ОСНОВЫ JAVASCRIPT

Принципы выполнения программ. Основные структуры языка. Преобразование типов данных. Функции, отложенное выполнение функций. Принципы работы с массивами, объектами, this. Основы работы с DOM. Обработка событий. Основы XMLHttpRequest.

Author: **Svyatoslav Kulikov**
Training And Education Manager
svyatoslav_kulikov@epam.com

Содержание

1. Общие сведения о JavaScript
2. Что нужно для работы JavaScript
3. Общий синтаксис JavaScript
4. Переменные и типы данных JavaScript
5. Определение и преобразование типов данных
6. Основные функции JavaScript, с которых надо начать
7. Операторы и управляющие конструкции JavaScript
8. Математические функции JavaScript
9. Функции JavaScript, определяемые пользователем
10. Работа с массивами в JavaScript

Содержание

Disclaimer: этот материал является кратким вводным курсом в JavaScript. Пожалуйста, не считайте его полным исчерпывающим руководством. Очень многие темы мы будем рассматривать упрощённо, на уровне, минимально достаточном для решения тривиальных задач.

Фактически, этот материал является ответом на вопрос «что надо знать о JavaScript, чтобы писать на PHP (или ином сервероориентированном высокоуровневом языке)» ☺.

ОБЩИЕ СВЕДЕНИЯ О JAVASCRIPT

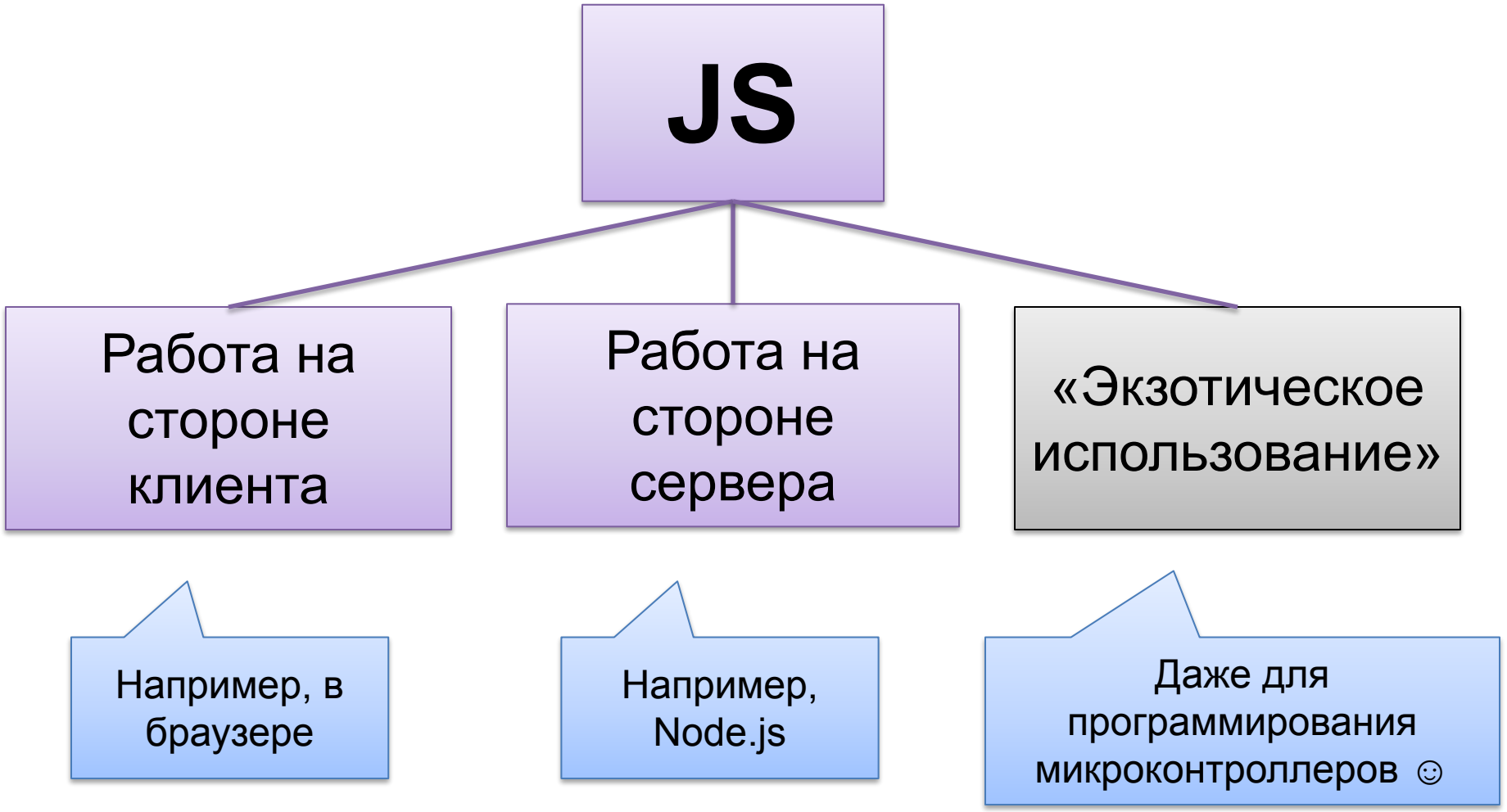
Общие сведения о JavaScript

JavaScript (JS) – язык программирования, предназначенный в первую очередь для расширения возможностей клиентской части веб-приложений.

Основные факты:

- Программы на JS хранятся в виде исходного текста.
- Большая часть синтаксиса JS пришла по наследию из языка C.
- JS – прототипно-ориентированный (об этом чуть позже), нестроготипизированный, интерпретируемый.
- В основном используется в браузерах (или иных клиентах), но существует и серверная реализация.

Что можно сделать с помощью JavaScript



ЧТО НУЖНО ДЛЯ РАБОТЫ JAVASCRIPT

Что нужно для работы JavaScript

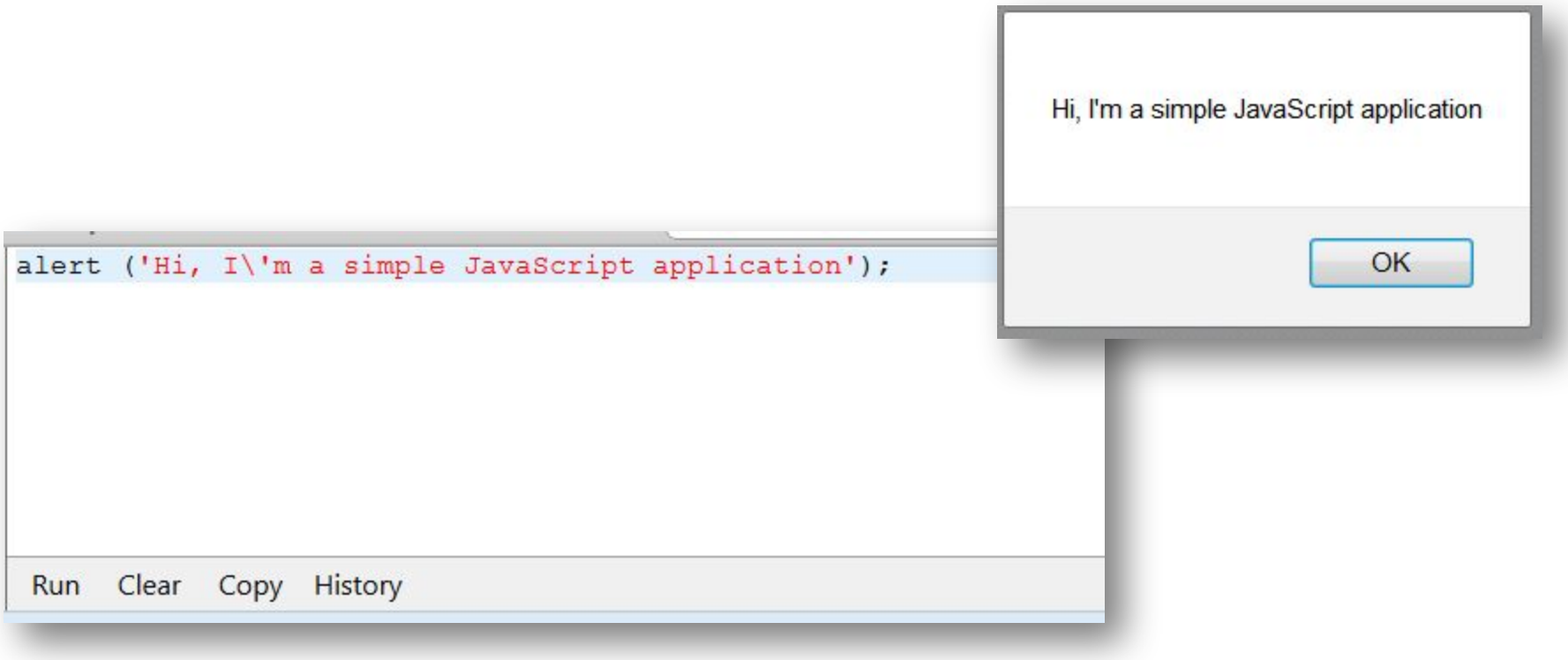
Поскольку мы будем в основном рассматривать JavaScript для работы с браузером, то нам понадобится:

- Браузер ☺.
- Редактор с подсветкой синтаксиса или полноценная IDE.

См. примеры в папке **01_js_samples_misc**

Что нужно для работы JavaScript

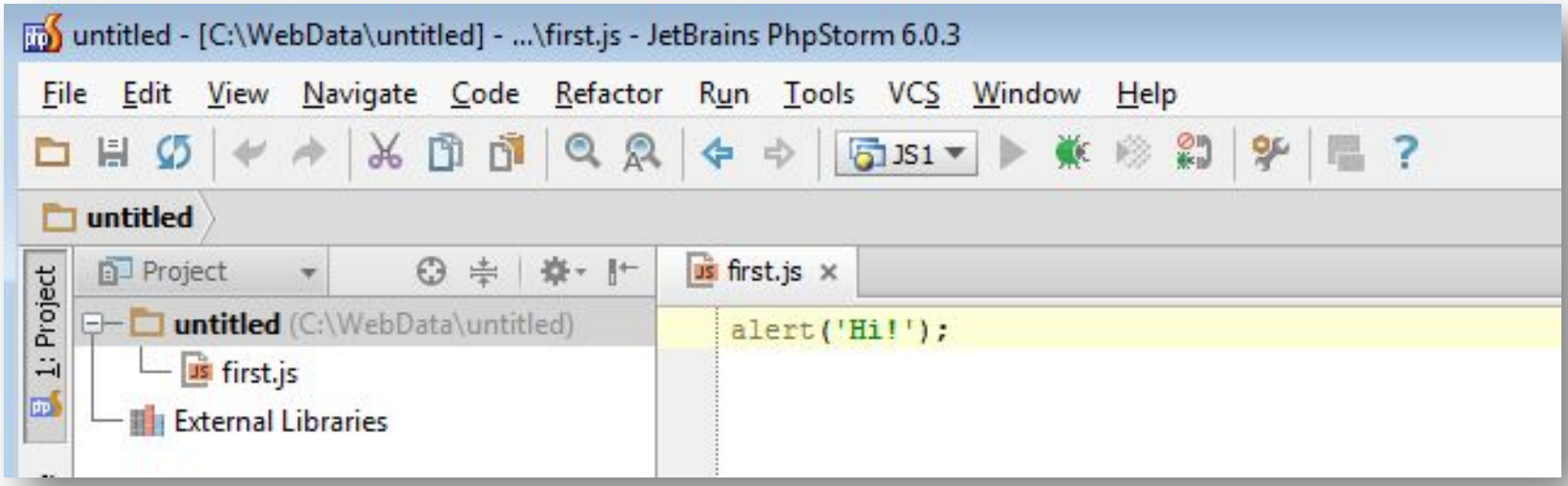
Говоря «браузер», мы подразумеваем Firefox, т.к. в нём доступно множество удобных инструментов, основной из которых для нас – Firebug (http://getfirebug.com)



Что нужно для работы JavaScript

Также для работы с JavaScript можно использовать:

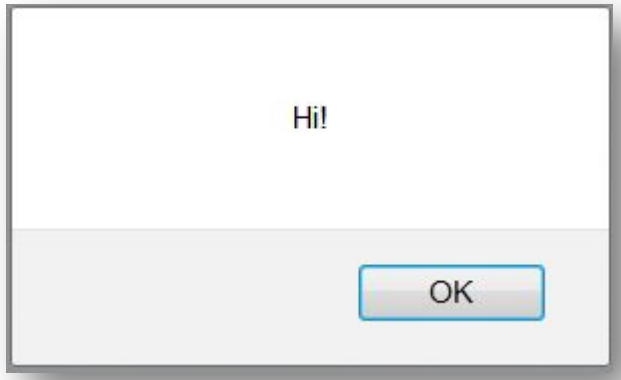
- PhpStorm
- WebStorm
- Eclipse
- NetBeans
- И т.д.



Запуск в браузере

Код JavaScript можно писать «внутри HTML» или подключать как внешние файлы:

```
<!DOCTYPE html>
 <html>
  <head>
   <meta charset="UTF-8">
   <title>JavaScript: первый пример</title>
   <script>
    alert('Hi!');
   </script>
   <script src="01_first_sample.js"></script>
  </head>
 <body>

  <span id="sample"></span>

</body>
</html>
```




```
window.onload=function(){document.getElementById('sample').innerHTML='Hi again!';};
```

См. пример в файле **01_js_samples_misc\01_first_sample.html**

Запуск в браузере

Поскольку наш курс посвящён основам JavaScript, мы не будем погружаться в дебри, и в процессе рассмотрения примеров будем использовать редактор с подсветкой синтаксиса для написания JavaScript-кода и браузер для его выполнения.

Приступим к рассмотрению самого JavaScript…

ОБЩИЙ СИНТАКСИС JAVASCRIPT

Команды JavaScript

Как уже было сказано, код на JavaScript можно заключать в тег <script></script> или подключать из внешних файлов <script src="script.js"></script>.

В отличие от PHP, <script> нельзя «разрывать», т.е. вот так – НЕЛЬЗЯ:

```
<script>
if (1==1)
 {
</script>

<script>
  alert('OK');
 }
</script>
```

Комментарии в JavaScript

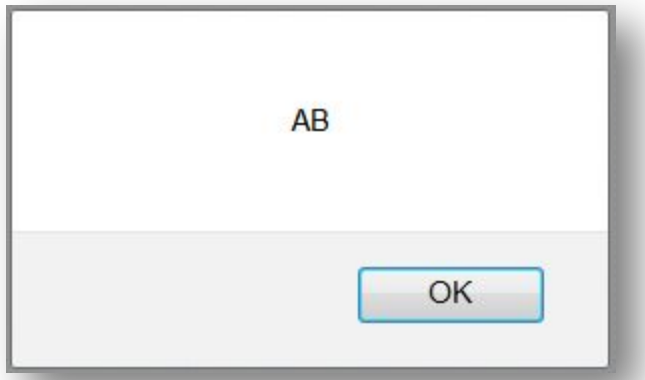
JavaScript предоставляет два способа вставки комментариев.

```
x=9; // комментарий

/*
   и это тоже комментарий
*/
```

Регистрочувствительность

JavaScript чувствителен к регистру – везде: в именах функций, переменных, объектов и т.д.

```
<script>
 var1 = 'A';
 vAr1 = 'B';
 alert(var1+vAr1);
</script>
```



Это – РАЗНЫЕ переменные! Пусть отличие и заключается всего лишь в регистре одной буквы. Частая ошибка: написать getelementbyid() вместо getElementById() и удивляться, что ничего не работает.

Точки с запятыми в JavaScript

В JavaScript НАДО писать точки с запятыми в конце строк. НАДО! Несмотря на то, что они являются опциональными, вы рискуете получить такой интересный эффект:

```
function f1()
 {
  return
  {
   x: 99
  }
 }

f1()
```

```
undefined
```

```
function f2()
 {
  return {
   x: 99
  }
 }


f2()
```

```
Object { x=99}
```

```
function f3()
 {
  return; {
   x: 99;
  }
 }


f3();
```

```
undefined
```

Точки с запятыми в JavaScript

Вторая частая ошибка – отсутствие «;» после «присвоения функции переменной»:

```
var func = function() { return false; };
```

Да, здесь нужна «;». Да, иногда работает и без неё. Нет, не всегда.

ПЕРЕМЕННЫЕ И ТИПЫ ДАННЫХ JAVASCRIPT

ВАЖНО!

- JavaScript – нестроготипизированный язык: переменные в нём могут менять свой тип в процессе выполнения программы.
- JavaScript не накладывает строгих ограничений на участие в выражениях переменных разных типов.
- Переменные не нужно объявлять отдельно, они «объявляются» через инициализацию при первом использовании.

Переменные в JavaScript

Имя переменной может состоять из букв, цифр, символов «$» и «_», при этом первый символ не должен быть цифрой.

```
var someVariable;
var newVariable15;
var $ = 15;
var _ = 17;
```

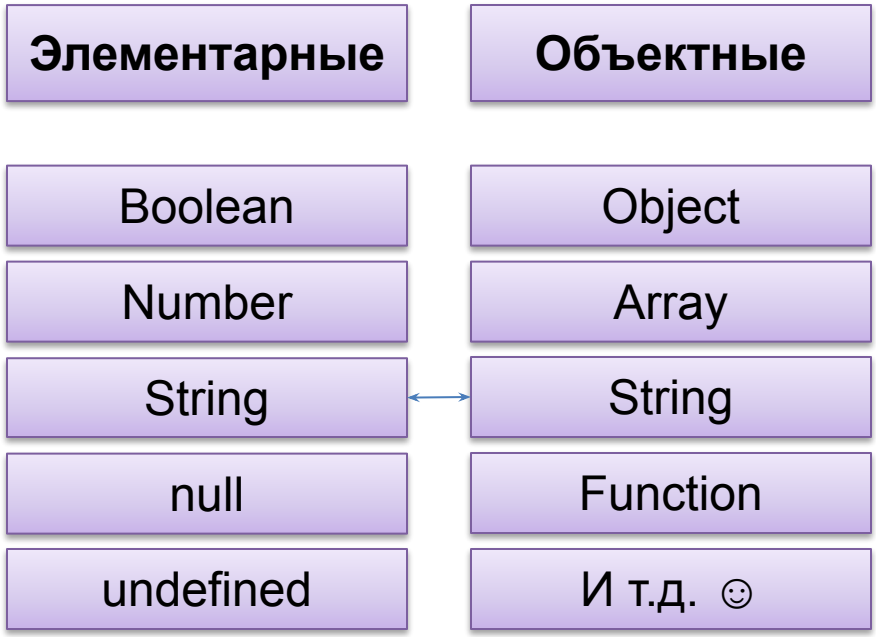
Переменные в JavaScript

Вопросы области видимости переменных мы рассмотрим чуть позже, но пока важно запомнить одно правило: всегда объявляйте ваши локальные переменные в функциях с использованием ключевого слова **var**:

```
var someVariable;
var newVariable15;
var $ = 15;
var _ = 17;
```

Типы данных в JavaScript

JavaScript поддерживает следующие типы данных: объектные и элементарные (которые тоже можно трактовать как объектные).



