Введение в синтаксис JavaScript

- как вставить скрипт в документ HTML (общие сведения);
- комментарии в JavaScript;
- как объявлять переменные и давать им правильные имена;
- разбор скрипта и синтаксис методов;
- метод alert();
- полезная мелочь: «заглушка» на временно не работающую ссылку

В создании web-страниц есть много разных тонкостей. Но есть и три кита. Это **HTML, CSS и JavaScript.**

Рекомендую организовать самообразование следующим образом: как только освоите синтаксис HTML и научитесь делать примитивные странички с текстом, картинками и таблицами, сразу подключайтесь к изучению CSS. Как только поймёте, как работать с CSS, начинайте осваивать JavaScript, параллельно пополняя «словарный запас» во всех трёх языках. Думаю, что через полгода сможете построить весьма красивый и функциональный сайт.

Как подступиться к JavaScript. Учебники либо слишком абстрактные — теория, теория, и непонятно, как приложить её к практике, либо, наоборот, слишком конкретные: вот тебе набор готовых рецептов, бери и пользуйся, а как это всё крутится — не твоего умишки дело.

Выбирайте как вам удобно.

Вставка в документ HTML

Наверняка видели в HTML-кодах такие тэги:

```
<script>
<!-- здесь какая-то непонятная абракадабра -->
</script>
```

Вот эта абракадабра между тэгами и есть скрипт.

Сам тэг **<script>** относится к языку HTML, и у него могут быть следующие атрибуты:

- language
- type
- src

```
<script language="javascript">
<!-- код скрипта -->
</script>
```

Этот атрибут необязателен. Его стоит использовать либо для уточнения версии языка (javascript1.1, javascript1.2 и т.п.), либо если используешь другой язык (например, **<script language=VBscript">**). То есть вреда этот атрибут в любом случае не принесёт, но в стандартной ситуации он вроде как лишний.

```
<script type="text/javascript">
<!-- код скрипта -->
</script>
```

Атрибут type, который указывает на тип текста: **text/javascript**. Он появился в версии HTML 4.0. Его-то я и рекомендую использовать.

Прежде чем перейти к атрибуту **src**, выясним, в какие разделы кода HTML можно вставлять

скрипты.

В любые. Но с умом.

Часто в скрипте указывается вывод конкретного текста, что называется, здесь и сейчас. Такой скрипт вставляется прямо в <body>, «на место происшествия».

Бывают скрипты с объявлениями переменных, которые могут быть использованы в других скриптах на странице, с функциями, которые можно вызвать из любого места кода HTML. Такие скрипты разумнее всего помещать между тэгами <head> и </head>.

Но можно сделать и так, чтобы использовать скрипт сразу на нескольких web-страницах. Для этого его код нужно записать в отдельный файл с расширением **.js** (например, **myscript_1.js**). Тэги <script> и </script> писать в нём уже не надо.

И вот тогда-то, для вызова скрипта с web-страницы, нам и понадобится атрибут **src**. Работает он так же, как и аналогичный атрибут тэга **<img>**:

```
<script type="text/javascript" src="scripts/myscript_1.js"></script>
```

Вот таким образом и помещается на разные страницы одна и та же шапка или меню, записанные в файле скрипта. Особенно это выручает на тех хостингах, где SSI не проходит.

Можно также вставлять маленькие скрипты в некоторые атрибуты тэгов HTML, но об этом — чуть позже.

Комментарии

Говорят, остались ещё такие браузеры, которые не понимают скриптов. Встречаются и маньяки-пользователи, которые настолько задвинуты на безопасности, что отключают скрипты. В этой ситуации скрипт выполняться не будет, но код его, та самая «абракадабра», просто вывалится на экран.

Так вот, чтобы не вываливалась, мы заключаем её в HTML-комментарии.

```
<script type="text/javascript">
<!--
Код скрипта
// -->
</script>
```

Ммм... а что это за два слэша перед закрытием комментария?

Закрывающий комментарий находится уже в «теле» скрипта. А JavaScript не понимает этих корявых HTML'ских скобок, для него это инородное тело, генерирующее ошибку. Значит, нужно скрыть этот закрывающий тэг от скрипта, поместив его как комментарий JavaScript. У JavaScript комментарии выглядят несколько изящнее: //. После этого знака скрипт не видит закрывающую скобку HTML, а HTML благополучно скрывает текст скрипта, и на экране не видно никаких «левых» записей.

Раз уж мы коснулись комментариев, то надо сказать, что в JavaScript они имеют две формы — такие же, как в C и C++ (а кстати и в CSS тоже).

```
// Эта форма комментария
// действует только на одну строку,
// то есть на каждой новой строке
// нужно выставлять знак комментария.
А это уже код скрипта...
/* А эта форма комментария
действует на сколько угодно строк
```

```
до тех пор, пока не натолкнётся
на закрывающий значок,
зеркально отражающий начальный. */
А теперь опять код скрипта...
```

Не путайте комментарии HTML и комментарии JavaScript!

Это разные языки, хотя и сосуществуют «в одном флаконе». Или, точнее, в одной банке. Как пауки...

Допустим, с помощью JavaScript Вы сделали из двух картинок что-то вроде анимированного баннера. Тогда Вы можете порадовать пользователей «убогих» браузеров (а Вы, я надеюсь, не полностью лишены альтруизма) лицезрением хотя бы одной из этих картинок с помощью тэга **<noscript>**:

```
<script type="text/javascript">
<!--
Код Вашего баннера
// -->
</script>
<noscript>
<img src="одна_из_картинок.gif">
</noscript>
```

В каких редакторах писать скрипты

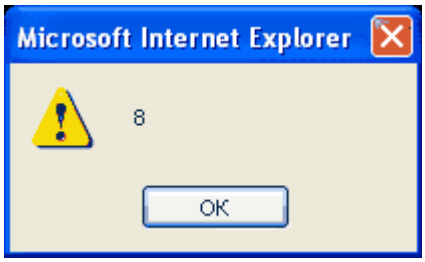
Возможно, существуют какие-то специальные редакторы для *JavaScript*. Обычно скрипты пишут в тех же редакторах, какие используют для создания web-страниц. Написание скрипта в этих редакторах ничем не отличается от написания его в простом блокноте, кроме подсветки кода. А она иногда очень помогает.

Сразу предупреждаю: практически этот пример абсолютно бесполезный. Но в нём сконцентрированы многие ключевые понятия языка javascript и его синтаксиса.

```
<html>
<head>
<title>Пример 1</title>
</head>
<body>
<script type="text/javascript">
<!--
var x = 5;
var y = x + 3;
alert(y);
// -->
</script>
</body>
</html>
```

Собственно скрипте

Если Вы скопируете этот код в чистую страницу Блокнота и сохраните её как файл.html, то при открытии файла увидите следующее:



Разберём, как это выходит.

```
var x = 5;
```

Здесь мы объявляем переменную x, которая равна 5. А знаете ли Вы, что такое **переменная**?

Если нет, прочитайте пояснение.

Как работает компьютер? Все данные сохраняются на диске, и место, где они лежат, должно быть помечено, чтобы было ясно, где что искать. Любая программа (а скрипт — это не что иное, как маленькая программа) работает с какими-то данными. Поэтому удобно сразу «забить место» для них. Вот этим местом, этим помеченным участком памяти и становится переменная. Почему «переменная»? Потому что этот участок может заполняться разными значениями. Например, когда мы работаем с калькулятором, то числа и действия с ними, которые мы вводим, записываются программой в соответствующие переменные. А при нажатии кнопки выполнения вступает в действие алгоритм, использующий те значения, которые мы ввели.

В коде программы переменные обозначаются именами, которые мы сами для них придумываем. Для создания имён существуют определённые правила, которые в разных языках программирования могут отличаться. Те ограничения, которые будут описаны ниже, относятся конкретно к языку JavaScript.

var — ключевое слово для объявления переменной (по-английски variable).
x — имя переменной.

> **Ограничения:** В имени переменной можно использовать только латинские буквы (любого регистра), цифры и символ подчёркивания. При этом переменная не должна начинаться с цифр. И никаких пробелов.

Правильные имена переменных:

myVar
my_var
text01
_text
button12bis

Неправильные имена переменных:

my Var
my-var
01text
текст01

Язык JavaScript чувствителен к регистру.

myvar, MyVar и *myVar* — разные переменные.

