

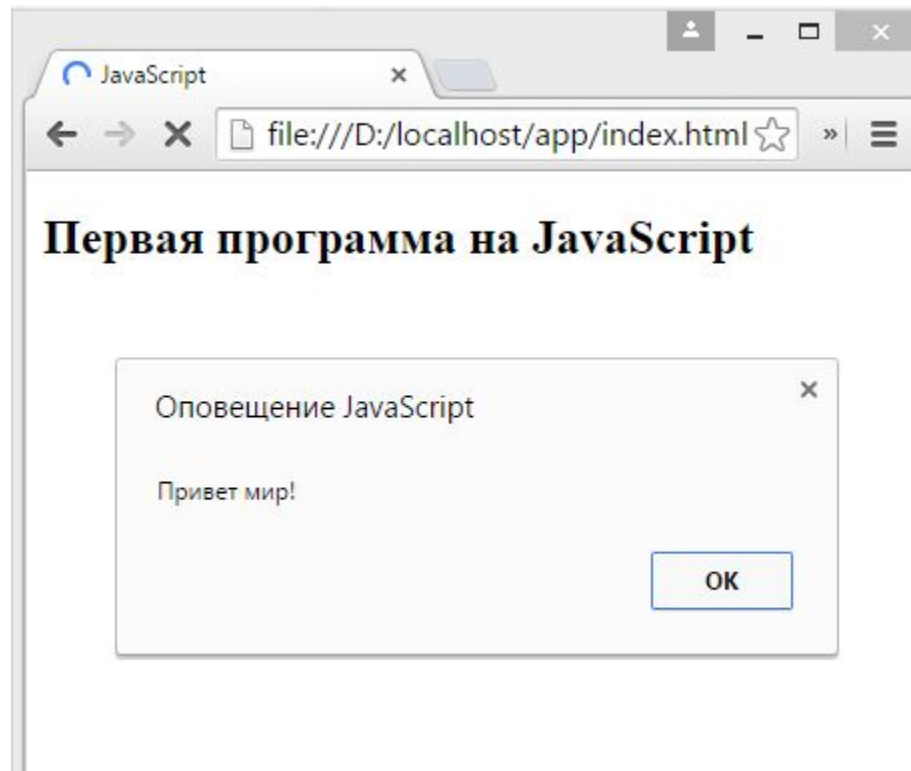
# **Введение в JavaScript**

## **Что такое JavaScript**

- **JavaScript** - это то, что делает живыми веб-страницы, которые мы каждый день просматриваем в своем веб-браузере.
- Создадим первую программу на javascript. Для начала определим для нашего приложения какой-нибудь каталог. Например, назовем его *app*. В этой папке создадим файл под названием **index.html**. То есть данный файл будет представлять веб-страницу.

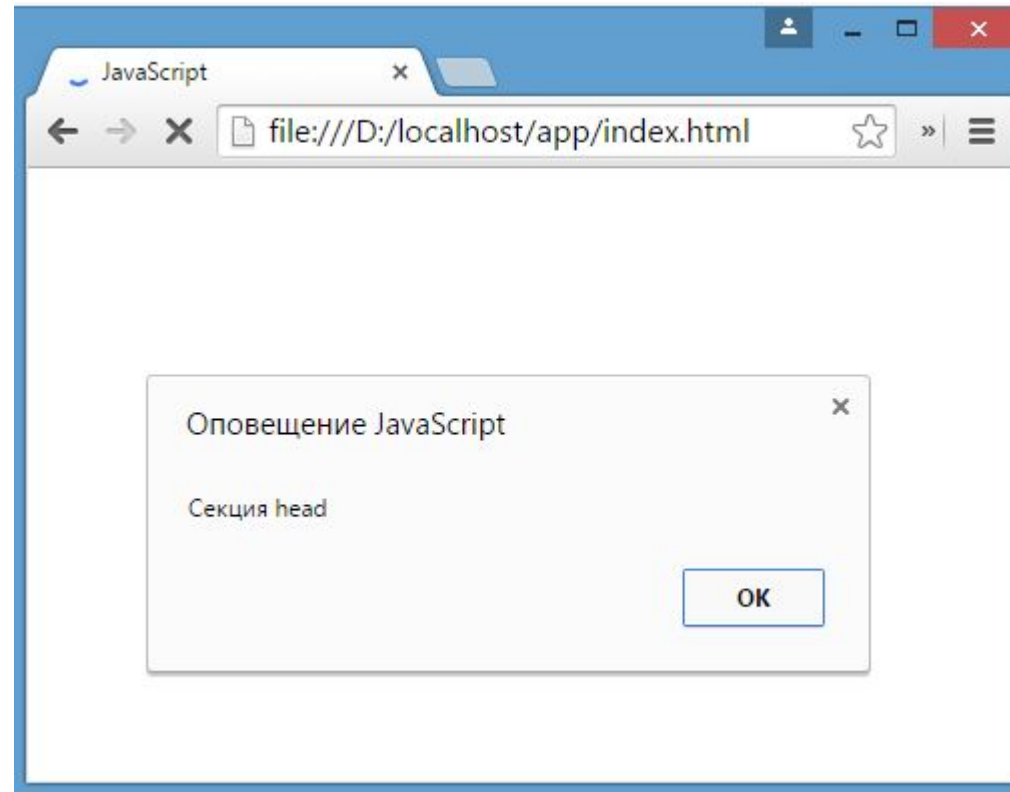
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>JavaScript</title>
</head>
<body>
  <h2>Первая программа на JavaScript</h2>
  <script>
    alert('Привет мир!');
  </script>
</body>
</html>
```

- Код **javascript** может содержать множество инструкций и каждая инструкция завершается точкой с запятой. Наша инструкция вызывает метод **alert()**, который отображает сообщение со строкой 'Привет мир!'.



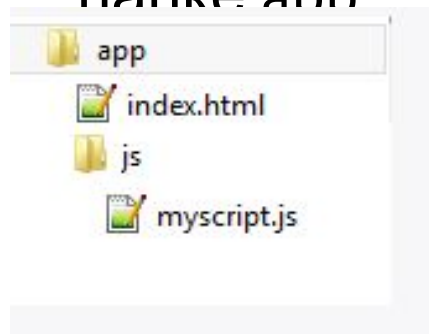
# Выполнение кода javascript

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>JavaScript</title>
  <script>
    alert("Секция head");
  </script>
</head>
<body>
  <h2>Первый заголовок</h2>
  <script>
    alert("Первый заголовок");
  </script>
  <h2>Второй заголовок</h2>
  <script>
    alert("Второй заголовок");
  </script>
</body>
</html>
```