Для каждого элементарного типа есть соответствующий объект, но использовать их не рекомендуется. Элементарные типы автоматически интерпретируются как объекты при вызовах методов.

Хорошая дополнительная информация: http://habrahabr.ru/post/150730/

```
alert(typeof "test"); // string
alert(typeof new String("test")); // object
```

Элементарные типы данных: Boolean

Boolean используется для хранения только двух логических значений: «истина» или «ложь».

```
a = true; // Не TRUE
b = false; // Не FALSE
```

Элементарные типы данных: Number

Number используется для хранения чисел и, фактически, представляет собой float64 (8-мибайтную дробь). Эта особенность приводит к очень неприятным последствиям при вычислениях.

```
a = 1;
b = 2;
c = a + b; // 3

a = 0.1;
b = 0.2;
c = a + b; // 0.30000000000000004
```

Элементарные типы данных: Number

В отличие от множества других языков в JavaScript операции с числами никогда не приводят к явному сообщению об ошибке, вместо этого используются специальные значения:

```
x =  1/0;                  // Infinity
x = -1/0;                  // -Infinity
x = Number('lalala');      // NaN
```

Элементарные типы данных: Number

Q: И как тогда получить вменяемый результат при работе с дробями?
A: Через методы toFixed(), Math.floor(), Math.round(), Math.ceil().

```
a = 0.1;
b = 0.2;
c = a + b; // 0.30000000000000004
d = c.toFixed(2); // "0.30"
e = Math.round(c*100)/100; // 0.3
```

Math.round(число) округляет число до целых.

Элементарные типы данных: String

String используется для хранения строк. Строки в JavaScript должны быть заключены в одинарные или двойные кавычки. В отличие от PHP, между этими вариантами НЕТ разницы.

```
a = "Text";
b = 'Text';
c = 'Hotel \'Minsk\'';
d = "Hotel 'Minsk'";
e = "Hotel \"Minsk\"";
f = "Long \
     line";
```

Если в строке содержится такая же кавычка, какими строка обрамлена, её надо экранировать.

Если строку надо «разорвать» для «продолжения со следующей строки», место разрыва «экранируется».

Элементарные типы данных: String

Напомним, что в JavaScript элементарные данные автоматически интерпретируются как объекты при вызове методов, т.е.:

```
a = "Text";
b = a.length; // 4
c = "Lalala".length; // 6
d = '\u1553'; // "ᕓ"
e = d.length; // 1
```

Элементарные типы данных: null

null говорит о том, что переменная не содержит допустимых Number, String, Boolean, Array или Object. Т.е. переменная ЕСТЬ, но значения у неё НЕТ.

ВАЖНО! null в JavaScript – НЕ то же самое, что null в PHP (в котором «переменной нет» === «переменная равна null»).

ВАЖНО! null в JavaScript – это объект, но НЕ экземпляр Object ☹.

```
var x = null;
```

Хорошее пояснение – здесь:
http://habrahabr.ru/post/171359/

Элементарные типы данных: undefined

Переменная равна undefined в случае, если она не существует, или была объявлена, но не проинициализирована.

```
var x;

if (x == undefined)
{
  // Какой-то код ☺.
}
```

Элементарные типы данных: разница между null и undefined

Итак, ещё раз:

- undefined – переменной нет, или ей не было присвоено значение.
- null – переменная есть, но её значение – «пустота».

```
var a;
var b = document.getElementById('non_existing_element');
var c = 999;
c = null;

console.log(typeof(a));
console.log(typeof(b));
console.log(typeof(c));

if (a == null)
{
 console.log('a is null');
}

if (a == undefined)
{
 console.log('a is undefined');
}

if (b == null)
{
 console.log('b is null');
}

if (b == undefined)
{
 console.log('b is undefined');
}

if (c == null)
{
 console.log('c is null');
}

if (c == undefined)
{
 console.log('c is undefined');
}
```

```
undefined
object
object
a is null
a is undefined
b is null
b is undefined
c is null
c is undefined
```

Объектные типы данных: Object

Object в JavaScript – это коллекция свойств и методов.
В общем случае можно выделить такие типы объектов:

- внутренние объекты (например, Array, String и т.д.);
- создаваемые объекты;
- объекты среды (например, window, document и т.д.);
- объекты ActiveX (не актуально для не MSIE).

ВАЖНО! Объекты в JavaScript – НЕ то же самое, что классические объекты в других языках программирования! Они НЕ создаются на основе описания классов!

Объектные типы данных: Object

Например, такое прекрасно работает в JavaScript, но не в других языках:

```
var someObject = new Object();
someObject.name = "SomeFruit";
someObject.price = 999;
someObject.getPrice =
    function ()
    {
        return this.price;
    };

console.log(someObject.name);
console.log(someObject.price);
console.log(someObject.getPrice());
```

```
SomeFruit
999
999
```

Объектные типы данных: Array

Array – специальный объект для хранения данных (в т. ч. разных типов). Многомерные массивы не поддерживаются в явном виде, но легко эмулируются.

ВАЖНО! JavaScript считает размером (длиной) массива «последний числовой ключ + 1» вне зависимости от реального положения дел.

Рассмотрим на примере…

Объектные типы данных: Array

Пример работы с массивами в JavaScript:

```
var names = new Array();

names[0] = "John";
names[1] = "Anna";
names[999] = "Jack";
names['test'] = "Test";

console.log(names[1]);
console.log(names.length);
console.log(names[555]);
```

```
Anna
1000
undefined
```

Объектные типы данных: Array

Если нужен многомерный массив…

```
var names = new Array();

names[0] = new Array("Иванов", "Иван", "Иванович");
names[1] = new Array("Петров", "Пётр", "Петрович");

console.log(names[0]);
console.log(names[1][1]);
console.log(names[1][999]);
```

```
["Иванов", "Иван", "Иванович"]
Пётр
undefined
```

Объектные типы данных: Array

И, наконец: если вы хотите получить «нормальный ассоциативный массив», вам для этого надо использовать… объект! ☺

```
var something = new Object();
something.name = "Test";
something.price = 100;
something.weight = 11.55;

for (var key in something)
{
    console.log(something[key]);
}
```

```
Test
100
11.55
```

Объектные типы данных: Array

Q: Можно ли в JavaScript обращаться к элементам массива напрямую в процессе вызова функции, возвращающей массив?
A: Да.

```
var fnc = function()
{
    return new Array(10, 20, 30);
}

console.log(fnc()[1]);
```

```
20
```

Объектные типы данных: Function

В JavaScript существует множество встроенных функций и возможность создавать собственные. Раз мы говорим о типах данных, рассмотрим пока вот такой пример:

```
var fnc = function()
{
    return "Test";
}

console.log(typeof(fnc));
fnc();
```

```
function
"Test"
```

Константы в JavaScript

Ранее JavaScript не поддерживал константы, и такой код может не сработать в старых версиях браузеров.

```
const a = 99;
console.log(a);
```

```
99
```

ОПРЕДЕЛЕНИЕ И ПРЕОБРАЗОВАНИЕ ТИПОВ ДАННЫХ

Определение типа переменной

Для определения типа переменной используется оператор

```
string typeof var
```

который возвращает тип переменной в виде строки:

- boolean;
- nember;
- string;
- object;
- function;
- null;
- undefined.

Определение типа переменной

Пример использования:

```
console.log(typeof 37); // number
console.log(typeof 3.14); // number
console.log(typeof Math.LN2); // number
console.log(typeof Infinity); // number
console.log(typeof NaN); // number (хоть Nan -- это и Not-A-Number)
console.log(typeof Number(1)); // number

console.log(typeof ""); // string
console.log(typeof "bla"); // string
console.log(typeof (typeof 1)); // string
console.log(typeof String("abc")); // string

console.log(typeof true); // boolean
console.log(typeof false); // boolean
console.log(typeof Boolean(true)); // boolean

console.log(typeof undefined); // undefined
console.log(typeof lalala); // undefined

console.log(typeof {a:1}); // object
console.log(typeof [1, 2, 4]); // object (используйте Array.isArray или Object.prototype.toString.call
                               // для различения объектов и массивов)
console.log(typeof new Date()); // object

console.log(typeof new Boolean(true)); // object
console.log(typeof new Number(1)); // object
console.log(typeof new String("abc")); // object

console.log(typeof function(){}); // function
console.log(typeof Math.sin); // function

console.log(typeof null); // object
```

Определение типа переменной

Пример распознавания типа объекта:

```
var arr = new Array(10, 20, 30);
if (typeof arr === "object")
{
    console.log("Да, это массив, но массив -- это ведь объект.");
}
if (Array.isArray(arr))
{
    console.log("Теперь мы точно знаем, что это -- массив.");
}

if (Object.prototype.toString.call(arr) ===  "[object Array]")
{
    console.log("Так тоже работает!");
}
```

```
Да, это массив, но массив -- это ведь объект.
Теперь мы точно знаем, что это -- массив.
Так тоже работает!
```

Определение принадлежности классу

Для определения того, является ли переменная экземпляром некоторого класса, используется оператор

boolean **instanceof** var

который возвращает true или false.

```
var someString = new String();
var someDate = new Date();

console.log(someString instanceof String); // true
console.log(someString instanceof Object); // true
console.log(someString instanceof Date);   // false

console.log(someDate instanceof Date);     // true
console.log(someDate instanceof Object);   // true
console.log(someDate instanceof String);   // false
```

Преобразование типа переменной

Для преобразования типа переменной используется синтаксис

```
переменная = Тип(переменная)
```

например

```
var a=1;
console.log(typeof a);
a = String(a);
console.log(typeof a);
```

```
number
string
```

Внимание! В большинстве случаев преобразование типа МЕНЯЕТ значение переменной!

Обязательно прочитайте: http://jibbering.com/faq/notes/type-conversion/

Преобразование к Boolean

false	**true**
false	true
0	Непустая строка
0.0	+/- Infinity
NaN	Всё остальное, что не перечислено слева.
Пустая строка	
null	
undefined	

Преобразование к Number

Из boolean	Из Number	Из string
false □ 0 true □ 1	См. далее ☺.	См. далее ☺.
Из null	**Из undefined**	**Из Object**
null □ 0	undefined □ NaN	Object □ NaN

Преобразование к Number

Самым простым способом преобразования переменной к числу является её использование в математической операции:

```
var str = "99";
var int = 1;
console.log(str+int);     // 991, конкатенация
console.log((+str)+int); // 100, сложение
```

Использование унарного + является одним из самых простых и быстрых способов преобразования переменной к числу, равно как использование двойного отрицания, т.е. !!, является одним из самых простых и быстрых способов преобразования к boolean.

Преобразование к Number

Также для преобразования к числу можно использовать parseInt() и parseFloat()

```
var str = "99.9";
parseInt(str);                  // 99
parseFloat(str);                // 99.9

var str = " 99.9 lalala";
parseInt(str);                  // 99
parseFloat(str);                // 99.9

var b = true;
parseInt(b);                    // NaN
```

Как и PHP, JavaScript пытается извлечь из строки число, стоящее в её начале (пробелы перед числом игнорируются). Но! Если PHP в любом случае получает число (0, если извлекать нечего), то JavaScript получает NaN во всех случаях, когда из строки не удалось извлечь число.

Преобразование к String

Из Boolean

true □ “true”

false □ “false”

Из null

null □ “null”

Из undefined

□ “undefined”

Из Array, Object, Function

Array □ строка с элементами

Object □ “[object Object]”

Function □ текст функции

Неявное преобразование к string происходит при выводе переменной в выходной поток.

Преобразование к Array

Из Boolean, String, null, undefined	**Из Object и Function**
Массив с одним элементом – исходным значением.	Массив с одним элементом – исходным значением.
Из целого числа	**Из дроби**
Массив соответствующего размера с undefined элементами.	Ошибка создания массива.

Преобразование к Array

```
var i = 2;
var d = 55.5;
var dt = new Date();
var obj = new Object;
obj.x = 100;
obj.y = 200;
var n = null;
var u;

Array(i);     // [undefined, undefined]
Array(d);     // RangeError: invalid array length
Array(dt);    // [Date {Thu Feb 27 2014 11:00:57 GMT+0300
                                    (Kaliningrad Standard Time)}]
Array(obj);   // [Object { x=100, y=200}]
Array(n);     // [null]
Array(u);     // [undefined]
```

Преобразование к Object

Из Boolean, Number, null, undefined

Пустой объект.

Из String

Фактически – массив. Объект с нумерованными свойствами (символами строки).

Из Function

Переменная остаётся функцией.

Эти нумерованные свойства-символы – ReadOnly! См. http://msdn.microsoft.com/en-us/library/ecczf11c%28v=vs.94%29.aspx
и

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Важно!

Преобразование типов в JavaScript достаточно нетривиально, а потому:

- Обязательно почитайте дополнительный материал, книги.
- Старайтесь не использовать преобразование там, где в этом нет необходимости.
- Обязательно тестируйте свой код. Могут быть очень неожиданные сюрпризы.

См. пример в файле **01_js_samples_misc\02_var_compare.html**

Преобразование и переключение типов

В JavaScript существует два механизма изменения типов данных:

- преобразование – переменная меняет тип (и, возможно, значение);
- переключение – копия переменной подвергается преобразованию, а сама переменная остаётся незатронутой.

Преобразование

```
a = 10;
a = String(a);
```

Переключение

```
a = 10;
b = true;
c = a + b;
```

Гибкое (мягкое) сравнение (==)

	true	false	1	0	-1	1	0	-1	null	Array()	str	""	undefined
true	true	false	true	false	false	true	false	false	false	false	false	false	false
false	false	true	false	true	false	false	true	false	false	true	false	true	false
1	true	false	true	false	false	true	false	false	false	false	false	false	false
0	false	true	false	true	false	false	true	false	false	true	false	true	false
-1	false	false	false	false	true	false	false	true	false	false	false	false	false
1	true	false	true	false	false	true	false	false	false	false	false	false	false
0	false	true	false	true	false	false	true	false	false	false	false	false	false
-1	false	false	false	false	true	false	false	true	false	false	false	false	false
null	false	false	false	false	false	false	false	false	true	false	false	false	true
Array()	false	true	false	true	false	false	false	false	false	true	false	true	false
str	false	false	false	false	false	false	false	false	false	false	true	false	false
""	false	true	false	true	false	false	false	false	false	true	false	true	false
undefined	false	false	false	false	false	false	false	false	true	false	false	false	true

Жёсткое (строгое) сравнение (===)

	true	false	1	0	-1	1	0	-1	null	Array()	str	""	undefined
true	true	false	false	false	false	false	false	false	false	false	false	false	false
false	false	true	false	false	false	false	false	false	false	false	false	false	false
1	false	false	true	false	false	false	false	false	false	false	false	false	false
0	false	false	false	true	false	false	false	false	false	false	false	false	false
-1	false	false	false	false	true	false	false	false	false	false	false	false	false
1	false	false	false	false	false	true	false	false	false	false	false	false	false
0	false	false	false	false	false	false	true	false	false	false	false	false	false
-1	false	false	false	false	false	false	false	true	false	false	false	false	false
null	false	false	false	false	false	false	false	false	true	false	false	false	false
Array()	false	false	false	false	false	false	false	false	false	true	false	false	false
str	false	false	false	false	false	false	false	false	false	false	true	false	false
""	false	false	false	false	false	false	false	false	false	false	false	true	false
undefined	false	false	false	false	false	false	false	false	false	false	false	false	true

Небольшая задача для закрепления материала

В обоих вариантах вам предлагается выяснить, чему равно значение переменной c после выполнения кода.

