

## 13.2. Основы JavaScript

**Цель:** Изучить основы JavaScript, ознакомиться с его синтаксисом. Научиться создавать функции, массивы, объекты и свойства объектов, простые программы.

### 13.2.1. Теоретические сведения

#### 13.2.1.1. Синтаксис

Шпаргалка с правилами синтаксиса:

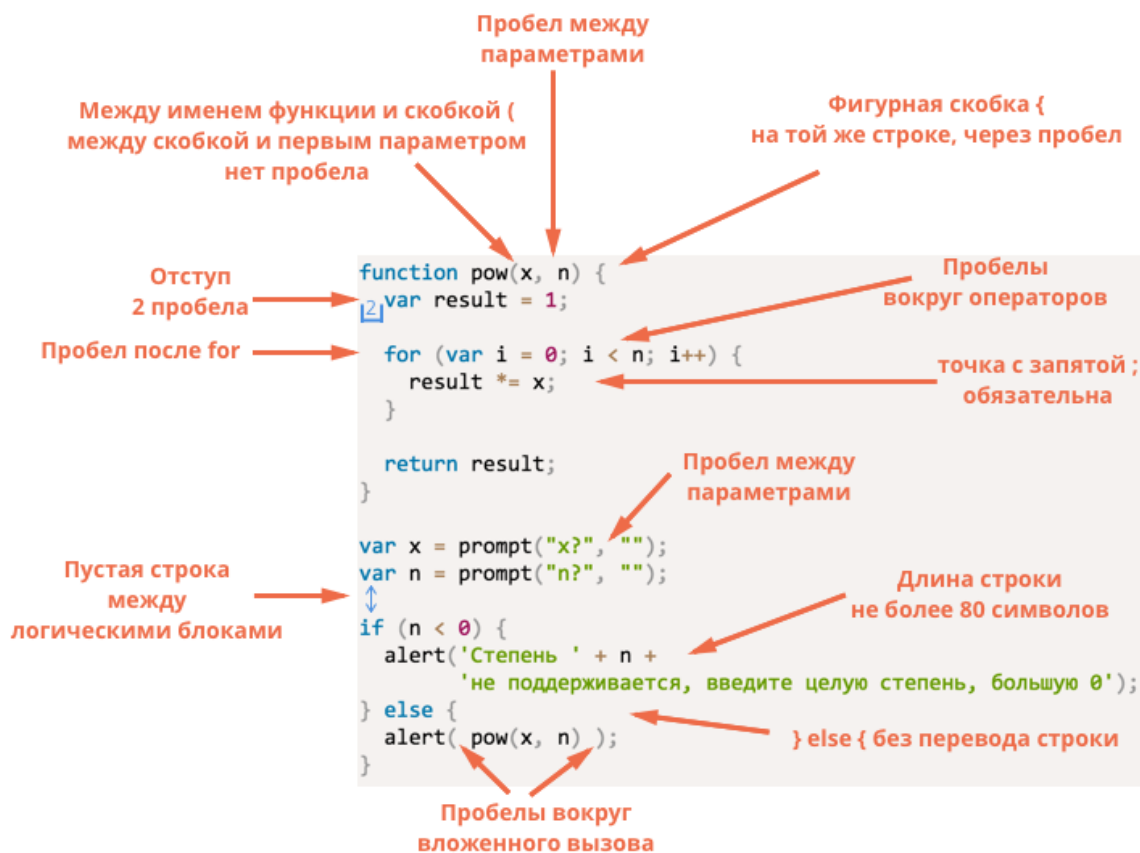


Рис. 13.1.

Не всё здесь однозначно, так что разберём эти правила подробнее.

Фигурные скобки пишутся на той же строке, так называемый «египетский» стиль. Перед скобкой – пробел.



**Плохо!**

**Фигурные скобки не имеют смысла**

```
if (n < 0) {alert('Степень ' + n + ' не поддерживается');}
```



**В одну строку без скобок - приемлемо, если эта строка короткая**

```
if (n < 0) alert('Степень ' + n + ' не поддерживается');
```



**Самый лучший вариант**

```
if (n < 0) {
    alert('Степень ' + n + ' не поддерживается');
}
```

Рис 13.2.

Есть языки, в которых точка с запятой не обязательна, и её там никто не ставит. В JavaScript перевод строки её заменяет, но лишь частично, поэтому лучше её ставить.

#### 13.2.1.2. Функции = Комментарии

Функции должны быть небольшими. Если функция большая – желательно разбить её на несколько. Вызов отдельной небольшой функции не только легче отлаживать и тестировать. Сравните, например, две функции `showPrimes(n)` для вывода простых чисел до `n`. Первый вариант использует метку:

```
function showPrimes(n) {
  nextPrime: for (var i = 2; i < n; i++) {

    for (var j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }
    alert(i); // простое
  }
}
```

Второй вариант – дополнительную функцию `isPrime(n)` для проверки на простоту:

```
function showPrimes(n) {
  for (var i = 2; i < n; i++) {
    if (!isPrime(i)) continue;
    alert(i); // простое
  }
}

function isPrime(n) {
  for (var i = 2; i < n; i++) {
    if (n % i == 0) return false;
  }
  return true;
}
```

Второй вариант проще и понятнее. Вместо участка кода мы видим описание действия, которое там совершается (проверка `isPrime`).

Есть два способа расположить функции, необходимые для выполнения кода.

1. Функции над кодом, который их использует.
2. Сначала код, а функции внизу.

#### 13.2.1.3. Комментарии

В коде нужны комментарии, но должен быть минимум комментариев, которые отвечают на вопрос «что происходит в коде?»

Если у вас образовалась длинная «простыня», то, возможно, стоит разбить её на отдельные функции, и тогда из их названий будет понятно, что делает тот или иной фрагмент.

А какие комментарии полезны и приветствуются?

Какие компоненты есть, какие технологии использованы, поток взаимодействия. О чём и зачем этот скрипт. Эти комментарии особенно нужны, если вы не один, а проект большой.

Для таких комментариев существует синтаксис JSDoc.

```
/**
 * Возвращает x в степени n, только для натуральных n
 *
 * @param {number} x Число для возведения в степень.
 * @param {number} n Показатель степени, натуральное число.
 * @return {number} x в степени n.
 */
function pow(x, n) {

}
```

Один из показателей хорошего разработчика – качество комментариев, которые позволяют эффективно поддерживать код, возвращаться к нему после любой паузы и легко вносить изменения.

#### 13.2.1.4. Переменные

Одним из ключевых моментов при программировании на любом языке являются переменные. И JavaScript не стал исключением из этого правила, и переменные в JavaScript являются основным инструментом при написании скриптов.

Переменные создаются с помощью ключевого слова "var". После этого ключевого слова идёт имя переменной. Серьёзных ограничений на имя переменной нет, но есть определённые правила:

1. Переменная не может состоять только из одних цифр.
2. Переменная не должна содержать пробельных символов.
3. Переменная не может быть ключевым словом (например, нельзя назвать переменную "var").
4. Переменная не может содержать различные спецсимволы (кавычки, апострофы, восклицательный и вопросительный знаки, точка, запятая и прочее), однако, дефис и знак подчёркивания использовать можно.