# Подключение внешнего файла JavaScript

- Еще один способ подключения кода JavaScript на веб-страницу представляет вынесение кода во внешние файлы и их подключение с помощью тега `<script>`
- Итак, в прошлой теме мы создали html-страницу `index.html`, которая находится в каталоге `app`. Теперь создадим в этом каталоге новый подкаталог. Назовем его `js`. Он будет предназначен для хранения файлов с кодом javascript. В этом подкаталоге создадим новый текстовый файл, который назовем `myscript.js`. Файлы с кодом javascript имеют расширение `.js`. То есть у нас получится следующая структура в папке `app`:



- Чтобы подключить файл с кодом javascript на веб-страницу, применяется также тег <script>, у которого устанавливается атрибут src. Этот атрибут указывает на путь к файлу скрипта. В нашем случае используется относительный путь. Так как веб-страница находится в одной папке с каталогом js, то в качестве пути мы можем написать js/myscript.js.

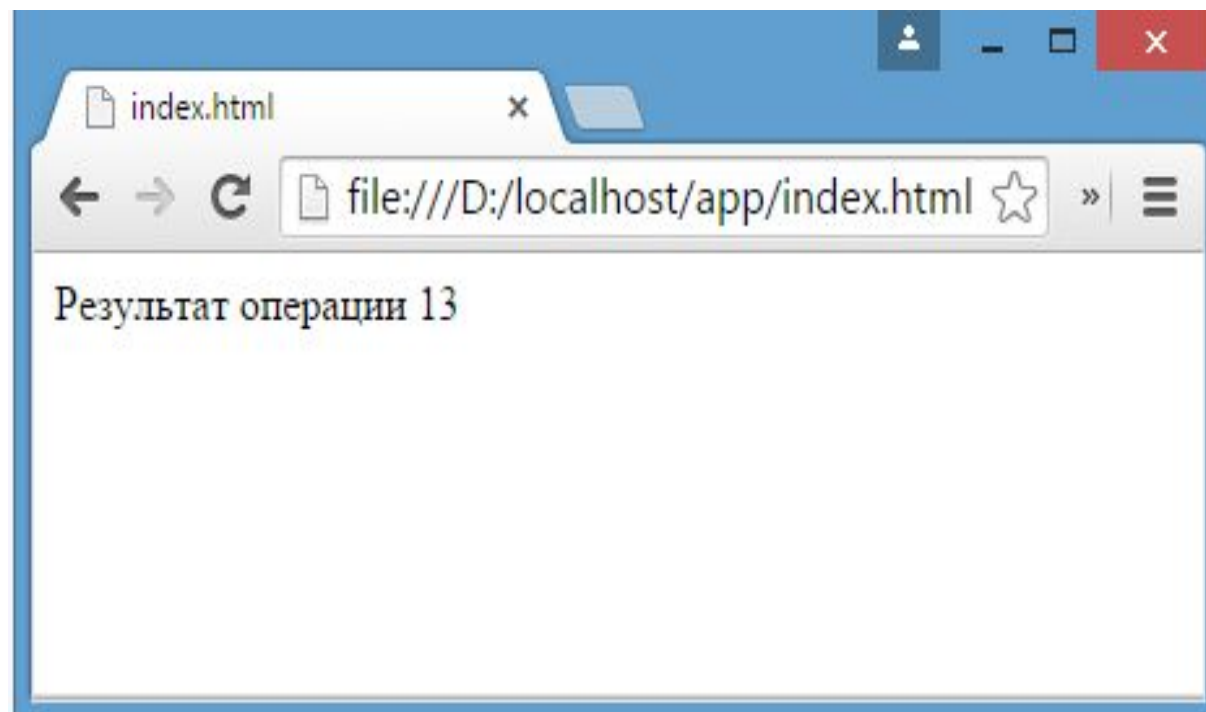
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
<title>JavaScript</title>
</head>
<body>
  <h2>Первая программа на JavaScript</h2>
  <script src="js/myscript.js"></script>
</body>
</html>
```

# Консоль браузера, console.log и document.write

## ❑ Метод document.write

- Также на начальном этапе нам может быть полезен метод `document.write()`, который пишет информацию на веб-страницу.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
<title>JavaScript</title>
</head>
<body>
  <h2>Первая программа на JavaScript</h2>
  <script>
    var a = 5 + 8;
    document.write("Результат операции ");
    document.write(a);
  </script>
</body>
</html>
```



# Основы javascript

## Переменные и константы

- Для хранения данных в программе используются **переменные**. Переменные предназначены для хранения каких-нибудь временных данных или таких данных, которые в процессе работы могут менять свое значение. Для создания переменных применяются ключевые слова **var** и **let**. Например, объявим переменную `myIncome`:

```
1 var myIncome;  
2 // другой вариант  
3 let myIncome2;
```

- Каждая переменная имеет имя. Имя представляет собой произвольный набор алфавитно-цифровых символов, знака подчеркивания (`_`) или знака доллара (`$`), причем названия не должны начинаться с цифровых символов. То есть мы можем использовать в названии буквы, цифры, подчеркивание. Однако все остальные символы запрещены.