Вариант 1:

```
a="10 cats";
b="5a dogs";
c=a/b;
```

NaN

Вариант 2:

```
a="10 cats";
b='5a dogs';
c=a/b;
```

NaN

Вариант 3:

```
a=true;
b="5a dogs";
c=a/b;
```

NaN

ОСНОВНЫЕ ФУНКЦИИ JAVASCRIPT, С КОТОРЫХ НАДО НАЧАТЬ

Функции, которые нужны всегда

В JavaScript существует несколько функций (и конструкций), которые нужны буквально с первого дня программирования. Рассмотрим их в сравнении с PHP:

```
<?php
echo $var;
print_r($var);
isset($arr[$elem]);
unset($var, $arr[$elem]);
include($filename);
exit('Some message…');
?>
```

```
<script>
// ??? ☺
</script>
```

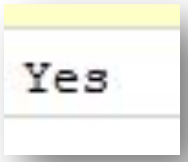
См. пример в файле **01_js_samples_misc\03_basic_functions.html**

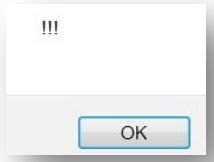
Вывод данных в выходной поток

Для вывода данных в выходной поток (в PHP за это отвечает конструкция echo) используется несколько вариантов:

- В консоли – просто указание переменной, результат последнего присваивания или console.log(x).
- В документе: document.write(x) или document.getElementById('id').innerHTML=x;
- Быстрый вывод в всплывающем окне: alert(x);

```
document.write('ABC');
document.getElementById('sample').innerHTML = 'OK';
console.log('Yes');
alert('!!!');
```

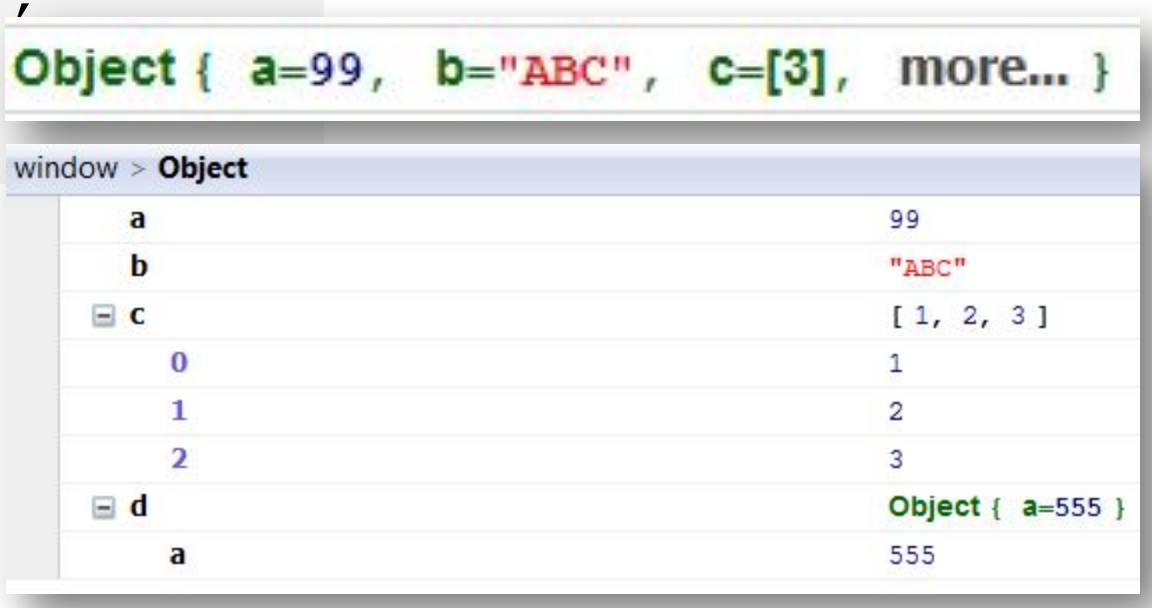




Отладочный вывод данных

При работе с консолью в Firebug вы можете анализировать в том числе сложные типы данных:

```
obj = new Object();
obj.a = 99;
obj.b = "ABC";
obj.c = new Array(1, 2 ,3);
obj.d = new Object();
obj.d.a = 555;
console.log(obj);
```



В PHP аналогичный эффект дают функции print_r() и var_dump().

Проверка существования

Проверка существование элемента массива или переменной осуществляется сравнением с undefined (НЕ строкой "undefined“!!!) (аналог в PHP – isset()):

```
var a = 99;
var b = null;
var c;

if (a === null) {console.log('a is null');}
if (a === undefined) {console.log('a is undefined');}

if (b === null) {console.log('b is null');}
if (b === undefined) {console.log('b is undefined');}

if (c === null) {console.log('c is null');}
if (c === undefined) {console.log('c is undefined');}
```

```
b is null
c is undefined
```

Проверка существования

ВАЖНО! Нельзя обращаться к элементу в иерархии «через голову» его родителей:

```
var obj = new Object();
obj.x = 5;
if (obj.x === undefined) {console.log('obj.x is
undefined');}
if (obj.y === undefined) {console.log('obj.y is
undefined');}
if (zzz.y ===                 console.log('zzz.y is
undefined');}
```

Ошибка!

```
obj.y is undefined
ReferenceError: zzz is not defined
```

Удаление переменных и элементов массива

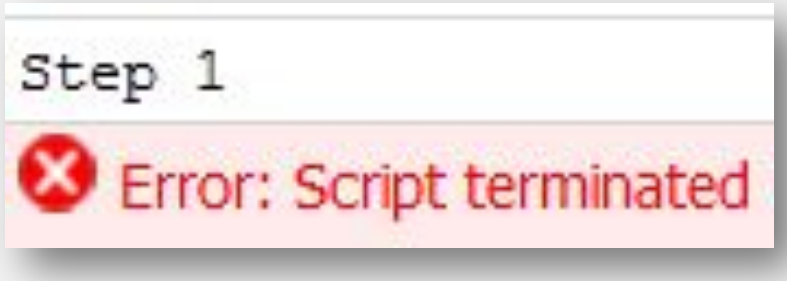
Удаление переменных и элементов массива осуществляется немного по-разному (аналог в PHP – unset()):

```
// Ещё один способ инициализации
var arr = [100, 200, 300, 400, 500];

arr.splice(3, 1);
console.log(arr); // [100, 200, 300, 500]

delete arr[2];
console.log(arr); // [100, 200, undefined, 500]

function Fnc(){};
Fnc.prototype = { x: 123 };
var example = new Fnc();
console.log(example.x); // 123
example.x = 456;
console.log(example.x); // 456
delete example.x;
console.log(example.x); // 123 (обращение к родительскому свойству)

z = 555;
delete z;
console.log(z); // 555 (не удаляется; особенно это относится к объявлению с var)
z = undefined;
console.log(z); // undefined
```

Досрочное прекращение выполнения скрипта

Для досрочного прекращения выполнения скрипта (аналог в PHP – **exit()** или **die()**) есть несколько альтернативных решений.

```
function fnc()
{
 console.log('Step 1');
 return false; // Выход из функции (тут классика).
 console.log('Step 2');
}

fnc();

// Остановка скрипта.
throw new Error('Script terminated');
console.log('Step 3');
```



Сборка скрипта из нескольких файлов

Прямых аналогов PHP-функций include_*/require_* в JavaScript нет. Решения по сборке скриптов из частей таковы:

- Подключить к HTML несколько JS-файлов.
- Использовать eval() (но это опасно).
- Использовать создание функции из её текста.

```
var someCode1 = 'console.log("First");'
eval(someCode1);

var someCode2 = 'return "Second";';
var fnc = new Function(someCode2);

console.log(fnc());
```

```
First
Second
```

Несколько очень простых примеров…

Сейчас в дополнение к уже показанному мы рассмотрим несколько примеров простых и часто используемых в JavaScript действий.

Итак…

Несколько очень простых примеров…

Определение информации о браузере:

```
br = navigator.userAgent.toLowerCase();
```

```
"mozilla/5.0 (windows nt 6.1; wow64; rv:27.0)
gecko/20100101 firefox/27.0"
```

Это очень упрощённый способ. Погуглите полный. **ВНИМАНИЕ!** Любые попытки «напрямую» определить версию браузера ненадёжны. Гуглите способы косвенного определения по поведению и поддерживаемым возможностям.

Несколько очень простых примеров…

Генерация случайных чисел в диапазоне:

```
function getRandomDouble (min, max)
{
 return Math.random() * (max - min) + min;
}

function getRandomInteger (min, max)
{
 return Math.floor(Math.random() * (max - min + 1)) + min;
}

console.log(getRandomDouble(100, 200));
console.log(getRandomInteger(100, 200));
```

```
150.1573960629088
145
```

Несколько очень простых примеров…

Изменение фона всех **td** на странице:

```
<!DOCTYPE html>
 <html>
  <head>
   <meta charset="UTF-8">
   <title>JavaScript: обработка элементов документа</title>
   <script>
   function makeCellsGreen()
   {
    var list=document.getElementsByTagName("td");
    for (var cell_key in list)
    {
     if (list[cell_key].style !== undefined)
     {
      list[cell_key].style.backgroundColor = 'green';
     }
    }
   }
   </script>

  </head>
 <body onload="makeCellsGreen()">
  <div id="sample"></div>

  <table>
   <tr>
    <td> </td>
   </tr>
   <tr>
    <td><a href="#">Link</a></td>
   </tr>
  </table>

</body>
</html>
```

См. пример в файле **01_js_samples_misc\04_scan_all_elements.html**

ОПЕРАТОРЫ И УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ JAVASCRIPT

ОПЕРАТОРЫ JAVASCRIPT

Виды операторов в JavaScript

Операторы в JavaScript бывают:

- Унарными – с одним операндом:

```
b = ! b;
i++;
```

- Бинарными – с двумя операндами (таких – абсолютное большинство):

```
c = a + b;
```

- Тернарными – с тремя операндами (такой оператор в JavaScript только один):

```
x = 10;
y = (x>=10) ? "OK" : "No";
```

Краткая выдержка из документации

http://javascript.ru/manual/operator

- delete
- function
- in
- instanceof
- new
- this
- typeof
- void
- Аксессоры: object.property и object["property"]
- Арифметические: +, -, *, /, %, ++, --
- Запятая: ,
- Логические: &&, ||, !
- Присваивания: =, +=, -=, *=, /=, >>=, <<=, >>>=, &=, |=, ^=
- Сравнения: ==, !=, ===, !==, >, >=, <, <=
- Побитовые: &, |, ^, ~, <<, >>, >>>
- Строковые: +, +=
- Условный: condition ? ifTrue : ifFalse

Выражения с несколькими операторами

Для простоты рекомендуется запомнить два правила:

1. JavaScript правильно учитывает приоритет арифметических операций.
2. В любых иных случаях или не используйте несколько операторов (особенно – разного типа) в одном выражении, или пользуйтесь скобками.

Для желающих понять суть проблемы глубже есть раздел документации, посвящённый ассоциативности операторов:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

Оператор присваивания

Оператор присваивания обозначается знаком = и активно используется для инициализации переменных и присваивания переменным и элементам новых значений:

```
a = 20;
document.getElementById('some_element').innerHTML = 'OK';
```

Для операций + - * / % (и многих других) поддерживается сокращённая форма записи оператора присваивания:

```
a = 2;
a += 1; // a = a + 1;
a %= 2; // a = a % 2;
```

Присваивание по ссылке

В JavaScript присваивания по ссылке НЕТ.

Очень упрощённо идею ссылок в JavaScript можно выразить так:

- Нельзя хранить объекты, можно хранить ссылки на объекты.
- Нельзя передавать по ссылке, но передаются ссылки на объекты.
- Ссылок на примитивные значения нет, они не имеют смысла.

Арифметические операторы

Операции с числами производятся в дробной форме.

Оператор	Действие	Пример
a= - a;	Смена знака	a=5; a=-a; // -5
c=a + b;	Сложение	c=3+x;
c=a - b;	Вычитание	c=17.6-z;
c=a * b;	Умножение	z=n*x*76;
c=a / b;	Деление	x=n/y;
c=a % b;	Деление по модулю (остаток)	c=5%2; // 1

Операторы инкремента и декремента

JavaScript поддерживает префиксные и постфиксные операторы инкремента и декремента числовых переменных. Инкрементирование или декрементирование логических переменных приводит их к числовому виду.

Оператор	Название	Результат
++a;	Префиксный инкремент	Увеличивает значение переменной на 1 и возвращает её значение (новое)
--a;	Префиксный декремент	Уменьшает значение переменной на 1 и возвращает её значение (новое)
a++;	Постфиксный инкремент	Возвращает значение переменной (старое) и увеличивает значение переменной на 1
a--;	Постфиксный декремент	Возвращает значение переменной (старое) и уменьшает значение переменной на 1

Небольшая задача для закрепления материала

Что получится в результате выполнения такого кода?

```
i=2;
i += i++ + ++i;
console.log(i);

// 8
```

А в PHP – 10.

Сейчас будет пояснение…

Небольшая задача для закрепления материала – пояснение

```
i=2;
i += i++ + ++i;
console.log(i);
```

1. i == 2;
2. Срабатывает префиксный инкремент ++i, выражение принимает вид:
3. i += i++ + 3;
4. Происходит подстановка закэшированного значения, выражение принимает вид:
5. i += 2 + 3; или: i += 5;
6. Происходит «разворачивание» «сокращённой записи сложения и присваивания» +=, выражение принимает вид:
7. i=2 + 5;
8. Выполнение выражения (п. 4) даёт i == 7;
9. Срабатывает постфиксный инкремент i++, который увеличивает значение i на единицу, т.е. теперь i == 8;

Поразрядные операторы

Эта группа операторов работает с битовыми представлениями значений операндов. В основном эти операторы применяются для создания т.н. «битовых масок», для решения задач криптографии и при генерации изображений.

Оператор	Действие
c=a & b;	Поразрядное И (AND)
c=a | b;	Поразрядное ИЛИ (OR)
c=a ^ b;	Поразрядное исключающее ИЛИ (XOR)
c= ~ a;	Поразрядное отрицание (NOT)
c=a << 1;	Поразрядный сдвиг влево
c=a >> 2;	Поразрядный сдвиг вправо

Логические операторы

Эта группа операторов используется для определения логического значения выражения (в т.ч. с несколькими операндами). Подробнее мы рассмотрим их в разделе, посвящённом управляющим конструкциям.

Оператор	Действие
c= !a;	Логическое отрицание (NOT)
c=a && b;	Логическое И (AND)
c=a || b;	Логическое ИЛИ (OR)

Операторы сравнения

Операторы сравнения позволяют сравнивать между собой значения переменных. Обратите внимание: при «мягком» сравнении («==») будет происходить переключение типов (см. две «большие таблицы» в разделе про типы данных).

Оператор	Название	Результат
a == b	Равно	TRUE, если a равно b
a === b	Равно по типу и значению	TRUE, если a равно b по типу и значению
a != b	Не равно	TRUE, если a НЕ равно b
a !== b	Не равно по типу или значению	TRUE, если a НЕ равно b по типу ИЛИ значению
a < b	Меньше	TRUE, если a < b
a > b	Больше	TRUE, если a > b
a <= b	Меньше либо равно	TRUE, если a <= b
a >= b	Больше либо равно	TRUE, если a >= b

Строковые операторы

В JavaScript есть один строковый оператор – плюс (+). Это оператор конкатенации (склеивания) строк. Он поддерживает сокращённую запись (конкатенацию с присваиванием).

```
a = 'Hello';
b = a + ' world!';
console.log(b);

a = 'Hello';
a += ' world!';
console.log(a);
```

```
Hello world!
Hello world!
```

Оператор «запятая»

Оператор «запятая» вычисляет оба операнда и возвращает значение второго, что позволяет включить несколько выражений в то место кода, где должно быть одно выражение.

```
for (var i=0, j=9; i <= 9; i++, j--)
{
 console.log("a[" + i + "][" + j + "]= " + a[i][j]);
}
```

Оператор проверки принадлежности классу

Для определения того, является ли переменная экземпляром некоторого класса, используется оператор

boolean **instanceof** var

который возвращает true или false.

```
var someString = new String();
var someDate = new Date();

console.log(someString instanceof String); // true
console.log(someString instanceof Object); // true
console.log(someString instanceof Date);   // false

console.log(someDate instanceof Date);     // true
console.log(someDate instanceof Object);   // true
console.log(someDate instanceof String);   // false
```

Оператор определения типа

Для определения типа переменной используется оператор

```
string typeof var
```

который возвращает тип переменной в виде строки:

- boolean;
- nember;
- string;
- object;
- function;
- null;
- undefined.

```
var a = 5;
console.log(typeof a); // "number"

var a = 'test';
console.log(typeof a); // "string"
```

Оператор доступа к свойству (аксессор)

Для доступа к свойству объекта или элементу массива используются т.н. «аксессоры».

. или [ ]

Это проще показать на примере:

```
arr = new Array(100, 200, 300);
obj = new Object({x:111, y:222});

arr.a  = 12345;
arr[b] = 67890;

obj.x  = 11111;
// obj[y] = 33333; // Так нельзя, это не элемент массива.

console.log(arr.a);  // 12345
console.log(arr[a]); // undefined
console.log(arr.b);  // undefined
console.log(arr[b]); // 67890
// console.log(obj[x]); // Так нельзя, это не элемент массива.
console.log(obj.y);  // 222
```

Оператор проверки наличия свойства

Для проверки наличия свойства или метода у объекта или элемента (по индексу!) в массиве используется оператор

boolean **in** Object

Если этот оператор применить не к объекту, будет ошибка.

```
obj = new Object({ a: 5});
console.log("a" in obj); // true
console.log("b" in obj); // false
console.log("toString" in obj); // true, т.к toString есть в прототипе.

arr = new Array("a", "b", "c");
console.log(1 in arr); // true
console.log(22 in arr); // false

delete arr[1];

console.log(1 in arr); // false, т.к. элемент удалён.
```

Оператор удаления

Для удаления переменной, элемента массива и т.д. используется оператор

boolean **delete** something

который возвращает false (если удаляемый объект существует и не может быть удалён) или true в любых других случаях..

```
x = 42;          // "Обычные переменные" являются свойствами глобального объекта window.
var y = 43;      // Переменная.
obj = new Number();
obj.x = 4;       // Свойство объекта.
obj.y = 5;       // Свойство объекта.

delete x;        // true (x объявлено без var).
delete y;        // false (объявлено с var, флаг DontDelete).
delete Math.PI;// false (встроенный объект, флаг DontDelete).
delete obj.x     // true (пользовательское свойство).

with(obj)
{
 delete y;       // true (эквивалент delete obj.y).
}

delete obj;      // true
```

Оператор исполнения

Исполнения кода без возврата значения и воздействия на контекст используется оператор

```
void (код)
```

Например:

```
someCode = 'x=999;';
x = 1;

void(someCode);
console.log(x); // 1

eval(someCode);
console.log(x); // 999
```

Операторы function, new, this

Оператор **function** используется для создания функций. Об этом будет подробно в разделе, посвящённом функциям.

Оператор **new** используется для создания объектов. Об этом будет подробно в разделе, посвящённом ООП.

Оператор **this** возвращает ссылку на объект, являющийся текущим контекстом вызова. Об этом будет подробно в разделе, посвящённом ООП.

УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ JAVASCRIPT

Оператор условия

Оператор условия (if) может использоваться в JavaScript в нескольких формах.
Самый простой вариант:

```
x = 10;
if (x == 10)
{
 console.log("Десять.");
}
```

Оператор условия

Оператор условия может содержать необязательную секцию else (PHP'шного elseif в JavaScript НЕТ!).

```
x = 10;
if (x == 10)
{
 console.log("Десять.");
}
else
{
 console.log("Не десять.");
}
```

Оператор условия

Оператор условия может быть вложенным (в любой секции).

```
x = 10;
y = 20;
if (x == 10)
{
 if (y == 20)
 {
  console.log("Десять. Двадцать.");
 }
}
```

Оператор условия

Условия могут быть сложными (составными), здесь пригодятся уже рассмотренные нами ранее логические операторы. Обратите внимание: в круглые скобки берётся как всё сложное условие целиком, так и каждое простое условие в его составе по отдельности:

```
x = 10;
y = 20;
if ((x == 10)&&(y==20))
{
  console.log("Десять и двадцать.");
}
```

Оператор переключения

Оператор переключения (switch) является наиболее удобным средством для организации т.н. «мультиветвления».

```
a = "One";

switch (a) // Любой тип данных, кроме массивов и объектов
{
 case 10:
  console.log('Десять.');
 break; // Наличие break обязательно в конце каждого case
 case 'One':
  console.log('Один.');
 break; // Наличие break обязательно в конце каждого case
 default:  // Эта секция может отсутствовать
  console.log('Иное значение.');
}
```

Цикл с предусловием

Цикл с предусловием может не выполниться ни разу, т. к. условие проверяется перед выполнением тела цикла:

```
i = 0;
while(++i <= 5)
 {
   console.log(i);
 }
```

Цикл с постусловием

Цикл с постусловием выполнится хотя бы один раз, т.к. условие проверяется после выполнения тела цикла:

```
i = 0;
do
 {
  console.log(i);
 }
while (++i <= 5);
```

Итерационный цикл

Итерационный цикл выполняется заданное количество раз, причём его синтаксис включает инициализацию счётчика, условие выхода и правило изменения счётчика:

```
var i;
for (i=0; i<=5; i++)
{
console.log(i);
}
```

Итерационный цикл

Специальная модификация итерационного цикла (for) позволяет проходить по любому массиву (по одному уровню для многомерных массивов), на каждом шаге извлекая ключ элемента:

```
arr = new Array(100, 200, new Array(1, 2, 3), 300, 400);
for (key_one in arr)
{
 if (Object.prototype.toString.call(arr[key_one]) !== '[object Array]')
 {
  console.log(arr[key_one]);
 }
 else
 {
  for (key_two in arr[key_one])
  {
   console.log(arr[key_one][key_two]);
  }
 }
}
```

```
100
200
1
2
3
300
400
```

Пропуск остатка итерации и выход из цикла

JavaScript позволяет пропустить выполнение части тела цикла и сразу перейти на следующую итерацию с помощью оператора **continue**. Досрочный выход из цикла выполняется оператором **break**.

```
for (var i=0; i<999; i++)
{
 if (i<500) continue;
 console.log(i);
 break;
}
```

Тернарный оператор

Управление выполнением программы можно реализовывать и с помощью тернарного оператора – сокращённого аналога оператора if:

```
x = 10;
y = (x == 10) ? "Десять" : "Не десять";

(y == "Десять") ? console.log("Да") : console.log("Нет");
```

МАТЕМАТИЧЕСКИЕ ФУНКЦИИ JAVASCRIPT

Общие сведения

Несмотря на то, что JavaScript редко используется для решения математических задач, есть ряд функций, использование которых может оказаться полезным.

Полный список математических функций, предоставляемых объектом Match, можно увидеть здесь: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

Настоятельно рекомендуется ознакомиться с этим списком функций – хотя бы для того, чтобы знать, что они есть, и «не изобретать велосипеды».

Округление чисел

Для округления чисел используются методы Math:

- До ближайшего целого: Math.round().
- До ближайшего меньшего целого: Math.floor().
- До ближайшего большего целого: Math.ceil().

```
Math.round(55.7777);   // 56
Math.floor(55.7777);   // 55
Math.ceil(55.7777);    // 56
```

Здесь есть отличный пример доработки для обеспечения округления с заданной точностью: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/round

Получение случайных чисел

Для получения случайных чисел используется метод Math.random(), который возвращает дробь от 0 (включительно) до 1 (не включительно).

```
console.log(Math.random());
```

Как получить целое число или дробь в заданном диапазоне, мы уже рассматривали ранее.

Перевод чисел между системами счисления

В JavaScript перевод между системами счисления проще всего выполнить так:

```
x = 255;
hexString = x.toString(16);
console.log(hexString);      // ff

y = parseInt("0xFFFF", 16); // 65535
console.log(y);
```

Определение минимального и максимального значения

Методы Math.max и Math.min позволяют искать минимальное и максимально значение в наборе чисел, а с небольшой доработкой – и в массиве чисел.

```
Math.max(10, 20, 5);    // 20
Math.min(-10, -20);     // -20
Math.max("ABC", "OK"); // NaN

function getMaxOfArray(numArray)
{
return Math.max.apply(null, numArray);
}

arr = new Array(1, 100, 5, 500);
console.log(getMaxOfArray(arr)); // 500
```

Определение корректности чисел

В JavaScript многие операции с числами могут привести к переполнению разрядной сетки. Проверить результат выполнения операций с числами на корректность можно с помощью функций isFinite() и isNaN():

```
x = 1/0;
y = -1/0;
z = 9e999999999999;
a = 5;
s = "Test";

console.log(isFinite(x)); // false
console.log(isFinite(y)); // false
console.log(isFinite(z)); // false
console.log(isFinite(a)); // true
console.log(isFinite(s)); // false

console.log(isNaN(x)); // false
console.log(isNaN(y)); // false
console.log(isNaN(z)); // false
console.log(isNaN(a)); // false
console.log(isNaN(s)); // true
```

ФУНКЦИИ JAVASCRIPT, ОПРЕДЕЛЯЕМЫЕ ПОЛЬЗОВАТЕЛЕМ

Небольшое введение

Прежде чем продолжить рассмотрение библиотечных функций JavaScript, нужно научиться писать свои собственные функции.

JavaScript следует такой логике при работе с функциями:

- Тип возвращаемых значений не указывается и может быть любым.
- Тип предаваемых параметров не указывается и может быть любым.
- Поддерживается переменное количество параметров.
- НЕ поддерживается указание значений параметров по умолчанию.
- НЕ поддерживается свободный выбор варианта передачи по ссылке или значению (объекты всегда передаются по ссылке, примитивы – по значению).

Объявление и вызов функции

В самом простом случае объявление и вызов функции выглядит так:

```
function sqr(x)
{
 return x*x;
}

console.log(sqr(2)); // 4
```

Область видимости переменных

В отличие от PHP в JavaScript переменная, используемая в функции без ключевого слова var, является ГЛОБАЛЬНОЙ.

```
a = 10;
b = 20;

function test()
{
 a = 1000;
 var b = 2000;
 var c = 40;
 d = 50;
}

console.log(a); // 10
console.log(b); // 20
test();
console.log(a); // 1000
console.log(b); // 20
//console.log(c); // Ошибка, переменная не существует вне функции.
console.log(d); // 50
```

Передача параметров по значению и по ссылке

В JavaScript нельзя выбрать, как передавать параметр в функцию. Действует правило: если параметр – объект, то передаётся ссылка, иначе – значение.

```
a = 10;
b = new Object();

function test(one, two)
{
 one = 50;
 two.z = 999;
}

test(a, b);

console.log(a); // 10
console.log(b); // Object { z=999 }
```

Параметры со значением по умолчанию

Несмотря на то, что в JavaScript нельзя указать значение параметра по умолчанию, это поведение можно эмулировать:

```
function fnc(a, b)
{
  a = typeof a !== 'undefined' ? a : 999;
  b = typeof b !== 'undefined' ? b : "Test";
  console.log(a, b);
}

fnc();          // 999 Test
fnc(5);         // 5 Test
fnc(7, "Yes"); // 7 Yes
```

Передача переменного количества параметров

В JavaScript в функцию можно передать больше параметров, чем указано при её объявлении. Внутри функции к параметрам можно получить доступ с помощью свойства arguments:

```
function fnc(a, b)
{
 console.log(a, b); // A B
 for (var i = 0; i < arguments.length; i++)
  {
    console.log(arguments[i]);  // A B C D
  }
}

fnc("A", "B", "C", "D");
```

Рекурсия

Понятие рекурсии относится к базовым понятиям информатики.

Но если надо его напомнить – говорите. Рассмотрим и запишем.

```
function fact(x)
{
 if (x == 0)
  {
   return 1;
  }
 return x*fact(x-1);
}

console.log(fact(5)); // 120
```

Анонимные функции и замыкания

JavaScript поддерживает:

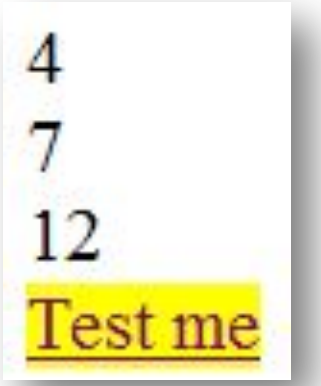
- анонимные функции – функций, которые объявляются в месте использования и не получают уникального идентификатора для доступа к ним.
- замыкания – функции, в теле которых присутствуют (не в качестве параметров) ссылки на переменные, объявленные вне тела этой функции. Такие функции ссылаются на свободные переменные в своём контексте.

Очень полезно:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures

Анонимные функции и замыкания

Создание и использование анонимных функций и замыканий выглядит так:

```
<!DOCTYPE html>
 <html>
  <head>
   <meta charset="UTF-8">
   <title>JavaScript: анонимные функции и замыкания</title>
   <script>

   // Просто функция, которая сработает по факту загрузки документа.
   function doIt()
   {
    // Анонимная функция
    var sqr = function(x){return x*x};
    document.getElementById('sample').innerHTML += sqr(2) + '<br>';

    // Использование замыкания
    var add5 = makeAdder(5);
    var add10 = makeAdder(10);
    document.getElementById('sample').innerHTML += add5(2) + '<br>';
    document.getElementById('sample').innerHTML += add10(2) + '<br>';
   }

   // Замыкание
   function makeAdder(x)
   {
    return function(y)
     {
      return x + y;
     };
   }
   </script>

  </head>
 <body onload="doIt()">
  <div id="sample"></div>

  <!-- ещё один пример использования анонимной функции
       (здесь -- в качестве обработчика события) -->
  <a href="#" onclick="(function(div){div.style.backgroundColor='yellow';})(this)">Test me</a>

</body>
</html>
```



См. пример в файле **01_js_samples_misc\05_anonymous_and_closures.html**

Пространства имён в JavaScript

Q: Поддерживает ли JavaScript т.н. «пространства имён» (namespaces)?
A: Явным образом – нет. Но их можно эмулировать так:

```
var yourNamespace = {

    fnc1: function() {
    },

    fnc2: function() {
    }
};

...

yourNamespace.fnc1();
```

Почитать подробнее:
http://msdn.microsoft.com/en-us/magazine/gg578608.aspx

РАБОТА С МАССИВАМИ В JAVASCRIPT

Общие сведения

Если немного расширить ранее озвученный набор фактов о массивах в JavaScript, получим, что они:

- Бывают только динамическими (причём меняться может и мерность массива, но не так легко, как в PHP).
- Могут содержать одновременно данные любых типов.
- В качестве ключей (индексов) могут использовать как числа, так и строки.
- Могут быть использованы для хранения таких классических структур как очереди, стеки, деревья и т.д.

Небольшой пример, поясняющий «многомерность массивов» в JavaScript:

```
var arr = new Array();
//arr[5][7] = 'Test'; // Ошибка!
arr[5] = new Array();
arr[5][7] = 'Test'; // Так – работает.
```

Q: В чём разница объявления массива разными способами?
A: Рассмотрим на примере.

```
var arr1 = new Array(5); // Массив с пятью undefined элементами
var arr2 = Array(3); // Массив с тремя undefined элементами
var arr3 = []; // Пустой массив
var arr4 = Array(); // Пустой массив
var arr5 = new Array(); // Пустой массив
var arr6 = new Array('A', 'B', 'C'); // Массив с элементами 'A', 'B', 'C'
var arr7 = Array('A', 'B', 'C'); // Массив с элементами 'A', 'B', 'C'
```

Случаи 1, 5, 6 являются классическими и наиболее рекомендуемыми.

Функции по работе с массивами

Сейчас мы рассмотрим основные функции по работе с массивами в JavaScript.

Полный их перечень можно увидеть, например, здесь: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Важно! В JavaScript рекомендуется использовать Array только для «классических строгоиндексированных» массивов, для ассоциативных лучше использовать Object.

Определение количества элементов в массиве

Для определения размера массива используется свойство length, которое «реагирует» не на реальное количество элементов, а возвращает «последний индекс + 1».

```
var arr = new Array(5);
arr[999] = 'Test';
console.log(arr.length); // 1000
```

Определение, является ли переменная массивом

Для определения того, является ли переменная массивом, можно использовать Array.isArray():

```
// Все эти примеры вернут true
Array.isArray([]);
Array.isArray([1]);
Array.isArray( new Array() );
Array.isArray( Array.prototype );

// Все эти пример вернут false
Array.isArray();
Array.isArray({});
Array.isArray(null);
Array.isArray(undefined);
Array.isArray(17);
Array.isArray("Array");
Array.isArray(true);
Array.isArray(false);
Array.isArray({ __proto__ : Array.prototype });
```

Поиск элемента в массиве

Для поиска элемента по ключу используется ранее рассмотренное решение со сравнением элемента с undefined. Для поиска элемента по значению используется **Array.indexOf(searchElement[, fromIndex])**.

```
var arr = [2, 5, 9, 5];
if (arr[999] === undefined)
{
    console.log('Element 999 is not set.')
}
console.log(index = arr.indexOf(5));    // 1
console.log(index = arr.indexOf(555)); // -1
```

Также см. Array.lastIndexOf(searchElement[, fromIndex])

Поиск элемента в массиве

Ещё одним способом проверки существования или некоего свойства элементов массива является использование **Array.every()** и **Array.some()**, проверяющие с помощью вызова функции тот факт, что все или хотя бы один элемент удовлетворяют некоторому критерию.

```
function isPositive(element, index, array)
{
  return (element > 0);
}
var arr = [2, -5, 8, 1, 4];
console.log(arr.some(isPositive));  // true (числа > 0 есть)
console.log(arr.every(isPositive)); // false (не все числа > 0)
```

Обработка всех элементов массива

Для обработки всех элементов массива можно использовать цикл for, а также методы **Array.filter()** и **Array.forEach()**:

```
function saveOnlyPositive(element)
{
    return element > 0;
}
var arr = [12, -5, -8, -130, 44];
arr = arr.filter(saveOnlyPositive); // [12, 44]


function sqrArrayElements(element, index, array)
{
    array[index] = element*element;
}
arr.forEach(sqrArrayElements);
console.log(arr); // [144, 1936]
```

Обработка всех элементов массива

И ещё один способ – использование Array.map() позволяет создать новый массив на основе существующего:

```
var arr = Array('A', 'B', 'C');
var codes = arr.map(function(x) { return x.charCodeAt(0); })
console.log(arr);   // ["A", "B", "C"]
console.log(codes); // [65, 66, 67]
```

Объединение массивов или значений в массивы

Использование **Array.concat()** позволяет объединить несколько массивов или несколько отдельных значений в массив:

```
var num1 = [1, 2, 3];
var num2 = [4, 5, 6];
var num3 = [7, 8, 9];

var nums = num1.concat(num2, num3);
nums = nums.concat(999, 1000, 1001);
console.log(nums);
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 999, 1000, 1001]
```

Объединение элементов массива в строку

Использование **Array.join()** позволяет объединить элементы массива в строку (используя разделитель, если это необходимо):

```
var arr = new Array(new Array('Иванов', 'Иван', 'Иванович'),
new Array('Петров', 'Пётр', 'Петрович'));
var str1 = arr.join(' ');
var str2 = arr[0].join(' ');
var str3 = arr[1].join();

// Иванов,Иван,Иванович Петров,Пётр,Петрович
console.log(str1);
console.log(str2); // Иванов Иван Иванович
console.log(str3); // Петров,Пётр,Петрович
```

Сортировка массива

Для сортировки массива применяется Array.sort() и (если необходимо) Array.reverse():

```
var arr = new Array('Один', 'Два', 'Три');
function compareByLengthReverse(a, b)
{
    if (a.length > b.length)
    {
        return -1;
    }
    else
    {
        if (a.length < b.length)
        {
            return 1;
        }
        else
        {
            return 0;
        }
    }
}

arr.sort();
console.log(arr); // ["Два", "Один", "Три"]
arr.sort(compareByLengthReverse);
console.log(arr); // ["Один", "Два", "Три"]
arr.reverse();
console.log(arr); // ["Три", "Два", "Один"]
```

Работа с массивом как со стеком

В JavaScript есть готовые решения для работы с массивом как со стеком: **Array.push()** добавляет элемент в конец массива и возвращает новую длину массива, **Array.pop()** удаляет элемент из конца массива и возвращает значение этого элемента.

```
var arr = Array(10, 20, 30);
console.log(arr.push(999)); // 4
console.log(arr);           // [10, 20, 30, 999]
console.log(arr.pop());     // 999
console.log(arr);           // [10, 20, 30]
```

Работа с массивом как со стеком

Вторым способом работы с массивом как со стеком является использование **Array.unshift()** и **Array.shift()**, которые работают аналогично **Array.push()** и **Array.pop()**, но добавляют/удаляют элемент в начале массива:

```
var arr = Array(10, 20, 30);
console.log(arr.unshift(999)); // 4
console.log(arr);              // [999, 10, 20, 30]
console.log(arr.shift());      // 999
console.log(arr);              // [10, 20, 30]
```

Работа с массивом как с очередью

Соответственно, использование **Array.unshift()/Array.pop()** и **Array.push()/Array. shift()**, позволяет работать с массивом как с очередью:

```
var arr = Array(10, 20, 30);
console.log(arr.unshift(999)); // 4
console.log(arr);              // [999, 10, 20, 30]
console.log(arr.pop());        // 30
console.log(arr);              // [999, 10, 20]
console.log(arr.push(555));    // 4
console.log(arr);              // [999, 10, 20, 555]
console.log(arr.shift());      // 999
console.log(arr);              // [10, 20, 555]
```

Извлечение и удаление части массива

Для извлечения части массива используется **Array.slice()**, а для модификации массива (в т.ч. удаления части элементов) – **Array.splice()**.

```
var arr = Array(10, 20, 30, 40, 50);
arr_new = arr.slice(2, 4);
console.log(arr_new); // [30, 40]

arr.splice(2, 0, 'Test'); // Добавить после 2-го элемента.
console.log(arr); // [10, 20, "Test", 30, 40, 50]
arr.splice(2, 1); // Удалить второй элемент.
console.log(arr); // [10, 20, 30, 40, 50]
arr.splice(3, 1, 'ABC'); // Заменить 3-й элемент строкой.
console.log(arr); // [10, 20, 30, "ABC", 50]
```

Задача для закрепления материала

Для лучшего понимания того, как работают массивы, рекомендуется выполнить следующее задание.