В этом скрипте переменной сразу при объявлении присваивается значение. Это не обязательно. Значение можно присвоить и позже. В конце строки стоит точка с запятой. Это в данном случае тоже не обязательно. Но в больших и сложных скриптах это иногда оказывается важным, поэтому я демонстрирую полную подробную запись. Строка выглядит просто **x = 5**, явное объявление **var** здесь тоже не обязательно. Но всё-таки желательно.

В следующей строке, как Вы уже можете догадаться, объявлена переменная **y**. Ей присвоено значение относительно уже объявленной **x**, на **3** больше, чем **x**.

А затем вызывается метод **alert()**.

Этот метод выводит на экран окно диалога с сообщением, указанным в скобках. Это синтаксис всех методов *javascript*: имя метода и скобки с его содержимым.

Содержимое этого метода — значение переменной y, и в окне диалога мы увидим восьмёрку. Поняли, почему?

Полезная мелочь

Мы с Вами познакомились с методом **alert()**, то вот его простое и полезное применение: иногда какие-то страницы сайта ещё не сделаны, а ссылки уже приготовлены. Неприятно бывает попадать на «**страницу 404**». Не очень также приятно ждать, пока она загрузится, а потом узнать, что раздел в разработке. Я всегда затыкаю подобные ссылки следующим образом:

```
<a href="javascript: alert('Страница в разработке');">Пункт меню</a>
```

Кстати, вот вам ещё один способ внедрения кода *JavaScript* в код HTML. Здесь он вставляется в атрибут **href** тэга ссылки и вводится через ключевое слово «*javascript:* » (с двоеточием и последующим пробелом: всегда обращайте внимание на синтаксические мелочи). Обратите также внимание на традиционные двойные кавычки значения атрибута HTML и «вложенные» одиночные кавычки в тексте самого скрипта.

Очень скоро мы узнаем и о специальных «событийных» атрибутах тэгов HTML, например, *onClick*, которые специально предназначены для введения кода *JavaScript* и не требуют ключевого слова.

Итак, мы узнали:

как объявить скрипт в документе HTML, какие формы имеют комментарии JavaScript, как объявлять переменные и давать им правильные имена, а также синтаксис методов и конкретно метод alert().

А также научились:

ставить «заглушку» на ссылку в виде сообщения **alert()**.

Введение в типы данных

- типы данных number и string;
- методы документа write() и writeln();
- склеивание строк и переменных

Числа и строки

Давайте изменим прошлый скрипт:

```
<html>
```

```
<head>
   <title>Пример 2</title>
</head>
<body>
<script type="text/javascript">
<!--
   var x = "5";
   var y = x + "3";
   alert(y);
// -->
</script>
</body>
</html>
```

Нашли различие?

А теперь посмотрим результат

Почему это произошло? Потому что сложились не числа, а строки.

Для того, чтобы компьютер правильно интерпретировал наши данные, они подразделяются на типы.

Во всех языках программирования существуют различные типы данных. В *JavaScript* таких типов 6, и их классификация заметно отличается от той, что распространена в большинстве языков. Сейчас нам важны два из них: **number** (число) и **string** (строка). **Number** всегда обозначается просто числом. А **string** — любым выражением, заключённым в кавычки. Всё, что заключено в кавычки, читается как набор символов: букв, цифр или других каких значков. При сложении строк к одному набору приклеивается другой. Так, к нашей пятёрке приклеилась тройка.

Кавычки можно использовать как двойные, так и одиночные. Бывает, что одни кавычки надо вложить в другие, как в моём примере со ссылкой из прошлого урока. Напомню:

```
<a href="javascript: alert('Страница в разработке');">Пункт меню</a>
```

Здесь в двойные кавычки заключено значение атрибута href тэга **<a>**, а строка для вложенного в него метода **alert()** заключена в одиночные кавычки.

А кстати, как Вы думаете, что выскочит, если последнюю строку скрипта записать как **alert("y")**?

А что делать, если мы хотим отобразить кавычки в сообщении? Например: Страница "Новости" ещё не готова. Для этого перед каждой отображаемой кавычкой нужно поставить обратный слэш — **«\»**.

```
<script type="text/javascript">
<!--
alert("Страница \"Новости\" ещё не готова");
// -->
```

```
</script>
```

НО:

Если вставим это в ссылку,

```
<a href="javascript: alert('Страница \"Новости\" ещё не готова');">Новости</a>
```

то нам выдадут ошибку, потому что HTML этих обозначений не понимает.

А можно ли что-нибудь сделать?

Можно. Обмануть HTML.

Как?

Нужно просто изменить стиль кавычек: внешние сделать одиночными, а внутренние — двойными:

```
<a href='javascript: alert("Страница \"Новости\" ещё не готова");'>Новости</a>
```

На клавиатуре они набираются ALT+0171 и ALT+0187. Их можно внедрить тремя способами:

1. Просто набрать с помощью ALT:

   ```
   <a href="javascript: alert('Страница «Новости» ещё не готова');">Новости</a>
   ```

2. Указать их как символы Unicode:

   ```
   <a href="javascript: alert('Страница &#171;Новости&#187; ещё не готова');">Новости</a>
   ```

3. Воспользоваться спецсимволами HTML:

   ```
   <a href="javascript: alert('Страница &laquo;Новости&raquo; ещё не готова');">Новости</a>
   ```

Примечание: последний способ отображает тот стиль кавычек, который используется в национальных настройках Windows. Так что те, у кого винды английские, увидят другие кавычки.

Пишем прямо в документ

А теперь ещё изменим скрипт, а заодно познакомимся с новым(и) методом(/ами).

```
<script type="text/javascript">
<!--
   var x = 5;
   var y = x + 3;
   document.writeln("<h2 align='center'>" + y + "</h2>");
// -->
</script>
```

или

```
<script type="text/javascript">
<!--
   var x = "авто";
   var y = x + "ручка";
   document.write("<h2 align='center'>" + y + "</h2>");
// -->
</script>
```

Для наглядности использования этих методов я включил туда и тэг *HTML* (кстати, вот опять необходимость двух уровней разных кавычек). Да, таким образом можно загнать в скрипт целую web-страницу (только нельзя разбивать строку, то есть весь код *HTML*, загнанный в метод, должен быть написан одной длинной строкой). Правда, для удобства можно многократно использовать этот метод:

```
<script type="text/javascript">
<!--
document.writeln("[первый кусок кода HTML]")
document.writeln("[второй кусок кода HTML]")
document.writeln("[третий кусок кода HTML]")
document.writeln("[и т.д.]")
// -->
</script>
```

В первом и последнем случаях я использовал метод документа **writeln()**, а во втором — также метод документа, но **write()**.

В чём разница?

Почти ни в чём. Долгое время я не мог понять разницы не находил ответов в руководствах (так и не нашёл). Но методом случайного тыка выяснил, что если написать:

```
<script type="text/javascript">
<!--
document.write("слово")
document.write("слово")
document.write("слово")
document.write("слово")
// -->
</script>
```

браузер выдаст:

словословословослово

А если написать:

```
<script type="text/javascript">
<!--
document.writeln("слово")
document.writeln("слово")
document.writeln("слово")
document.writeln("слово")
// -->
</script>
```

то выйдет:

слово слово слово слово

Кроме того, если первый из этих скриптов заключить в тэг **<pre></pre>**, то ничего не произойдёт. А со вторым сработает:

слово
слово
слово
слово

То есть получается, что метод **write()** просто выводит данные, содержащиеся в нём, а метод

writeln() (это означает «write line» — «писать строку») представляет их как строку кода *HTML*, что реально отражается в браузере как текст с последующим пробелом. Таким образом, в примере с кусками кода целесообразнее использовать **writeln()**: на странице это в большинстве случаев никак не отразится, но именно с такой разбивкой вводим *HTML*-код руками.

И ещё немного о сложении строк

Метод **document.write()** может содержать как и непосредственно строки (взятые в кавычки), так и имена переменных, содержащих строки (имена переменных вводятся в метод без кавычек). Можно комбинировать и то, и другое.

