МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
КЫРГЫЗСКОЙ РЕСПУБЛИКИ

КЫРГЫЗСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ им. И. РАЗЗАКОВА

Кафедра «Программное обеспечение компьютерных систем»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторных работ по дисциплине «Web дизайн» по разделу «Программирование на JavaScript» для студентов специальности 552801.04 – «Программное обеспечение ВТ и АС»

БИШКЕК – 2011

«РАССМОТРЕНО»
на заседании кафедры
«Программное обеспечение
компьютерных систем»
Прот. № 3 от 7.09.2011 г.

«ОДОБРЕНО»
Методическим советом
ФИТ
Прот. №3 от 19.10.2011г.

УДК 519.682

Составители: СТАМКУЛОВА Г.К., КЫДЫРАЛИЕВ Н.Н.

Методические указания к выполнению лабораторных работ по дисциплине «Web дизайн» по разделу «Программирование на JavaScript» для студентов специальности 552801.04 – «Программное обеспечение ВТ и АС» / КГТУ им. И.Раззакова; сост.: Г.К.Стамкулова, Н.Н.Кыдыралиев. – Б.: ИЦ «Текник», 2011. – 35 с.

Работа посвящена для самостоятельного освоения программирования на языке JavaScript часть 3.

Предназначено для студентов специальности «Программное обеспечение вычислительной техники и автоматизированных систем» всех форм обучения.

Таблиц: 4.
Рисунков: 6.
Библиогр. 8 названий.

Рецензент д.ф.-м.н., профессор Салиев А.Б.

Тех. редактор ***Субанбердиева Н.Е.***

Подписано к печати 16.11.2011 г. Формат бумаги 60x84 1/16.
Бумага офс. Печать офс. Объем 2,25 п.л. Тираж 50 экз. Заказ 390. Цена 33 сом.

Бишкек, ул. Сухомлинова, 20. ИЦ "Текник" КГТУ им. И.Раззакова, т.: 54-29-43
e-mail: beknur@mail.ru

Содержание

Введение

JavaScript — объектно-ориентированный скриптовый язык программирования.

JavaScript обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.

Основные архитектурные черты: динамическая типизация, слабая типизация, автоматическое управление памятью, прототипное программирование, функции как объекты первого класса.

JavaScript обладает рядом свойств объектно-ориентированного языка, но реализованное в языке прототипирование обусловливает отличия в работе с объектами по сравнению с традиционными объектно-ориентированными языками. Кроме того, JavaScript имеет ряд свойств, присущих функциональным языкам — функции как объекты первого класса, объекты как списки, карринг, анонимные функции, замыкания — что придаёт языку дополнительную гибкость.

Несмотря на схожий с Си синтаксис, JavaScript по сравнению с языком Си имеет коренные отличия:

- объекты, с возможностью интроспекции;
- функции как объекты первого класса;
- автоматическое приведение типов;
- автоматическая сборка мусора;
- анонимные функции.

В языке отсутствуют такие полезные вещи , как:

- модульная система: JavaScript не предоставляет возможности управлять зависимостями и изоляцией областей видимости;
- стандартная библиотека: в частности, отсутствует интерфейс программирования приложений по работе с файловой системой, управлению потоками ввода/вывода, базовых типов для бинарных данных;
- стандартные интерфейсы к веб-серверам и базам данных;
- система управления пакетами, которая бы отслеживала зависимости и автоматически устанавливала их

Лабораторная работа № 1

Массивы

- объявление массивов;
- работа с массивами на примере несложного меню.

Как и многое другое, массив в JavaScript является объектом. Но это с точки зрения архитектуры языка. С нашей же точки зрения его можно представить как объединённую группу переменных, где мы можем работать как с каждой переменной в отдельности, так и со всей группой.

Массив (по-английски array) объявляется так:

```
имя_массива = new Array()
```

В скобках можно

а) указать количество элементов массива — **new Array(8);**

б) перечислить элементы массива (в кавычках и через запятую) — **new Array("эники", "беники", "ели", "вареники");**

Примечание: перечисляемые элементы массива являются строками, а не именами переменных. Поэтому необходимы кавычки и поэтому же можно не придерживаться правил имён и даже писать русскими буквами.

в) не указывать ничего (чтобы сделать назначения в дальнейшем).

У массива есть свойство **length** — длина, или, как говорят программисты, размерность. Это свойство указывает количество элементов массива. У массива с пустыми скобками размерность равна нулю.

Размерность можно динамически изменять. Определив «пустой» массив, можно потом присвоить значение и порядковый номер одному из его элементов. Как только мы это сделаем, изменится и размерность массива:

```
yoklmn = new Array()
yoklmn[3] = "tratata"
```

Обратите внимание на **квадратные скобки**, в которые заключается порядковый номер массива.

Теперь, даже если другие элементы не определены, массив имеет размерность в 4 элемента.

Первый элемент массива всегда имеет **нулевой номер**.

Теперь смотрите:

Длина пустого массива = = 0

Длина массива с одним (нулевым) элементом = = 1

Длина массива с двумя элементами ([0], [1]) = = 2

И т.д.

То есть **размерность массива всегда на один номер больше номера последнего элемента**.

Скопируйте этот код в **<body>** пустой страницы HTML и посмотрите, что получится.

```
<script type="text/javascript">
yoklmn = new Array()
yoklmn[3] = "tratata"
document.write(yoklmn.length)
</script>
```

Создание меню

Чтобы понять, как работают массивы, давайте создадим простенькое меню для домашней страницы, которое будет отображаться на всех страницах сайта.

Главная	О сайте	Обо мне	Ссылки	Гостевая книга

Вот так, предположим, выглядело бы это меню в коде HTML.

```
<style type="text/css">
.txtmenu {
text-align: center;
color: #FF8080;
font-weight: bold;
}
a.lnkmenu:link, a.lnkmenu:visited, a.lnkmenu:active {
color: #FFC;
text-decoration: none;
}
a.lnkmenu:hover {
color: #FF8080;
text-decoration: none;
}
</style>

<table width="600" border="1" cellspacing="0" cellpadding="0"
align="center"
bgcolor="#800000">
<tr>
<td><div class="txtmenu"><a href="index.htm" class="lnkmenu">Главная
</a></div></td>
<td><div class="txtmenu"><a href="aboutsite.htm" class="lnkmenu">О сайте
</a></div></td>
<td><div class="txtmenu"><a href="aboutme.htm" class="lnkmenu">Обо мне
</a></div></td>
<td><div class="txtmenu"><a href="links.htm" class="lnkmenu">Ссылки
</a></div></td>
```

```
<td><div class="txtmenu"><a href="http://адрес_гостевой"
class="lnkmenu">
Гостевая книга</a></div></td>
</tr>
</table>
```

Ссылка страницы на саму себя будет неактивна. То есть тэг **<a>** на свою страницу выводиться не будет. Обратите внимание, стиль **color** для ** ** определён тот же, что и для ссылки при наведении мышкой, розовенький, **#FF8080**. Лишённый тэга **<a>**, пункт меню всё время будет отображаться этим цветом.