Дано: многомерный массив произвольной мерности и произвольного размера (в т.ч. в любом измерении).

Сделать: удалить из массива все строки, а каждое отрицательное число возвести в квадрат.

РАБОТА СО СТРОКАМИ В JAVASCRIPT

Общие сведения

Если немного расширить ранее озвученный набор фактов о строках в JavaScript, получим, что:

- на длину строки нет никаких искусственных ограничений (только объём памяти, доступный скрипту);
- JavaScript умеет рассматривать строки как массивы символов.

Функции по работе со строками

Сейчас мы рассмотрим основные функции по работе со строками в JavaScript.

Полный их перечень можно увидеть здесь: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

В JavaScript большое количество задач решается путём выполнения операций над строками, потому рассмотрим это подробно.

Мультибайтовая кодировка строк, определение длины

JavaScript считает, что строки представлены в кодировке UTF-16, и считает длину строки в символах, а не в байтах (за определение длины строки отвечает свойство **length**):

```
var str = new String("한국어");
console.log(str.length); // 3
```

Экранирование символов

В некоторых случаях некоторые символы строк должны быть экранированы, чтобы строка могла быть использована в нужном контексте.
В JavaScript нет готового аналога PHP'шной функции htmlspecialchars(), но есть альтернативные решения:

```
// Вариант 1 – написание своей функции
function escapeHtml(text) {
  return text
      .replace(/&/g, "&amp;")
      .replace(/</g, "&lt;")
      .replace(/>/g, "&gt;")
      .replace(/"/g, "&quot;")
      .replace(/'/g, "&#039;");
}

var text_string = '<htmltag/>';

// Вариант 2 – через «трюк» с HTML
var div = document.createElement('div');
var text = document.createTextNode(text_string);
div.appendChild(text);
console.log(div.innerHTML); // &lt;htmltag/&gt;
console.log(escapeHtml(text_string)); // &lt;htmltag/&gt;
```

Работа с отдельными символами строки

Для получения символа строки или его кодового представления можно использовать **String.charAt()** и **String.charCodeAt()**, а также обращение к строке как к массиву символов:

```
var str = 'Тест';
console.log(str.charAt(1));     // е
console.log(str.charCodeAt(1)); // 1077
console.log(str[1]);            // е
str[1] = '-'; // Так -- не работает!
console.log(str);               // Тест
```

Поиск, замена, разбиение и склеивание строк

Для поиска вхождения подстроки в строку можно использовать **String.indexOf()** и **String.lastIndexOf()**:

```
var str = 'Тестовая строка';
console.log(str.indexOf('т'));        // 3
console.log(str.lastIndexOf('т'));    // 10
console.log(str.indexOf('abc'));      // -1
console.log(str.lastIndexOf('abc')); // -1
```

Поиск, замена, разбиение и склеивание строк

Для «вырезания» части строки или разбиения строки в массив подстрок используются: **String.slice()**, **String.substr()**, **String.substring()**, **String.split()**.

```
var str = 'Тестовая строка';
console.log(str.slice(3, 7));       // това
console.log(str.substr(3, 2));      // то
console.log(str.substring(3, 7));   // това
console.log(str.split(' '));
// ["Тестовая", "строка"]

console.log(str.split(''));
// ["Т", "е", "с", "т", "о", "в", "а", "я",
// " ", "с", "т", "р", "о", "к", "а"]
```

Поиск, замена, разбиение и склеивание строк

Для замены подстроки в строке используется **String.replace()**, для «склеивания строк» – **String.concat()**, для очистки строки от концевых пробелов – **String.trim()**.

```
var str = 'Тестовая строка';
console.log(str.replace('ст', '***'));
console.log(str = str.concat('. И ещё одна.      '));
console.log(str.trim());
```

```
Те***овая строка
Тестовая строка. И ещё одна.~~~~~
Тестовая строка. И ещё одна.
```

Тильды символизируют пробелы, которых на экране не видно ☺.

Управление регистром

Для управления регистром строки используются **String.toLowerCase()** и **String.toUpperCase()**:

```
var str = 'Тестовая строка';
console.log(str.toLowerCase());
// тестовая строка

console.log(str.toUpperCase());
// ТЕСТОВАЯ СТРОКА
```

Преобразование строки в веб-ссылку

Для автоматизации получения из строки полноценной гиперссылки и «ссылки-закладки» используются **String.link()** и **String.anchor()**.

```
var humanText = "Tut.by";
var URL = "http://tut.by/";
console.log(humanText.link(URL));

var contentsStart = "Table of Contents";
console.log(contentsStart.anchor("contents_anchor"));
```

```
<a href="http://tut.by/">Tut.by</a>
<a name="contents_anchor">Table of Contents</a>
```

Работа с регулярными выражениями

О регулярных выражениях будет чуть позже, но пока отметим, что для применения их к строке используются **String.match()**, **String.search()** и **String.replace()**.

```
var str = "Индекс: 220001";
var regex1 = /\d{6}/;
var regex2 = /\d{10}/;
console.log(str.match(regex1));                  // см. ниже
console.log(str.match(regex2));                  // null
console.log(str.search(regex1));                 // 8
console.log(str.search(regex2));                 // -1
console.log(str.replace(regex1, '------')); // Индекс: ------
console.log(str.replace(regex2, '------')); // Индекс: 220001
```

```
["220001"]
0 "220001"
index 8
Input "Индекс: 220001"

null
8
-1
Индекс: ------
Индекс: 220001
```

Хэш-функции

Q: Есть ли в JavaScript поддержка хэш-функций?

A: Встроенной нет. Но есть вот такое:
https://code.google.com/p/crypto-js/

Задания для закрепления материала

Дан текст (о регистре символов в котором мы ничего не знаем).

1. Привести текст к виду: “Каждая Первая Буква Слова В Верхнем Регистре, Остальные В Нижнем”.

2. Привести текст к виду: “Только первая буква – в верхнем регистре, остальные – в нижнем”.

3. Привести текст к виду: “кАЖДАЯ пЕРВАЯ бУКВА сЛОВА в нИЖНЕМ рЕГИСТРЕ, оСТАЛЬНЫЕ в вЕРХНЕМ”.

Задания для закрепления материала

4. Определить, сколько раз в строке встречается каждый из присутствующих там символов (например, в слове test: t = 2, e = 1, s = 1).
5. Определить длину самого короткого и самого длинного слова в строке, вывести все самые короткие и самые длинные слова.
6. Определить среднюю длину слова в строке.

ФУНКЦИИ JAVASCRIPT ПО РАБОТЕ С ДАТОЙ И ВРЕМЕНЕМ

Общие сведения

Для работы с датой и временем в JavaScript используется объект Date.

Как и PHP, JavaScript работает с датой-временем в формате UNIXTIME, но отсчитывает с 1-го января 1970-го года не секунды, а милисекунды.

Полную информацию по работе с датой и временем см. здесь:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

Получение текущих даты и времени

Для получения информации о текущей дате и текущем времени используется статический метод Date.now().

```
console.log(Date.now()); // 1393925189676
```

Получение текущих даты и времени

Для получения «человекочитаемой» информации о текущей дате и текущем времени огромное количество методов объекта Date, из которых мы на примере рассмотрим основные:

```
dt = new Date();
console.log(dt.getDate());              // 4 (4-е число)
console.log(dt.getDay());               // 2 (вторник)
console.log(dt.getFullYear());          // 2014
console.log(dt.getHours());             // 12
console.log(dt.getMilliseconds());      // 233
console.log(dt.getMinutes());           // 32
console.log(dt.getMonth());             // 2 (март, нумерация с 0)
console.log(dt.getSeconds());           // 29
console.log(dt.getTime());              // 1393925549233 (unixtime)
console.log(dt.getTimezoneOffset());    // -180 (-3 часа в минутах)
```

Создание определённого значения даты-времени

Для создания определённого значения даты-времени используется огромное количество сеттеров, из которых мы также рассмотрим основные:

```
dt = new Date();
dt.setDate(31);
dt.setFullYear(2015);
dt.setHours(23);
dt.setMilliseconds(0);
dt.setMinutes(30);
dt.setMonth(11);
dt.setSeconds(15);
console.log(dt);
// Date {Thu Dec 31 2015 23:30:15 GMT+0300 (Kaliningrad Standard Time)}
```

Получение даты-времени в разных форматах

И, наконец, рассмотрим на примерах получение даты-времени в нескольких наиболее распространённых форматах:

```
console.log(dt);
console.log(dt.toDateString());
console.log(dt.toISOString());
console.log(dt.toJSON());
console.log(dt.toLocaleDateString());
console.log(dt.toLocaleString());
console.log(dt.toLocaleTimeString());
console.log(dt.toString());
console.log(dt.toTimeString());
console.log(dt.toUTCString());
```

```
Date {Thu Dec 31 2015 23:30:15 GMT+0300 (Kaliningrad Standard Time)}
Thu Dec 31 2015
2015-12-31T20:30:15.000Z
2015-12-31T20:30:15.000Z
Thursday, December 31, 2015
Thursday, December 31, 2015 23:30:15
23:30:15
Thu Dec 31 2015 23:30:15 GMT+0300 (Kaliningrad Standard Time)
23:30:15 GMT+0300 (Kaliningrad Standard Time)
Thu, 31 Dec 2015 20:30:15 GMT
```

Дата и время: Q&A

Q: Можно ли сразу «создать нужную дату»?
A: Да.

```
new Date(year, month [, day, hour, minute, second, millisecond])
```

Q: Есть ли в JavaScript аналог функции PHP date()?
A: Нет.

Q: И checkdate() нет?
A: Нет.

Q: И sleep() / usleep() нет?
A: Нет.

Задача для закрепления материала

Написать скрипт, который строит календарь за указанный год с указанием дней недели.

[Упрощённый вариант] Календарь может представлять собой просто «ленту дат».

[Усложнённый вариант] Календарь должен представлять собой таблицу 3x4, где каждая ячейка представляет собой один месяц – т.е. «обычный календарь», который каждый из вас видел сотни раз в жизни.

ОБРАБОТКА СОБЫТИЙ В JAVASCRIPT, РАБОТА С DOM

Общая информация

Событие в JavaScript – это информация (как правило, от браузера) о том, что что-то произошло.

В общем случае можно выделить такие виды событий:

- DOM-события, которые инициируются элементами DOM (например, click, mouseover и т.д.); см. о событиях в HTML-элементах в предыдущем разделе.
- События окна (например, resize при изменении размера окна браузера).
- Прочие события (например load, readystatechange и т. д.)

Отличное, очень полное и простое пояснение работы событий: http://javascript.ru/tutorial/events/intro

Обработчики событий

Чтобы иметь возможность реагировать на возникновение события, нужно назначить для него обработчик. Это можно сделать несколькими способами:

- Через атрибут HTML-тега.
- Через свойство объекта.
- Через специальные решения (в общем случае делящиеся на W3C-совместимые и Microsoft-совместимые).

Обработчики событий: установка через атрибут HTML-тега

Самый простой способ установки обработчика событий – через атрибут HTML-тега. Можно использовать анонимную функцию или указывать существующую функцию (это – лучше и удобнее).

```
<span onclick="alertOnClick()">Кликни здесь (будет alert).</span><br>

<span onclick="(function(elem){elem.style.color='red';})(this)">Кликни здесь
(ссылка поменяет цвет).</span><br>

<span onclick="(function(){alert('Сработал обработчик клика в виде анонимной
функции.')}).call(this)">Кликни здесь (будет ещё один alert).</span><br>
```

См. пример в файле **01_js_samples_misc\06_events_01.html**

Обработчики событий: установка через свойство объекта

Чуть более хитрый способ навесить обработчик – модифицировать соответствующее свойство элементов. Если надо навесить обработчик на много элементов, удобно использовать замыкания.

```
<script>
  function addHandlers()
  {
   spanlist = document.getElementsByTagName("span");
   for (var i=0; i<spanlist.length; i++)
    {
     spanlist[i].onclick = microLog;
     spanlist[i].ondblclick = function()
      {
       document.getElementById('message').innerHTML += 'Вы выполнии двойной клик по слову<br>';
       //this.style.color='red'; Тут это НЕ сработает!
      }

     spanlist[i].onmouseover = (function (elem) {
                               return function() {
                               this.style.color='red';
                               // Или: elem.style.color='red';
                               };
                               })(spanlist[i]);

     spanlist[i].onmouseout = (function (elem) {
                              return function() {
                              this.style.color='black';
                              // Или: elem.style.color='red';
                              };
                              })(spanlist[i]);
   }
  }

  function microLog()
  {
   document.getElementById('message').innerHTML += 'Вы кликнули по слову<br>';
  }
  </script>
```

См. пример в файле **01_js_samples_misc\07_events_02.html**

Обработчики событий: специальные решения

Самый правильный способ – использование специальных решений. Сначала – Microsoft-style:

```
elem = document.getElementById('some_id');
handler = function() {
    alert('OK');
};
elem.attachEvent("onclick", handler) // Навесить обработчик.
elem.detachEvent("onclick", handler) // Снять обработчик.
```

Обработчики событий: специальные решения

Теперь – W3C-style:

```
element.addEventListener(имя_события, обработчик, стадия);
```

Стадия (фаза) перехвата = true, стадия (фаза) всплывания = false. Об этом – совсем скоро. Обычно ставится false.

```
element.removeEventListener(имя_события, обработчик, стадия);
```

Обработчики событий: специальные решения

Пример W3C-style:

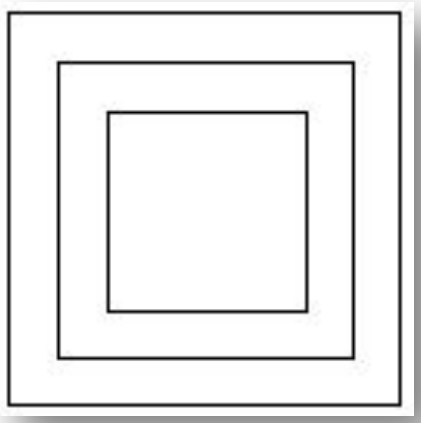
```
function addHandlers()
   {
    var spanlist = document.getElementsByTagName("span");
    for (var i=0; i<spanlist.length; i++)
     {
      spanlist[i].addEventListener("click", microLog, false);

      // Здесь заодно вспомним о замыканиях.
      // Можно получить функцию отдельно...
      var handler_red = makeRed(spanlist[i]);
      spanlist[i].addEventListener("mouseover", handler_red, false);

      // ... или сразу при указании обработчика.
      spanlist[i].addEventListener("mouseout", makeBlack(spanlist[i]), false);
     }
   }

   function makeRed(elem)
   {
    return function()
     {
      this.style.color='red';
     };
   }

   function makeBlack(elem)
   {
    return function()
     {
      elem.style.color='black';
     };
   }

   function microLog()
   {
    document.getElementById('message').innerHTML += 'Вы кликнули по слову<br>';
   }
```

См. пример в файле **01_js_samples_misc\08_events_03.html**

Порядок выполнения событий, стадии перехвата и всплытия

В случае, если событие срабатывает на вложенном элементе, оно сработает и на его родителях.

```
<div id="outer">
   <div id="middle">
      <div id="inner">
      </div>
   </div>
</div>
```



Клик здесь…

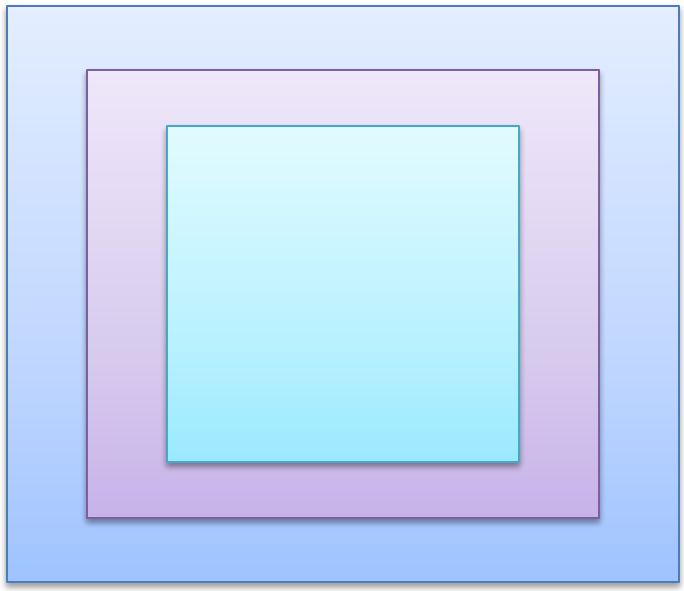
… распространится сюда…

… и сюда.

Порядок выполнения событий, стадии перехвата и всплытия

Если абстрагироваться от кроссбраузерной совместимости, то в общем случае событие «распространяется» в двух направлениях: сначала от самого внешнего элемента к внутреннему (стадия перехвата), а потом обратно (стадия всплытия).