- правильные названия переменных:

```
1 $commision
2 someVariable
3 product_Store
4 income2
5 myIncome_from_deposit
```

- Следующие названия являются некорректными и не могут использоваться:

```
1 222lol
2 @someVariable
3 my%percent
```

- Через запятую можно определить сразу несколько переменных.

```
1 var myIncome, procent, sum;  
2 let a, b, c;
```

```
1 var income = 300;  
2 income = 400;  
3 console.log(income);  
4  
5 let price = 76;  
6 price = 54;  
7 console.log(price);
```

## Константы

С помощью ключевого слова **const** можно определить **константу**, которая, как и переменная, хранит значение, однако это значение не может быть изменено.

```
1 const rate = 10;
```

- Если мы попробуем изменить ее значение, то мы столкнемся с ошибкой:

```
1 const rate = 10;  
2 rate = 23; // ошибка, rate - константа, поэтому мы не можем изменить ее значение
```

## Типы данных

- Все используемые данные в javascript имеют определенный тип. В JavaScript имеется пять примитивных типов данных:
  - String: представляет строку
  - Number: представляет числовое значение
  - Boolean: представляет логическое значение true или false
  - undefined: указывает, что значение не установлено
  - null: указывает на неопределенное значение

# Числовые данные

```
1 var x = 45;  
2 var y = 23.897;
```

## □ Строки

- Тип `string` представляет строки, то есть такие данные, которые заключены в кавычки. Например, "Привет мир". Причем мы можем использовать как двойные, так и одинарные кавычки: "Привет мир" и 'Привет мир'. Единственным ограничением является то, что тип закрывающей кавычки должен быть тот же, что и тип открывающей, то есть либо обе двойные, либо обе одинарные.

```
1 var companyName1 = "Бюро 'Рога и копыта'";  
2 var companyName2 = 'Бюро "Рога и копыта"';
```

# Тип Boolean

- Тип Boolean представляет булевы или логические значения true и false (то есть да или нет):

```
1 var isAlive = true;
2 var isDead = false;
```

- **null и undefined**
- Нередко возникает путаница между null и undefined. Итак, когда мы только определяем переменную без присвоения ей начального значения, она представляет тип undefined:

```
1 var isAlive;
2 console.log(isAlive); // выведет undefined
```

```
1 var isAlive;
2 console.log(isAlive); // undefined
3 isAlive = null;
4 console.log(isAlive); // null
5 isAlive = undefined; // снова установим тип undefined
6 console.log(isAlive); // undefined
```

# object

- Тип object представляет сложный объект. Простейшее определение объекта представляют фигурные скобки:

```
1 var user = {name: "Tom", age:24};
2 console.log(user.name);
```

- **Оператор typeof**

- **С помощью оператора typeof можно получить тип**

```
1 var name = "Tom";
2 console.log(typeof name); // string
3
4 var income = 45.8;
5 console.log(typeof income); // number
6
7 var isEnabled = true;
8 console.log(typeof isEnabled); // boolean
9
10 var undefVariable;
11 console.log(typeof undefVariable); // undefined
```

# Операции с переменными

- Математические операции

## Сложение:

```
1 var x = 10;  
2 var y = x + 50;
```

## Вычитание:

```
1 var x = 100;  
2 var y = x - 50;
```

## Умножение:

```
1 var x = 4;  
2 var y = 5;  
3 var z = x * y;
```

## Деление:

```
1 var x = 40;  
2 var y = 5;  
3 var z = x / y;
```

## Инкремент:

```
1 var x = 5;  
2 x++; // x = 6
```

- Оператор инкремента ++ увеличивает переменную на единицу. Существует префиксный инкремент, который сначала увеличивает переменную на единицу, а затем возвращает ее значение. И есть постфиксный инкремент, который сначала возвращает значение переменной, а затем увеличивает его на единицу:

```
1 // префиксный инкремент
2 var x = 5;
3 var z = ++x;
4 console.log(x); // 6
5 console.log(z); // 6
6
7 // постфиксный инкремент
8 var a = 5;
9 var b = a++;
10 console.log(a); // 6
11 console.log(b); // 5
```



- **Декремент** уменьшает значение переменной на единицу. Также есть префиксный и постфиксный декремент:

```
1 // префиксный декремент
2 var x = 5;
3 var z = --x;
4 console.log(x); // 4
5 console.log(z); // 4
6
7 // постфиксный декремент
8 var a = 5;
9 var b = a--;
10 console.log(a); // 4
11 console.log(b); // 5
```

# • Операции присваивания

- =

Приравнивает переменной определенное значение: `var x = 5;`

- +=

Сложение с последующим присвоением результата. Например:

```
1 var a = 23;  
2 a += 5; // аналогично a = a + 5  
3 console.log(a); // 28
```

- -=

Вычитание с последующим присвоением результата. Например:

```
1 var a = 28;  
2 a -= 10; // аналогично a = a - 10  
3 console.log(a); // 18
```

- \*=

Умножение с последующим присвоением результата:

```
1 var x = 20;  
2 x *= 2; // аналогично x = x * 2  
3 console.log(x); // 40
```

- /=

Деление с последующим присвоением результата:

```
1 var x = 40;  
2 x /= 4; // аналогично x = x / 4  
3 console.log(x); // 10
```

# Преобразования данных

- Возникает необходимость преобразовать одни данные в другие

```
1 var number1 = "46";  
2 var number2 = "4";  
3 var result = number1 + number2;  
4 console.log(result); //464
```