Для решения этой задачи нам понадобится скрипт, в котором будут использованы массивы. Скрипт будет состоять из двух файлов.js. В первом будут заданы все параметры, и ссылка на него будет находиться в **<head>** наших web-страниц, а во втором файле будет скрипт, выводящий готовое меню на страницу, и ссылка на него будет там, где это меню должно появиться.

Как это будет работать?

Скрипт будет определять **<title>**заголовок**</title>** каждой страницы, а потом через **if** будет решать, какой из пунктов меню нужно выводить без ссылки.

Сначала запишем в переменные повторяющийся текст тэгов, отграничив ссылки от остального (двойные кавычки в тэгах превратим в одиночные):

```
var div1 = "<td><div class='txtmenu'>"
var lnk1 = "<a href="
var lnk2 = "' class='lnkmenu'>"
var lnk3 = "</a>"
var div2 = "</div></td>"
```

Пункт меню со ссылкой будет выглядеть так:

```
div1 + lnk1 + "URL_страницы" + lnk2 + "текст_меню" + lnk3 + div2
```

Без ссылки — так:

```
div1 + "текст_меню" + div2
```

Примечание: всё, что находится внутри **document.write()**, вводится без перевода каретки. Переносы строк, которые Вы можете увидеть в примерах кода, — результат автопереноса в браузере. Если Вы скопируете эти коды, то ненужных переносов не будет.

В первом файле скрипта объявляем и назначаем переменные для тэгов и создаём три массива.

Первый — для заголовков в тэге< **<title>**.

Второй — для URL-адресов страниц.

Третий — для заголовков пунктов меню.

Вот как этот файл будет выглядеть:

```
var div1 = "<td><div class='txtmenu'>"
var lnk1 = "<a href="
var lnk2 = "' class='lnkmenu'>"
var lnk3 = "</a>" var div2 = "</div></td>"

titArray = new Array()
titArray [0] = "Мой сайт - Главная страница"
titArray [1] = "Мой сайт - О сайте"
titArray [2] = "Мой сайт - Обо мне"
titArray [3] = "Мой сайт - Ссылки"
titArray [4] = "Мой сайт - Гостевая книга"

urlArray = new Array()
urlArray [0] = "index.htm"
urlArray [1] = "aboutsite.htm"
urlArray [2] = "aboutme.htm"
urlArray [3] = "links.htm"
urlArray [4] = "http://адрес_гостевой_книги"

mnuArray = new Array()
mnuArray [0] = "Главная"
mnuArray [1] = "О сайте"
mnuArray [2] = "Обо мне"
mnuArray [3] = "Ссылки"
mnuArray [4] = "Гостевая книга"
```

Сохраним его и приступим к созданию второго. Не забудьте, что помимо **<td>** у таблицы имеются **<tr>** и **<table>**. Сразу откроем и закроем таблицу:

```
    document.write("<table width='600' border='1' cellspacing='0'
cellpadding='0' align='center' bgcolor='#800000'><tr>")

/* скрипт меню*/

document.write("</tr></table>")
```

А теперь будем заполнять середину.

Первый пункт меню.

Если
(заголовок страницы — «Мой сайт - Главная страница»)
{выводим меню без ссылки**}**
В противном случае
{выводим меню со ссылкой**}**
Заголовок страницы достаётся через **document.title**.

Переводим это на JavaScript:

```
if (document.title == titArray[0])
{document.write(div1+mnuArray[0]+div2)}
else
{document.write(div1+lnk1+urlArray[0]+lnk2+mnuArray[0]+lnk3+div2)}
```

Всё это копируем и для остальных пунктов меню, только соответственно меняем номера элементов массива. Целиком второй файл выглядит так:

```
document.write("<table width='600' border='1' cellspacing='0'
cellpadding='0' align='center' bgcolor='#800000'><tr>")
if (document.title == titArray[0])
{document.write(div1+mnuArray[0]+div2)}
else
{document.write(div1+lnk1+urlArray[0]+lnk2+mnuArray[0]+lnk3+div2)}
if (document.title == titArray[1])
{document.write(div1+mnuArray[1]+div2)}
else
{document.write(div1+lnk1+urlArray[1]+lnk2+mnuArray[1]+lnk3+div2)}
if (document.title == titArray[2])
{document.write(div1+mnuArray[2]+div2)}
else
{document.write(div1+lnk1+urlArray[2]+lnk2+mnuArray[2]+lnk3+div2)}
if (document.title == titArray[3])
{document.write(div1+mnuArray[3]+div2)}
else
{document.write(div1+lnk1+urlArray[3]+lnk2+mnuArray[3]+lnk3+div2)}
if (document.title == titArray[4])
{document.write(div1+mnuArray[4]+div2)}
else
{document.write(div1+lnk1+urlArray[4]+lnk2+mnuArray[4]+lnk3+div2)}
document.write("</tr></table>")
```

Не забудьте пристегнуть к web-страницам и файл со стилями CSS.

Меню готово.

Лабораторная работа № 2

Ещё о массивах

- многомерные массивы;
- встроенные массивы;
- усовершенствованный фотоальбом;
- оператор-помощник **with**.

Многомерные массивы

Многомерные массивы в JavaScript — это массивы, содержащие внутри себя другие массивы.

Чтобы это проиллюстрировать, представим себе заготовку двухуровнего меню в виде списка.

Сначала объявим массив:

```
list = new Array()
```

Теперь объявим его первый элемент как массив из трёх элементов — наших основных пунктов меню.

```
list[0] = new Array("Меню 1", "Меню 2", "Меню 3")
```

В этом «массиве в массиве» у нас три элемента, которые можно вызвать как 0, 1 и 2 (помним, что отсчёт ведётся с нуля).

Теперь, вызывая **list[0][0]**, мы получим **«Меню 1»**, вызывая **list[0][1] — «Меню 2»**, и т.д.

Следующие элементы главного массива заготавливаем как массивы пунктов подменю:

```
list[1] = new Array("Меню 1.1", "Меню 1.2", "Меню 1.3")
list[2] = new Array("Меню 2.1", "Меню 2.2")
list[3] = new Array("Меню 3.1", "Меню 3.2", "Меню 3.3", "Меню 3.4")
"tratata"
```

Вызываются аналогично: например, **list[1][2]** (**«Меню 1.3»**) или **list[2][0]** (**«Меню 2.1»**).