Последовательность срабатывания обработчиков будет такой:
1) outer-catch
2) middle-catch
3) inner-bubble
4) inner-catch
5) middle-bubble
6) outer-bubble



Порядок выполнения событий, стадии перехвата и всплытия

Вот код, который демонстрирует эту логику:

```
<!DOCTYPE html>
 <html>
  <head>
   <meta charset="UTF-8">
   <title>JavaScript: установка обработчиков событий</title>
   <script>
   function addHandlers()
   {
    document.getElementById("inner").addEventListener("click", innerClickedBubble, false);
                    document.getElementById("middle").addEventListener("click", middleClickedBubble, false);
                    document.getElementById("outer").addEventListener("click", outerClickedBubble, false);

                    document.getElementById("inner").addEventListener("click", innerClickedCatch, true);
                    document.getElementById("middle").addEventListener("click", middleClickedCatch, true);
                    document.getElementById("outer").addEventListener("click", outerClickedCatch, true);
   }

   function innerClickedBubble()
   {
    document.getElementById('log').innerHTML += 'inner-bubble<br>';
   }

   function middleClickedBubble()
   {
    document.getElementById('log').innerHTML += 'middle-bubble<br>';
   }

   function outerClickedBubble()
   {
    document.getElementById('log').innerHTML += 'outer-bubble<br>';
   }

   function innerClickedCatch()
   {
    document.getElementById('log').innerHTML += 'inner-catch<br>';
   }

   function middleClickedCatch()
   {
    document.getElementById('log').innerHTML += 'middle-catch<br>';
   }

   function outerClickedCatch()
   {
    document.getElementById('log').innerHTML += 'outer-catch<br>';
   }
   </script>

   <style>
    #outer {
                    width: 100px;
                    height: 100px;
                    border: 1px solid black;
                    }

                    #middle {
                    width: 75px;
                    height: 75px;
                    margin-left: 12px;
                    margin-top: 12px;
                    border: 1px solid black;
                    }

                    #inner {
                    width: 50px;
                    height: 50px;
                    margin-left: 12px;
                    margin-top: 12px;
                    border: 1px solid black;
                    }
   </style>
   </style>
  </head>
  <body onload="addHandlers()">

                    <div id="outer">
                                    <div id="middle">
                                                    <div id="inner">
                                                    </div>
                                    </div>
                    </div>

                    <div id="log"></div>

  </body>
 </html>
```

См. пример в файле **01_js_samples_misc\09_events_04_event_order.html**

Порядок выполнения событий, стадии перехвата и всплытия

На обеих стадиях можно блокировать дальнейшее распространение события:

```
event.stopPropagation();
```

Заодно упомянём полезную вещь – отключения события по умолчанию:

```
event.preventDefault();
```

Если обработчик возвращает **false**, это тоже блокирует событие по умолчанию, но не останавливает дальнейшее распространение событий.

Важно! Другие обработчики этого же события на этом же элементе сработают в любом случае!

Порядок выполнения событий, стадии перехвата и всплытия

Вот код, который демонстрирует блокировку распространения:

```
<!DOCTYPE html>
 <html>
  <head>
   <meta charset="UTF-8">
   <title>JavaScript: установка обработчиков событий</title>
   <script>
   function addHandlers()
   {
    document.getElementById("inner").addEventListener("click", innerClickedBubble, false);
                    document.getElementById("middle").addEventListener("click", middleClickedBubble, false);
                    document.getElementById("outer").addEventListener("click", outerClickedBubble, false);

                    document.getElementById("inner").addEventListener("click", innerClickedCatch, true);
                    document.getElementById("middle").addEventListener("click", middleClickedCatch, true);
                    document.getElementById("outer").addEventListener("click", outerClickedCatch, true);
   }

   function innerClickedBubble(event)
   {
    document.getElementById('log').innerHTML += 'inner-bubble<br>';
                    event.stopPropagation();
   }

   function middleClickedBubble(event)
   {
    document.getElementById('log').innerHTML += 'middle-bubble<br>';
                    event.stopPropagation();
   }

   function outerClickedBubble(event)
   {
    document.getElementById('log').innerHTML += 'outer-bubble<br>';
                    event.stopPropagation();
   }

   function innerClickedCatch(event)
   {
    document.getElementById('log').innerHTML += 'inner-catch<br>';
                    event.stopPropagation();
   }

   function middleClickedCatch(event)
   {
    document.getElementById('log').innerHTML += 'middle-catch<br>';
                    event.stopPropagation();
   }

   function outerClickedCatch(event)
   {
    document.getElementById('log').innerHTML += 'outer-catch<br>';
                    event.stopPropagation();
   }
   </script>

   <style>
    #outer {
                    width: 100px;
                    height: 100px;
                    border: 1px solid black;
                    }

                    #middle {
                    width: 75px;
                    height: 75px;
                    margin-left: 12px;
                    margin-top: 12px;
                    border: 1px solid black;
                    }

                    #inner {
                    width: 50px;
                    height: 50px;
                    margin-left: 12px;
                    margin-top: 12px;
                    border: 1px solid black;
                    }
   </style>
  </head>
  <body onload="addHandlers()">

                    <div id="outer">
                                    <div id="middle">
                                                    <div id="inner">
                                                    </div>
                                    </div>
                    </div>

                    <div id="log"></div>

  </body>
 </html>
```

См. пример в файле
01_js_samples_misc\10_events_05_event_order_no_propagation.html

Объект event и его свойства

Мы снова опустим вопросы кроссбраузерной совместимости и скажем, что много полезной и интересной информации можно извлечь из свойств «самого события» (да, оно предаётся как объект).

Мы не станем разбирать все свойства события (их очень много), но доработаем наш пример для их показа и посмотрим, что там есть интересного…

Очень хороший дополнительный материал:
http://javascript.ru/tutorial/events/properties

Объект event и его свойства

Этот код логирует события в консоль:

```
<!DOCTYPE html>
 <html>
  <head>
   <meta charset="UTF-8">
   <title>JavaScript: установка обработчиков событий</title>
   <script>
   function addHandlers()
   {
    document.getElementById("inner").addEventListener("click", innerClickedBubble, false);
                    document.getElementById("middle").addEventListener("click", middleClickedBubble, false);
                    document.getElementById("outer").addEventListener("click", outerClickedBubble, false);
   }

   function innerClickedBubble(event)
   {
                    console.log(this.id);
                    console.log(event);
   }

   function middleClickedBubble(event)
   {
                    console.log(this.id);
                    console.log(event);
   }

   function outerClickedBubble(event)
   {
                    console.log(this.id);
                    console.log(event);
   }


   </script>

   <style>
    #outer {
                    width: 100px;
                    height: 100px;
                    border: 1px solid black;
                    }

                    #middle {
                    width: 75px;
                    height: 75px;
                    margin-left: 12px;
                    margin-top: 12px;
                    border: 1px solid black;
                    }

                    #inner {
                    width: 50px;
                    height: 50px;
                    margin-left: 12px;
                    margin-top: 12px;
                    border: 1px solid black;
                    }
   </style>
  </head>
  <body onload="addHandlers()">

                    <div id="outer">
                                    <div id="middle">
                                                    <div id="inner">
                                                    </div>
                                    </div>
                    </div>

                    <div id="log"></div>

  </body>
 </html>
```

MOZ_SOURCE_CURSOR		4
MOZ_SOURCE_ERASER		3
MOZ_SOURCE_KEYBOARD		6
MOZ_SOURCE_MOUSE		1
MOZ_SOURCE_PEN		2
MOZ_SOURCE_TOUCH		5
MOZ_SOURCE_UNKNOWN		0
NONE	0	
buttons	0	
defaultPrevented		false
mozInputSource		1
mozMovementX		72
mozMovementY		154
mozPressure	0	
multipleActionsPrevented		false
getModifierState		getModifierState()
initNSMouseEvent		initNSMouseEvent()
stopImmediatePropagation		stopImmediatePropagation()
altKey	false	
bubbles	true	
button	0	
cancelBubble	false	
cancelable	true	
clientX	72	
clientY	76	
ctrlKey	false	
currentTarget	div#middle	
detail	1	
eventPhase	3	
explicitOriginalTarget		div#inner
isChar	false	
isTrusted	true	
layerX	72	
layerY	76	
metaKey	false	
originalTarget	div#inner	
pageX	72	
pageY	76	
rangeOffset	0	
rangeParent	div#inner	
relatedTarget	null	
screenX	72	
screenY	176	
shiftKey	false	
target	div#inner	
timeStamp	703677937	
type	"click"	
view	Window 11_events_06_event_object.html	
which	1	
getPreventDefault		getPreventDefault()
initEvent	initEvent()	
initMouseEvent		initMouseEvent()
initUIEvent	initUIEvent()	
preventDefault		preventDefault()
stopPropagation		stopPropagation()
ALT_MASK	1	
AT_TARGET	2	
BUBBLING_PHASE		3
		1
		2
	8	
		32768
		-32768
	4	

См. пример в файле
01_js_samples_misc\11_events_06_event_object.html

Работа с DOM в контексте событий

Мы уже рассмотрели множество примеров определения обработчиков событий и их функционирования. В завершение этого подраздела рассмотрим, как с помощью JavaScript можно модифицировать DOM-структуру документа (например, в ответ на событие).

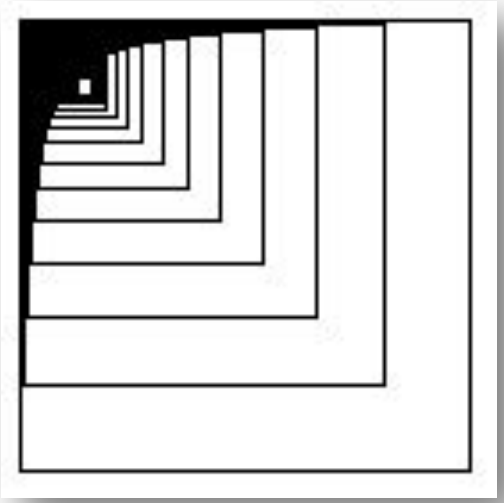
Рассмотрим это на очень простом наборе примеров:

- Вложенные div’ы.
- Рамки вокруг надписей.
- Управление таблицей.

Работа с DOM в контексте событий

Вложенные div'ы (при клике на div мы будем добавлять внутрь него ещё один div):

```
<!DOCTYPE html>
 <html>
  <head>
   <meta charset="UTF-8">
   <title>JavaScript: примеры работы с DOM</title>
   <script>
   function addHandlers()
   {
          document.getElementById("div1").addEventListener("click", onClickListener, false);
   }

   function onClickListener(event)
   {
          var parent_div = event.currentTarget;
          var child_div = document.createElement('div');

          event.stopPropagation();
          parent_div.removeEventListener('click', onClickListener);
          child_div.style.width = '80%';
          child_div.style.height = '80%';
          child_div.style.border = '1px solid black';
          parent_div.appendChild(child_div);
          child_div.addEventListener("click", onClickListener, false);
   }
   </script>

   <style>
    #div1 {
          width: 100px;
          height: 100px;
          border: 1px solid black;
          }
   </style>
 </head>
 <body onload="addHandlers()">

          <div id="div1">

          </div>

 </body>
</html>
```

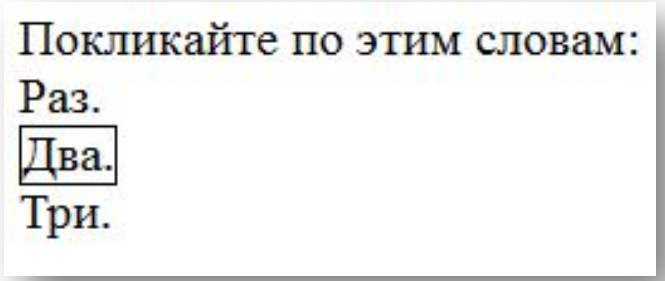


См. пример в файле
01_js_samples_misc\12_examples_01_divs.html

Работа с DOM в контексте событий

При клике по надписям мы будем создавать вокруг них рамки:

```
<!DOCTYPE html>
 <html>
  <head>
   <meta charset="UTF-8">
   <title>JavaScript: примеры работы с DOM</title>
   <script>
   function addHandlers()
   {
    var spanlist = document.getElementsByTagName("span");
    for (var i=0; i<spanlist.length; i++)
          {
      spanlist[i].addEventListener("click", makeBox(spanlist[i]), false);
     }
   }

   function makeBox(elem)
   {
    var clickHandler = function()
     {
           this.style.border='1px solid black';
           this.removeEventListener('click', clickHandler);
     };
         return clickHandler;
   }
   </script>

  </head>
 <body onload="addHandlers()">
  Покликайте по этим словам:<br>
  <span>Раз.</span><br>
  <span>Два.</span><br>
  <span>Три.</span>
 </body>
</html>
```



См. пример в файле
01_js_samples_misc\13_examples_02_boxes.html

Работа с DOM в контексте событий

Мы сделаем кнопку добавления строки в таблицу, а также в каждой строке кнопку для её удаления:

```
<!DOCTYPE html>
 <html>
  <head>
   <meta charset="UTF-8">
   <title>JavaScript: примеры работы с DOM</title>
   <script>
   function deleteRow(row)
   {
    document.getElementById('tbl').deleteRow(row.rowIndex);
   }

   function addRow()
   {
          var row = document.getElementById('tbl').insertRow(0);
    var cell1=row.insertCell(0);
    var cell2=row.insertCell(1);
          cell1.className='bordered';
          cell2.className='bordered';

          var txt = document.createElement('input');
          txt.type = 'text';
          txt.className = 'bordered';
          cell1.appendChild(txt);

    var remove_button = document.createElement('button');
          remove_button.innerHTML = 'Удалить';
          remove_button.onclick=function(){deleteRow(row)};
    cell2.appendChild(remove_button);
   }
   </script>
   <style>
    .bordered {border: 1px solid black;}
   </style>
  </head>
 <body>
  <button onclick='addRow()'>Добавить строку</button><br>
  <table id='tbl' class='bordered'>
  </table>
 </body>
</html>
```

См. пример в файле
01_js_samples_misc\14_examples_03_table.html

ОТЛОЖЕННОЕ ВЫПОЛНЕНИЕ ФУНКЦИЙ В JAVASCRIPT

Общие сведения

Отложенное выполнение функций в JavaScript может использовать по множеству причин:

- В текущий момент ещё не готовы объекты, с которыми должна работать функция.
- Ещё просто не подошло время выполнения функции.
- Нужно дождаться завершения некоторых операций.
- И т.д. ☺

См. очень хороший и подробный материал:
http://javascript.ru/tutorial/events/timing

Отложенное выполнение с setTimeout и setInterval

Две основных функции для отложенного выполнения кода – это:

- timeoutID=setTimeout(expression, msec)
- timeoutID=setInterval(expression, msec)

Для отмены их действия вызываются:

- clearTimeout(timeoutID)
- clearInterval(timeoutID)

ОЧЕНЬ ВАЖНО! setTimeout() предназначена для ОДНОКРАТНОГО выполнения кода через указанное время, а setInterval() – для многократного (до тех пор, пока не будет выполнен вызов clearInterval()).

Отложенное выполнение с setTimeout и setInterval

Ещё одним способом отложенного выполнения кода является использование обработчиков событий. Самый яркий пример – обработчик onload():

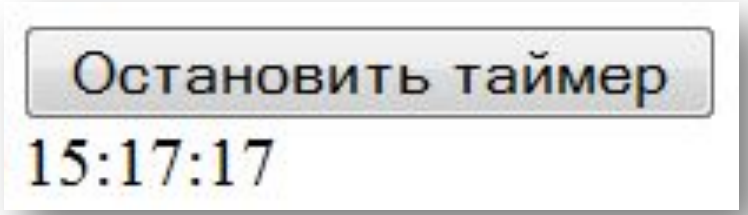
```
<body onload="addHandlers()">
```

Такое решение позволяет произвести выполнение некоторого кода в момент, когда всё тело документа уже загружено и, соответственно, элементы внутри тела уже существуют.

Рассмотрим все три способа в одном примере: создадим таймер, который запускается через пять секунд после загрузки документа и работает до момента отключения.

Отложенное выполнение: setTimeout, setInterval, onload

```
<!DOCTYPE html>
 <html>
  <head>
   <meta charset="UTF-8">
   <title>JavaScript: отложенное выполнение функции</title>
   <script>
   function prepareTimer()
   {
    setTimeout(startTimer, 5000);
   }

   function startTimer()
   {
         document.getElementById('timer').style.display = 'block';
         showTimer();
         interval = setInterval(showTimer, 1000); // НЕ var interval !!!
   }

   function showTimer()
   {
    var dt = new Date();
         hours = ((h = dt.getHours())<10) ? '0'+h : h;
         minutes = ((m = dt.getMinutes())<10) ? '0'+m : m;
         seconds = ((s = dt.getSeconds())<10) ? '0'+s : s;
         document.getElementById('hour').innerHTML = hours;
         document.getElementById('minute').innerHTML = minutes;
         document.getElementById('second').innerHTML = seconds;
   }

   function stopTimer()
   {
    document.getElementById('timer').style.display = 'none';
         clearInterval(interval);
   }

   </script>
   <style>
    .bordered {border: 1px solid black;}
   </style>
  </head>
 <body onload='prepareTimer()'>
  <div id='timer' style='display: none'>
   <button onclick='stopTimer()'>Остановить таймер</button><br>
    <span id='hour'></span>:<span id='minute'></span>:<span id='second'></span>
   </div>
 </body>
</html>
```



См. пример в файле
01_js_samples_misc\15_timer.html

РАБОТА С XML / JSON В JAVASCRIPT

Общие сведения

Сейчас мы посмотрим на то, как его можно использовать в JavaScript. А ещё мы посмотрим на…

JSON (JavaScript Object Notation) – специальный текстовый формат обмена данными, основанный на JavaScript.

Зачем нужны эти форматы?

В JavaScript использование XML или JSON связано, как правило, с обменом данными с сервером с использованием XMLHttpRequest. Какой из форматов выбирать – в большинстве случаев дело вкуса, но мы рассмотрим использование обоих вариантов на примере.

Итак: сервер (код на PHP) генерирует таблицу со случайным количеством рядов и строк, передаёт её скрипту на JavaScript в обоих форматах, а тот, в свою очередь, «парсит» данные и обновляет содержимое страницы.