Например:

```
<html>
<head>
   <title>Пример 3</title>
<script type="text/javascript">
var a = " понедельник", b = "о вторник", c = " среду", d = " четверг", e = " пятницу",
f = " субботу", g = " воскресенье", h = " узнаете свежие новости.", i = " увидите
новые поступления.";
</script>
</head>
<body>
<script type="text/javascript">
<!--
document.write("Заходите к нам в" + a + ", и Вы" + h + "<br>")
document.write("Заходите к нам в" + b + ", и Вы" + i + "<br>")
document.write("Заходите к нам в" + d + ", и Вы" + h + "<br>")
document.write("Заходите к нам в" + f + ", и Вы" + i)
// -->
</script>
</body>
</html>
```

Обратите внимание на «химию» при написании «во вторник», которая позволяет внедрить изменённый вариант предлога, не нарушая общего алгоритма (например, если бы повторяющийся текст нам захотелось загнать в переменные).

(Все перечисленные в объявлении переменные нужно писать в одну длинную строку. Если в **document.write()** не ввести тэг
, то весь отображаемый текст будет в одну строку.)

С помощью *JavaScript* можно также сделать, чтобы нужные значения автоматически вставлялись и изменялись в зависимости от текущей даты или от страницы, на которой этот текст находится. Когда мы дойдём до изучения этих возможностей, то ещё вернёмся к этой болванке.

Первое свиданье с функцией

- делаем текст невидимым;
- учимся записывать функцию;
- оператор if...else;

- анатомия функции и новые объекты

Немного CSS

Обратимся к **CSS**. Среди свойств **CSS** есть свойство **display**.

- none
- block
- inline-block
- inline
- table-header-group
- table-footer-group
- list-item

Нам нужны два: **none** и **block**.

«display: none;» означает «скрыть», а **«display: block;»** — «показать».

Вот как выглядит тот код который использует это свойство:

```
<p align="justify"><b>Стоп! А знаете ли Вы, что такое переменная?</b></p>
<p align="justify">Если нет, то нажмите эту <a href="javascript:
displ('var')"><i>ссылочку</i></a> и прочитайте пояснение.</p>
<div id="var" style="display: none;">
<p align="justify">Как работает компьютер? Все данные сохраняются на диске, и
место, где они лежат, должно быть помечено, чтобы было ясно, где что искать.
Любая программа (а скрипт — это не что иное, как маленькая программа)
работает с какими-то данными. Поэтому удобно сразу «забить место» для них. Вот
этим местом, этим <b>помеченным участком памяти</b> и становится переменная.
Почему «переменная»? Потому что этот участок может заполняться разными
значениями. Например, когда мы работаем с калькулятором, то числа и действия с
ними, которые мы вводим, записываются программой в соответствующие
переменные. А при нажатии кнопки выполнения вступает в действие алгоритм,
использующий те значения, которые мы ввели.</p>
<p align="justify">В коде программы переменные обозначаются именами, которые
мы сами для них придумываем. Для создания имён существуют определённые
правила, которые в разных языках программирования могут отличаться. Те
ограничения, которые будут описаны ниже, относятся конкретно к языку
JavaScript.</p>
<p align="right"><a href="javascript: displ('var')"><i>Закрыть пояснение.</i></a></p>
</div>
```

Начнём разбираться.

Скрытые **<p>** -абзацы упрятаны в ** **, которому дано персональное имя-идентификатор **(id="var")** и «невидимый» стиль **(style="display: none;")**.

В двух ссылках, открывающей и закрывающей этот текст, мы видим один и тот же пока непонятный код. (Кстати, закрыть текст можно точно так же и верхней ссылкой, просто я поставил ещё одну, чтобы никто голову не ломал, как это сделать. Но это уже вопрос так называемого «юзабилити».)

В коде ссылок фигурирует знакомое **"var"**, то есть «имя собственное» нашего **div**'а. А вот **displ()** — это имя функции, которая помещена между <head> и </head>.

Учимся писать функции

Это очень простенькая функция. Вот как она выглядит:

```
<script type="text/javascript">
function displ(nnn) {
if (document.getElementById(nnn).style.display == 'none')
{document.getElementById(nnn).style.display = 'block'}
else {document.getElementById(nnn).style.display = 'none'}
}
</script>
```

Запомните синтаксис любой функции. У функции, как и у переменной, должно быть своё **уникальное имя**. После имени идут, как и в методе, **круглые скобки**. В них помещаются **аргументы** функции, тоже со своими уникальными именами. Если аргументов несколько, они перечисляются через запятые. В них функция передаёт нужные значения. Бывают функции, которые не требуют аргументов. Тогда круглые скобки всё равно ставятся, но остаются пустыми. Всё содержание или, как говорят программисты, «тело» функции (от англ. "body") заключено в **фигурные скобки**:

```
function имя_функции(аргумент1, аргумент2) {
код в теле функции
}
```

Можно и так:

```
function имя_функции(аргумент1, аргумент2)
{
код в теле функции
}
```

И так:

```
function имя_функции(аргумент1, аргумент2)
{код в теле функции}
```

И даже так:

```
function имя_функции(аргумент1, аргумент2) {код в теле функции}
```

То есть принудительного разрыва или, наоборот, «неразрыва» строки между скобками и другими элементами не требуется. Но **обе скобки должны быть**.

Синтаксис любого языка программирования надо зарубить себе на носу, особенно *HTML*'щикам, так как *HTML* допускает некоторые варианты и, так сказать, «попустительство». Настоящий язык программирования — **НЕТ**!

Итак, наша функция **displ(nnn)** имеет аргумент **nnn**, который, как видно из фрагмента кода HTML, приведённого выше, берёт на себя id (идентификатор, уникальное имя) того элемента (тэга), который мы хотим показать или спрятать.

Новым элементом **JavaScript в данном примере является** оператор **if...else**.

Оператор if...else

Как правило, программы пишут для того, чтобы они, эти программы, облегчали наш труд и делали вместо нас какие-то действия. Для выполнения этих действий, или, иными словами, операций, языки программирования напичканы всевозможными операторами. Самый простой и употребительный оператор — **if...else**.

«**Если (if)** магазин открыт, я пойду в магазин. **В противном случае (else)** я пойду пить пиво.»

«Магазин открыт» является **условием**, при котором выполняется **действие 1** («я пойду в магазин»).

При отсутствии этого условия (**else**) выполняется **действие 2** («я пойду пить пиво»).

Напишем этот бред по-яваскриптски:

```
if (магазин открыт)
{я пойду в магазин}
else {я пойду пить пиво}
```

И снова обращаем внимание на синтаксис.

Условие — в круглых скобках.

Действия — в фигурных скобках.

Есть ещё и более краткая запись: без ключевых слов **if** и **else**, а с помощью специальных символов. Но на ранних этапах не рекомендую этим увлекаться. Попробуйте разобраться в этой маленькой головоломке сами:

```
магазин открыт?я пойду в магазин:я пойду пить пиво
```

Объекты, объекты, объекты...

Посмотрим «иф» из нашей функции.

Его условие — **document.getElementById(nnn).style.display == 'none'**

Что бы это значило?

Мы уже встречали такой объект document с методом **write()**. Помните — **document.write("всякий там разный текст")**?

Объект **document** — это наш с вами документ, файл, в который мы пишем код *HTML*. И у документа много разных методов. Есть и такой: **getElementById()**, то есть «взять элемент по его идентификатору». А в скобках — этот самый идентификатор. Обратите внимание на прописные и строчные буквы в имени метода. *JavaScript* чувствителен к регистру!

Найденный этим методом элемент — тоже объект, только рангом пониже, чем **document**. И у него есть свои методы и свойства. Некоторые из них соответствуют атрибутам тэгов *HTML*, вот как, например, свойство **style**.

Которое тоже, в свою очередь, объект со своими свойствами, среди которых и **display**.

То есть все объекты, являющиеся методами и свойствами других объектов, нанизываются через точку по нисходящей иерархии.

И вот у этого последнего свойства-объекта нам нужно поймать значение **none**. Обратите внимание на сдвоенный знак равенства. Это, собственно, и есть знак равенства в *JavaScript*. Обычный, одиночный — это не равенство, а назначение, приравнивание.

Что же получилось? По-русски говоря, «если в документе есть элемент по имени такой-то, и он невидим» — вот оно, наше условие.

А теперь всё очень просто. Если этот самый элемент невидим, то он должен стать видимым. То есть в фигурные скобки действия скопируем то же самое, но вместо **none** поставим **block**.