Теперь формируем список:

```
/*Первый уровень, первый пункт*/
document.writeln("<ul><li>" + list[0][0] + "</li>")
/*Второй уровень*/
document.writeln("<ul><li>" + list[1][0] + "</li>")
document.writeln("<li>" + list[1][1] + "</li>")
```

```
document.writeln("<li>" + list[1][2] + "</li></ul>")
/*Первый уровень, второй пункт*/
document.writeln("<li>" + list[0][1] + "</li>")
/*Второй уровень*/
document.writeln("<ul><li>" + list[2][0] + "</li>")
document.writeln("<li>" + list[2][1] + "</li></ul>")
/*Первый уровень, третий пункт*/
document.writeln("<li>" + list[0][2] + "</li>")
/*Второй уровень*/
document.writeln("<ul><li>" + list[3][0] + "</li>")
document.writeln("<li>" + list[3][1] + "</li>")
document.writeln("<li>" + list[3][2] + "</li>")
document.writeln("<li>" + list[3][3] + "</li></ul></ul>")
```

Результат:

- Меню 1
  - Меню 1.1
  - Меню 1.2
  - Меню 1.3
- Меню 2
  - Меню 2.1
  - Меню 2.2
- Меню 3
  - Меню 3.1
  - Меню 3.2
  - Меню 3.3
  - Меню 3.4

Для чего это нужно?

Чтобы для настройки и «оживления» меню можно было программно обращаться к его пунктам как к элементам массива.

Таким же образом можно загнать в многомерный массив, скажем, таблицу и написать скрипт, который при нажатии определённых кнопок или заголовков будет выполнять сортировку (по алфавиту, по величине и т.д.).

Встроенные массивы

Поскольку JavaScript предназначен прежде всего для web-страниц, некоторые элементы HTML встроены в него как массивы (иногда их называют коллекциями).

К таким массивам-коллекциям относятся формы — **forms()** и изображения — **images()**.

Допустим, на нашей web-странице несколько раз встречается тэг **<img>**. Первый **<img>** будет автоматически определяться как **images[0]**, и т.д. по порядку появления в коде. То же и с формами.

getElementById() — присваиваем тэгу **id** и обращаемся к нему этим методом.

Но есть от встроенных массивов и настоящая польза. У элементов этих массивов есть свойства, присущие их объектам-прототипам. Так, элементы коллекции **forms()** имеют свойства **method**, **action**, **name**, а элементы **images()** — свойства **src**, **width**, **height**. Таким образом, мы можем создавать некие «абстрактные» элементы массива, а потом определять для них конкретный тэг. Это хорошо работает в слайд-шоу, когда в одном тэге программно заменяются картинки, определённые в скрипте.

Чтобы создать такой элемент, нужно объявить обычный массив и переопределить его элементы:

```
    imgslide = new Array()
imgslide[0] = new Image()
imgslide[1] = new Image()
```

Обратите внимание: элементы называются в единственном числе — **Image()** (и с большой буквы), а не **images()** (с маленькой буквы), как вся коллекция.

Объявим две переменных счётчика и два массива — для картин и для скульптур. Элементы массивов сразу объявим как **images**.

```
var i = 0, j = 0;

imgslide = new Array()
imgslide[0] = new Image()
imgslide[1] = new Image()
imgslide[2] = new Image()
imgslide[3] = new Image()
imgslide[4] = new Image()

imgslide2 = new Array()
imgslide2[0] = new Image()
imgslide2[1] = new Image()
imgslide2[2] = new Image()
imgslide2[3] = new Image()
imgslide2[4] = new Image()
```

Высота у всех изображений одинаковая — 300 px. А вот ширина немножко разная. Воспользуемся свойствами элементов встроенного массива. Кроме ширины **width** укажем имена и пути файлов **src**.

```
imgslide[0].src = "album/plakha.jpg"
imgslide[0].width = "225"
```

Что, так и будем по два раза каждый элемент прописывать?

Для сокращения кода существует вспомогательный оператор **with**.

Один раз упомянув в круглых скобках объект, в фигурных можно выписать все его свойства и методы через точку с запятой:

```
with (объект) {свойство_или_метод; свойство_или_метод}
```

Теперь наш код приобретёт более компактный вид:

```
with (imgslide[0]) {src = "album/plakha.jpg"; width = "225"}
with (imgslide[1]) {src = "album/grav.jpg"; width = "224"}
with (imgslide[2]) {src = "album/history.jpg"; width = "200"}
with (imgslide[3]) {src = "album/porog.jpg"; width = "202"}
with (imgslide[4]) {src = "album/dedal.jpg"; width = "232"}

with (imgslide2[0]) {src = "album/applegir.jpg"; width = "178"}
with (imgslide2[1]) {src = "album/atlant.jpg"; width = "181"}
with (imgslide2[2]) {src = "album/afrodita.jpg"; width = "193"}
with (imgslide2[3]) {src = "album/turgenev.jpg"; width = "199"}
with (imgslide2[4]) {src = "album/obelisk.jpg"; width = "222"}
```

Все картины и скульптуры как-то называются. С названиями легче. Давайте сделаем, чтобы и названия выпрыгивали.

Для этого заготовим ещё два массива — тоже для картин и для скульптур.

```
imgname = new Array()
imgname[0] = "Плаха"
imgname[1] = "И сказал Бог: «Да будет свет!»<br><small>по гравюре Г. Доре</small>"
imgname[2] = "История"
imgname[3] = "Порог"
imgname[4] = "Дедал"

imgname2 = new Array()
imgname2[0] = "Девочка с яблоком"
imgname2[1] = "Почетный приз «Атлант»"
imgname2[2] = "Пеннорожденная Афродита"
imgname2[3] = "И.С.Тургенев"
imgname2[4] = "Памятник Л.Богданову"
```

Принцип работы будет такой: в левой половине — слайд с картинами и под ним две кнопки: «вперёд» и «назад». Под кнопками — меняющееся название. Справа — аналогичная конструкция для скульптур (всё это можно поместить в таблицу с двумя **<td>** по 50%). Кнопки мы возьмём из стандартного элемента **form**.

Сейчас напишем этакую многофункциональную функцию, которая отрегулирует работу всей этой системы.

Сначала сформулируем задачи:

Кнопка «вперёд» должна «бесконечно» прокручивать картинки вперёд, то есть, дойдя до конца, возвращаться к началу.

Кнопка «назад» должна делать то же самое, но в обратном направлении.

Пары кнопок должны работать автономно, прокручивая только свой слайд.

Нам нужны всего две таких цепочки:

вперёд-назад — да-нет;

картины-скульптуры — да-нет.