Пример использования XML/JSON в JavaScript и PHP

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <title>JavaScript: XML и JSON</title>
  <script>
  function loadXML()
  {
   if (typeof xmlhttp == 'undefined')
   {
    xmlhttp=new XMLHttpRequest();
   }
   xmlhttp.onload = updateWithXML;
   xmlhttp.open("GET", "http://127.0.0.1/xml_and_json.php?xml=true", false);
   xmlhttp.send();
  }

  function loadJSON()
  {
   if (typeof xmlhttp == 'undefined')
   {
    xmlhttp=new XMLHttpRequest();
   }
   xmlhttp.onload = updateWithJSON;
   xmlhttp.open("GET", "http://127.0.0.1/xml_and_json.php", false);
   xmlhttp.send();
  }

  function updateWithXML()
  {
   var tbl = document.getElementById('xml_table');
   if ((typeof tbl != 'undefined')&&(tbl != null))
   {
    tbl.parentNode.removeChild(tbl);
   }

   tbl = document.createElement('table');
   tbl.id = 'xml_table';
   tbl.className = 'bordered';

   var xmlDoc=xmlhttp.responseXML;
   var rows = xmlDoc.getElementsByTagName('tr');
   for (var i=0; i<rows.length; i++)
   {
    var row = rows[i];
    var table_row = tbl.insertRow(-1);
    for (var j=0; j<row.childNodes.length; j++)
    {
     var cell = row.childNodes[j];
     var table_cell = table_row.insertCell(-1);
     var text = cell.childNodes[0].textContent;
     table_cell.className = 'bordered';
     table_cell.innerHTML = text;
    }
   }
   document.getElementById('xml_table_container').appendChild(tbl);
  }


 function updateWithJSON()
  {
   var tbl = document.getElementById('json_table');
   if ((typeof tbl != 'undefined')&&(tbl != null))
   {
    tbl.parentNode.removeChild(tbl);
   }

   tbl = document.createElement('table');
   tbl.id = 'json_table';
   tbl.className = 'bordered';

   var jsonDoc=JSON.parse(xmlhttp.responseText);
   var rows = jsonDoc.tr;

   // В JSON-формате вместо массива получается строка, если элемент один.
   // Это первый вариант "хака" по обхождению такого поведения.
   if (typeof rows.length == 'undefined')
   {
    rows[0] = new Array();
    rows[0].td = rows.td;
    rows.length = 1;
    rows.td = null;
   }

   for (var i=0; i<rows.length; i++)
   {
    var row = rows[i];
    var table_row = tbl.insertRow(-1);

    // В JSON-формате вместо массива получается строка, если элемент один.
    // Это второй вариант "хака" по обхождению такого поведения.
    if (typeof row.td == 'string')
    {
     var save_value = row.td;
     row.td = new Array();
     row.td[0] = save_value;
    }

    for (var j=0; j<row.td.length; j++)
    {
     var cell_and_text = row.td[j];

     var table_cell = table_row.insertCell(-1);
     table_cell.className = 'bordered';
     table_cell.innerHTML = cell_and_text;
    }
   }
   document.getElementById('json_table_container').appendChild(tbl);
  }

  </script>
  <style>
   .bordered {border: 1px solid black;}
  </style>
 </head>
 <body>
  <button onclick='loadXML()'>Обновить таблицу на основе XML</button> <button onclick='loadJSON()'>Обновить таблицу на основе JSON</button><br>
  <div id='xml_table_container'></div>
  <div id='json_table_container'></div>
 </body>
</html>
```

```
<?php

$tree = new SimpleXMLElement('<table/>');

$rows = mt_rand(1, 5);
$cols = mt_rand(1, 5);

for ($i=0; $i<$rows; $i++)
{
    $row = $tree->addChild('tr');
    for ($j=0; $j<$cols; $j++)
    {
        $row->addChild('td', mt_rand(1, 100));
    }
}

if (isset($_GET['xml']))
{
    header("Content-type: application/xml");
    echo $tree->asXML();
}
else
{
    header("Content-type: text/plain");
    echo json_encode($tree);
}

?>
```

Обновить таблицу на основе XML   Обновить таблицу на основе JSON

87	41	68	100	30
2	68	38	72	85
27	6	4	34	59
100	44	2	51	50

5	41	14	68	45
14	60	93	10	4
60	42	71	55	77
39	27	15	81	13

См. примеры в папке
02_js_samples_xml_json

ОБРАБОТКА ОШИБОЧНЫХ СИТУАЦИЙ И ИСКЛЮЧЕНИЙ В JAVASCRIPT

Обработка ошибочных ситуаций

Для обработки ошибочных ситуаций в JavaScript в отличие от PHP существует, фактически, один универсальный способ – обработка исключений.

Мы здесь рассмотрим примеры обработки и порождения исключений.

Пример обработки исключений с try … catch … finally

Итак, представим, что мы пытаемся установить значение несуществующего свойства несуществующего объекта.

```
try
{
 non_existing_object.non_existing_property = 5;
}
catch(e)
{
 console.log(e);
}
finally
{
 console.log('Finally...');
}
```

```
ReferenceError: non_existing_object is not defined
Finally...
```

Виды исключений

Фактически, это и всё, что нам доступно. Даже т.н. «условные исключения» (реакция на исключения разных типов) пока ещё не стандартизированы. Сам JavaScript порождает исключения следующих типов (см. объект Error: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error):

- EvalError – возникает в процессе работы eval().
- RangeError – возникает при нарушении диапазонов (массив отрицательной длины, Number.to*() на числах вне допустимого диапазона, не числах и т.д.)
- ReferenceError – аозникает при обращении к несуществующему объекту или элементу.
- SyntaxError – возникает в процессе анализа синтаксиса аргумента eval().
- TypeError – возникает в случае, когда операция выполняется над элементом недопустимого типа.
- URIError – возникает, если в функции encodeURI() или decodeURI() переданы неверные параметры.

Порождение собственных исключений

Мы можем порождать исключения любого из только что перечисленных типов или свои собственные:

```
function test(x)
{
 if (x<0)
 {
  throw new RangeError('x mut be >= 0'); // Ещё можно так: throw String('x mut be >= 0');
 }

 if (typeof x == 'string')
 {
  // В создаваемом объекте вы можете определять ЛЮБЫЕ свойства и методы.
  throw {
    name:        'Type Error',
    level:       'Script stop',
    message:     'x must be a number',
    htmlMessage: '<a href="no_manual_sorry.html">No manual available</a>',
    toString:    function(){return this.name + ": " + this.message}
       }
 }

}

try
{
 test(-1);   // Закомментируйте эту строку, чтобы увидеть реакцию на второй случай.
 test('abc');
}
catch (e)
{
 console.log(e);
}
```

ООП В JAVASCRIPT

Общие сведения об ООП

ООП – парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

Настоятельно рекомендуется вдумчиво прочитать:
http://javascript.ru/tutorial/object/inheritance
http://javascript.ru/tutorial/object/thiskeyword

См. примеры в папке
03_js_samples_oop

ООП в JavaScript – что не так?

Итак, в JavaScript реализовано не классическое ООП (с классами), а т.н. «прототипное наследование».

Логика ООП состоит не в создании классов и объектов, а в… создании объектов СРАЗУ, «из ничего».

Объект можно создать даже вот так:

```
obj = {a: 'Yes', b: 'No', sqr : function(x){return x*x}};
console.log(obj);
console.log(obj.sqr(5));
```

```
Object { a="Yes", b="No", sqr=function()}
25
```

ООП в JavaScript, создание объектов

Чуть более классический вариант – создание функции (напоминаем, в JavaScript функция – тоже объект) с использованием ключевого слова new.

```
function Person(name, profession)
{
 this.name = name;
 this.profession = profession;
 this.make_some_work = function(){return 'Done';};
}

var Pupkin = new Person('Pupkin', 'worker');
console.log(Pupkin.make_some_work());
```

```
Done
```

Как можно понять из предыдущего примера, мы не можем создать «классический класс», не можем использовать модификаторы прав доступа (public, protected, private), не можем создавать «нормальные» статические методы и т.д.

В общем случае это – не проблема, т.к. для простых скриптов нет необходимости в сложных решениях, но мы всё же рассмотрим, как организуется наследование.

Итак, чтобы «унаследовать» «класс» от объекта (не класс от класса!!!) надо сделать так:

```
// Объявляем первый "класс".
function Person(name, profession)
{
 this.name = name;
 this.profession_by_education = profession;
 this.make_some_work = function(){return 'Done';};
}

// Объявляем второй "класс".
function Professional(name, profession)
{
 this.name = name;
 this.profession_in_real_life = profession;
 this.make_good_work = function(){return 'Well done!';};
}

// Создаём первый экземпляр.
var pupkin = new Person('Pupkin', 'worker');

// Наследуем второй "класс" от _ЭКЗЕМПЛЯРА_ первого.
Professional.prototype = pupkin;

// Создаём второй экземпляр.
var smith = new Professional('Smith', 'programmer');

console.log(pupkin.make_some_work());          // Done
console.log(smith.make_good_work());           // Well done!
console.log(smith.make_some_work());           // Done
console.log(smith.profession_by_education);    // worker
pupkin.profession_by_education = 'writer';
console.log(smith.profession_by_education);    // writer
```

```
console.log(typeof pupkin); // object
console.log(typeof smith);  // object
console.log(pupkin instanceof Person);  // true
console.log(smith instanceof Person);   // true
console.log(pupkin instanceof Professional);  // false
console.log(smith instanceof Professional);   // true
```

Ещё один способ наследования – с использованием Object.create():

```
function Creature(name, type)
{
    this.name = name;
    this.type = type || 'unknown type';
}

Creature.prototype.getName = function()
{
    return this.name;
}

Creature.prototype.getType = function()
{
    return this.type;
}


function Human(name)
{
    Creature.call(this, name, 'human');
    this.name = name || 'No name';
}

Human.prototype = Object.create(Creature.prototype);

var pupkin = new Human('Pupkin');
var smith = new Human('Smith');

console.log(pupkin.getName()); // Pupkin
console.log(pupkin.getType()); // human
console.log(smith.getName());  // Smith
console.log(smith.getType());  // human
console.log(smith instanceof Human);    // true
console.log(smith instanceof Creature); // true
```

Q: Как в JavaScript объявлять конструктор?

A: «Никак». Это просто код внутри «объекта-функции», работающий с this:

```
function Creature(name, type)
{
    this.name = name;
    this.type = type || 'unknown type';
}

alien = new Creature('Alien');
console.log(alien);

strange_alien = Creature('Strange alien');
console.log(strange_alien);
console.log(window.name);
```

Без ключевого слова new указатель this будет указывать не на экземпляр «функции-класса», а на глобальный объект window.

```
Creature { name="Alien", type="unknown type"}
undefined
Strange alien
```

Q: Как защититься от ошибок создания экземпляров без new?

A: Вот так:

```
function Creature(name, type)
{
    // Если this -- не наш «класс»,
    // создаём экземпляр «вручную» и возвращаем его.
    if (!(this instanceof Creature))
    {
     return new Creature(name, type);
    }

     this.name = name;
     this.type = type || 'unknown type';
}
```

Q: Да что вообще с этим this такое творится?!

A: Если очень кратко, то логика такая:

```
// Просто вызов функции
test_function('text'); // this == window

// Обращение к конструктору
var x = new TestFunction('text'); // this == новый объект

// Вызов метода
obj.test_function('text'); // this == obj

// Вызов apply или call
test_function.call(obj, params); // this == obj
test_function.apply(obj, params); // this == obj

// Вызов обработчика
<span onclick="span_click(this)"> // this == span
```

Q: А как всё же сделать «классический ООП» -- со статическими методами и свойствами, константами классов, синглтонами и прочим-разным-привычным?

A: Не надо. Несмотря на то, что ответы на любую часть этого вопроса легко выгугливаются (даже реализация примесей (mixins)), всё же JavaScript используется «очень по-своему», и не надо пытаться «написать PHP/C#/Java на JavaScript».

Q: Я прочитал всё это трижды и всё равно ничего не понял. Что делать?

A: Читать дальше. Вот здесь http://habrahabr.ru/post/131714/ есть ещё одно объяснение про «ООП в JS». Возможно, оно покажется более простым.

Задание для закрепления материала

Реализуйте с помощью JavaScript классическую задачу «о геометрических фигурах»: есть «абстрактная фигура», от которой наследуется эллипс, прямоугольник и треугольник. От эллипса наследуется круг, от прямоугольника – квадрат, от треугольника – равносторонний и прямоугольный треугольники. Для всех фигур нужно вычислять периметр и площадь. Каждая фигура задаётся координатами вершин или координатами центра и радиусом (радиусами).

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ В JAVASCRIPT

Общие сведения

Регулярные выражения вы изучали раньше.

JavaScript ***почти*** следует логике PCRE в своих регулярных выражениях, так что подавляющее большинство решений из PHP будет работать и в JavaScript.

Чтобы понять нюансы достаточно глубоко, смотрите документацию по JavaScript, а мы рассмотрим основное и вкратце.

Объект RegExp

Объект RegExp, отвечающий за работу с регулярными выражениями, можно создать так:

```
// Классический способ:
var re = new RegExp("регулярное_выражение"[, модификаторы]);

// Perl-style способ:
var re = /регулярное_выражение/модификаторы;
```

```
// Классический способ:
var re_classic = new RegExp('/[a-z]+\\w/', 'i');

// Perl-style способ:
var re_perl = /[a-z]+\w/i;
```

Использование объекта RegExp

У объекта RegExp есть следующие методы:

- exec – выполнить поиск и вернуть массив с результатами и дополнительной информацией.
- test – проверить на совпадение и вернуть true или false.

У объекта String есть следующие методы по работе с регулярными выражениями:

- match – выполнить поиск и вернуть массив с результатами и дополнительной информацией (или null, если ничего не найдено).
- search – проверить на совпадение и вернуть индекс совпадения или -1, если совпадения нет.
- replace – произвести поиск и замену.
- split – разрезать строку на массив подстрок, используя в качестве разделителя регулярное выражение или фиксированную строку.

Использование RegExp.exec()

Рассмотрим пример использования RegExp.exec()

```
var text = 'Это текст с кодами товаров: ABC-1000,
def-2000.';
var re = new RegExp('([a-z]{3})-(\\d{4})', 'ig');
var result;

while ((result = re.exec(text)) !== null)
{
 console.log(result);
}
```

```
["ABC-1000", "ABC", "1000"]
0     "ABC-1000"
1     "ABC"
2     "1000"
index 28
input "Это текст с кодами товаров: ABC-1000, def-2000."

["def-2000", "def", "2000"]
0     "def-2000"
1     "def"
2     "2000"
index 38
input "Это текст с кодами товаров: ABC-1000, def-2000."
```

Использование RegExp.test()

Рассмотрим пример использования RegExp.test()

```
var text = 'Это текст с кодами товаров: ABC-1000,
def-2000.';
var re_4d = new RegExp('\\d{4}', '');
var re_5d = new RegExp('\\d{5}', '');

console.log(re_4d.test(text)); // true
console.log(re_5d.test(text)); // false
```

Использование Sring.match()

Рассмотрим пример использования Sring.match()

```
var text = 'Это текст с кодами товаров: ABC-1000,
def-2000.';
var re_4d = new RegExp('\\d{4}', '');
var re_5d = new RegExp('\\d{5}', '');

console.log(text.match(re_4d)); // См. ниже.
console.log(text.match(re_5d)); // null
```

```
["1000"]
0   "1000"
index  32
input  "Это текст с кодами товаров: ABC-1000, def-2000."
```

Использование Sring.search()

Рассмотрим пример использования Sring.search()

```
var text = 'Это текст с кодами товаров: ABC-1000,
def-2000.';
var re_4d = new RegExp('\\d{4}', '');
var re_5d = new RegExp('\\d{5}', '');

console.log(text.search(re_4d)); // 32
console.log(text.search(re_5d)); // -1
```

Использование Sring.replace()

Рассмотрим пример использования Sring.replace()

```
var text = 'Это текст с кодами товаров: ABC-1000,
def-2000.';
var re_4d = new RegExp('\\d{4}', 'g');

console.log(text.replace(re_4d, '?'));

// Это текст с кодами товаров: ABC-?, def-?.