В противном же случае (а «противный случай» наступает, когда наш элемент отображается) он снова должен стать невидимкой. Значит в действие для **else** просто копируем то, что было в условии.

Теперь посмотрите: вместо идентификатора элемента мы вписали имя аргумента функции. Теперь, если мы вызовем эту функцию из кода *HTML*, поставив в качестве аргумента реально существующий идентификатор, то функция обратится именно к нему.

И такая вот подстава. Вы обратили внимание на одиночные кавычки? Почему они здесь, когда вроде бы ничего не мешает им быть двойными?

Оказывается, мешает!

В документе *HTML* функция выполняется внутри атрибута **href**, в котором она уже закавычена. И с двойными кавычками она не выполнится. Это проверено методом тыка() на собственной шкуре и крыше и бессонных ночах.

Переменные

Мы уже умеем их называть. Объявляли мы их только одним способом, а способов этих несколько.

1. Непосредственным назначением:

```
myName = "Андрей";
```

2. Без назначения, с помощью ключевого слова:

```
var myHome;
```

3. С помощью ключевого слова и с непосредственным назначением:

```
var myAge = 46;
```

То есть если мы сразу задаём переменной конкретное значение, то ключевое слово **var** не обязательно.

Если мы объявляем переменную «про запас», ничего пока на неё не назначая, то ключевое слово **var** обязательно.

С помощью одного ключевого слова можно назначить несколько переменных, как с непосредственным назначением, так и без него, переменные разделяются запятыми:

```
var myName = "Андрей", myHome, myAge = 46;
```

Переменные имеют область действия, они делятся на *глобальные* и *локальные*. Переменные, объявленные в теле функции, действуют только внутри функции. Они локальны. Остальные — глобальны и действуют с момента объявления до конца программы. Поэтому глобальные переменные «про запас» часто объявляют в самом начале скрипта. И ещё одно *«правило для дураков»*: нельзя давать переменным имена, которые соответствуют ключевым словам. Но поскольку «в дураках» может оказаться любой недостаточно сведущий человек, я об этом упоминаю. И даже приведу небольшой список наиболее употребительных ключевых слов:

var, if, else, const, true, false, function, super, switch, for, while

Некоторые из них нам уже встречались, другие встретятся в следующих уроках. Вообще-то их гораздо больше, и по возможности полный словарь ключевых слов я также собираюсь включить в свою шпаргалку.

Примечание: Я тут недавно употребил слово *var* для имени элемента. Возможно, это не слишком удачно с педагогической точки зрения. Но вполне безопасно, так как это не переменная, а строка, которая в кавычках.

Типы данных

Вот полная таблица всех типов данных из учебника В. Дунаева:

Тип данных	Примеры	Описание значений
Строковый (string)	"Привет" "д.т. 123-4567"	Последовательность символов, заключенная в кавычки, двойные или одиночные
Числовой (number)	3.14 -567 +2.5	Число, последовательность цифр, перед которой может быть указан знак числа (+ или -); перед положительными числами не обязательно ставить знак «+»; целая и дробная части чисел разделяются точкой. Число записывается без кавычек
Логический (булевский, boolean)	true false	true (истина, да) или false (ложь, нет); возможны только два значения
Null		Отсутствие какого бы то ни было значения
Объект (object)		Программный объект, определяемый своими свойствами. В частности, массив также является объектом
Функция (function)		Определение функции — программного кода, выполнение которого может возвращать некоторое значение

Что такое логический (булев) тип данных?

Предположим, нам нужно из сотни вариантов выбрать только те, которые соответствуют определённому условию. Сколько их и есть ли они вообще, мы не знаем, так как проверяем базу данных, которая всё время обновляется. Мы зададим нужное условие и назначим на него булеву переменную со значением **true**. И зададим ещё одно условие: девочки **(true)** налево, мальчики **(false)** направо. Вот примерно так это выглядит «на пальцах». А когда доберёмся до конкретных примеров, поговорим серьёзнее.

Всё-таки удивительный человек был этот Джордж Буль (между прочим, отец писательницы Войнич, которая «Овода» написала). При жизни его считали хоть и гением, но большим чудаком, носившимся с какой-то никому не нужной логической алгеброй. А вот потом выяснилось, что программистам без неё, ну, никак.

Null следует отличать от нулевого значения. Нулевое значение — это, всё-таки, значение. А вот **Null** — это отсутствие какого-либо значения, даже нулевого.

Об *объектах* и *функциях* как типах данных будем говорить, когда рассмотрим их поближе.

Скажу только, что в других языках протраммирования **типы данных** понимаются несколько по-другому. Там бывает несколько числовых типов (короткое целое число, длинное целое число, число с плавающей запятой и т.д.), а **функции** — это функции, и никакого отношения к типам данных не имеют. Но это так, к слову. Вдруг Вы уже изучали какой-нибудь другой язык и теперь сильно удивились.

Операторы

Операторов много, и все они очень разные. От простых операторов сложения и вычитания до сложных логических операторов, выполняющих хитроумные последовательности различных действий.

Пробежимся по их классификации.

Я предлагаю разделить все операторы на две группы: *простые* и *сложные*. Под **простыми** будем понимать те, которые выполняют какое-то одно конкретное действие и записываются неким условным обозначением. **Сложные** выполняют целый алгоритм и включают в свою

запись фрагменты кода этого алгоритма.

Из простых нам уже встречались **комментарии**, которые просто отсекают фрагмент текста от общего кода, и один из операторов **присвоения** в виде простого знака равенства, с помощью которого мы присваивали значения переменным. Встречался нам и оператор **равенства** (из группы операторов сравнения) в виде сдвоенного знака равенства.

Из сложных — мы познакомились с одним из операторов **условного перехода: if...else**.

Термины «простые» и «сложные» — моё «изобретение». Внутри этих групп я придерживаюсь традиционной классификации.

Вот самая общая сводная таблица групп операторов:

<table>
<tr><th>Простые</th><th colspan="2">Сложные</th></tr>
<tr><td>Комментарии</td><td rowspan="2">Условного перехода</td><td>if...else</td></tr>
<tr><td>Арифметические</td><td>switch</td></tr>
<tr><td>Сравнения</td><td rowspan="3">Цикла</td><td>for</td></tr>
<tr><td>Присвоения</td><td>while</td></tr>
<tr><td>Логические</td><td>do...while</td></tr>
</table>

Простые операторы

Ещё раз о комментариях

О комментариях уже было сказано всё, что о них можно сказать. Лишний раз призываю: не путайте комментарии *HTML* и комментарии *JavaScript*.

И маленький совет: если Вы хотите удалить кусок кода, но предполагаете, что от Вам ещё может пригодиться, не стирайте его, а просто закомментируйте. Это бывает полезно при отладке скрипта, когда, перепробовав массу вариантов, мы всё же возвращаемся ближе к исходному.

Арифметические операторы

Вот их полный список:

Оператор	Название	Пример
+	Сложение	X+Y
-	Вычитание	X-Y
*	Умножение	X*Y
/	Деление	X/Y
%	Деление по модулю	X%Y
++	Увеличение на 1	X++
--	Уменьшение на 1	Y--

Для «школьных» операторов, думаю, комментарии излишни. Кроме одного: так же, как и в школе, минус употребляется не только для вычитания, но и для обозначения отрицательного числа.

Что такое *«деление по модулю»*? Этот оператор возвращает не частное, а **остаток** от деления. Например, **12%3** возвратит **0**, а **14%3** возвратит **2**. Понятно?

Операторы увеличения и уменьшения действуют так:

x++ то же, что **x+1**

y-- то же, что **y-1**

Можно использовать как эту, так и обычную «школьную» запись. Сначала я так и делал, не понимая этого «выверта». Но потом обнаружил, что в некоторых длинных кодах с применением операторов цикла это, оказывается, удобно. Впрочем, даже такой механический язык, как язык программирования, оставляет некоторую возможность для индивидуального стиля программиста.

Операторы сравнения

Оператор	Название	Пример
==	Равно	X == Y
!=	Не равно	X != Y
>	Больше, чем	X > Y
>=	Больше или равно	X >= Y
<	Меньше, чем	X < Y
<=	Меньше или равно	X <=Y

Эти операторы сравнивают не только числа, но и строки, и булевы значения.

Каким образом?

Ну, с булевыми значениями просто. Значение **true** всегда равно **единице**, а **false** — **нулю**.

А строки...