Эти два булевых выражения и станут аргументами нашей функции. Функции назовём её **dem_plus(n,k)**. Приведу её всю, а дальше будем разбираться, что к чему.

```
    function dem_plus(n, k)
{
var dlina;
switch (k)
{
 case true:
 // определяем размерность первого массива
 // и подставляем значение в код для кнопок
 dlina = imgslide.length
   if (n == true)
   { i++;
     if (i == dlina)
     i = 0;
   }
   else
   {i--;
     if (i == -1)
     i = (dlina - 1);
   }
   with(document.images("pic"))
   {src = imgslide[i].src;
   width=imgslide[i].width}
   document.getElementById("picname").innerHTML = imgname[i]
```

```
 break
 case false:
 // определяем размерность второго массива
 // и снова подставляем, используя новый счётчик
 dlina = imgslide2.length
  if (n == true)
  { j++;
   if (j == dlina)
   j = 0;
  }
  else
  {j--;
   if (j == -1)
   j = (dlina - 1);
  }
  with(document.images("sculp"))
  {src = imgslide2[j].src;
  width = imgslide2[j].width}
  document.getElementById("sculpname").innerHTML = imgname2[j]
 }
}
```

Количество картин и скульптур в нашем примере одинаково. Но оно может быть и разным. Для этого нам и нужны два счётчика. Но это ещё не всё. Для правильной работы функция должна «знать» фактическую длину каждого массива. Поэтому объявим в ней переменную **dlina**, чтобы, воспользовавшись свойством массива **length**, динамически определить «потолок» для каждой группы слайдов.

(Вспомните, что переменная, объявленная внутри функции, «живёт» только в теле этой функции. Но здесь нам этого вполне достаточно.)

Сначала разнесём разные слайды. С этим управится второй аргумент — **k**. Тут никаких особых действий — просто «да» и «нет». Не будем конструировать «если бы да кабы», а используем простой переключатель **switch**. И в случае **true** (картины) будем использовать счётчик **i**, а в случае **false** (скульптуры) — **j**.

Аргумент **n** отвечает за «вперёд-назад». Если он **true**, то вперёд. Счётчик (**i** или **j**) будет считать картинки по возрастающей.

Теперь внимание: пошла арифметика, в которой можно и запутаться.

Какова размерность массива картин (скульптур)? Как мы помним, это составит их реальное количество + 1 (см. прошлый урок). Значит, **dlina** даст нам на 1 номер больше, чем их количество.

Значит, когда счётчик перейдёт на значение **dlina** (элемента с таким номером **уже нет**, понятно, почему?), нужно взмахнуть волшебной палочкой и превратить его в **0**. Вот видите, можно и так циклы делать, не используя специальных операторов. Но здесь нам не нужен самодвижущийся цикл. «Движок» — это «юзер» с мышкой.

В противном случае (**else**) делаем «пач-пач-пач» (назад). За нулём — что там в школе проходили? Минус единичка. Её-то мы и превращаем в...

Будьте внимательны! В **dlina - 1**. Именно это и будет номером последней картинки, который нам нужен.

Кончилась абстракция, теперь мы имеем дело с конкретными объектами на web-странице. Давайте отвлечёмся от функции и подготовим плацдарм.

Вот наши окошки для слайдов. Можно воспользоваться и порядковыми номерами из коллекции. Но давайте дадим дизайнеру возможность оформить страницу, не заморачиваясь сохранностью её содержимого. Поэтому лучше дадим нашим окошкам имена собственные: **pic** и **sculp**.

Да, у нас же ещё и названия! Так что зададим имена и тем абзацам или заголовкам (у меня **<h3>**), в которых эти названия должны появляться.

Теперь вернёмся к функции. Зададим для подготовленных окошек имя файла и ширину картинки (можно опять через **with**), а также текст названия (из массивов **imgname** и **imgname2**). В качестве номера элемента указываем счётчик. Если счётчик будет работать правильно, он сам будет подставлять нужные номера.

Метод **innerHTML** возвращает то, что находится внутри указанного тэга HTML (в данном случае — текст). Подробнее об этих методах будем говорить позже.

Разбросаем всё по случаям **true** (картины) **false** (скульптуры).

Не забудем поставить **break**!

Нам нужно **либо** одно, **либо** другое, а не всё сразу!

Аккуратно закрываем все скобки. Функция готова.

Вызывать её будем из кнопок формы (из атрибута **onClick**) с нужными аргументами (**true** или **false**).

В реальные тэги HTML (по умолчанию) вставим параметры нулевых слайдов.

Вот примерно что должно быть в **<body>** (проанализируйте, как расподожены аргументы **true** и **false** в вызовах функции):

```
    d<table width="100%" border="0" cellspacing="0" cellpadding="0"
align="center">
<tr>
<td width="50%" align="center">
<h3>Живопись</h3>
<img src="album/plakha.jpg" alt="" name="pic" id="pic" width="225"
height="300" border="0"></td>
<td width="50%" align="center">
```

```
<h3>Скульптура</h3>
<img src="album/applegir.jpg" alt="" name="sculp" id="sculp" width="178"
height="300" border="0"></td>
</tr>
<tr>
<td align="center"><form name="form1">
<input type="button" value="Назад" onClick="dem_plus(false, true)">
<input type="button" value="Вперед" onClick="dem_plus(true, true)">
</form>
<h3 id="picname">Плаха</h3></td>
<td align="center"><form name="form2">
<input type="button" value="Назад" onClick="dem_plus(false, false)">
<input type="button" value="Вперед" onClick="dem_plus(true, false)">
</form>
<h3 id="sculpname">Девочка с яблоком</h3></td>
</tr>
</table>
```

Лабораторная работа № 3
Как правильно писать код

Объявление переменных

Когда слово var не обязательно? Только когда объявляется переменная с назначением: **x = "Вася"**. В остальных случаях — обязательно.

Но чтобы не запоминать, где что можно, а где нельзя, предлагаю везде использовать ключевое слово **var**.

Точка с запятой

Каков смысл точки с запятой?

Это конец строки. Каждое выражение (объявление или инструкция) должно занимать отдельную строку. Но строка эта может быть как реальной (переводом каретки), так и «виртуальной» — ограниченной точкой с запятой. Всё, что следует после точки с запятой на этой же реальной строке, программа воспринимает как новую строку.