- Обе переменных представляют строки, а точнее строковые представления чисел.
- В этом случае мы можем использовать операции преобразования. Для преобразования строки в число применяется функция `parseInt()`:

```
1 var number1 = "46";
2 var number2 = "4";
3 var result = parseInt(number1) + parseInt(number2);
4 console.log(result); // 50
```

- Для преобразования строк в дробные числа применяется функция **parseFloat()**:

```
1 var number1 = "46.07";
2 var number2 = "4.98";
3 var result = parseFloat(number1) + parseFloat(number2);
4 console.log(result); //51.05
```

- Строка может иметь смешанное содержимое, например, **"123hello"** . Но метод **parseInt()** все равно попытается выполнить преобразование:

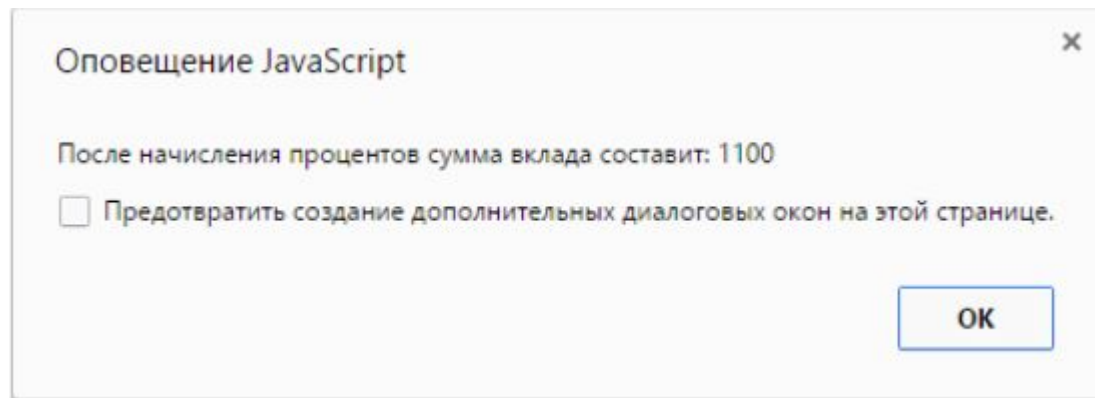
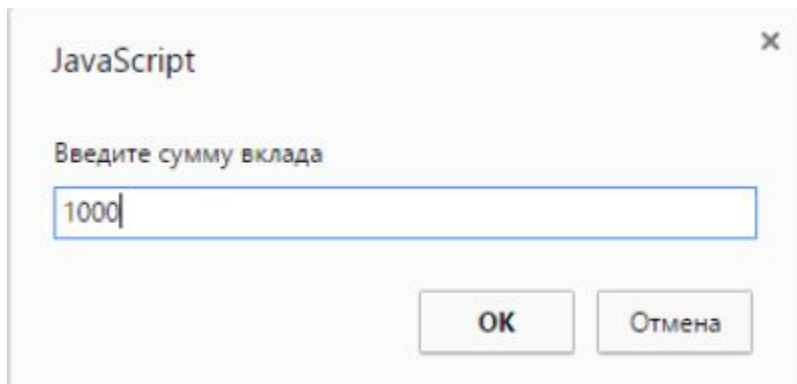
```
1 var num1 = "123hello";
2 var num2 = parseInt(num1);
3 console.log(num2); // 123
```

- Если методу не удастся выполнить преобразование, то он возвращает значение **NaN** (Not a Number), которое говорит о том, что строка не представляет число и не может быть преобразована.
- С помощью специальной функции **isNaN()** можно проверить, представляет ли строка число. Если строка не является числом, то функция возвращает **true**, если это число - то **false**:

```
1 var num1 = "javascript";
2 var num2 = "22";
3 var result = isNaN(num1);
4 console.log(result); // true - num1 не является числом
5
6 result = isNaN(num2);
7 console.log(result); // false - num2 - это число
```

```
<script>
  var strSum = prompt("Введите сумму вклада", 1000);
  var strPercent = prompt("Введите процентную ставку", 10);
  var sum = parseInt(strSum);
  var procent = parseInt(strPercent);
  sum = sum + sum * procent / 100;
  alert("После начисления процентов сумма вклада составит: " + sum);
</script>
```

- С помощью функции **prompt()** в браузере выводится диалоговое окно с предложением ввести некоторое значение. Второй аргумент в этой функции указывает на значение, которое будет использоваться по умолчанию.



# Массивы

- Для работы с наборами данных предназначены массивы. Для создания массива применяется выражение **new Array()**:

```
1 var myArray = new Array();
```

- Существует также более короткий способ инициализации массива:

```
var myArray = [];
```

- В данном случае мы создаем пустой массив. Но можно также добавить в него начальные данные:

```
1 var people = ["Tom", "Alice", "Sam"];  
2 console.log(people);
```

В этом случае в массиве `myArray` будет три элемента.

Графически его можно пред

Индекс	Элемент
0	Tom
1	Alice
2	Sam

ИК:



- Для обращения к отдельным элементам массива используются индексы. Отсчет начинается с нуля, то есть первый элемент будет иметь индекс 0, а последний - 2:

```
1 var people = ["Tom", "Alice", "Sam"];
2 console.log(people[0]); // Tom
3 var person3 = people[2]; // Sam
4 console.log(person3); // Sam
```

- Если мы попробуем обратиться к элементу по индексу больше размера массива, то мы получим undefined:

```
1 var people = ["Tom", "Alice", "Sam"];
2 console.log(people[7]); // undefined
```

- Также по индексу осуществляется установка значений для элементов массива:

```
1 var people = ["Tom", "Alice", "Sam"];
2 console.log(people[0]); // Tom
3 people[0] = "Bob";
4 console.log(people[0]); // Bob
```