У каждой буквы и любой закорючки из шрифта есть числовые значения *ANSII*, *Unicode* и т.д. Вот эти значения и сравнивают операторы. Числовое значение строки равно сумме значений всех её символов. В какой кодировке? Это может зависеть от типа и версии браузера, от местных национальных настроек. Во всяком случае, у цифр, латинских букв и «препинаков» коды единообразные. В числовых значениях кириллицы и «санскритицы» (деванагари) у меня всегда проходил *Unicode*. Хотя и не уверен, что это «истина для всех». Вообще проблеме кодировок можно посвятить целую диссертацию...

Но если не вдаваться в тонкости, то, например, латинское маленькое «**a**» во всех популярных кодировках равно **97**, «**b**» — **98**, и т.д. То есть **ab == 195**, **ac == 196**, значит, **ab < ac** (или **ac > ab**).

Вы можете спросить, кому это всё нужно. Может быть, лично Вам это никогда и не понадобится. А вдруг да и понадобится — всякое бывает... Вот тогда-то Вы и вспомните... что есть страничка, на которой это можно подглядеть.

Операторы присвоения

Кроме основного оператора присвоения, обозначающегося знаком равенства, существует несколько дополнительных, в которых присвоение комбинируется с арифметическими действиями. В следующей таблице показаны примеры этих операторов и более привычные выражения, которые они заменяют. Эти операторы, как и два последних арифметических, пришли сюда из языка C (си).

О С Н О В Н О Й	
Оператор	**Пример**

=	var x = "моя строка"

Д О П О Л Н И Т Е Л Ь Н Ы Е		
Оператор	**Пример**	**Эквивалентное выражение**
+=	X+=Y	X = X + Y
-=	X-=Y	X = X - Y
*=	X*=Y	X = X * Y
/=	X/=Y	X = X / Y
%=	X%=Y	X = X % Y

Вообще *JavaScript* по своему синтаксису относится к языкам семейства *C* и похож он на *C* и *C++* примерно так же, как какой-нибудь креольский на классический французский.

Логические операторы

Оператор	**Название**	**Пример**
!	Отрицание (НЕ)	!X
&&	И	X && Y
||	ИЛИ	X || Y

Предположим, есть две фирмы. Одной руководит Иванов, другой — Петров. И у каждого есть вакансии. Но Иванову нужен человек не старше 35 лет **и** со стажем не менее 10 лет, а у Петрова условия чуть помягче: **либо** не старше 35, **либо** со стажем не менее 10. И пришли к одному и другому 4 человека. Запустили Иванов с Петровым свои компьютеры, и вот что эти компьютеры выдали:

X (возраст)	**Y (стаж)**	**X && Y (Иванов)**	**X || Y (Петров)**
35 лет	10 лет	true (да)	true (да)
28 лет	8 лет	false (нет)	true (да)
48 лет	15 лет	false (нет)	true (да)
36 лет	9 лет	false (нет)	false (нет)

Оператор if...else

Принцип работы и синтаксис мы уже знаем. Напомню ещё раз:

```
if (условие)
{этот код работает при выполненном условии}
else {этот код работает, если условие не выполнено}
```

А если при невыполненном условии нам вообще никакого кода не надо?

Тогда так:

```
if (условие)
{этот код работает при выполненном условии}
// И больше ничего
```

А можно ли смоделировать в этом операторе следующую ситуацию?

Предположим, у Вас очень необычный дизайн, который никак не подогнать под разные типы браузеров. И вот Вы героически решили написать три почти одинаковых страницы, одна из которых правильно смотрится в *IE*, другая — в *Netscape* и *Mozilla*, а третья — в *Opera*. А теперь надо сделать такой скрипт, который при нажатии на ссылку проверил бы браузер пользователя и открыл бы ему нужную страницу.

Если (условие1 — IE), то {код1 — открытие index_ie.html}, если нет, то если (условие2 — NS или MOZILLA), то {код2 — открытие index_ns.html}, если нет, то если (условие3 — OPERA), то {код3 — открытие index_op.html}.

Этот алгоритм несовершенен: если браузер не подходит ни под одно из определений, то по ссылке не откроется ничего. Но при таком раскладе яснее проступит синтаксис вложенных операторов, что я и хочу в первую очередь продемонстрировать.

Для определения типа браузера у объекта **navigator** существует свойство **appName**

У *IE* это свойство имеет значение *"Microsoft Internet Explorer"*, у *NS* и *Mozilla* — *"Netscape"*, а у *Opera* — *"Opera"*.

Заготовим тэги для скрипта

```
<script type="text/javascript">

</script>
```

и начнём его заполнять. Для начала предлагаю объявить переменные для имён браузеров: и основной код компактнее будет, и потренируемся, кстати.

```
<script type="text/javascript">
  var IE = (navigator.appName == "Microsoft Internet Explorer");
  var OP = (navigator.appName == "Opera");
  var NS = (navigator.appName == "Netscape");

</script>
```

А теперь добавим основной код, и скрипт будет выглядеть так:

```
<script type="text/javascript">
  var IE = (navigator.appName == "Microsoft
Internet Explorer");
  var OP = (navigator.appName == "Opera");
  var NS = (navigator.appName == "Netscape");

  if (IE) {self.parent.location = "index_ie.html";}
  else {
    if (NS) {self.parent.location = "index_ns.html";}
    else {
      if (OP) {self.parent.location =
"index_op.html";}
    }
  }
</script>
```

Обратите внимание на фигурные скобки. Первый альтернативный код включает в себя все вложенные условия и выполнения и закрывается скобкой в самом конце. Все вложенные операторы вкладываются друг в друга, как матрёшки. Ниже я изобразил «голую» схему, в которую вложил больше операторов, и соединил стрелочками их начала и концы.

Оба кода расположены традиционной программистской «лесенкой», но это делается не по

требованиям языка, а для нашего «человеческого» удобства чтения.

```
if (условие){код 1}
else {
  if (условие){код 2}
  else {
    if (условие){код 3}
    else {
      if (условие){код 4}
      else {
        if (условие){код 5}
        else {код 6}
      }
    }
  }
}
```

я деталь: последний виток завершён кодом else,
ых) случаях. Если мы, допустим, зададим
ывали и все остальные, не учтённые нами
ь так:

```
<script type="text/javascript">
  var OP = (navigator.appName == "Opera");
  var NS = (navigator.appName == "Netscape");
/* Уже излишне объявлять переменную для IE,
так как он задан по умолчанию */
  if (OP) {self.parent.location = "index_op.html";}
  else {
    if (NS) {self.parent.location = "index_ns.html";}
    else {self.parent.location = "index_ie.html";}
//страница по умолчанию
  }
</script>
```

Но тогда бы получилось меньше вложений, а моей задачей было объяснить технику записи вложенных операторов.

И вот ещё один маленький скрипт с этим оператором, который может оказаться весьма полезным. Привожу его, во-первых, для того, чтобы из сегодняшнего теоретического занудства Вы извлекли максимум пользы, а во-вторых, чтобы Вам примелькалось словосочетание, которое я сегодня сознательно не объяснял: **self.parent.location**. Ну и **self.parent.frames.length** тоже ничего. Это я Вас потихоньку веду к иерархии объектов. Через подсознание.

Так вот. Есть на свете такие фреймы, которые кто-то любит, а кто-то ненавидит. Если Вы совсем зелёный новичок, то, наверно, любите. Если пару раз столкнулись с проблемой фрейма в поисковике, то, возможно, успели возненавидеть. А суть в том, что если дана ссылка на страницу, которая находится во фрейме, а сама по себе является как бы неполноценной (то есть без навигации и т.п.) то по этой ссылке Вы попадёте прямо на неё в первозданно-недоделанном виде, а не туда, где она симпатично упакована во фрейм. Но от этого есть элементарное противоядие. Когда я его узнал, то вновь возлюбил фреймы.

Итак, в «голову» *(head)* этой злосчастной страницы нужно вживить ма-аленький такой имплант, который переадресовывает её на страницу-«хозяйку».

Вместо *адрес/имя_основного_документа.html* вписываете эту самую «хозяйку», ну, скажем, там, *http://mysite.ru/index.html*.

```
<script type="text/javascript">
   if (self.parent.frames.length == 0)
   {self.parent.location = "адрес/имя_основного_документа.html";}
</script>
```

Теперь, щёлкнув по ссылке на эту подчинённую страницу, мы откроем её в нужном фрейме. Но всё же чрезмерно не увлекайтесь фреймами: у них и другие сюрпризы бывают.