После определения имени переменной можно её инициализировать, и через знак "=" написать её значение.

```
var x = 5;
```

В этом примере создаётся переменная с именем "x", которой присваивается значение "5".

Теперь создадим ещё одну переменную в JavaScript скрипте, которой присвоим значение переменной "x" + 1.

```
var y = x + 1;
```

В примере выше мы создали переменную "y", у которой значение переменной "x" + 1, то есть в нашем случае - это "6". Разумеется, помимо операции сложения, есть также и операции вычитания (-), умножения (\*) и деления (/). Поэтому Вы можете попрактиковаться, создавая новые переменные в JavaScript и меняя их значения.

Выводить переменные можно, например, используя функцию document.write(), передав в качестве параметра имя переменной:

```
document.write(x);
```

#### 13.2.1.5. Типы переменных

Начнём с самого простого - целого типа. Целый тип создаётся с помощью присвоения переменной целого значения, например, так:

```
var number = -323;
```

Следующий тип переменной - это вещественный тип, или, как принято его называть, double. Так же как и с другими типами, для создания вещественной переменной необходимо инициализировать переменную одним из подобных значений. Вот пример:

```
var number = 32.3291;
```

Обратите внимание, что целая часть от дробной отделяется не "запятой", а "точкой".

Строковый тип - это какая-либо строка. Для примера такая: "string", "732", "My name's Michael". Обратите внимание, что строка задаётся в кавычках. В частности, строка "732" отличается от числа 732. Создаётся строка аналогично другим типам в JavaScript:

```
var str = "some string";
```

Ещё один тип переменной - это булевский. Тип этой переменной может содержать одно из двух значений: true (истина) или false (ложь). Переменная используется в условных операторах:

```
var bool = true;
```

И последний тип - это массив. Самый сложный тип из всех, однако, очень важный и присутствующий почти во всех языках программирования. Это тип, который содержит в себе несколько различных переменных, причём, возможно, что даже разных типов.

Пример вывода элемента массива в окно браузера и его перезапись.

```
<script type="text/javascript">
  var array = new Array(3, 7, 12, true, 4.5, "some string", true);
  document.write("array[0]=", array[0]);
  array[2] = array[1] + 1;
  document.write("<br>");
  document.write("array[2]=", array[2]);
</script>
```

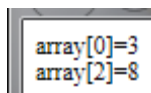


Рис. 13.3.

### 13.2.1.6. Создание функций

У функции есть определённые параметры, которыми она манипулирует, и возвращает результат.

```
<script type="text/javascript">
    function hello() {
        alert("Привет");
        alert("Привет");
        alert("Привет");
    }
    hello();
</script>
```

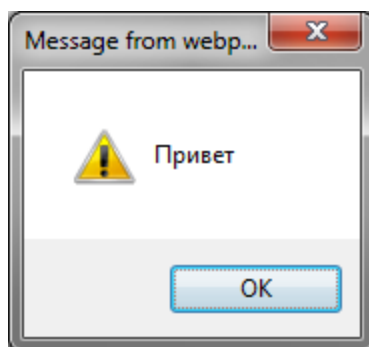


Рис. 13.4.

Теперь поговорим о функциях с параметрами.

```
<script type="text/javascript">
    function sum(x, y) {
        var sum = x + y;
        document.write("sum=", sum);
    }
    sum(5, 4);
</script>
```

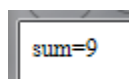


Рис. 13.5.

Вначале ключевое слово `function`, потом название функции. Внутри скобок указаны два параметра, которые требуются ( $x$  и  $y$ ). Внутри функции ещё одна переменная `sum`, которой присвоена сумма  $x$  и  $y$ . И затем вывод в окно браузера результата. После функции вызываем её, передав параметры 5 и 4.

В примере выше мы результат сразу печатали, однако, для данной функции будет наиболее логично ничего не печатать, а вернуть результат:

```
<script type="text/javascript">
    function sum(x, y) {
        var sum = x + y;
        return sum;
    }
    var z = sum(4, 5) + sum(1, -3);
    document.write("sum(4, 5) + sum(1, -3)=", z);
</script>
```

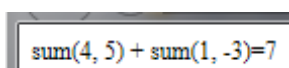


Рис 13.6.

#### 13.2.1.6.1. Параметры и область видимости

Параметры функции – такие же переменные, но их начальные значения задаются при вызове функции, а не в её коде.

#### 13.2.1.6.2. Вложенные области видимости

```
<script type="text/javascript">
  var landscape = function() {
    var result = "";
    var flat = function(size) {
      for (var count = 0; count < size; count++)
        result += "_";
    };
    var mountain = function(size) {
      result += "/";
      for (var count = 0; count < size; count++)
        result += "'";
      result += "\\ ";
    };
    flat(3);
    mountain(4);
    flat(6);
    mountain(1);
    flat(1);
    return result;
  };
  console.log(landscape());
  // → ___/''''\____/'\__
</script>
```

### 13.2.1.6.3. Стек вызовов



Обрабатывается она примерно так: вызов `greet` заставляет проход прыгнуть на начало функции. Он вызывает встроенную функцию `console.log`, которая перехватывает контроль, делает своё дело и возвращает контроль. Потом он доходит до конца `greet`, и возвращается к месту, откуда его вызвали. Следующая строчка опять вызывает `console.log`.

Место, где компьютер запоминает контекст, называется стеком. Каждый раз при вызове функции, текущий контекст помещается наверх стека. Когда функция возвращается, она забирает верхний контекст из стека и использует его для продолжения работы.

Хранение стека требует места в памяти. Когда стек слишком сильно разрастается, компьютер прекращает выполнение и выдаёт что-то вроде “stack overflow” или “too much recursion”.