Вот несколько примеров:

```
    //Правильно.
var x
y = 5
x = 3
document.write(x + y)
```

```
//Правильно, хотя точки с запятыми здесь не обязательны.
var x;
y = 5;
x = 3;
document.write(x + y);

/*Правильно, но где логика
в этой единственной точке с запятой?*/
var x
y = 5
x = 3
document.write(x + y);

/*Правильно. Первое выражение (через запятую) -
однострочное.*/
var x, y = 5
x = 3
document.write(x + y)

/*А вот это неправильно: однострочные выражения
нельзя разбивать на строки.*/.
var x,
y = 5
x = 3
document.write(x + y)

/*Правильно, но по-человечески нелогично:
зачем первую строку делать "виртуальную",
а остальные - реальные?*/
var x; y = 5
x = 3
document.write(x + y)

/*Правильно, и логика присутствует:
на одной строке - объявления,
на другой - инструкция.*/
var x, y = 5; x = 3
document.write(x + y)

/*Правильно, но по-человечески
```

```
опять же как-то неестественно.*/
var x, y = 5;
x = 3; document.write(x + y)

//Правильно.
var x, y = 5; x = 3; document.write(x + y)

//Неправильно.
var x, y = 5 x = 3 document.write(x + y)
```

Вырабатывайте свой стиль записи кода, но желательно, чтобы он был культурным. Либо Вы ставите точки с запятыми там и только там, где это необходимо, либо везде.

Скобки

Всё, что находится в **круглых** скобках, должно быть **на одной строке**, удобно это или неудобно. Когда круглые скобки показывают **заголовок оператора** (перед его телом в фигурных), точку с запятой после них ставить нельзя. В арифметических выражениях круглые скобки используются так же, как в школьных примерах. Иногда можно их использовать просто для наглядности, как я сделал это в последней функции: **i = (dlina – 1)**, чтобы показать, что **(dlina – 1)** — это не «что-то минус один», а конкретное число, которое назначается счётчику (а почему минус один — вспомните про «нулевой месяц январь»). Эти скобки можно и не ставить.

Фигурные скобки не реагируют на перевод каретки, и располагать их можно самым причудливым образом. Если при этом использовать ещё и «лесенку» с табуляцией (как в стихах Маяковского), то можно добиться очень наглядной записи хитро закрученных «слоёных» инструкций. Если в фигурные скобки помещается только одно выражение, то их можно отбросить. Но опять же: с точки зрения культуры — либо всегда отбрасывать, либо никогда.

JavaScript специально приспособлен для web-страниц и использует довольно сложную и ветвистую объектную модель браузера. Это один наш «заяц». Другой «заяц» — это собственно объекты JavaScript с их многочисленными свойствами и методами.

Изучать это параллельно и постараться ничего не упустить — очень сложно.

Наверно, самый лучший способ — это изучать конкретные скрипты по мере возрастающей сложности в обоих направлениях.

Очень многие скрипты — часы, календари, приветствия по времени суток, дата последнего обновления, подсчёт дней до и после события — связаны с объектом даты/времени. Объект довольно запутанный, и «нулевой месяц январь» — самая милая и невинная его причуда. Есть также разночтения в разных браузерах, но есть и алгоритмы, сводящие их практически на нет.

Но дата и время связаны с одной стороны с чистой математикой, с другой — с выводом строковых данных в браузер. Поэтому не мешает кое-что знать о свойствах и методах строковых и числовых объектов, а также о взаимодействии их с объектами браузера.

Лабораторная работа № 4

Объектная модель JavaScript

- объекты;
- методы;
- свойства;
- иерархия объектов браузера.

К объектно-ориентированным языкам относится и JavaScript. То есть, «персонажами» или «именами существительными» являются объекты. Но кроме того, у объектов есть ещё свойства («прилагательные») и методы («глаголы»). Вот это всё и составляет объектную модель языка.

У каждого объекта свои методы и свойства. Бывают свойства и методы, закреплённые за несколькими объектами, и с разными объектами они могут работать по-разному. Бывает, что объекты превращаются в свойства и методы других объектов.

рисунок (объект);
рисовать (метод);
нарисованный (свойство).
В английском — ещё нагляднее:
a **stone** [камень] — объект;
a **stone** house [каменный дом] — свойство;
to **stone** house [облицовывать дом камнем] — метод.

Объекты

Объекты бывают

- **встроенные (внутренние)** — то есть объекты языка JavaScript. К ним относятся:
  - **String** — строка текста;
  - **Array** — массив;
  - **Date** — дата и время;
  - **Math** — математические функции;
  - **Object** — конструктор для создания **пользовательских объектов**;
  - **Дополнительные объекты**

- **объекты браузера** — создаются автоматически при загрузке документа в браузер:
  - **window** — объект верхнего уровня в иерархии объектов браузера;
  - **document** — содержит свойства, которые относятся к текущему HTML-документу;
  - **location** — содержит свойства, описывающие местонахождение текущего документа, например, адрес URL;
  - **navigator** — содержит информацию о версии браузера;
  - **history** — содержит информацию обо всех ресурсах, к которым пользователь обращался во время текущего сеанса;
- **связанные с тэгами HTML и стилями CSS** — в JavaScript большинству тэгов HTML и стилей CSS соответствуют свойства объекта document, которые сами также являются объектами;
- **пользовательские объекты** — это объекты, которые создаём мы сами с помощью конструктора **Object**.

Методы

Методы — это своеобразные «рычаги» для управления объектами. За объектами в JavaScript закреплено множество различных методов. К «высшему пилотажу» относится создание собственных методов для объектов. В коде методы указываются вместе со своими объектами (через точку):

```
объект.метод(параметры)
```

Например:

```
document.write("<p>Это первый абзац.</p>")
```

Свойства

Наличие свойств — специфическая особенность объектно-ориентированных языков. Собственно, наличие свойств и делает объект объектом, «вещью», которая имеет свой «характер» и с которой можно работать. Как и методы, свойства указываются вместе с объектами, через точку. Но параметры свойств ставятся не в скобки, а через оператор назначения или равенства:

```
объект.свойство = параметр
объект.свойство == параметр
```

Например:

```
document.images[4].height = 300
navigator.appName == "Netscape"
```

Как определить, когда какой оператор ставить?

Когда мы **назначаем** свойству его параметр, то ставим присвоение (=).
Когда используем **уже заданный параметр**, то ставим равенство (==).
Например:

```
if (document.images[4].height == 300) {какая_нибудь_инструкция}
```