# Многомерные массивы

- Массивы могут быть одномерными и многомерными. Каждый элемент в многомерном массиве может представлять собой отдельный массив.

```
1 var numbers1 = [0, 1, 2, 3, 4, 5 ]; // одномерный массив
2 var numbers2 = [[0, 1, 2], [3, 4, 5] ]; // двумерный массив
```

- Визуально оба массива можно представить следующим образом:

Одномерный массив numbers1



Двухмерный массив numbers2



- Поскольку массив `numbers2` двумерный, он представляет собой простую таблицу. Каждый его элемент может представлять отдельный массив.

```
1 var people = [  
2     ["Tom", 25, false],  
3     ["Bill", 38, true],  
4     ["Alice", 21, false]  
5 ];  
6  
7 console.log(people[0]); // ["Tom", 25, false]  
8 console.log(people[1]); // ["Bill", 38, true]
```

```
1 var people = [  
2     ["Tom", 25, false],  
3     ["Bill", 38, true],  
4     ["Alice", 21, false]  
5 ];  
6 people[0][1] = 56; // присваиваем отдельное значение  
7 console.log(people[0][1]); // 56  
8  
9 people[1] = ["Bob", 29, false]; // присваиваем массив  
10 console.log(people[1][0]); // Bob
```

- элемент `people[0][1]` будет ссылаться на ячейку таблицы, которая находится на пересечении первой строки и второго столбца (первая размерность - 0 - строка, вторая размерность - 1 - столбец).

# Условные конструкции

- Условные конструкции позволяют выполнить те или иные действия в зависимости от определенных условий.
- **Выражение if**
- Конструкция if проверяет некоторое условие и если это условие верно, то выполняет некоторые действия.

```
1 if(условие) действия;
```

```
1 var income = 100;  
2 if(income > 50) alert("доход больше 50");
```

- Здесь в конструкции if используется следующее условие: `income > 50`. Если это условие возвращает `true`, то есть переменная `income` имеет значение больше 50, то браузер отображает сообщение. Если же значение `income` меньше 50, то никакого сообщения не отображается.
- Если необходимо выполнить по условию набор инструкций, то они помещаются в блок из фигурных скобок:

```
1 var income = 100;
2 if(income > 50){
3
4     var message = "доход больше 50";
5     alert(message);
6 }
```

```
1 var income = 100;
2 var age = 19;
3 if(income < 150 && age > 18){
4
5     var message = "доход больше 50";
6     alert(message);
7 }
```

- Конструкция `if` позволяет проверить наличие значения.  
Например:

```
1 var myVar = 89;
2 if(myVar){
3     // действия
4 }
```

- В конструкции `if` мы также можем использовать блок `else`.  
Данный блок содержит инструкции, которые выполняются, если условие после `if` ложно, то есть равно `false`:

```
1 var age = 17;
2 if(age >= 18){
3
4     alert("Вы допущены к программе кредитования");
5 }
6 else{
7     alert("Вы не можете участвовать в программе, так как возраст меньше 18");
8 }
```

- С помощью конструкции **else if** мы можем добавить альтернативное условие к блоку **if**:

```
1 var income = 300;
2 if(income < 200){
3
4     alert("Доход ниже среднего");
5 }
6 else if(income >= 200 && income <= 400){
7
8     alert("Средний доход");
9 }
10 else{
11
12     alert("Доход выше среднего");
13 }
```

```
if(income < 200){
    alert("Доход ниже среднего");
}
else if(income >= 200 && income < 300){
    alert("Чуть ниже среднего");
}
else if(income >= 300 && income < 400){
    alert("Средний доход");
}
else{
    alert("Доход выше среднего");
}
```

- В данном случае выполнится блок **else if**. При необходимости мы можем использовать несколько блоков **else if** с разными условиями:

# True или false

- В javascript любая переменная может применяться в условных выражениях, но не любая переменная представляет тип boolean.
- **undefined**  
Возвращает false
- **null**  
Возвращает false
- **Boolean**  
Если переменная равна false, то возвращается false. Соответственно если переменная равна true, то возвращается true
- **Number**  
Возвращает false, если число равно 0 или NaN (Not a Number), в остальных случаях возвращается true



- Например, следующая переменная будет возвращать false:

```
1 var x = NaN;
2 if(x){ // false
3
4 }
5
```

- **String**

Возвращает false, если переменная равна пустой строке, то есть ее длина равна 0, в остальных случаях возвращается

true

```
1 var y = ""; // false - так как пустая строка
2 var z = "javascript"; // true - строка не пустая
3
```

- **Object**

Всегда возвращает true

```
1 var user = {name:"Tom"}; // true
2 var isEnabled = new Boolean(false) // true
3 var car = {} // true
```

# Конструкция switch..case

- Конструкция switch..case является альтернативой использованию конструкции if..else if..else и также позволяет обработать сразу несколько условий:

```
1  var income = 300;
2  switch(income){
3
4      case 100 :
5          console.log("Доход равен 100");
6          break;
7      case 200 :
8          console.log("Доход равен 200");
9          break;
10     case 300 :
11         console.log("Доход равен 300");
12         break;
13 }
```

- После ключевого слова **switch** в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора **case**. И если совпадение будет найдено, то будет выполняться определенный блок **case**.
- В конце каждого блока **case** ставится оператор **break**, чтобы избежать выполнения других блоков.
- Если мы хотим также обработать ситуацию, когда совпадения не будет найдено, то можно добавить блок **default**.

```
1 var income = 300;
2 switch(income){
3
4     case 100 :
5         console.log("Доход равен 100");
6         break;
7     case 200 :
8         console.log("Доход равен 200");
9         break;
```

```
10     case 300 :
11         console.log("Доход равен 300");
12         break;
13     default:
14         console.log("Доход неизвестной величины");
15         break;
16 }
```