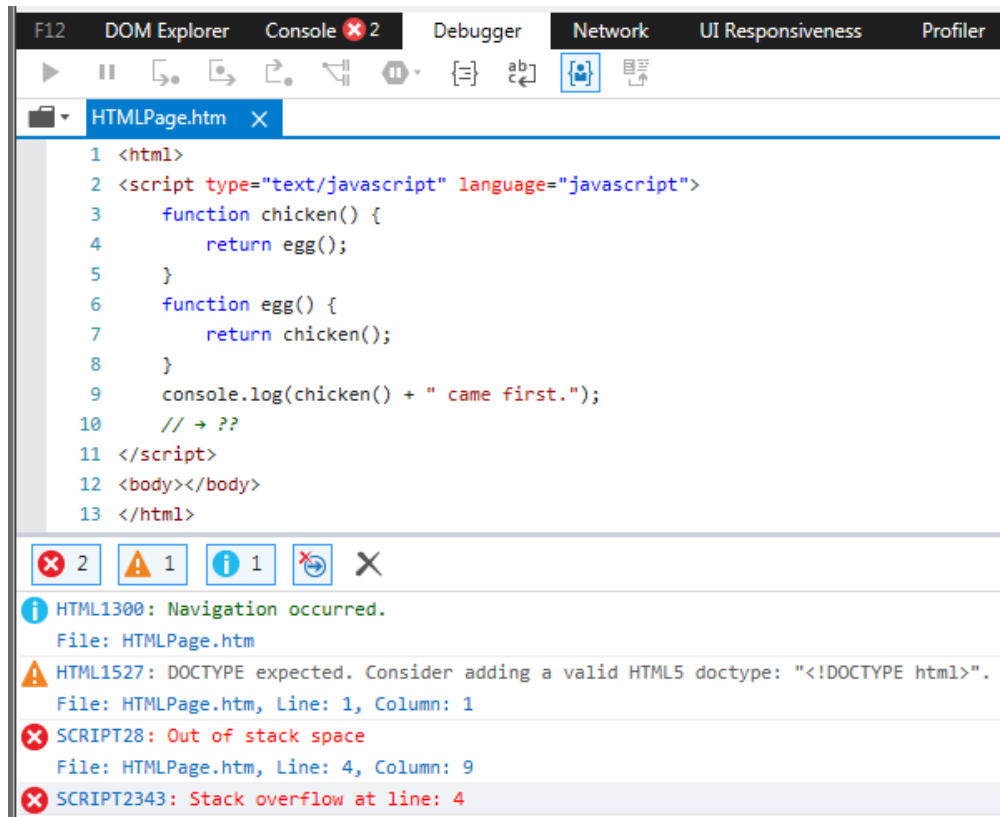


Рис. 13.8.

#### 13.2.1.6.4. Пример передачи функции по ссылке

Функцию легко можно передавать в качестве аргумента другой функции.

Например, `map` берет функцию `func`, применяет ее к каждому элементу массива `arr` и возвращает получившийся массив.

```
map(run, [10, 20, 30]) // = [1,2,3]
```

Или можно создать анонимную функцию непосредственно в вызове `map`:

```
map(function(a) { return a * 3 }, [1, 2, 3]) // = [3,6,9]
```

#### 13.2.1.6.5. Замыкания

Возможность использовать вызовы функций как переменные вкупе с тем фактом, что локальные переменные каждый раз при вызове функции создаются заново, приводит нас к интересному вопросу. Что происходит с локальными переменными, когда функция перестает работать?

Следующий пример иллюстрирует этот вопрос. В нём объявляется функция `wrapValue`, которая создаёт локальную переменную. Затем она возвращает функцию, которая читает эту локальную переменную и возвращает её значение.

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

function wrapValue(n) {
  var localVariable = n;
  return function() { return localVariable; };
}

var wrap1 = wrapValue(1);
var wrap2 = wrapValue(2);
console.log(wrap1());
// → 1
console.log(wrap2());
// → 2
```

```
</script>
```

```
1  
2
```

Рис. 13.10.

Эта возможность работать со ссылкой на какой-то экземпляр локальной переменной называется замыканием. Функция, замыкающая локальные переменные, называется замыкающей. Она не только освобождает вас от забот, связанных с временем жизни переменных, но и позволяет творчески использовать функции.

#### 13.2.1.6.6. Рекурсия

Функция вполне может вызывать сама себя, если она заботится о том, чтобы не переполнить стек. Такая функция называется рекурсивной.

Однако, у такой реализации есть проблема – в обычной среде JavaScript она раз в 10 медленнее, чем версия с циклом. Проход по циклу выходит дешевле, чем вызов функции.

Но рекурсия не всегда лишь менее эффективная альтернатива циклам. Некоторые задачи проще решить рекурсией. Чаще всего это обход нескольких веток дерева, каждая из которых может ветвиться.

Вот вам загадка: можно получить бесконечное количество чисел, начиная с числа 1, и потом либо добавляя 5, либо умножая на 3. Как нам написать функцию, которая, получив число, пытается найти последовательность таких сложений и умножений, которые приводят к заданному числу? К примеру, число 13 можно получить, сначала умножив 1 на 3, а затем добавив 5 два раза. А число 15 вообще нельзя так получить.

Рекурсивное решение:

```
<script type="text/javascript">  
    function findSolution(target) {  
        function find(start, history) {  
            if (start == target)  
                return history;  
            else if (start > target)  
                return null;  
            else  
                return find(start + 5, "(" + history + " + 5)") ||  
                       find(start * 3, "(" + history + " * 3)");  
        }  
        return find(1, "1");  
    }  
  
    console.log(findSolution(24));  
    // → (((1 * 3) + 5) * 3)  
</script>
```

```
(((1 * 3) + 5) * 3)
```

Рис. 13.12.

Внутренняя функция `find` занимается рекурсией. Она принимает два аргумента – текущее число и строку, которая содержит запись того, как мы пришли к этому номеру. И возвращает либо строку, показывающую нашу последовательность шагов, либо `null`.

#### 13.2.1.6.7. Вырачиваем функции

Существует два более-менее естественных способа ввода функций в программу.

Первый – вы пишете схожий код несколько раз.

Второй способ – вы обнаруживаете потребность в некоей новой функциональности, которая достойна помещения в отдельную функцию. Вы начинаете с названия функции, и затем пишете её тело.

То, насколько сложно вам подобрать имя для функции, показывает, как хорошо вы представляете себе её функциональность. Возьмём пример. Нам нужно написать программу, выводящую два числа, количество коров и куриц на ферме, за которыми идут слова «коров» и «куриц». К числам нужно спереди добавить нули так, чтобы каждое занимало ровно три позиции.

```
007 Коров
```


```
011 Куриц
```

Рис. 13.13.

Но только мы собрались отправить фермеру код, он звонит и говорит нам, что у него в хозяйстве появились свиньи, и не могли бы мы добавить в программу вывод количества свиней?

Можно, конечно. Но когда мы начинаем копировать и вставлять код, мы понимаем, что надо остановиться и подумать. Должен быть способ лучше. Пытаемся улучшить программу:

```
<script type="text/javascript">
  // Добавить Нулей
  function zeroPad(number, width) {
    var string = String(number);
    while (string.length < width)
      string = "0" + string;
    return string;
  }
  // Вывести Инвентаризацию Фермы
  function printFarmInventory(cows, chickens, pigs) {
    console.log(zeroPad(cows, 3) + " Коров");
    console.log(zeroPad(chickens, 3) + " Куриц");
    console.log(zeroPad(pigs, 3) + " Свиней");
  }
  printFarmInventory(7, 16, 3);
</script>
```



|            |
|------------|
| 007 Коров  |
| 011 Куриц  |
| 003 Свиней |

Рис. 13.14.

Функция с хорошим, понятным именем `zeroPad` облегчает понимание кода. И её можно использовать во многих ситуациях, не только в нашем случае. К примеру, для вывода отформатированных таблиц с числами.

Мы можем написать как простейшую функцию, которая дополняет число нулями до трёх позиций, так и навороченную функцию общего назначения для форматирования номеров, поддерживающую дроби, отрицательные числа, выравнивание по точкам, дополнение разными символами, и т.п.

#### 13.2.1.7. Циклы

Начнём с самого первого цикла (и самого популярного) - цикла `for`.