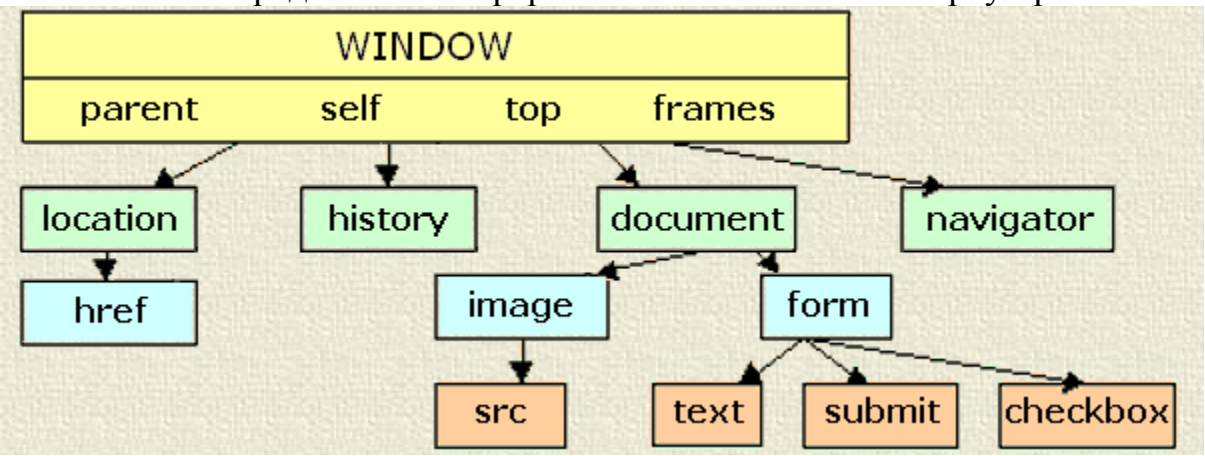
Кроме того, есть свойства **только для чтения**, с уже заданными патаметрами, которые нельзя переназначить. Например, приведённое выше свойство **appName** объекта **navigator**. Такие свойства вызываются **только** через сдвоенный оператор равенства (==).

Понятие иерархии объектов

Это понятие связано со структурой языка. Вкратце же — есть старшие (родительские) объекты и младшие (дочерние). Все **встроенные** объекты имеют одного родителя, который предпочитает остаться невидимкой. Поэтому при их использовании (за исключением особо хитрых случаев, о которых речь пойдёт в третьей серии) об иерархии можно не думать и даже не знать.

На сегодняшний день для нас с Вами важна **иерархия объектов браузера**, поскольку именно с этими объектами работает подавляющее большинство скриптов.

На этой схеме представлена иерархия основных объектов браузера.



Старший объект **window** обычно не упоминается в коде. Помните, в 1 уроке нам встречался метод **alert()**? Это метод объекта **window**, и он записывается без упоминания своего «хозяина». Впрочем, выражение **window.alert()** тоже не было бы ошибкой. Но «краткость — сестра таланта».

Parent, **self**, **top** и **frames** — это не объекты, а «псевдонимы» объекта **window**, особенности их употребления мы разберём в дальнейшем.

Дочерний объект **location** служит для определения адреса страницы. Объект **href** является его свойством (его имя совпадает с атрибутом тэга **<a>**, но это разные вещи: атрибут ссылки **href** не является объектом, дочерним для **location**)..

Объекты **history**, **navigator** и **document** — также дочерние объекты **window**. Самый разветвлённый из них — **document**. Помимо элементов коллекций **images** и **forms**, о которых мы говорили в прошлом уроке, он содержит все объекты, связанные с тэгами HTML и стилями CSS.

Свойства элементов коллекций **images** и **forms**, в свою очередь, являются дочерними объектами для объектов **image** и **form**

Поддержка JavaScript

Многие уже столкнулись с тем, что разные браузеры понимают скрипты по-разному, а то и вовсе не понимают. Поэтому при использовании объектов, их свойств и методов необходимо учитывать разные типы браузеров.

Лабораторная работа № 5

Встроенные объекты

Ниже перечислены встроенные объекты, свойства и методы которых доступны в сценариях JavaScript без предварительного определения этих объектов.

Объект	Описание
Array	Массив
Boolean	Логические данные
Date	Календарная дата
Function	Функция
Global	Глобальные методы
Math	Математические константы и функции
Number	Числа
Object	Объект
String	Строки

Встроенные объекты удобны для выполнения различных операций со строками, календарными датами, массивами, числами и так далее. Они освобождают программиста от выполнения различных рутинных операций вроде преобразования строк или вычисления математических функций.

Как работать со встроенными объектами? Достаточно просто. Программа создает реализации объектов, а затем обращается к свойствам и методам объектов. В качестве примера, имеющего практическое значение, рассмотрим документ HTML, в котором отображается текущая дата и время.

Листинг 5.1.

```
<HTML>
  <HEAD>
    <TITLE>Текущая дата и время</TITLE>
  </HEAD>
  <BODY BGCOLOR=WHITE>
    <H1>Текущая дата и время</H1>
    <SCRIPT LANGUAGE="JavaScript">
    <!--

      var dt;
      var szDate="";

      dt = new Date();
      szDate = "Date: " + dt.getDate() + "."
        + dt.getMonth() + "." + dt.getYear();

      document.write(szDate);
      document.write("<BR>");
      document.write("Time: " + dt.getHours()
        + ":" + dt.getMinutes() + ":" + dt.getSeconds());

    // -->
    </SCRIPT>
  </BODY>
</HTML>
```

Здесь сценарий JavaScript создает объект Data, применяя для этого ключевое слово new, и конструктор Date без параметров:

```
var dt;
dt = new Date();
```

Создаваемый таким образом объект Data инициализируется текущей локальной датой, установленной у пользователя (а не на сервере Web, с которого был загружен соответствующий документ HTML). В следующей строке формируется текстовая строка даты:

```
szDate = "Date: " + dt.getDate() + "."
  + dt.getMonth() + "." + dt.getYear();
```

Значение календарного числа, номера месяца и года здесь получается при помощи методов getDate, getMonth и getYear, соответственно. Эти методы вызываются для объекта dt, содержащего текущую дату.

Текстовая строка даты выводится в документ HTML с помощью метода write, определенного в объекте document:

```
document.write(szDate);
```

Объект Date содержит также информацию о текущем времени. Эта информация извлекается для отображения с помощью методов getHours, getMinutes и getSeconds (соответственно, часы, минуты и секунды):

```
document.write("Time: " + dt.getHours()
  + ":" + dt.getMinutes() + ":" + dt.getSeconds());
```

Встроенный объект Math

Хотя сценарии JavaScript редко применяют для математических вычислений, в нем все же есть встроенный класс Math, предназначенный как раз для этого.

Свойства

Ниже перечислены свойства класса Math. Все эти свойства являются математическими константами, поэтому сценарий JavaScript не может изменять их значение.

E

Это свойство представляет собой константу e. Приблизительное значение этой константы равно 2,72.

Вот пример использования свойства E:

```
var nE;
nE = Math.E;
```

PI

Свойство PI - это число . Оно также является константой с приблизительным значением, равным 3,14.

Пример использования свойства PI (вычисления длины окружности по ее радиусу):

```
var nL;
var nR;
nL = 2 * Math.PI * nR;
```

LN2

Свойство LN2 - константа со значением натурального логарифма числа 2, то есть ln2.

