObj != null) {
    dragObj[currentObj].left= e.pageX - dx;
    dragObj[currentObj].top= e.pageY - dy;
  }
}

function endDrag(e) {
  currentObj= null;
  window.releaseEvents(Event.MOUSEMOVE);
}

function init() {
  // задать 'перемещаемые' слои
  dragObj[0] = document.layers["layer0"];
  dragObj[1] = document.layers["layer1"];
  dragObj[2] = document.layers["layer2"];
}

function whichObj(e) {

  // определить, по какому объекту был произведен щелчок

  var hit= null;
  for (var i= 0; i < dragObj.length; i++) {
    if ((dragObj[i].left < e.pageX) &&
        (dragObj[i].left + dragObj[i].clip.width > e.pageX) &&
        (dragObj[i].top < e.pageY) &&
        (dragObj[i].top + dragObj[i].clip.height > e.pageY)) {
          hit= i;
          dx= e.pageX- dragObj[i].left;
          dy= e.pageY- dragObj[i].top;
          break;
    }
  }
  return hit;
}


// -->
</script>
</head>
<body onLoad="init()">

<layer name="layer0" left=100 top=200 clip="100,100" bgcolor="#0000ff">
<font size=+1>Object 0</font>
</layer>

<layer name="layer1" left=300 top=200 clip="100,100" bgcolor="#00ff00">
<font size=+1>Object 1</font>
</layer>

<layer name="layer2" left=500 top=200 clip="100,100" bgcolor="#ff0000">
```

```
<font size=+1>Object 2</font>
</layer>

</body>
</html>
```

можно видеть, что в разделе <*body*> нашей HTML-страницы мы определяем три слоя. После того, как была загружена вся страница, при помощи программы обработки события *onLoad*, указанной в тэге <*body*>, вызывается функция *init()*:

```
function init() {
  // define the 'dragable' layers
  dragObj[0] = document.layers["layer0"];
  dragObj[1] = document.layers["layer1"];
  dragObj[2] = document.layers["layer2"];
}
```

Массив *dragObj* влючает все слои, которые пользователь может перемещать. Каждый такой слой получает в множестве dragObj некий номер.можно видеть, что мы используем тот же самый код, что использовался ранее для перехвата событий, связанных с мышью:

```
window.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP);

window.onmousedown= startDrag;
window.onmouseup= endDrag;
window.onmousemove= moveIt;
```

К функции *startDrag()* я добавил следующую сроку:

```
currentObj= whichObj(e);
```

Функция *whichObj()* определяет, по какому объекту был произведен щелчок. Возвращает она номер соответствующего слоя. Если ни один слой не был выделен, то возвращается значение *null*. Полученное значение хранится в переменной *currentObj*. Это означает, что из *currentObj* можно извлечь номер слоя, который в данный момент необходимо перемещать (либо это будет *null*, если никакого слоя перемещать не надо).

В функции whichObj*()*для каждого слоя мы проверяем свойства *left*, *top*, *width* и *height*. По этим значеням мы и можем проверять, по которому из объектов пользователь щелкнул клавишей.

"Оставляемые" объекты

Теперь мы имеем все, что необходимо, чтобы реализовать механизм drag & drop. С помощью нашего скрипта пользователь может перемещать объекты по web-странице. Однако мы еще ничего не говорили об размещении перемещенных объектов. Предположим, Вы хотите создать онлайновый магазин. У нас есть несколько изделий, которые можно поместить в корзину. Пользователь должен переносить эти изделия в корзинку и оставлять их там. Это означает, что мы должны регистрировать моменты, когда пользователь опускает некий объект в корзину - иными словами, что он хочет купить его.

Какую часть кода мы должны изменить, чтобы сделать такое? Мы должны проверить, в какой месте оказался объект после того, как было зафиксировано событие *MouseUp* - то есть мы должны сделать некоторые добавления к функции *endDrag()*.Например мы могли бы проверять, попадает ли в этот момент курсор мыши в границы некого прямоугольника. Если это так, то Вы вызываете функцию, регистрирующую все изделия, которые необходимо

купить (например, можно поместить их в некий массив). Ну и после этого можно показывать это изделие уже в корзинке.

Реализации

Есть несколько путей для совершенствования нашего скрипта. Во-первых, мы могли бы изменять порядок следования слоев, как только пользователь щелкает клавишей мыши по какому-либо объекту. Иначе выглядело бы странным, если бы Вы перемещали объект, а он при этом прятался от Вас за окружающие предметы. Очевидно, что эту проблему можно решить, меняя лишь порядок следования слоев в функции *startDrag()*.

Контрольные вопросы промежуточной контрольной .

Вариант 1.

1) Опишите свойства класса TEXT в CSS
2) Опишите процедуру формирования набора фреймов

Вариант 2.

1) Как описать цвета в CSS?
2) Приведите пример навигации между фреймами.

Вариант 3.

1) Что такое потоки в Java?
2) Опишите свойства класса FONT в CSS.

Вариант 4.

1) Как установить гиперссылку?
2) Назначение комментариев Java.

Вариант 5.

1) Что такое аплеты Java?
2) Что такое CSS (Каскадная таблица стилей)

Вариант 6.

1) Опишите понятие динамическая Web-страница.
2) Роли сервера.

Вариант 7.

1) Приведите форматы используемых графических изображений в Web-технологиях
2) Свойства Color и Background в CSS.

Вариант 8.

1) Опишите свойства класса FONT в CSS.
2) Приведите пример описания маркированных и нумерованных списков.

Вариант 9.

1) Опишите процесс представления графического изображения в виде ссылки.
2) Как описать функции с помощью скриптов Java.

Вариант 10.
1) Какие тэги HTML используются для создания «бегущей строки».
2) Как описаны целые числа в Java.

Вариант 11.
1) Операции инкремента/декремента в Java.
2) Назначение мета-тэгов HTML?

Вариант 12.
1) Приведите пример создания вложенной таблицы.
2) В каких случаях указывается язык сценариев (<SCRIPT LANGUAGE”"JavaScript1.1">)

Вопросы для проведения итоговой контрольной работы по курсу Web-технологии

Вариант 1.
1. Опишите классы JavaScript.
2. Назначение тэга TITLE.
3. Опишите свойства класса TEXT в CSS

Вариант 2.
1.Приведите пример использования тега Style
2.С помощью каких тэгов осуществляют размещение таблицы в HTML(учитывая выравнивание)
3.Опишите иерархию объектов в модели документа DOM.

Вариант 3.
1. Приведите и опишите теги HTML для вставки графических объектов.
2. Как связаны объектная модель документа DOM с HTML?
3. Опишите свойства класса FONT в CSS.

Вариант 4.
1. Типы переменных JavaScript..
2. Что такое CSS (Каскадная таблица стилей)
3. Как установить гиперссылку?

Вариант 5.
1. Назначение тэга TITLE.
2. Назначение языка JavaScript.
3. Как описать цвета в CSS?

Вариант 6.
1. Как осуществляется связь JavaScript с HTML?
2. Приведите пример навигации между фреймами.
3. Приведите связь Web-технологий с сетевыми технологиями.

Вариант 7.
1.С помощью каких тэгов осуществляют размещение таблицы в HTML(учитывая выравнивание)
2.Опишите инструмент FORM (форма).
3.Опишите понятие динамическая Web-страница DHTML.

Вариант 8.
1. Перечислите типы данных JavaScript..
2. Как осуществляется преобразование целого в символ? (JavaScript.)
3. Свойства Color и Background в CSS.

Вариант 9.
1. Приведите пример создания обычной гиперссылки в HTML
2. С помощью какого скрипта можно указать выполнение задачи - синтаксис (в JavaScript.)
3. Опишите свойства класса FONT в CSS.

Вариант 10.
1. Как можно изменить цвет текста и фон в HTML документе
2. Как оформляют окна с сообщениями в JavaScript.
3. Как описать цвета в CSS?

Вариант 11.
1. Приведите пример описания маркированных и нумерованных списков.
2. Опишите свойства класса TEXT в CSS.
3. Назначение операторов JavaScript.

Вариант 12.
1.Что такое скролинг текста в HTML.
2. Как оформляют окна с сообщениями в JavaScript (ALERT; CONFIRM; PROMT)
3. Роли сервера.

Вариант 13.
1. Существуют ли запрещающие Javascript? (например - копирование с сайта)
2. Как можно изменить цвет текста и фон в HTML документе.
3. Что такое CSS (Каскадная таблица стилей)

Вариант 14.
1. Назначение JavaScript.
2. С помощью каких тэгов осуществляют размещение таблицы в HTML(учитывая выравнивание).
3. Опишите свойства класса FONT в CSS

Вариант 15.
1. Приведите связь Web-технологий с сетевыми технологиями.
2. Приведите пример навигации между фреймами
3. Как можно изменить цвет текста и фон в HTML документе.

Вариант 16.
1. Какие объекты существуют в JavaScript?
2. Приведите пример создания обычной гиперссылки в HTML.
3. Свойства Color и Background в CSS.

Вариант 17.
1. Назначение фреймов "неразборчиво написано".
2. Типы переменных JavaScript.
3. Опишите свойства класса TEXT в CSS

Вариант 18.

1. Приведите пример описания маркированных и нумерованных списков.
2. Опишите классы JavaScript.
3. Опишите свойства класса FONT в CSS.

Вариант 19.

1. Приведите и опишите теги HTML для вставки графических объектов.
2. Перечислите операторы используемые в JavaScript.
3. Опишите объектную модель документа DOM.