# Циклы

- Циклы позволяют в зависимости от определенных условий выполнять некоторое действие множество раз. В JavaScript имеются следующие виды циклов:

- **for**

- **for..in**

- **for..of**

- **while**

- **do..while**

```
1  for ([инициализация счетчика]; [условие]; [изменение счетчика]){  
2  
3      // действия  
4  }
```

## ❖ Цикл for

- Цикл for имеет следующее формальное определение:

Например, используем цикл for для перебора элементов массива:

```
1 var people = ["Tom", "Alice", "Bob", "Sam"];
2 for(var i = 0; i<people.length; i++){
3
4     console.log(people[i]);
5 }
```

Первая часть объявления цикла - `var i = 0` - создает и инициализирует счетчик - переменную `i`. И перед выполнением цикла ее значение будет равно 0. По сути это то же самое, что и объявление переменной. Вторая часть - условие, при котором будет выполняться цикл. В данном случае цикл будет выполняться, пока значение `i` не достигнет величины, равной длине массива `people`. Получить длину массива можно с помощью свойства `length`: `people.length`. Третья часть - приращение счетчика на единицу. И так как в массиве 4 элемента, то блок цикла сработает 4 раза, пока значение `i` не станет равным `people.length` (то есть 4). И каждый раз это значение будет увеличиваться на 1. Каждое отдельное повторение цикла называется итерацией. Таким образом, в данном случае сработают 4 итерации. А с помощью выражения `people[i]` мы сможем получить элемент массива для его последующего вывода в

- **for..in**

- Цикл **for..in** предназначен для перебора массивов и объектов. Его формальное определение:

```
1 for (индекс in массив) {  
2     // действия  
3 }
```

- Например, переберем элементы массива:

```
1 var people = ["Tom", "Alice", "Bob", "Sam"];  
2 for(var index in people){  
3  
4     console.log(people[index]);  
5 }
```

- **Цикл for...of**

- Цикл **for...of** похож на цикл **for...in** и предназначен для перебора коллекций, например,

```
1 let users = ["Tom", "Bob", "Sam"];  
2 for(let val of users)  
3     console.log(val);
```

- **Цикл while**

- Цикл while выполняется до тех пор, пока некоторое условие истинно. Его формальное определение:

```
1 while(условие){
2
3     // действия
4 }
```

```
1 var people = ["Tom", "Alice", "Bob", "Sam"];
2 var index = 0;
3 while(index < people.length){
4
5     console.log(people[index]);
6     index++;
7 }
```

- **do..while**

- В цикле do сначала выполняется код цикла, а потом происходит проверка условия в инструкции while. И пока это условие истинно, цикл повторяется. Например:

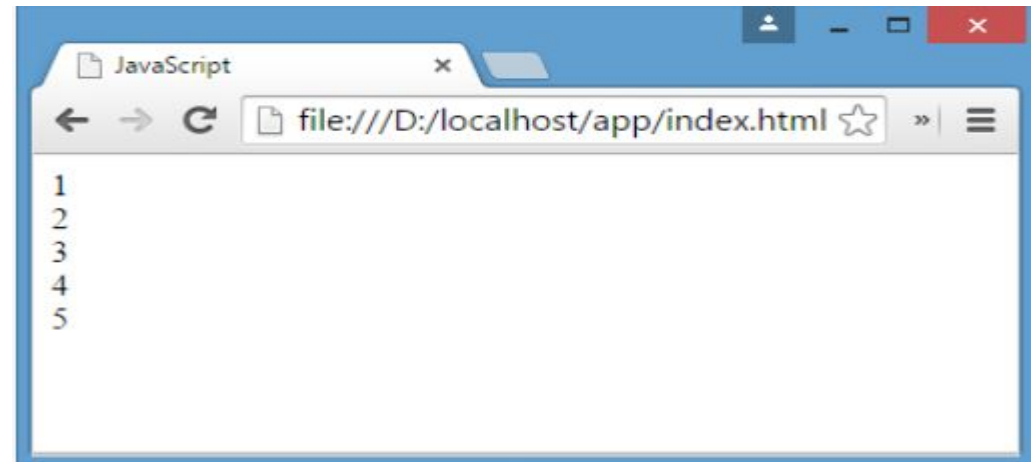
```
1 var x = 1;
2 do{
3     console.log(x * x);
4     x++;
5 }while(x < 10)
```



# Операторы continue и break

- Иногда бывает необходимо выйти из цикла до его завершения. В этом случае мы можем воспользоваться оператором **break**:

```
1 var array = [ 1, 2, 3, 4, 5, 12, 17, 6, 7 ];
2 for (var i = 0; i < array.length; i++)
3 {
4     if (array[i] > 10)
5         break;
6     document.write(array[i] + "<br>");
7 }
```