Пример использования:

```
var nValue;
nValue = Math.LN2;
```

LN10
Свойство LN10 - константа со значением натурального логарифма числа 10, то есть ln10.
Пример использования:
var nValue;
nValue = Math.LN10;

LOG2E
Это свойство является константой со значением, равным логарифму числа 2 по основанию e, то есть $\log_e 2$.
Пример использования:
var nValue;
nValue = Math.LOG2E;

LOG10E
Свойство LOG10E - это логарифм числа e по основанию 10, то есть $\log_{10} e$.
Пример использования:
var nValue;
nValue = Math.LOG10E;

QRT2
Свойство SQRT2 - это значение квадратного корня из 2.
Пример использования:
var nValue;
nValue = Math.SQRT2;

SQRT1_2
Свойство SQRT1_2 - это значение квадратного корня из 0,5.
Пример использования:
var nValue;
nValue = Math.SQRT1_2;

Методы класса Math

abs
Вычисление абсолютного значения. Пример использования:
var nValueAbs;
nValueAbs = Math.abs(nValue);
Здесь в переменную nValueAbs записывается абсолютное значение переменной nValue.

acos
Вычисление арккосинуса. Пример использования:

```
var nValue;
nValue = Math.acos(nAngle);
```

asin

Вычисление арксинуса. Пример использования:

```
var nValue;
nValue = Math.asin(nAngle);
```

atan

Вычисление арктангенса. Пример использования:

```
var nValue;
nValue = Math.atan(nAngle);
```

ceil

Вычисление наименьшего целого значения, большего или равного аргументу функции. Пример использования:

```
var nValue;
nValue = Math.ceil(nArg);
```

cos

Вычисление косинуса. Пример использования:

```
var nValue;
nValue = Math.cos(nAngle);
```

exp

Экспоненциальная функция, значение которой равно числу e, возведенному в степень аргумента функции.

Пример использования:

```
var nValueExp;
nValueExp = Math.exp(nValue);
```

floor

Вычисление наибольшего целого значения, меньшего или равного аргументу функции.

Пример использования:

```
var nValue;
nValue = Math.floor(nArg);
```

log

Вычисление натурального логарифма аргумента функции. Пример использования:

```
var nValue;
nValue = Math.log(nArg);
```

max

Определение наибольшего из двух значений. Пример использования:

```
var nValue1;
var nValue2;
var nValueMax;
nValueMax = Math.max(nValue1, nValue1);
```

min
Определение наименьшего из двух значений. Пример использования:
```
var nValue1;
var nValue2;
var nValueMin;
nValueMin = Math.min(nValue1, nValue1);
```
pow
Возведение числа в заданную степень. Пример использования:
```
var nValue;
nValue = Math.pow(2, 3);
```
Здесь число 2 возводится в степень 3, а результат, равный 8, записывается в переменную nValue.

random
Метод random возвращает случайное число в интервале от 0 до 1. Пример использования:
```
var nRandomValue;
nRandomValue = Math.random();
```
round
Метод round предназначен для выполнения округления значения аргумента до ближайшего целого. Если десятичная часть числа равна 0,5 или больше этого значения, то выполняется округление в большую сторону, если меньше - в меньшую. Пример использования:
```
var nValue;
nValue = Math.round(1.8);
```
После выполнения округления значение nValue будет равно 2.

sin
Вычисление синуса. Пример использования:
```
var nValue;
nValue = Math.sin(nAngle);
```
sqrt
Вычисление квадратного корня от аргумента. Пример использования:
```
var nValueSqrt;
nValueSqrt = Math.sqrt(nArg);
```
tan
Вычисление тангенса. Пример использования:
```
var nValue;
nValue = Math.tan(nAngle);
```

Лабораторная работа № 6

События, связанные с объектами

С каждым объектом браузера связывается определенный набор событий, обработка которых возможна в сценарии JavaScript.

Например, с объектом **window** связаны события **onLoad** и **onUnload**. Первое из этих событий возникает, когда браузер завершает загрузку окна и всех расположенных в нем фреймов (если эти фреймы определены в окне). Второе событие возникает, когда пользователь завершает работу с документом, закрывая окно браузера или переключаясь на другой документ.

Сценарий JavaScript может, например, при обработке события **onLoad** выводить для пользователя приветственное сообщение или запрашивать дополнительную информацию. При завершении работы с окном (когда возникает событие **onUnload**) сценарий может освобождать какие-либо ресурсы, связанные с этим окном, или выводить сообщение на экран монитора.

Объекты на базе классов, создаваемых программистом.

В языке JavaScript для создания собственных классов используется подход, отличный от подходов, принятых во многих языках программирования. Класс JavaScript создается как функция, в которой определены свойства, играющие роль данных. Методы также определяются как функции, но отдельно.

Приведем конкретный пример. Пусть нужно создать класс, предназначенный для хранения записи воображаемой телефонной базы данных. В этом классе нужно предусмотреть свойства для хранения имени, фамилии, номера телефона, адреса, а также специального признака для записей, закрытых от несанкционированного доступа. Дополнительно требуется разработать методы, предназначенные для отображения содержимого объекта в табличном виде.

Прежде всего займемся созданием собственного класса с названием myRecord. Пусть пока в нем не будет методов, добавим их позже. Искомый класс создается следующим образом:

```
function myRecord(name, family, phone, address)
{
  this.name    = name;
  this.family  = family;
  this.phone   = phone;
  this.address = address;
  this.secure  = false;
}
```

Нетрудно заметить, что описание данного класса есть ни что иное, как функция конструктора.

Параметры конструктора предназначены для установки свойств объекта при его создании на базе класса. Свойства определяются простыми ссылками на их имена с указанием ключевого слова this. Это ключевое слово здесь указывает, что в операторе выполняется ссылка на свойства того объекта, для которого вызван конструктор, то есть для создаваемого объекта.
Обратите внимание, что конструктор инициализирует свойство с именем secure, записывая в него значение false. Соответствующий параметр в конструкторе не предусмотрен, что вполне допустимо.
Как пользоваться определенным классом? На базе этого класса можно создать произвольное количество объектов. Ниже приведен фрагмент сценария JavaScript, где на базе класса myRecord создается два объекта rec1 и rec2:

```
var rec1;
var rec2;
rec1 = new myRecord("Иван", "Иванов",
  "000-322-223", "Малая Большая ул., д. 225, кв. 226");
rec2 = new myRecord("Петр", "Петров",
  "001-223-3334", "Большая Малая ул., д. 552, кв. 662");
rec2.secure = true;
```