```
for (i = 0; i < 100; i++)
  document.write(i + " ");
```

Здесь мы задали переменную для итерации (`i`), которой присвоили значение 0. Дальше проверяется условие: `i < 100`. Если оно выполняется, то выполняется одна итерация цикла. После выполнения каждой итерации происходит `i++` (то есть увеличение переменной `i` на 1). Снова проверяется условие, и если оно истинно, то выполняется ещё одна итерация. И так до тех пор, пока условие `i < 100` не станет ложным.

Теперь поговорим о второй разновидности циклов в JavaScript - `while`. В принципе, цикл очень похож на все `for`, но здесь общий вид другой:

```
var i = 0;
while (i < 100) {
  i++;
  document.write(i + " ");
}
```

Перед началом цикла мы создали переменную `i`, которой присвоили начальное значение. Затем перед запуском цикла проверяется условие, и если оно истинно, то запускается итерация цикла, в которой мы увеличиваем переменную для итерации (иначе произойдёт заикливание). И выводим эту переменную.

И, наконец, последний вид циклов в JavaScript - цикл `do-while`.

Очень похож на цикл `while`, однако, здесь есть всего одно, но очень принципиальное отличие. Если цикл `while` сначала проверяет условие, а потом уже выполняет или нет итерацию. То цикл `do-while` сначала именно выполняет итерацию, и только потом проверяет условие. И если оно ложно, то выходит из цикла. Другими словами, независимо от условия данный цикл гарантированно выполнится хотя бы 1 раз.

```
var i = 0;
do {
  i++;
  document.write(i + " ");
}
```



```
    } while (i < 100)
```

Операторы цикла break и continue.

Начнём с break. Данный оператор позволяет досрочно выскочить из цикла.

```
for (i = 0; i < 100; i++) {  
    if (i == 50) break;  
    document.write(i + " ");  
}
```

Вы можете запустить этот скрипт и обнаружите, что вывелись только числа до 49, так как при i = 50 цикл прервался, благодаря оператору break.

Continue позволяет перейти к следующей итерации цикла.

```
for (i = 0; i < 100; i++) {  
    if (i == 50) continue;  
    document.write(i + " ");  
}
```

Если Вы запустите этот скрипт, то увидите, что не хватает числа 50. Это произошло потому, что при i = 50, мы переходим к следующей итерации цикла, перед которой i увеличивается на 1 и становится равным 51-му.

### 13.2.1.8. Switch case

Смысл этого оператора очень прост. Переменная variable проверяется на своё значение, и если оно совпадает с одним из вариантов (value\_1, value\_2, ..., value\_n), то выполняется соответствующий блок операторов. Если нет ни одного совпадения, то выполняется блок default (которого, кстати, может и не быть).

Допустим, Вы просите пользователя ввести число от 1 до 3. Каждое число отвечает за определённое действие. Вот пример реализации такого "меню":

```
<script type="text/javascript">  
var change = prompt("Выберите действие: 1 - Купить автомобиль; 2 - Продать  
автомобиль; 3 - Обменять автомобиль");  
/*var change = prompt("Выберите действие:\n" +  
    "1 - Купить автомобиль\n" +  
    "2 - Продать автомобиль\n" +  
    "3 - Обменять автомобиль");*/  
  
switch (change) {  
    case "1":  
    {  
        document.write("Вы хотите купить автомобиль");  
        break;  
    }  
    case "2":  
    {  
        document.write("Вы хотите продать автомобиль");  
        break;  
    }  
    case "3":  
    {  
        document.write("Вы хотите обменять автомобиль");  
        break;  
    }  
    default:  
    {  
        document.write("Вы ввели некорректную команду");  
        break;  
    }  
}  
</script>
```

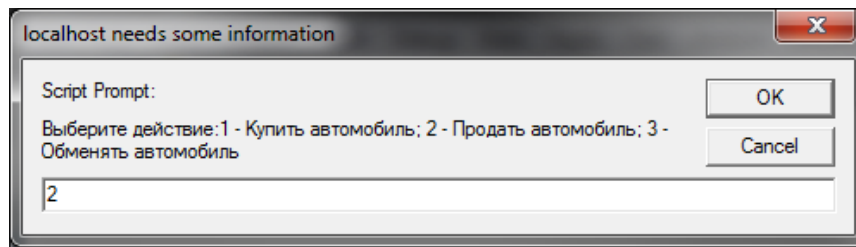


Рис. 13.19.

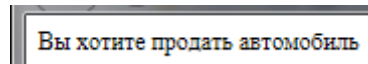


Рис. 13.20.

Первой строкой мы запрашиваем у пользователя его выбор. На следующей строчке начинается оператор switch case. В качестве переменной для анализа мы выбрали change, значение для которой ввёл пользователь. Дальше мы анализируем её. Обратите, что значения стоят в кавычках. Это потому, что мы получаем от пользователя не число, а строку, и мы её должны сравнивать с другими строками. Если введено что-то другое (не "1", не "2" и не "3"), то выводится "Вы ввели некорректную команду".

### 13.2.1.9. Объекты Javascript

Объект в javascript представляет собой обычный ассоциативный массив или, иначе говоря, "хэш". Он хранит любые соответствия "ключ => значение" и имеет несколько стандартных методов.

Метод объекта в javascript - это просто функция, которая добавлена в ассоциативный массив.

#### 13.2.1.9.1. Создание объекта и добавление свойств

Следующие два варианта создания объекта эквивалентны:

```
var o = new Object()  
var o = {}
```

Есть два синтаксиса добавления свойств в объект. Первый - точка, второй - квадратные скобки:

```
o.test = 5  
o["test"] = 5
```

Свойства так же могут быть удалены. Удаляет свойство оператор delete:

```
o.test = 5  
delete o.test
```

#### 13.2.1.9.2. Доступ к свойствам

Доступ к свойству осуществляется точно так же:

```
alert(o.test)  
alert(o['test'])
```

Если у объекта нет такого свойства, то результат будет 'undefined'

```
var o = {}  
alert(o.nosuchkey) // => undefined
```

Никакой ошибки при обращении по несуществующему свойству не будет, просто вернется специальное значение undefined.

### 13.2.1.10. Объект Math

Объект Math в JavaScript отвечает за математические операции.

```
document.write('Math.E=', Math.E);  
document.write("<br>");  
document.write('Math.PI=', Math.PI);
```

Если Вы запустите этот скрипт, то увидите значения двух самых популярных констант математики.

Теперь займёмся методами объекта Math в JavaScript. Первый метод - это abs(x), который принимает в качестве параметра число, и возвращает его модуль. Например, так:

```
var x = -15.2;
document.write('Math.abs(x)=', Math.abs(x));
```

Результатом будет число "15.2".

Следующим методом будет random(). Очень популярный метод, который генерирует случайным образом число от 0 до 1. Причём, 0 входит, а 1 уже не входит. Давайте с Вами получим число от 0 до 10.

```
document.write('Math.random() * 10=', Math.random() * 10);
```

Данная строка выведет число от 0 до 10 (причём дробное). Обратите внимание, что 0 быть может, а 10 быть не может.

Метод sqrt(x) считает квадратный корень из числа. Применение очевидное и очень простое:

```
document.write('Math.sqrt(9)=', Math.sqrt(9));
```

В данном примере, после запуска скрипта мы увидим число "3".

Метод log(x) считает натуральный логарифм числа.