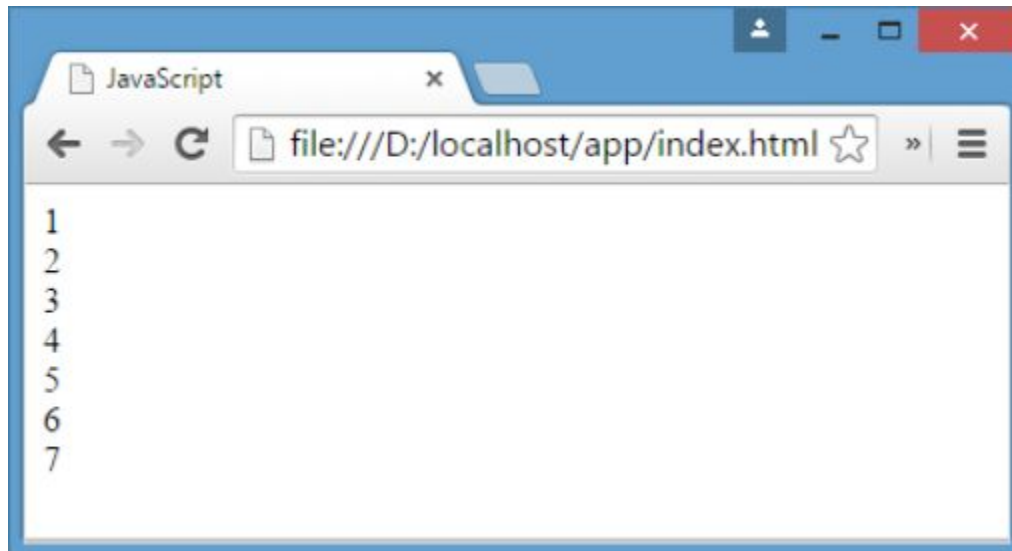


- Данный цикл перебирает все элементы массива, однако последние четыре элемента не будут выведены в браузере, поскольку проверка `if (array[i] > 10)` прервет выполнение цикла с помощью оператора `break`, когда перебор массива дойдет до элемента 12.



- Если нам надо просто пропустить итерацию, но не выходить из цикла, мы можем применять оператор **continue**:

```
1 var array = [ 1, 2, 3, 4, 5, 12, 17, 6, 7 ];
2 for (var i = 0; i < array.length; i++)
3 {
4     if (array[i] > 10)
5         continue;
6     document.write(array[i] + "</br>");
7 }
```



# Функциональное программирование

## Функции

- Функции представляют собой набор инструкций, выполняющих определенное действие или вычисляющих определенное значение.
- Синтаксис определения функции:

```
1 function имя_функции([параметр [, ...]]){  
2  
3     // Инструкции  
4 }
```

- Определение функции начинается с ключевого слова **function**, после которого следует имя функции. Наименование функции подчиняется тем же правилам, что и наименование переменной: оно может содержать только цифры, буквы, символы подчеркивания и доллара (\$) и должно начинаться с буквы, символа подчеркивания или доллара.
- После имени функции в скобках идет перечисление параметров. Даже если параметров у функции нет, то просто идут пустые скобки. Затем в фигурных скобках идет тело функции, содержащее набор инструкций.
- Определим простейшую функцию:

```
1 function display(){  
2  
3     document.write("функция в JavaScript");  
4 }
```

- Также мы можем динамически присваивать функции для переменной:

```
1 function goodMorning(){
2
3     document.write("Доброе утро");
4 }
5 function goodEvening(){
6
7     document.write("Добрый вечер");
8 }
9 var message = goodMorning;
10 message(); // Доброе утро
11 message = goodEvening;
12 message(); // Добрый вечер
```

# Параметры функции

- Рассмотрим передачу параметров:

```
1 function display(x){ // определение функции
2
3     var z = x * x;
4     document.write(x + " в квадрате равно " + z);
5 }
6
7 display(5); // вызов функции
```

```
1 function display(x, y){
2
3     if(y === undefined) y = 5;
4     if(x === undefined) x = 8;
5     let z = x * y;
6     console.log(z);
7 }
8 display(); // 40
9 display(6); // 30
10 display(6, 4) // 24
```

- **Необязательные параметры**
- Если для параметров не передается значение, то по умолчанию они имеют значение "undefined".

- Есть и другой способ определения значения для параметров по умолчанию:

```
1 function display(x = 5, y = 10){
2     let z = x * y;
3     console.log(z);
4 }
5 display(); // 50
6 display(6); // 60
7 display(6, 4) // 24
```

- При необходимости мы можем получить все переданные параметры через глобально доступный массив **arguments**:

```
1 function display(){
2     var z = 1;
3     for(var i=0; i<arguments.length; i++)
4         z *= arguments[i];
5     console.log(z);
6 }
7 display(6); // 6
8 display(6, 4) // 24
9 display(6, 4, 5) // 120
```

# Результат функции

- Функция может возвращать результат. Для этого используется оператор `return`:

```
1  var y = 5;  
2  var z = square(y);  
3  document.write(y + " в квадрате равно " + z);  
4  
5  function square(x) {  
6      return x * x;  
7  }
```

## Функции в качестве параметров

- Функции могут выступать в качестве параметров других функций:

```
1 function sum(x, y){
2     return x + y;
3 }
4
5 function subtract(x, y){
6     return x - y;
7 }
8
9 function operation(x, y, func){
10
11     var result = func(x, y);
12     console.log(result);
13 }
14
15 console.log("Sum");
16 operation(10, 6, sum); // 16
17
18 console.log("Subtract");
19 operation(10, 6, subtract); // 4
```



# Область видимости переменных

- Глобальные переменные
- Все переменные, которые объявлены вне функций, являются глобальными:

```
var x = 5;  
let d = 8;  
function displaySquare(){  
  
    var z = x * x;  
    console.log(z);  
}
```

- Здесь переменные `x` и `d` являются глобальными. Они доступны из любого места программы.
- А вот переменная `z` глобальной не является, так как она определена внутри функции.

# Локальные переменные

- Переменная, определенная внутри функции, является локальной.

```
1 function displaySquare(){
2
3     var z = 10;
4     console.log(z);
5
6     let b = 8;
7     console.log(b);
8 }
```

- Переменные `z` и `b` являются локальными, они существуют только в пределах функции.