Объекты создаются при помощи оператора new. Конструктору передаются параметры для инициализации свойств создаваемых объектов.
Что же касается свойства с именем secure, то в объекте rec2 оно инициализируется уже после создания последнего. В него записывается значение true. Свойство secure объекта rec1 не изменяется, поэтому в нем хранится значение false.
Теперь добавим в определенный класс новые методы с именами printTableHead, printTableEnd и printRecord. Первые два из этих методов выводят в документ HTML, соответственно, начальный и конечный фрагмент таблицы, а третий - строки таблицы, отражающие содержимое записей. В сокращенном виде новое определение класса myRecord представлено ниже:

```
function printTableHead()
{
  . . .
}
function printTableEnd()
{
  . . .
}
function printRecord()
{
  . . .
}
function myRecord(name, family, phone, address)
{
```

```
  this.name    = name;
  this.family  = family;
  this.phone   = phone;
  this.address = address;
  this.secure  = false;
  this.printRecord    = printRecord;
  this.printTableHead = printTableHead;
  this.printTableEnd  = printTableEnd;
}
```

Здесь перед определением конструктора расположены определения для функций-методов нашего класса. Кроме этого, в конструктор добавлено определение новых свойств:

```
this.printRecord    = printRecord;
this.printTableHead = printTableHead;
this.printTableEnd  = printTableEnd;
```

Эти свойства хранят ссылки на методы, определенные выше. После такого определения класса можно создавать объекты и обращаться к определенным методам:

```
rec1.printTableHead();
rec1.printRecord();
rec1.printTableEnd();
rec2.printTableHead();
rec2.printRecord();
rec2.printTableEnd();
```

Приведем полный исходный текст получившийся программы.

Листинг 6.1.

```
<HTML>
  <HEAD>
    <TITLE>Просмотр записей</TITLE>
    <SCRIPT LANGUAGE="JavaScript">
    <!--

    function printTableHead()
    {
      var szSec = "";
      if(this.secure)
        szSec = " (Secure)";
      else
        szSec = " (Unsecure)".fontcolor("red");

      document.write("<TABLE BORDER>");
      document.write("<CAPTION ALIGN=LEFT>" +
```

```
    this.name + " " + this.family + szSec +
    "</CAPTION>");
   document.write("<TH ALIGN=LEFT>Поле записи</TH>"
    + "<TH ALIGN=LEFT>Содержимое</TH>");
  }

  function printTableEnd()
  {
   document.write("</TABLE>");
   document.write("<P> ");
  }

  function printRecord()
  {
   document.write("<TR><TD>Name:</TD><TD>" +
    this.name.italics() + "</TD></TR>");
   document.write("<TR><TD>Family:</TD><TD>" +
    this.family.italics() + "</TD></TR>");
   document.write("<TR><TD>Phone:</TD><TD>" +
    this.phone.italics() + "</TD></TR>");
   document.write("<TR><TD>Address:</TD><TD>" +
    this.address.italics() + "</TD></TR>");
  }

  function myRecord(name, family, phone, address)
  {
   this.name = name;
   this.family = family;
   this.phone = phone;
   this.address = address;
   this.secure = false;
   this.printRecord = printRecord;
   this.printTableHead = printTableHead;
   this.printTableEnd = printTableEnd;
  }

  // -->
  </SCRIPT>
</HEAD>
<BODY BGCOLOR=WHITE>
  <H1>Просмотр записей</H1>
  <SCRIPT LANGUAGE="JavaScript">
```

```
    <!--

    var rec1;
    var rec2;

    rec1 = new myRecord("Иван", "Иванов",
      "000-322-223", "Малая Большая ул., д. 225, кв. 226");

    rec2 = new myRecord("Петр", "Петров",
      "001-223-3334", "Большая Малая ул., д. 552, кв. 662");
    rec2.secure = true;
    rec1.printTableHead();
    rec1.printRecord();
    rec1.printTableEnd();
    rec2.printTableHead();
    rec2.printRecord();
    rec2.printTableEnd();

    // -->
    </SCRIPT>
  </BODY>
</HTML>
```

Определение нового класса myRecord и его методов расположено в области заголовка документа HTML, как это принято делать.

Метод printTableHead выводит в документ HTML заголовок таблицы. Внешний вид этого заголовка зависит от содержимого свойств объекта. Прежде всего метод printTableHead проверяет свойство secure, получая его значение при помощи ключевого слова this:

```
var szSec = "";
if(this.secure)
  szSec = " (Secure)";
else
  szSec = " (Unsecure)".fontcolor("red");
```

Здесь это ключевое слово означает, что необходимо использовать свойство того объекта, для которого был вызван метод printTableHead. Если содержимое свойства secure равно true, в текстовую переменную szSec записывается строка " (Secure)". Если же оно равно false, в эту переменную заносится строка " (Unsecure)", причем для строки устанавливается красный цвет.

Так как в JavaScript все текстовые строки (в том числе и литералы) являются объектами встроенного класса String, то для них можно вызывать определенные в этом классе методы. В частности, метод fontcolor позволяет установить цвет строки.

Далее метод printTableHead выводит в документ HTML оператор <TABLE> с параметром BORDER, с которого начинается определение таблицы, имеющей рамку. Надпись над таблицей задается с помощью динамически формируемого оператора <CAPTION>. В эту надпись включается имя и фамилия, извлеченные из соответствующих свойств объекта, для которого был вызван метод printTableHead. Затем этот метод выводит надписи для столбцов таблицы.

Метод printTableEnd выводит в документ HTML оператор </TABLE>, завершающий определение таблицы, а также пустой параграф для отделения таблиц, следующих друг за другом:

```
function printTableEnd()
{
  document.write("</TABLE>");
  document.write("<P> ");
}
```

Последний метод, определенный в классе, называется printRecord. Он печатает содержимое первых четырех свойств объекта как строку таблицы, определенной в документе HTML только что описанной функцией printTableHead. Обратите внимание, что содержимое свойств объекта печатается наклонным шрифтом, для чего мы вызываем метод italics:

```
document.write("<TR><TD>Name:</TD><TD>" +
  this.name.italics() + "</TD></TR>");
```

Во второй части сценария, расположенной в теле документа HTML, создается два объекта rec1 и rec2 на базе класса myRecord, а затем устанавливается свойство secure объекта rec2 в состояние true.

Далее сценарий последовательно выводит в документ HTML две таблицы, соответствующие созданным объектам, вызывая для этого методы printTableHead, printRecord и printTableEnd.

Как видите, применение собственного класса позволили сильно упростить задачу инициализации и отображения содержимого записей воображаемой телефонной базы данных. Фактически эта задача сведена к вызовам нескольких методов, определенных заранее в нашем классе.

Литература

1. Кэмпбел Марк. Строим Web сайт. Москва, 2006.
2. Chuck Musciano and Bill Kennedy, "HTML: The Definitive Guide", O"Reily&Associates, Inc (1996).
3. html.doc, "Время Microsoft", журнал Аурамедиа, спецвыпуск. "Решения Microsoft", выпуск 5 (1996).
4. Michael J. Hannah, "HTML Reference Manual" (1996), http://www.sandia.gov/ sci_compute/ html_ref.html