Оператор switch

С английского **switch** переводится как «переключать». Это оператор-переключатель. Вот его структура (в квадратных скобках обозначены необязательные части):

```
switch (выражение) {
   case вариант1:
      код
      [break]
   case вариант2:
      код
      [break]
   [default:
код]
}
```

Выражение — это контрольное, тестовое значение, с которым будут сравниваться **варианты**.

Выражение может быть числовым, строковым или булевым значением. При булевом значении возможны только 2 варианта: **true** и **false** (или 1 и 0). При остальных вариантов может быть сколько угодно.

Для каждого варианта пишется определённый код. Весь перебор вариантов заключён в фигурные скобки.

Сначала рассмотрим вариант работы оператора без использования **break** и **default**.

Когда последовательное тестирование вариантов натыкается на вариант, совпадающий с тестовым, то выполняется код этого варианта и все последующие коды.

Например:

```
var x = 0;
switch (x) {
   case 11:
      alert("Эй!");
   case 0:
      alert("Осторожно!");
   case "stroka":
      alert("Злая");
   case 235:
      alert("собака!");
}
```

Мы объявили переменную и назначили на неё числовое значение 0. Далее я специально ввёл в оператор этакий шизофренический набор вариантов (11, 0, "stroka", 235), чтобы Вы поняли, что от какой-либо последовательности или логичности в этом наборе ровным счётом ничего не зависит.

Вариант 11 проносится мимо, а начиная с варианта 0 при выполнении начинают по очереди выпадать алерты: «Осторожно!» «Злая» «собака!»

Ключевое слово **break** прерывает дальнейшее выполнение кода. Давайте «брекнем» наш нолик:

```
var x = 0;
switch (x) {
   case 11:
      alert("Эй!");
   case 0:
      alert("Осторожно!");
      break
```

```
   case "stroka":
      alert("Злая");
   case 235:
      alert("собака!");
}
```

Теперь выскакивает только один алерт: «Осторожно!»

Если нет ни одного варианта, соответствующего выражению, то ничего выполняться не будет. Но если мы напишем код для ключевого слова **default**, то именно он будет выполняться при отсутствии соответствующих вариантов. В следующем примере изменено значение переменной, используемой в выражении, и добавлен вариант **default**, который в данном случае и будет выполняться.

```
var x = 8;
switch (x) {
   case 11:
      alert("Эй!");
   case 0:
      alert("Осторожно!");
   case "stroka":
      alert("Злая");
   case 235:
      alert("собака!");
   default:
      alert("Да заходи, не бойся!");
}
```

А теперь выполним с помощью этого оператора предыдущее задание по загрузке определённой страницы в определённый браузер:

```
var mybrowser = (navigator.appName)
switch (mybrowser) {
   case "Netscape":
      self.parent.location = "index_ns.html";
      break
   case "Opera":
      self.parent.location = "index_op.html";
}
```

Мы полагаем, что скрипт находится на главной по умолчанию странице *index_ie.html*, поэтому незачем давать её ни как **case**, ни как **default**. Понятно также, что с помощью директивы **break** мы предотвращаем попытку выполнения невыполнимого кода, которая лично у меня, когда я тестировал этот пример, привела к глюкам (и стал я жертвой собственного фанатизма: «Лисичка» почему-то вообще перестала открываться, обиделась, что ли :).

Оператор for

Его синтаксис напоминает оператор **if**: условие в круглых скобках, код — в фигурных. Но условие более сложное:

```
for (начало; область действия; шаг)
{код}
```

Начало: здесь задаётся начальное условие, при котором оператор включается.

Область действия: границы, в которых выполняется код.

Шаг: способ продвижения цикла от начального к конечному условию.

Арифметическая прогрессия

Предположим, нам надо узнать и записать на странице сумму всех чисел подряд от 1 до 10.

```
var i, x = 0;
for (i = 1; i <= 10; i++)
{x += i}
document.write(x)
```

Результат:

55

Разберём:

Объявляем переменную счётчика **i** и переменную, собирающую знчения — **x**. Назначим ей нулевое значение. Все значения должны куда-то складываться, а неназначенная переменная — это **Null** («посмотришь на его место, и нет его»).

Начальное значение счётчика — 1. Граница — до 10 включительно, то есть меньше или равно 10. Шаг — 1 (каждое следующее целое число).

Привыкайте к специальным обозначениям. Если не помните, что такое ++ или +=, зайдите в прошлый урок и посмотрите табличку.

В коде **x += i** каждое новое значение счётчика приплюсовывается к переменной **x**.

Затем выходим из цикла и публикуем на странице итоговое значение.

А теперь чуть поинтереснее.

Вытащим на страницу все промежуточные значения счётчика. Для этого поместим **document.write** не выходя из цикла (то есть внутри фигурных скобок). А чтобы числа не печатались сплошной нераздельной строкой, добавим в него разделитель:

```
var i, x = 0;
for (i = 1; i <= 10; i++)
{x += i
document.write(x + "; ")
}
```

Результат:

1; 3; 6; 10; 15; 21; 28; 36; 45; 55;

А как сделать, чтобы вся строка заканчивалась, скажем, точкой x

Для этого скомбинируем этот оператор с уже известным нам оператором **if...else**. То есть при последнем значении i (10) у нас должна выпасть точка, при остальных — точка с запятой и пробел:

```
var i, x = 0;
for (i = 1; i <= 10; i++)
{x += i
   if (i == 10) {document.write(x + ".")}
   else {document.write(x + "; ")}
}
```

Результат:

1; 3; 6; 10; 15; 21; 28; 36; 45; 55.

Примечание: Кстати, вот оно, реальное различие методов **write** и **writeln**. Вместо пробела здесь можно применить метод **writeln**. Если мы запишем

```
.....
.....
   if (i == 10) {document.write(x + ".")}
   else {document.writeln(x + ";")}
.....
```

то пробел делать не обязательно, его поставит сам метод.

Поскольку конечное значение счётчика — уже существующее условие, а не назначаемая величина, ставим сдвоенное равенство. И не запутайтесь в фигурных скобках.

Таблица умножения

А сейчас, допустим, нам захотелось вывести таблицу умножения — вот такую:

2x2=4	3x2=6	4x2=8	5x2=10	6x2=12	7x2=14	8x2=16	9x2=18
2x3=6	3x3=9	4x3=12	5x3=15	6x3=18	7x3=21	8x3=24	9x3=27
2x4=8	3x4=12	4x4=16	5x4=20	6x4=24	7x4=28	8x4=32	9x4=36
2x5=10	3x5=15	4x5=20	5x5=25	6x5=30	7x5=35	8x5=40	9x5=45
2x6=12	3x6=18	4x6=24	5x6=30	6x6=36	7x6=42	8x6=48	9x6=54
2x7=14	3x7=21	4x7=28	5x7=35	6x7=42	7x7=49	8x7=56	9x7=63
2x8=16	3x8=24	4x8=32	5x8=40	6x8=48	7x8=56	8x8=64	9x8=72
2x9=18	3x9=27	4x9=36	5x9=45	6x9=54	7x9=63	8x9=72	9x9=81
2x10=20	3x10=30	4x10=40	5x10=50	6x10=60	7x10=70	8x10=80	9x10=90

Да чтобы руками эту таблицу не форматировать и опечаток не наделать.

Давайте найдём алгоритм для составления такой таблицы.

Как мы помним из *HTML*, таблицы надстраиваются горизонтальными рядами **<tr>**, которые, в свою очередь, делятся на ячейки **<td>**.

Рассмотрим содержимое первого горизонтального ряда.

Первый множитель увеличивается на единицу. Второй остаётся без изменений.

А в вертикальных столбцах — наоборот. Первый стоит на месте, а второй приращивается.

Рядов **<tr>** в нашей таблице девять (от 2 до 10), а колонок в каждом из них по 8 (от 2 до 9).

Попробуем сначала сформировать все **<tr>** через счётчик **i**.

```
var i;
for (i = 2; i <= 10; i++)
{document.write("<tr>");
document.write("</tr>")}
```

Почему от 2 до 10, а не от 1 до 9?

Догадайтесь с трёх раз...