// Без модификатора g (global) будет
// заменено только первое вхождение.
```

Использование Sring.split()

Рассмотрим пример использования Sring.split()

```
var text = 'Это текст с кодами товаров: ABC-1000,
def-2000.';
var re_4d = new RegExp('\\d{4}', 'g');

console.log(text.split(re_4d));

// ["Это текст с кодами товаров: ABC-", ", def-", "."]
```

Задание для закрепления материала

Задачи:

1) Определить, содержит ли текст хотя бы одно трёхзначное число.

2) Показать все двухзначные числа, стоящие строго в начале или строго в конце строки.

3) Для дат в формате [Д]Д.[М]М.[ГГ]ГГ выделить день, месяц, год.

4) Выделить в тексте дублирующиеся слова.

5) Определить числа с запятой или пробелом в качестве разделителя разрядов.

ИСПОЛЬЗОВАНИЕ XMLHTTPREQUEST

Общие сведения

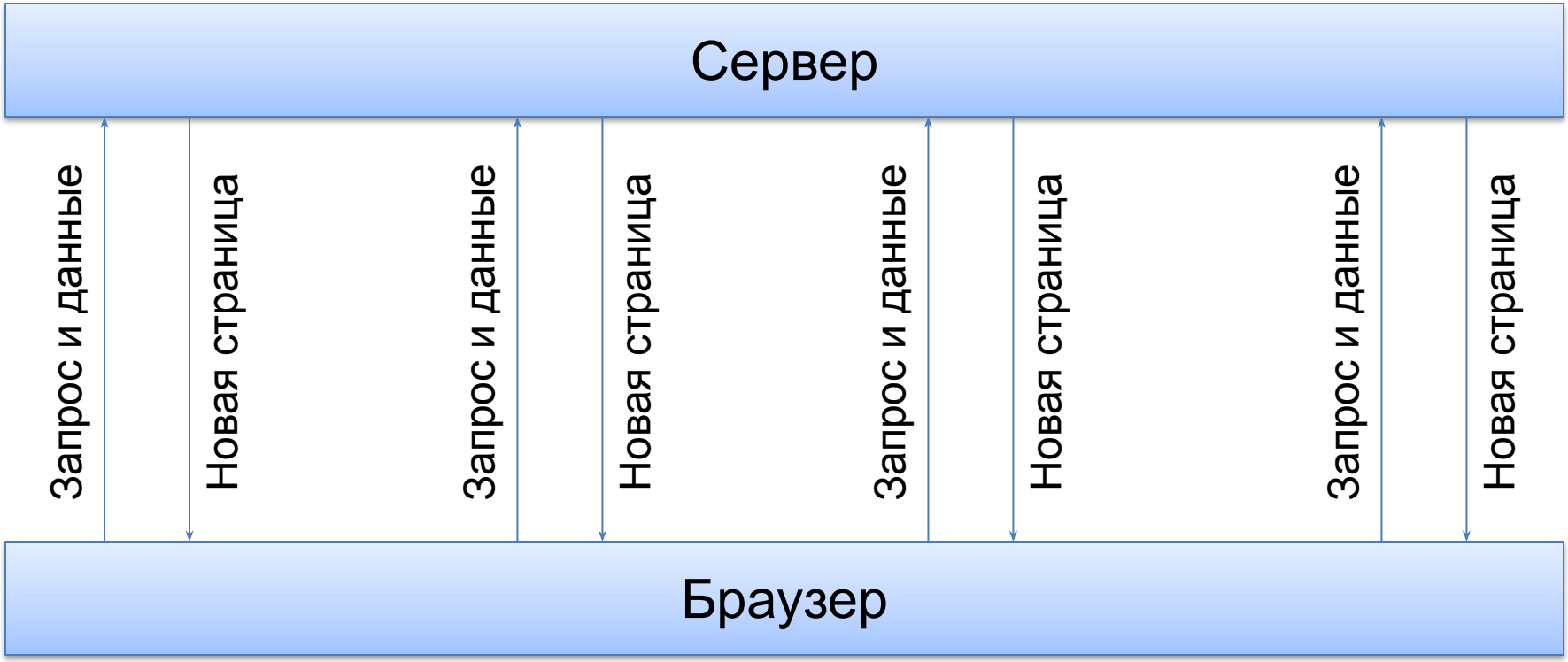
XMLHttpRequest – специальное решение (фактически, API), позволяющее выполнять т.н. фоновые запросы к веб-серверу, не обновляя всю страницу целиком.

XMLHttpRequest лежит в основе AJAX (Asynchronous Javascript and XML, асинхронный JavaScript и XML) – набора решений, позволяющих строить максимально интерактивные веб-приложения.

См. примеры в папке
04_js_samples_ajax

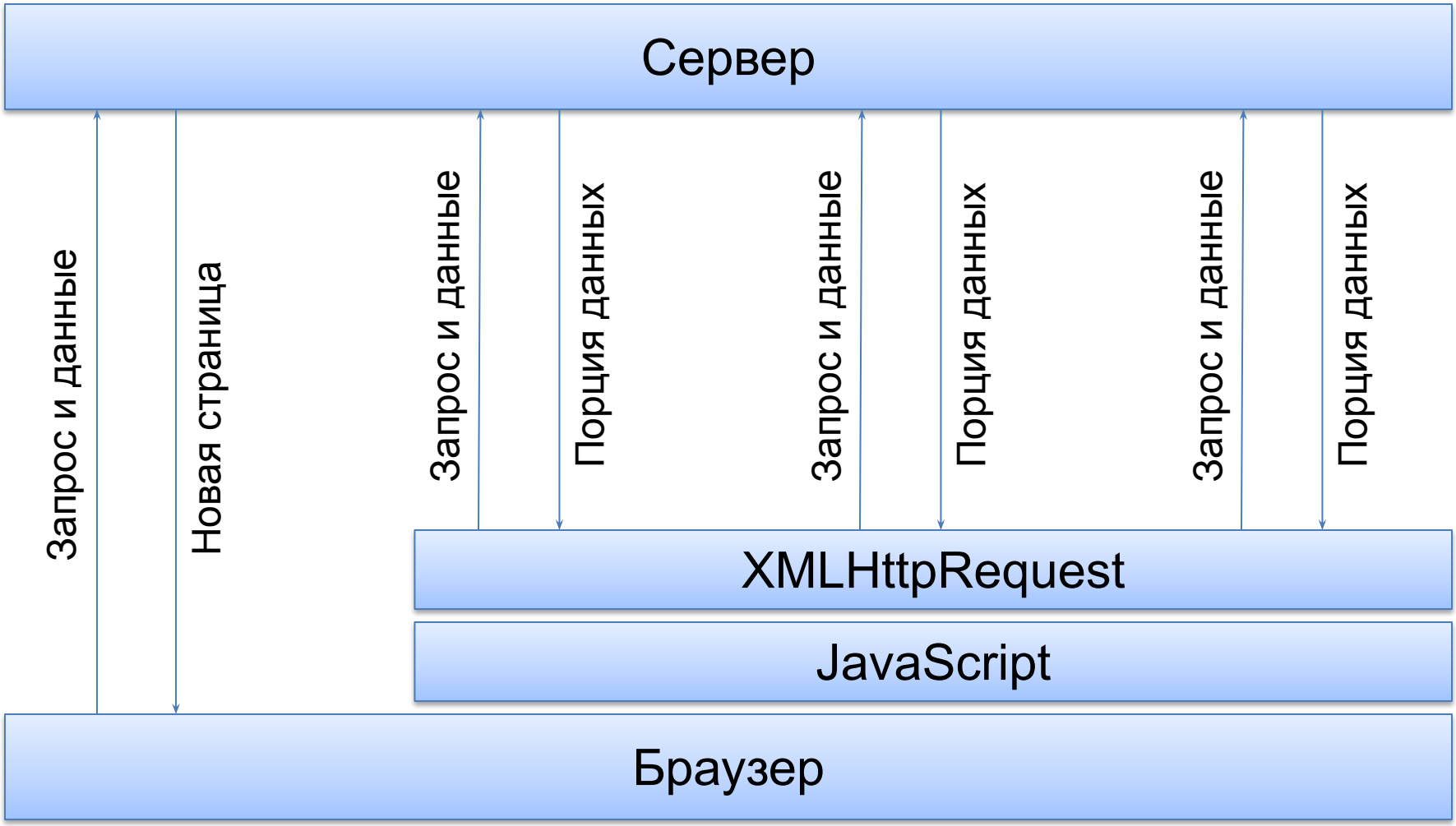
Как это работает

В классическом варианте любые изменения страницы, требующие информации с сервера, приводят к повторной загрузке всей страницы целиком:



Как это работает

В случае использования AJAX картина становится такой:



Как это реализовать

Общий принцип прост, и мы его уже изучили (и даже видели пример в главе, посвящённой XML и JSON):

1) Написать код, вызываемый в результате реакции на какое-то событие (как вариант – по таймеру).

2) Создать экземпляр XMLHttpRequest, выполнить запрос и получить ответ.

3) Обработать ответ и внести правки в страницу (или выполнить какие-то иные действия – JavaScript ведь может многое).

Как это реализовать

С первой и третей частями мы уже хорошо познакомились в соответствующих главах (см. всё, что связано с обработкой событий и управлением DOM).

Потому – сразу переходим ко второй (создать экземпляр XMLHttpRequest, выполнить запрос и получить ответ) и посмотрим, как это сделать максимально универсально и кроссбраузерно.

Итак…

Как это реализовать

Поскольку реализация XMLHttpRequestзависит от браузера, воспользуемся таким решением:

```
function getXHR()
{
    var xmlhttp = false;
    var XMLHttpFactories = [
    function () {return new XMLHttpRequest()},
    function () {return new ActiveXObject("Msxml2.XMLHTTP")},
    function () {return new ActiveXObject("Msxml3.XMLHTTP")},
    function () {return new ActiveXObject("Microsoft.XMLHTTP")}];

    for (var i=0;i<XMLHttpFactories.length;i++) {
        try
        {
            xmlhttp = XMLHttpFactories[i]();
        }
        catch (e)
        {
            continue;
        }
        break;
    }
    return xmlhttp;
}
```

Как это реализовать

Получив экземпляр XMLHttpRequest, следует его «настроить»: указать обработчики изменения состояния. Это можно сделать через два свойства:

- onreadystatechange – указывается функция, которая будет вызываться при любом изменении состояния (крайне не рекомендуется для синхронных запросов; впрочем, и сами синхронные запросы крайне не рекомендуются);
- onload – указывается функция, которая будет вызываться после завершения выполнения запроса.

Как это реализовать

У XMLHttpRequest есть следующие состояния:

- 0 (UNSENT) – метод open() ещё не был вызван.
- 1 (OPENED) – метод send() ещё не был вызван.
- 2 (HEADERS_RECEIVED) – метод send() был вызван, и уже доступен статус ответа и заголовки ответа.
- 3 (LOADING) – идёт загрузка, в свойстве responseText содержится часть полученных данных.
- 4 (DONE) – операция завершена.

Как это реализовать

Метод

```
void open(
    DOMString method,
    DOMString url,
    optional boolean async,
    optional DOMString user,
    optional DOMString password
);
```

инициализирует запрос. Здесь особо стоит отметить параметр async, который в общем случае должен быть true, т.е. в противном случае (при долгих запросах-ответах) в зависимости от браузера можно получить низкую производительность или вовсе «замирание»/«подвисание» приложения или даже всего браузера.

Как это реализовать

Метод

```
void send();
```

отправляет запрос и либо ждёт его завершения (синхронный запрос), либо сразу же завершается (асинхронный запрос).

Из других полезных методов стоит отметить:

- abort() – отмена всей операции.
- getResponseHeader(String header) – возвращает указанный заголовок ответа или null, если заголовка нет, или он пока не получен.
- setRequestHeader(String header, String value) – позволяет указать заголовок HTTP-запроса.
- overrideMimeType(String mimetype) – позволяет принудительно указать MIME-тип документа (полезно, например, если нужно будет проанализировать XML-ответ, но сервер не выставил правильный заголовок).

Как это реализовать

Много полезной информации можно получить из свойств:

- readyState – текущее состояние операции (см. onreadystatechange).
- status – код HTTP-ответа сервера (200, 404 и т.д.)
- statusText – «текст кода HTTP-ответа», т.е. "OK" и т.д.
- response – «тело ответа» (см. документацию, там много нюансов).
- responseText – текст ответа сервера (как правило, или используется сам по себе для вставки в страницу, или содержит JSON).
- responseXML – XML-вид ответа сервера (при условии, что сервер ДЕЙСТВИТЕЛЬНО передал в ответ XML-данные).
- timeout – позволяет устанавливать таймаут в тысячных долях секунд (по умолчанию == 0, т.е. время выполнения запроса не ограничено).
- ontimeout – позволяет указать функцию, которая будет вызвана при наступлении таймаута.

Демонстрация на примере

Теперь соберём всё это вместе и посмотрим, как работает синхронный и асинхронный запрос:

```
<!DOCTYPE html>
 <html>
  <head>
   <meta charset="UTF-8">
   <title>JavaScript: AJAX</title>
   <script>
   function getXHR()
   {
    var xmlhttp = false;
    var XMLHttpFactories = [
    function () {return new XMLHttpRequest()},
    function () {return new ActiveXObject("Msxml2.XMLHTTP")},
    function () {return new ActiveXObject("Msxml3.XMLHTTP")},
    function () {return new ActiveXObject("Microsoft.XMLHTTP")}];

    for (var i=0;i<XMLHttpFactories.length;i++) {
        try
        {
            xmlhttp = XMLHttpFactories[i]();
        }
        catch (e)
        {
            continue;
        }
        break;
    }
    return xmlhttp;
   }

   function sampleSync()
   {
                   var xhr = getXHR();
                   xhr.onreadystatechange = createWatcher(xhr, 'Sync: ');

                   // ВАЖНО! AJAX-запросы НЕ работают "кросс-доменно"!!!!!
                   // Т.е. нельзя "просто открыть эту HTML'ку в браузере, и всё заработает". Надо открывать её как http://127.0.0.1/ajax.html
                   // При попытке выполнить кросс-доменный запрос, вы будете получать ошибку NS_ERROR_FAILURE, и запрос будет остановлен.
                   // Решений два:
                   // 1) Сервер при передаче HTML, в котором вызывается ваш запрос, должен вернуть заголовок header('Access-Control-Allow-Origin: *'); (это может всё равно не работать).
                   // 2) Создавать т.н. server-side-proxy -- скрипт, который будет перенаправлять ваши запросы на другие сайты и возвращать ответы.
                   xhr.open('get', 'http://127.0.0.1', false);
                   xhr.send();
   }

   function sampleAsync()
   {
                   var xhr = getXHR();
                   xhr.onreadystatechange = createWatcher(xhr, 'Async: ');

                   // ВАЖНО! AJAX-запросы НЕ работают "кросс-доменно"!!!!!
                   // Т.е. нельзя "просто открыть эту HTML'ку в браузере, и всё заработает". Надо открывать её как http://127.0.0.1/ajax.html
                   // При попытке выполнить кросс-доменный запрос, вы будете получать ошибку NS_ERROR_FAILURE, и запрос будет остановлен.
                   // Решений два:
                   // 1) Сервер при передаче HTML, в котором вызывается ваш запрос, должен вернуть заголовок header('Access-Control-Allow-Origin: *'); (это может всё равно не работать).
                   // 2) Создавать т.н. server-side-proxy -- скрипт, который будет перенаправлять ваши запросы на другие сайты и возвращать ответы.
                   xhr.open('get', 'http://127.0.0.1', true);
                   xhr.send();
   }

   function createWatcher(xhr, msg)
   {
    return function()
                   {
     document.getElementById('log').innerHTML += msg + xhr.readyState + '<br>';
                    if (xhr.readyState == 4)
                    {
                     var log = document.getElementById('log');
                     log.innerHTML += msg + 'Finished' + '<br>';
                     log.innerHTML += msg + 'AllResponseHeaders: ' + xhr.getAllResponseHeaders() + '<br>';
                     log.innerHTML += msg + 'Status: ' + xhr.status + '<br>';
                     log.innerHTML += msg + 'Status: ' + xhr.statusText + '<br>';
                    }
                   }
   }

   </script>
  </head>
  <body>
   <button onclick='sampleSync()'>Синхронный запрос</button> <button onclick='sampleAsync()'>Асинхронный запрос</button><br>
   <div id='log'></div>
  </body>
 </html>
```

КРОССДОМЕННЫЕ ЗАПРОСЫ

Суть проблемы

Браузеры по соображениям безопасности ограничивают возможности скриптов по взаимодействию с другими доменами, т.е. учитывают комбинацию схемы, имени хоста и порта для того, чтобы определить, откуда был загружен скрипт, и разрешить ему выполнять только запросы по тому же адресу.

Почитать подробнее:
http://en.wikipedia.org/wiki/Same-origin_policy

Решение проблемы

Однако иногда есть объективная необходимость выполнять такие (т.н. «кроссдоменные») запросы. Сделать это помогает…

JSONP (JSON with padding) – расширение формата JSON, предоставляющее возможность запросить данные с «чужого домена».

Почитать подробнее:
http://en.wikipedia.org/wiki/JSONP

Как это работает

Сначала рассмотрим серверную часть. Здесь важны две вещи:

1. Верные заголовки HTTP-ответа.
2. Правильное формирование контента.

Рассмотрим на примере выполнения кроссдоменного запроса (будем посылать запрос на виртуальную машину)…

См. примеры в папке **05_js_samples_jsonp**

Как это работает

Серверная часть:

```
<?php

if (isset($_GET['callback']))
{
    // Just for security reasons.
    $callback = preg_replace('/[^a-z0-9$_]/si', '', $_GET['callback']);

    // Set headers.
    header('Access-Control-Allow-Origin: *');
    header('Content-Type: application/javascript;charset=UTF-8');

    // Collect some useful data.
    $response = array('a' => '999');

    // Create proper JSONP format response and send it.
    $response = $callback.'('.json_encode($response).')';
    echo $response;
}

?>
```

Как это работает

На клиентской части нужно сделать две вещи:

1. Подготовить функцию, имя которой передано на сервер в качестве параметра запроса.
2. Динамически создать элемент <script>.

Идея состоит в том, что вы не ВЫПОЛНЯЕТЕ запрос к «чужому домену», а создаёте запрос, который загружает «скрипт» (JSON-данные, обёрнутые в вызов функции) с нужного вам домена. Как только этот «скрипт» загрузится, функция будет вызвана. А у вас в документе она уже есть – и всё.

Как это работает

Клиентская часть:

```
<!DOCTYPE html>
 <html>
  <head>
   <meta charset="UTF-8">
   <title>JavaScript: JSONP</title>
   <script>

   function getCrossDomainData()
   {
        var script = document.createElement("script");
        script.src = 'http://192.168.56.101/jsonp.php?callback=jsonptest';
        document.body.appendChild(script);
   }

   function jsonptest(data)
   {
        console.log(data);
        document.getElementById('log').innerHTML += data.a;
   }
   </script>
  </head>
 <body>
  <button onclick='getCrossDomainData()'>Запросить данные</button><br>
  <div id='log'></div>
 </body>
</html>
```

Как это работает

Итак, важное и ещё раз:

- Это – НЕ XMLHttpRequest! Да, это динамическая подгрузка данных, но она работает БЕЗ XMLHttpRequest.
- Алгоритм такой:
  - Создать на клиенте функцию.
  - Добавить на клиенте в тело документа скрипт, …
  - Которому указать в SRC нужный адрес на «чужом» домене.
  - Когда скрипт загрузится, выполнится ваша клиентская callback-функция и получит в качестве аргумента присланные данные.

С JavaScript – всё!

О ещё некоторых небольших особенностях работы с JavaScript мы поговорим в следующем разделе, посвящённом jQuery.

СПАСИБО ЗА ВНИМАНИЕ!

ВОПРОСЫ?

Основы JavaScript

Author: **Svyatoslav Kulikov**
Training And Education Manager
svyatoslav_kulikov@epam.com