```
document.write('Math.log(Math.E * Math.E)=', Math.log(Math.E * Math.E));
```

Ещё один метод считает степень числа. Называется метод - pow(x, y). Принимает два параметра, первый - это основание числа, а второй - это его степень. Сразу пример:

```
document.write('Math.pow(2, 5)=', Math.pow(2, 5));
```

И, напоследок, рассмотрим группу методов, выполняющие тригонометрические функции:

```
var x = 0.1;
document.write("x = 0.1 <br>");
document.write('Math.sin(x)=', Math.sin(x) + "<br>"); //Синус числа
document.write('Math.cos(x)=', Math.cos(x) + "<br>"); //Косинус числа
document.write('Math.tan(x)=', Math.tan(x) + "<br>"); //Тангенс числа
document.write('Math.asin(x)=', Math.asin(x) + "<br>"); //Арксинус числа
document.write('Math.acos(x)=', Math.acos(x) + "<br>"); //Арккосинус числа
document.write('Math.atan(x)=', Math.atan(x) + "<br>"); //Арктангенс числа.
```

#### 13.2.1.11. Используя функцию

Это, вероятно, один из самых распространенных способов. Вы определяете нормальную функцию JavaScript, а затем создаете объект, используя новое ключевое слово. Чтобы определить свойства и методы для объекта, созданного с помощью функции (), вы используете это ключевое слово, как показано в следующем примере.

```
function Apple(type) {
    this.type = type;
    this.color = "red";
    this.getInfo = getAppleInfo;
}
// anti-pattern! keep reading...
function getAppleInfo() {
    return this.color + ' ' + this.type + ' apple';
}
```

Чтобы создать экземпляр объекта с помощью функции конструктора Apple, задайте некоторые свойства и методы вызова, вы можете сделать следующее:

```
var apple = new Apple('macintosh');
apple.color = "reddish";
alert(apple.getInfo());
```

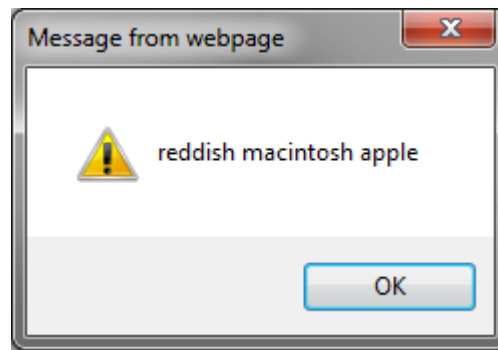


Рис. 13.23.

#### 13.2.1.11.1. Методы, определенные внутри

В приведенном выше примере вы видите, что метод `getInfo ()` класса `Apple` был определен в отдельной функции `getAppleInfo ()`. Хотя это прекрасно работает, у него есть один недостаток - вы можете в конечном итоге определить множество этих функций, и все они находятся в «глобальной названии». Это означает, что у вас могут быть конфликты имен, если вы (или другая библиотека, которую используете) решили создать другую функцию с тем же именем. Чтобы предотвратить загрязнение глобального пространства имен, вы можете определить свои методы в функции конструктора, например,

```
<script type="text/javascript" language="javascript">
    function Apple(type) {
        this.type = type;
        this.color = "red";
        this.getInfo = function() {
            return this.color + ' ' + this.type + ' apple';
        };
    }

    var apple = new Apple('macintosh');
    apple.color = "reddish";
    alert(apple.getInfo());
</script>
```

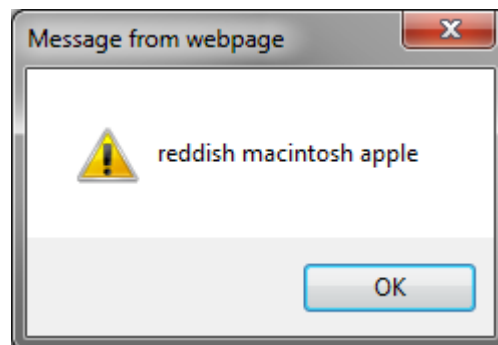


Рис. 13.24.

Использование этого синтаксиса ничего не меняет, как вы создаете экземпляр объекта и используете его свойства и методы.

#### 13.2.1.12. Использование объектных литералов

Литералы - это более короткий способ определения объектов и массивов в JavaScript. Чтобы создать пустой объект, можно использовать:

```
var o = {};
```

вместо «нормального» способа:

```
var o = new Object ();
```

Таким образом вы можете сразу создать объект. Вот те же функции, что описаны в предыдущих примерах, но с использованием синтаксиса объектного литерала:

```
var apple = {
```

```

    type: "macintosh",
    color: "red",
    getInfo: function() {
        return this.color + ' ' + this.type + ' apple';
    }
}

```

В этом случае вам не нужно создавать экземпляр класса, он уже существует. Поэтому вы просто начинаете использовать этот экземпляр.

```

//var apple = new Apple('macintosh');
apple.color = "reddish";
alert(apple.getInfo());

```

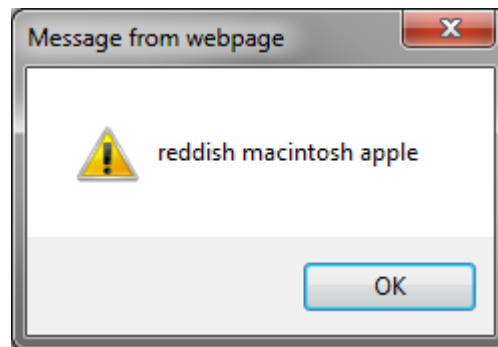


Рис. 13.25.

Литеральный объект, также называемый инициализатором объекта, представляет собой набор парных имен и значений, разделенных запятыми. Вы можете создать литерал объекта, как показано ниже:

```

<script type="text/javascript" language="javascript">
    var car = {
        model: 'bmw',
        color: 'red',
        price: 2000
    }
    console.log(JSON.stringify(car));
</script>

```

```

{"model":"bmw","color":"red","price":2000}

```

Рис. 13.26.

Вы можете добавлять свойства динамически в объект, в том числе после создания объекта. Здесь мы добавляем динамическое свойство `car.type`:

```

var car = {
    model: 'bmw',
    color: 'red',
    price: 2000
}
console.log(JSON.stringify(car));
car.type = 'manual'; // dynamic property
console.log(JSON.stringify(car));

```

Литеральный объект - это простое выражение, которое создает объект каждый раз, когда оператор выполняется в коде.

### 13.2.1.13. Объект Array

Объект Array в JavaScript позволяет создавать массивы. Также он содержит свойства и методы, позволяющие управлять массивом.

Рассмотрим всего одно свойство, но самое важное и часто используемое. Вот оно:

```

var arr = new Array(1, 0, -4, 10, 12);
document.write("arr: ", arr);
document.write("<br>arr.length=", arr.length);

```

Свойство `length` содержит длину массива. Очевидно, что используется данное свойство преимущественно в циклах, где перебираются элементы массива.

Теперь рассмотрим методы объекта `Array`. Начнём с метода `join()`:

Данный метод возвращает строку, составленную из элементов массивов, разделённых запятой. Если использовать параметр в методе `join()`, то, соответственно, вместо запятой будет этот параметр. Например, так:

```
document.write(arr.join("--"));
```

На выходе получится следующее: "1--0--4--10--12".

Метод `pop()` вытаскивает последний элемент из массива и возвращает его. То есть у массива исчезает последний элемент, который метод `pop()` возвращает. Вот пример:

```
var last_element = arr.pop();
document.write("arr.pop()=", last_element);
```

Здесь из массива исчезает последний элемент, который мы записываем в переменную `last_element` и выводим в окно браузера.

Метод `shift()` работает аналогично, только исчезает не последний элемент, а первый, при этом все остальные элементы сдвигаются на одну позицию вниз.