Ну конечно, мы их потом будем использовать и для значений содержимого таблицы, а оно начинается с 2x2.

Наш первый **<tr>** должен выглядеть вот так (× — это спецсимвол HTML для отображения

«школьного» знака умножения):

```
<tr>
   <td>2&times;2=4</td>
   <td>3&times;2=6</td>
   <td>4&times;2=8</td>
   <td>5&times;2=10</td>
   <td>6&times;2=12</td>
   <td>7&times;2=14</td>
   <td>8&times;2=16</td>
   <td>9&times;2=18</td>
</tr>
```

То есть внутри него ещё один цикл — из **</td>**.

Эти циклы можно вкладывать, как и операции с **if...else**. Вложим?

Для второго цикла определим счётчик **j**.

```
var i, j;
document.write("<table border=\"1\" cellspacing=\"0\" cellpadding=\"2\"
align=\"center\">")
for (i = 2; i <= 10; i++)
{document.write("<tr>");
   for (j = 2; j < 10; j++)
   {document.write("<td>" + "</td>")}
document.write("</tr>")
}
document.write("</table>")
```

Обратили внимание? Если **i** <= **10**, то **j** < **10** (без равенства), так как горизонтальных ячеек на одну меньше.

Пустая таблица свёрстана, осталось заполнить её содержимым. Первый множитель приращивается в каждом **</td>**, значит, он соответствует значению **j**. А второй, возрастающий в каждом **</tr>**, — значению **i**:

j + "×" + i + "=" + (i*j)

Подставим это в наш код:

```
var i, j;
document.write("<table border=\"1\" cellspacing=\"0\" cellpadding=\"2\"
align=\"center\">")
for (i = 2; i <= 10; i++)
{document.write("<tr>");
   for (j = 2; j < 10; j++)
   {document.write("<td>" + j + "&times;" + i + "=" + (i * j) + "</td>")}
document.write("</tr>")
}
document.write("</table>")
```

Таблица умножения готова. Таким же способом можно сделать, например, таблицу символов *Unicode*. Полезная штука. Подобные алгоритмы используются в скриптах меню или календарей.

Оператор while

Даже если Вы ещё никогда не программировали, то наверняка работали в разных программах просто как пользователь. И знаете, что одного и того же результата иногда можно достичь разными способами.

То же и в программировании. С помощью цикла **while** можно делать всё то же, что и с помощью цикла **for**. Что-то удобнее делать в одном цикле, что-то — в другом. Что-то быстрее в одном, что-то безопаснее в другом. Кто-то привык к одному, кто-то — к другому.

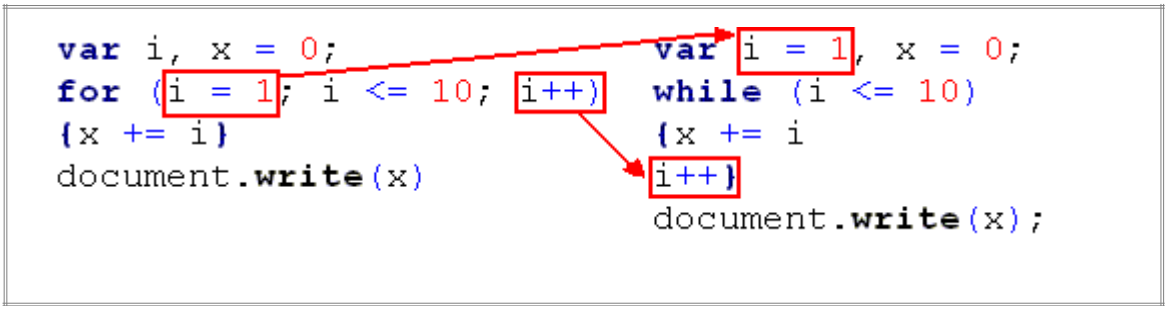
Чтобы Вам было понятнее, мы будем использовать те же самые примеры, но в новой «аранжировке».

While означает «в то время как» или «до тех пор, пока». В то время как имеется условие и до тех пор, пока оно существует, выполняется код.

Синтаксис до боли знакомый:

```
while (условие)
{код}
```

Попробуем «проиграть» в этом цикле наш первый пример: сумму чисел от 1 до 10. Вот как выглядят оба кода:



В цикле **for** значение счётчика **i** задавалось в заголовке цикла. В цикле **while** оно не задаётся, мы задаём его при объявлении переменной.

Шаг здесь переносится в самый конец. То есть мы задаём условие, назначаем на него действие и только потом указываем шаг для его выполнения.

Расширенный вариант, с выводом всех значений и блоком **if** для разделителей, попробуйте конвертировать сами. А потом взгляните на **ответ**.

А теперь немножко помучаемся с таблицей умножения.

Здесь у нас были вложенные циклы. Цикл **while** тоже можно вкладывать, но мы наткнёмся на один нюанс, связанный с последовательностью действий и сохранением данных в переменной, и придётся немножко обмануть программу.

А сначала попробуем комбинированный вариант: внешний цикл сконвертируем в **while**, а внутренний оставим **for**.

Вот наш внешний цикл, выводящий **<tr>**.

```
var i = 2, j; /* Сразу объявим переменную и для внутреннего цикла.
Назначать не будем, так как внутренний цикл у нас FOR.*/
document.write("<table border='1' cellspacing='0' cellpadding='2' align='center'>")
while (i <= 10)
{ document.write("<tr>")
   // Здесь будет
   // вложенный цикл.
document.write("</tr>")
```

```
i++
}
document.write("</table>")
```

Теперь просто скопируем в отведённое место вложенный цикл **for**:

```
var i = 2, j;
document.write("<table border='1' cellspacing='0' cellpadding='2' align='center'>")
while (i <= 10)
{ document.write("<tr>")
   for (j = 2; j < 10; j++)
   {document.write("<td>" + j + "&times;" + i + "=" + (i * j) + "</td>")}
document.write("</tr>")
i++
}
document.write("</table>")
```

Теперь разберёмся с проблемой вложенных циклов **while**. Сейчас я приведу **неправильный** скрипт, и попробуем понять, почему он неправильный и как его исправить. Это будет неплохая разминка для мозгов.


```
var i = 2, j = 2;
document.write("<table border='1' cellspacing='0' cellpadding='2'
align='center'>")
while (i <= 10)
{ document.write("</tr>")
   while (j < 10)
   {document.write("<td>" + j + "&times;" + i + "=" + (i * j) + "</td>")
   j++
   }
document.write("</tr>")
i++
}
document.write("</table>")
```

Можете скопировать и запустить. Вы увидите заполненную первую строку и какую-то белую полоску под ней. Эта полоска — результат сгенерированных пустых **<tr>**. То есть внешний цикл честно отрабатывает до конца, а вложенный виснет на первом круге.

Вопрос: чему равна переменная **j** после первого прохождения большого цикла?

Ответ: она равна 10.

Вопрос: чему она должна быть равна к началу прохождения второго витка?

Ответ: 2, как задано.

Вопрос: как её сбросить?..

Давайте подумаем.

Есть оператор **break**, который мы использовали в **switch**. Здесь он тоже используется (потом мы о нём специально поговорим). Но сразу скажу, нашей беде он не поможет. С его помощью можно только поменять циклы (то есть будет выводиться только один вертикальный столбец).

Внимание. Думаем. Вникаем. Нам нужна переменная, которая бы содержала неизменную двойку. И нам нужна переменная, которая бы многократно проходила цикл от этой двойки. А что нам мешает создать ещё одну переменную, допустим, **k**, приравнять её к **j** и запустить в цикл? Да ничего, разве что тормоза в голове.

Смотрите, она **(k)** пройдёт цикл, прирастая до 10, а в следующем витке вновь приравняется к неизменной **j** и снова пройдёт тот же цикл:

```
var i = 2, j = 2, k;
document.write("<table border='1' cellspacing='0' cellpadding='2' align='center'>")
while (i <= 10)
{ document.write("<tr>")
   k = j
   while (k < 10)
   {document.write("<td>" + k + "&times;" + i + "=" + (i * k) + "</td>")
   k++
   }
document.write("</tr>")
i++
}
document.write("</table>")
```

Вывод: лучше, всё-таки, использовать **for**. Но чтобы это понять, надо попробовать и **while**.