```
var first_element = arr.shift();
document.write("arr.shift()=", first_element);
```

Для того чтобы развернуть массив используется метод `reverse()`. Пример его использования:

```
var a = arr.reverse();
//document.write("arr.reverse()= ", a);
document.write("arr.reverse()=", a.join("; "));
```

В данном примере метод возвращает перевёрнутый массив `arr` и записывает его в массив `a`. Очень полезный метод, позволяющий получить новый массив из части другого массива – `slice()`.

```
var b = arr.slice(2, 4);
document.write("b = arr.slice(2, 4)= " , b.join("; ") + "<br>");
```

В этом примере мы присваиваем массиву `a` часть массива `arr`, которая начинается со 2-го элемента (включительно) до 4-ого (исключительно). Например, если массив `arr` был с такими значениями: 2, 4, 5, 1, 2. То массив `a` будет иметь следующие значения: 5, 1. Также не забывайте, что нумерация начинается с 0. Поэтому элемент с индексом "2" - это элемент со значением 5.

Если в методе `slice()` указать только один параметр, то будет браться массив начиная с индекса, указанного в параметре и до конца массива.

И последний метод - метод `sort()`. Данный метод сортирует массив. Использовать метод `sort()` легко:

```
arr.sort();
```

Тут всё просто: массив сортируется, однако, какое правило сортировки? По алфавиту, по числам, по возрастанию, по убыванию и прочее. В этом и состоит сложность. Поэтому разработчики этого метода предоставили нам задать эти правила, добавив один необязательный параметр в метод `sort()`, который должен быть именем функции, описывающей "правила сортировки".

Эта функция должна определять критерии, при которых один элемент больше другого. Давайте, например, сделаем так: если элемент равен "5", то тогда одно заведомо больше другого. Если ни один из элементов не равен 5, то тогда сравниваем по обычным правилам.

```
function myCompare(x, y) {
    if (x == 5) return 1;
    if (y == 5) return -1;
    if (x > y) return 1;
    else if (x == y) return 0;
    else return -1;
}
```

```
arr.sort(myCompare);
```

#### 13.2.1.14. Объект `Date`

При работе с датой и временем в JavaScript используется объект `Date`.

Первый конструктор без параметров, и он возвращает текущее время и дату:

```
var date = new Date();  
document.write("date= ", date);
```

В результате, Вы увидите что-то в этом духе: "Thu Oct 14 2010 11:42:06 GMT+0400".  
Второй конструктор объекта Date - это конструктор с одним параметром:

```
var dt1 = new Date(135253235);  
document.write("Date(135253235)= ", dt1);
```

В результате Вы увидите следующее: "Fri Jan 02 1970 16:34:13 GMT+0300".

Следующий конструктор позволяет создать объект Date с заданием следующих параметров: года, месяца и числа:

```
var date = new Date(2010, 0, 12);  
document.write(date);
```

Результат: "Tue Jan 12 2010 00:00:00 GMT+0300". Также заметьте, что 0-ой месяц - это январь, а 11-ый - это декабрь.

И последний конструктор класса Date в JavaScript позволяет создать объект Date со всеми параметрами даты и времени: год, месяц, число, часы, минуты и секунды.

```
var dt2 = new Date(2010, 0, 12);  
document.write("Date(2010, 0, 12)= ", dt2);
```

Получится вот такая строка: "Tue Jan 12 2010 23:45:11 GMT+0300". Вот и все конструкторы объекта Date в JavaScript, которые нам предоставили разработчики.

И последнее – метод setTime(). Принимает в качестве параметра число миллисекунд прошедших с 01.01.1970:

```
date.setTime(39293012);  
document.write(date);
```

В результате Вы увидите вот это: "Thu Jan 01 1970 13:54:53 GMT+0300".

#### 13.2.1.15. Объект Window

Для создания всплывающих окон в JavaScript используется объект Window.

Окна создаются через метод объекта Window - open():

```
var win = window.open("https://www.tut.by/", "My Window");
```

В данном примере мы создаём окно, в которое будет подгружен сайт: "http://tut.by", а имя нового окна будет: "My Window". Это простейший вариант создания окна в JavaScript. Однако, у метода open() существует ещё один необязательный параметр, с настройками нового окна:

```
var win = window.open("https://www.tut.by/", "My Window",  
    "width=400, height=500, menubar=yes, toolbar=no, location=yes,  
scrollbars=yes");
```

Обратите внимание на то, что этот параметр имеет очень жёсткие условия в плане синтаксиса: никаких пробелов быть вообще не должно. А именно эту ошибку чаще всего и допускают. Теперь о параметрах:

Width - отвечает за ширину нового окна.

Height - отвечает за высоту нового окна.

Menubar - если стоит "yes", то в новом окне будет меню, если стоит "no", то, соответственно, меню не будет.

Toolbar - отвечает за наличие панели инструментов.

Location - отвечает за наличие адресной строки в новом окне.

Scrollbars - отвечает за наличие полос прокрутки.

Теперь перейдём к методам, которые используют очень часто, - таймерам в JavaScript. Есть всего два метода для работы с таймерами. Первый метод - это setTimeout(). Эта функция принимает два параметра: функцию (либо код), которую нужно выполнить, и второй параметр - задержку (в миллисекундах), через которую надо выполнить функцию (либо код). Вот пример:

```
setTimeout(func, 3000);  
function func() {  
    alert("Прошло 3 секунды");
```

```
}
```

Если Вы запустите данный скрипт, то через 3 секунды после начала выполнения Вы увидите информационное сообщение. Обратите внимание, что метод `setTimeout()` применён к глобальному окну, поэтому мы не писали так: `"window.setTimeout()"`. Хотя, конечно, так тоже можно писать.

Самая популярная задача - это часы на сайте. Думаю, многие с ними встречались. Давайте реализуем простейший вариант (простейший, потому что без графики):

```
<html>
<head>
<script type="text/javascript">
    function startTime() {
        var date = new Date();
        var hours = date.getHours();
        var minutes = date.getMinutes();
        var seconds = date.getSeconds();
        if (hours < 10) hours = "0" + hours;
        if (minutes < 10) minutes = "0" + minutes;
        if (seconds < 10) seconds = "0" + seconds;
        document.getElementById("time").innerHTML = hours + ":" + minutes + ":"
+ seconds;
        setTimeout(startTime, 1000);
    }
</script>
</head>
<body onLoad = 'startTime()'>
    Время на сайте: <b><span id="time"></span></b>
</body>
</html>
```

Начнём с функции `startTime()`, которая в самом начале узнаёт текущие время и дату. Затем получает отдельно часы, минуты и секунды. Дальше добавляются ведущие нули для однозначных чисел (чтобы не было, например, такого 15:1:12, а было 15:01:12). После этого внутри элемента с `id = "time"` появляется строка с текущим временем. И дальше идёт самое интересное: мы внутри самой функции вызываем через одну секунду её ещё раз. Всё вышесказанное вновь выполняется (но время уже на одну секунду больше), снова меняется содержимое элемента `"time"`, затем вновь запуск через одну секунду. И, наконец, чтобы вся эта конструкция запустилась, мы при загрузке документа (событие `Load` и обработчик `onLoad`) вызываем функцию `startTime()`. Вот такой классический алгоритм реализации таймера.



Рис. 13.27.

#### 13.2.1.16. Объект Image

Для работы с изображениями в JavaScript используется объект `Image`. Данный объект является очередным свойством объекта `Document`.

Прежде чем начать обрабатывать изображение необходимо его создать. Разумеется, создание происходит в HTML, поэтому знакомый Вам тег:

```
<img name = 'img' src = 'image1.jpg' />
```

Теперь мы можем обратиться к этому объекту через JavaScript:

```
document.write(document.img)
```

Как видите, обращение к объекту `Image` очень простое: сначала пишется объект `Document`, а затем его свойство с именем объекта `Image` (это имя мы задали в атрибуте `"name"`). В результате выполнения этого скрипта Вы увидите: `"[object HTMLImageElement]"`. Это сработал метод `toString()`.

Теперь переходим к свойствам. Начнём со свойства `border`. Данное свойство отвечает за размер рамки вокруг изображения. Разумеется, мы его можем и прочитать, и записать. Давайте изменим размер рамки нашего изображения:

```
document.img.border = 5;
```

Разумеется, есть два свойства, отвечающие за ширину и высоту изображения. Это свойства `width` и `height` объекта JavaScript. Давайте их выведем:



```
document.write("Ширина изображения - " + document.img.width + "<br />");
document.write("Высота изображения - " + document.img.height);
```

И последнее свойство, которое мы рассмотрим - это src. Данное свойство отвечает за путь к картинке. И давайте с Вами решим такую задачу: есть картинка и есть кнопка. При нажатии на кнопку картинка меняется.

```
<script type="text/javascript">
    var flag = 0;
    function changeImage() {
        if (flag == 0) {
            document.img.src = 'image1.jpg';
            flag = 1;
        }
        else {
            document.img.src = 'image2.jpg';
            flag = 0;
        }
    }
</script>
</body>
<img name = 'img' alt="" src = 'image1.jpg' />
<form>
    <input type = 'button' value = 'Сменить изображение' onClick =
'changeImage()' />
</form>
```

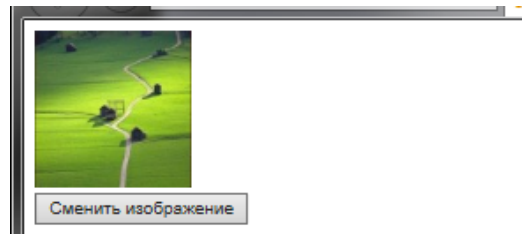


Рис. 13.30.

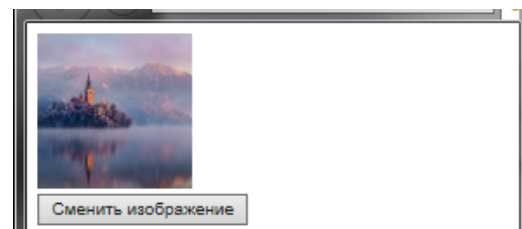


Рис. 13.31.

Сначала мы описываем саму функцию. Создаётся переменная flag. Это некий флаг, который переключается при смене изображения. Далее идёт функция changeImage(), которая и занимается сменой изображения. Изображению присваивается тот путь к картинке, которому соответствует флаг. После смены изображения меняется и значение флага (чтобы в следующий раз было другое изображение). За пределами скрипта создаётся форма с одной кнопкой. Здесь обратите внимание на атрибут "onClick". Этот атрибут отвечает за обработку события "Клик мыши по кнопке".

#### 13.2.1.17. Сведения о браузере

Существует достаточно большое количество браузеров, а также их версий. И проблема в том, что одни браузеры позволяют выполнить определённый скрипт, а другие не позволяют. Встаёт вопрос: а как узнать браузер пользователя в JavaScript, чтобы через условие решить: надо выполнять скрипт или нет.

Для таких целей существует объект Navigator, а точнее два его свойства: appName и appVersion.

Свойства appName и appVersion доступны только для чтения (это и логично), поэтому изменить их у Вас не получится. Теперь встаёт вопрос, а как использовать их в операторе IF:

```
<script type="text/javascript">
    document.write("Вы используете браузер:<br>" + navigator.appName);
    document.write("<br><br>");

    document.write("Версия Вашего браузера:<br>" + navigator.appVersion);
```

```

document.write("<br><br>");

var browser = navigator.appName;
//убираем пробелы вначале и в конце строки
browser = browser.replace(/^\s*/, '').replace(/\s*$/, '');
if (browser == "Netscape") {
    document.write("Это браузер NetScape<br>");
    document.write("Тут можно выполнять скрипты для обладателей браузеров
NetScape");
}
else
    document.write("Вы используете браузер: " + navigator.appName);
</script>

```

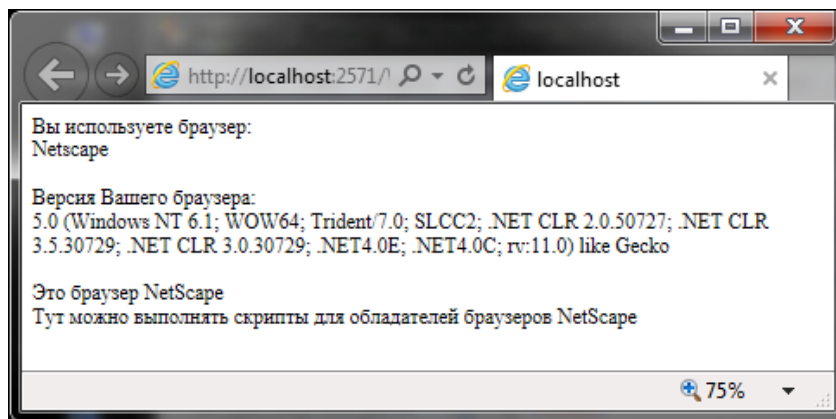


Рис. 13.32.

### 13.2.1.18. События

Тема событий очень важная и очень интересная. Благодаря ей, можно делать очень много интересных вещей. Событие в JavaScript - это определённое действие, которые вызвано либо пользователем, либо браузером. Например, событием может быть клик мыши по кнопке, движение мыши, наведение фокуса на элемент, изменение значения в каком-нибудь текстовом поле, изменение размеров окна браузера и так далее.

| Событие   | Объект   | Причина возникновения            |
|-----------|--|----------------------------------|
| Abort     | Image  | Прерывание загрузки изображения  |
| Blur      | Button, Checkbox, FileUpload, Frame, Layer, Password, Radio, Reset, Select, Submit, Text, Textarea, Window | Потеря фокуса элемента           |
| Change    | FileUpload, Select, Text, Textarea   | Смена значения                   |
| Click     | Area, Button, Checkbox, Document, Link, Radio, Reset, Submit   | Клик мыши на элементе            |
| DbClick   | Area, Document, Link   | Двойной клик на элементе         |
| DragDrop  | Window   | Перемещение в окно браузера      |
| Focus     | Button, Checkbox, FileUpload, Frame, Layer, Password, Radio, Reset, Select, Submit, Text, Textarea, Window | Установка фокуса на элементе     |
| KeyDown   | Document, Image, Link, Textarea  | Нажатие клавиши на клавиатуре    |
| KeyPress  | Document, Image, Link, Textarea  | Удержание клавиши на клавиатуре  |
| KeyUp     | Document, Image, Link, Textarea  | Отпускание клавиши на клавиатуре |
| Load      | Document, Image, Layer, Window   | Загрузка элемента                |
| MouseDown | Button, Document, Link   | Нажата кнопка мыши               |
| MouseMove | Window   | Мышь в движении                  |
| MouseOut  | Area, Layer, Link  | Мышь выходит за границы элемента |
| MouseOver | Area, Layer, Link  | Мышь находится над элементом     |
| MouseUp   | Button, Document, Link   | Отпущена кнопка мыши             |
| Move      | Frame  | Перемещение элемента             |
| Reset     | Form   | Сброс формы                      |
| Resize    | Frame, Window  | Изменение размеров               |
| Select    | Text, Textarea   | Выделение текста                 |
| Submit    | Form   | Передача данных                  |
| Unload    | Window   | Выгрузка текущей страницы        |

Таблица 13.1.

Теперь разберёмся с тем, как использовать события в JavaScript. Существуют, так называемые, обработчики событий. Обработчики событий как раз и определяют, что будет происходить при возникновении определённого события. Обработчики событий в JavaScript имеет следующий общий вид:

`onНазваниеСобытия`

То есть вначале идёт приставка "on", а дальше название события, например, такие обработчики событий: `onFocus`, `onClick`, `onSubmit` и так далее.

На странице имеются три ссылки. Каждая из ссылок отвечает за разный цвет фона (допустим, белый, жёлтый и зелёный). Вначале фон белый. При наведении мыши на определённую ссылку цвет фона меняется. При отведении мыши цвет фона возвращается на цвет по умолчанию. При щелчке мыши по ссылке цвет фона сохраняется, как по умолчанию.