Оператор do...while

Этот оператор очень похож на предыдущий. Различие, образно говоря, такое: если **while** сначала подумает, а потом сделает, то **do...while** сначала сделает, а потом подумает.

while (условие) **проверяет** {код} **делает**	do {код} **делает** while (условие) **проверяет**

Если оператор **while** не найдёт нужного условия при проверке, то его код вообще не будет выпонняться. А код оператора **do...while** хотя бы один раз выполняется всегда. А выполнение (или невыполнение) дальнейшего цикла уже зависит от проверки условия.

Ниже приводится сравнение кодов выведения нашей арифметической прогрессии в этих двух операторах.

```
var i = 0, x = 10;
while (i <= 10)
{x += i
if (i==10) {document.write(x+".")}
else {document.write(x+"; ")}
i++
}
```

```
var i = 0, x = 10;
do {x += i
if (i==10) {document.write(x+".")}
else {document.write(x+"; ")}
i++
}
while (i <= 10)
```

А вот «аранжировка для do...while» нашей таблицы умножения (здесь тоже приходится использовать «лишнюю» переменную):

```
var i = 2, j = 2, k;
document.write("<table border='1' cellspacing='0' cellpadding='2' align='center'>")
do
{document.write("<tr>")
   k = j
   do
   {document.write("<td>" + k + "&times;" + i + "=" + (i * k) + "</td>")
   k++
   }
   while (k < 10)
document.write("</tr>")
```

```
i++
}
while (i <= 10)
document.write("</table>")
```

В общем-то, мы закончили наше первое знакомство с операторами. Но есть ещё некоторые ключевые слова, вроде уже встречавшихся нам **break** или **default**. Это, по сути дела, тоже маленькие вспомогательные операторы, и им мы посвятим следующий урок, прежде чем перейдём к массивам.

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()
```

Правила для имён массивов такие же, как и для имён переменных (см. *урок 1*).

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.

Yoklmn, почему четыре?

О программистах ходит много анекдотов, например, такой:

— По порядку рррас-считайсь!
— Нулевой!..

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**.

Не вру, можете проверить.

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>
```

Кстати, иногда бывает очень удобна такая вот «динамически создаваемая шпаргалка». Когда затрудняетесь определить какое-нибудь значение в Вашем скрипте, выведите его через **document.write**. JavaScript не ошибётся.

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>
```

Но мы придумаем одну хитрость: 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 (не поленитесь тщательно прописать их), а потом через **if** будет решать, какой из пунктов меню нужно выводить без ссылки.

Сначала загоним 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
```

Напоминаю: 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 приступиь 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.

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.).

О методах сортировки элементов массива я расскажу, когда мы будем говорить о методах JavaScript.

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.

Иногда это бывает удобно, но не всегда.

В Ассоциации «Русская Традиция» при Питерском Союзе композиторов, в которой я состою, двое из музыкантов являются также и одарёнными художниками. Я сделал на сайте Ассоциации маленькую галерею (можете *посмотреть*) и использовал эту встроенную коллекцию. А потом решил резко поменять дизайн всего сайта. Но поскольку в элементах дизайна тоже есть графика, то нумерация сбилась, и я двое суток сидел и правил цифры. А на третьи сутки до меня дошло, что лучше это сделать, например, через **getElementById()** — помните? — 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.

Усложняем слайд-шоу № 9

В «Диких уроках HTML» *урок № 9* посвящён созданию слайд-шоу на JavaScript. Мы немного усложним этот скрипт.

Там фотоальбом посвящён кошкам. Я тоже люблю и кошек, и собак, но ещё больше — друзей. Есть у меня друг Гена Змитрович, талантливый скульптор и художник. И в нашем уроке я хочу представить несколько его работ. А поскольку он выступает в двух ипостасях — как художник и как скульптор — у нас будет два слайд-шоу, с картинами и скульптурами, но управляться они будут одним скриптом.

Этот скрипт — своего рода контрольное задание. Кроме массивов, в нём используется ряд операторов, которые мы изучили, а также он продемонстрирует, как можно использовать логический (булев) тип данных.

Если Вы сможете сами соорудить подобный скрипт, значит, уже перешли из категории «чайников» в категорию «кофейников»:)

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

(В первоначальном варианте я для простоты дал один счётчик, но с ним функция работала не совсем корректно. Так что пусть будет немного сложнее, но без халтуры.)

```
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**. Сейчас научу, как им пользоваться.

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. Картины у Змитровича философские. А среди скульптур есть и памятник, и кубок, и просто миниатюры. 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.

Вот сейчас и пора вспомнить гениального Джорджа Буля, который утверждал, что самые сложные проблемы можно решить через последовательность цепочек «да» и «нет».

Нам нужны всего две таких цепочки:

вперёд-назад — да-нет;
картины-скульптуры — да-нет.

Эти два булевых выражения и станут аргументами нашей функции. Поскольку это «родная дочь» функции **dem(n)** из Дикого урока № 9, 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>
```

Объектная модель JavaScript

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

Если Вы любознательны и пытливы, Вам, должно быть, где-нибудь уже попадалась такая аббревиатура: ООП, что значит «объектно-ориентированное программирование». 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.

Должен предупредить, что с точки зрения теории программирования эти мои объяснения не вполне корректны. Но если я буду объяснять «как надо», боюсь, что меня поймут только

программисты (которым объяснять вообще ничего не надо). В дальнейшем, при необходимости, мы исследуем и «подводную часть айсберга», когда Вы будете к этому готовы. Но для особо любознательных могу дать некоторые начальные пояснения (*посмотреть*).

У каждого объекта свои методы и свойства. Бывают свойства и методы, закреплённые за несколькими объектами, и с разными объектами они могут работать по-разному. Но с какими-то объектами они вооще не работают. Как и в обычном языке: бывает жёлтый цвет, и даже жёлтая пресса. Но на «жёлтую температуру»... программа выполнит недопустимую операцию.

Бывает, что объекты превращаются в свойства и методы других объектов. Это тоже встречается в человеческом языке:

Бывает, что объекты превращаются в свойства и методы других объектов. Это тоже встречается в человеческом языке:

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** — см. *здесь.*;
- **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**. Для этого нужно поднатореть в программировании. Надеюсь, что когда-нибудь дойдём и до них.

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}
```

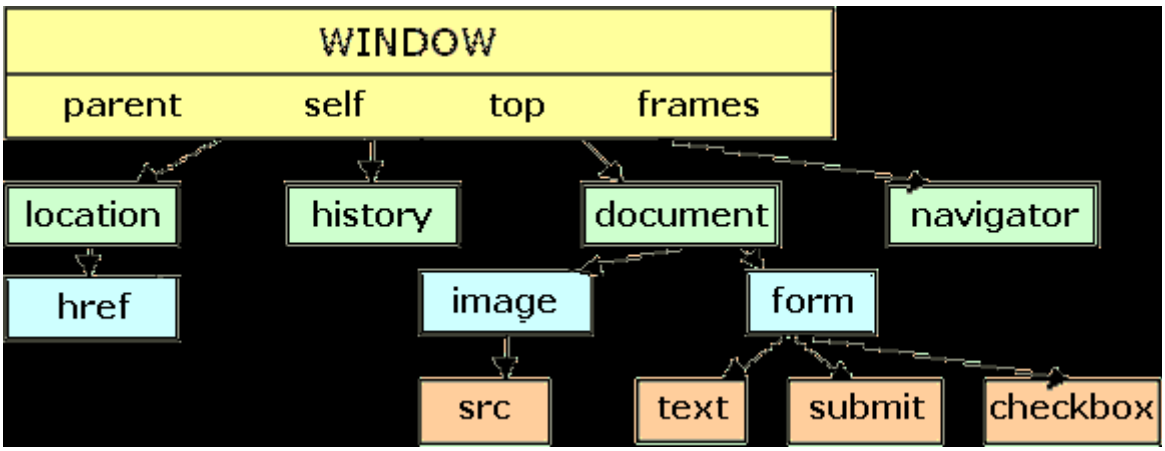
Кроме того, есть свойства **только для чтения**, с уже заданными патаметрами, которые нельзя переназначить. Например, приведённое выше свойство **appName** объекта **navigator**. Такие свойства вызываются **только** через сдвоенный оператор равенства (==).

Понятие иерархии объектов

Это понятие связано со структурой языка (подробнее см. скрытое примечание в начале урока). 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**