```
<html>
<head>
<style>
  a {
    color: blue;
    text-decoration: underline;
    cursor: pointer;
  }
</style>
<script type="text/javascript">
var default_color = "white";

function setTempColor(color) {
  document.bgColor = color;
}

function setDefaultColor(color) {
  default_color = color;
  document.bgColor = default_color;
}

function defaultColor() {
  document.bgColor = default_color;
}
</script></script>
</head>
<body>
  <a onMouseOver = "setTempColor('white');"
    onMouseOut = "defaultColor()"
    onClick = "setDefaultColor('white');">Белый</a>
  <a onMouseOver = "setTempColor('yellow');"
    onMouseOut = "defaultColor()"
    onClick = "setDefaultColor('yellow');">Жёлтый</a>
  <a onMouseOver = "setTempColor('green');"
    onMouseOut = "defaultColor()"
    onClick = "setDefaultColor('green');">Зелёный</a>
</body>
</html>
```



Рис. 13.33.



Рис. 13.34.



Рис. 13.35.

Вначале идут обычные HTML-теги, с которых начинается любая HTML-страница. Далее мы создаём стиль, в котором требуем, чтобы все ссылки на данной странице были синего цвета, подчёркнутые, и чтобы указатель мыши на них был в виде "Pointer". Далее уже начинается JavaScript. Мы создаём переменную `default_color`, отвечающую за цвет по умолчанию. Далее идут три функции:

- Функция `setTempColor()` отвечает за временное изменение цвета.
- Функция `setDefaultColor()` отвечает за изменение цвета по умолчанию.
- Функция `defaultColor()` устанавливает цвет фона по умолчанию.

Потом идут ссылки с атрибутами в виде обработчиков событий. При наведении мышки на ссылку возникает событие `MouseOver`, соответственно, обработчик события `onMouseOver` вызывает функцию `setTempColor()` и передаёт соответствующий параметр. При снятии мышки с элемента возникает событие `MouseOut`, а дальше вызывается функция `defaultColor()`, которая делает цветом фона цвет по умолчанию. И, наконец, при клике мышки по ссылке (обработчик `onClick`) вызывается функция `setDefaultColor()`, которая устанавливает цвет заданный в параметре цветом фона по умолчанию. Как видите, всё достаточно просто.

#### 13.2.1.19. Запуск нескольких функций в `onload`

Данное событие срабатывает непосредственно при загрузке страницы. И очень часто при загрузке страницы должна выполняться какая-то функция. Но что делать, если нужно запустить несколько функций JavaScript?

Решение действительно тривиальное, достаточно запустить функцию, в которой запускаются все необходимые другие функции:

```
<html>
<head>
<script type="text/javascript">
    document.write("Запуск нескольких функций в onload<br>");
    function a() {
        b()
        c();
    }
    function b() {
        alert("Функция b() запущена");
    }
    function c() {
        alert("Функция c() запущена");
    }
</script>
</head>
<body onload="a()">
</body>
</html>
```

Если открыть данную страницу, то увидите, как при загрузке страницы работают функции `b()` и `c()`.

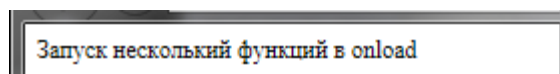


Рис. 13.36.

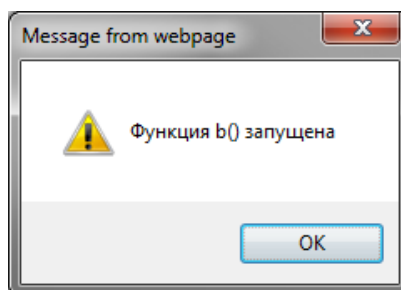


Рис. 13.37.

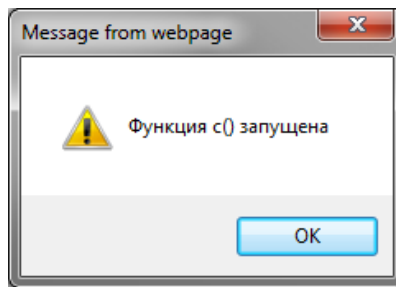


Рис. 13.38.

### 13.2.2. Контрольные вопросы

1. Как лучше располагать функции в коде?
2. С помощью чего задаются переменные в JavaScript?
3. Какие существуют правила для переменных?
4. Какие значения принимает тип переменной `boolean`?
5. Что такое стек вызовов и что может приводить к ошибке «stack overflow»?
6. Перечислите три вида циклов.
7. Какие существуют методы объекта `Array`?
8. Назовите параметры объекта `Window`. За что они отвечают?
9. Что такое событие в JavaScript?
10. Какие существуют типы переменных в JavaScript?
11. Что такое параметры функции?
12. Что такое рекурсия?
13. Какие существуют способы ввода функций в программу?

### 13.2.3. Задания

1. Создайте программу на JavaScript которая будет показывать в браузере информацию о названии, версии браузера. (R01401)
2. Добавьте в нее часы, а также таймер, по которому через 5 секунд будет вызываться окно с любым набранным Вами текстом. (R01402)
3. Создайте программу, которая выводит на экран набранный с клавиатуры массив. (R01403)
4. Используя функцию `math.min`, напишите функцию, принимающую два аргумента и возвращающую минимальный из них. (R01404)
5. Напишите программу, которая будет считать степень числа, используя рекурсивную функцию. (R01405)