# Замыкания

- **Замыкание (closure)** представляют собой конструкцию, когда функция, созданная в одной области видимости, запоминает свое лексическое окружение даже в том случае, когда она выполняет вне своей области видимости.
- Замыкание технически включает три компонента:
  - внешняя функция, которая определяет некоторую область видимости и в которой определены некоторые переменные - лексическое окружение
  - переменные (лексическое окружение), которые определены во внешней функции

```
1 function outer(){           // внешняя функция
2     var n;                  // некоторая переменная
3     return inner(){        // вложенная функция
4         // действия с переменной n
5     }
6 }
```

эти переменные

- Рассмотрим замыкания на простейшем примере:

```
1 function outer(){
2     let x = 5;
3     function inner(){
4         x++;
5         console.log(x);
6     };
7     return inner;
8 }
9 let fn = outer(); //
10 // вызываем внутреннюю
11 fn(); // 6
12 fn(); // 7
13 fn(); // 8
```

# Переопределение функций

- Функции обладают возможностью для переопределения поведения. Переопределение происходит с помощью присвоения анонимной функции переменной, которая называется так же, как и переопределяемая функция:

```
1 function display(){
2     console.log("Доброе утро");
3     display = function(){
4         console.log("Добрый день");
5     }
6 }
7
8 display(); // Доброе утро
9 display(); // Добрый день
```

# Передача параметров по значению и по ссылке

- Передача параметров по значению
- Строки, числа, логические значения передаются в функцию по значению.

```
1 function change(x){
2     x = 2 * x;
3     console.log("x in change:", x);
4 }
5
6 var n = 10;
7 console.log("n before change:", n); // n before change: 10
8 change(n);                          // x in change: 20
9 console.log("n after change:", n);  // n after change: 10
```

- **Передача по ссылке**

- Объекты и массивы передаются по ссылке. То есть функция получает сам объект или массив, а не их копию.

```
1 function change(user){
2     user.name = "Tom";
3 }
4
5 var bob = {
6     name: "Bob"
7 };
8 console.log("before change:", bob.name); // Bob
9 change(bob);
10 console.log("after change:", bob.name); // Tom
```

# Стрелочные функции

- Стрелочные функции (arrow functions) представляют сокращенную версию обычных функций. Стрелочные функции образуются с помощью знака стрелки (`=>`), перед которым в скобках идут параметры функции, а после - собственно тело функции.

```
1 let sum = (x, y) => x + y;  
2 let a = sum(4, 5);      // 9  
3 let b = sum(10, 5);    // 15
```

```
1 let sum = (x, y) => console.log(x + y);  
2 sum(4, 5);             // 9  
3 sum(10, 5);           // 15
```

- Но также в качестве тела функции может применяться выражение, которое ничего не возвращает и просто выполняет некоторое действие:



- Если функция принимает один параметр, то скобки вокруг него можно опустить:

```
1 var square = n => n * n;  
2  
3 console.log(square(5)); // 25  
4 console.log(square(6)); // 36  
5 console.log(square(-7)); // 49
```

# Объектно-ориентированное программирование

## ❖ Объекты

- **Объект** - может хранить свойства, которые описывают его **состояние**, и методы, которые описывают его **поведение**.

## ❖ Создание нового объекта

- Первый способ заключается в использовании конструктора `Object()`.

```
1 | var user = new Object();
```

- Второй способ создания объекта представляет использование фигурных скобок:

```
1 | var user = {};
```

## • Свойства объекта

- После создания объекта мы можем определить в нем свойства. Чтобы определить свойство, надо после названия объекта через точку указать имя свойства и присвоить ему значение.

```
1 var user = {};  
2 user.name = "Tom";  
3 user.age = 26;
```

```
1 console.log(user.name);  
2 console.log(user.age);
```

```
1 var user = {  
2  
3     name: "Tom",  
4     age: 26  
5 };
```

```
1 var name = "Tom";  
2 var age = 34;  
3 var user = {name, age};  
4 console.log(user.name); // Tom  
5 console.log(user.age); // 34
```

ценный способ определения

# Методы объекта

- Методы объекта определяют его поведение или действия, которые он производит. Методы представляют собой функции.

```
1  var user = {};  
2  user.name = "Tom";  
3  user.age = 26;  
4  user.display = function(){  
5  
6      console.log(user.name);  
7      console.log(user.age);  
8  };  
9  
10 // вызов метода  
11 user.display();
```

- Также методы могут определяться непосредственно при определении объекта:

```
1  var user = {  
2  
3      name: "Tom",  
4      age: 26,  
5      display: function(){  
6  
7          console.log(this.name);  
8          console.log(this.age);  
9      }  
10 };
```

- Чтобы обратиться к свойствам или методам объекта внутри этого объекта, используется ключевое слово **this**. Оно означает ссылку на текущий объект.

# Вложенные объекты и массивы в объектах

- Одни объекты могут содержать в качестве свойств другие объекты.

```
1 var country = {
2
3   name: "Германия",
4   language: "немецкий",
5   capital:{
6
7     name: "Берлин",
8     population: 3375000,
9     year: 1237
10  }
11 };
12 console.log("Столица: " + country.capital.name); // Берлин
13 console.log("Население: " + country["capital"]["population"]); // 3375000
14 console.log("Год основания: " + country.capital["year"]); // 1237
```

# Объекты в функциях

- Функции могут возвращать значения. Но эти значения не обязательно должны представлять примитивные данные - числа, строки, но также могут быть сложными объектами

```
1 function createUser(pName, pAge) {
2     return {
3         name: pName,
4         age: pAge,
5         displayInfo: function() {
6             document.write("Имя: " + this.name + " возраст: " + this.age + "<br/>");
7         }
8     };
9 };
10 var tom = createUser("Tom", 26);
11 tom.displayInfo();
12 var alice = createUser("Alice", 24);
13 alice.displayInfo();
```

# Инкапсуляция

- Инкапсуляция является одним из ключевых понятий объектно-ориентированного программирования и представляет сокрытие состояния объекта от прямого доступа извне. По умолчанию все свойства объектов являются публичными, общедоступными, и мы к ним можем обратиться из любого места программы.

## □ Наследование

JavaScript поддерживает наследование, то позволяет нам при создании новых типов объектов при необходимости унаследовать их функционал от уже существующих.



# Классы

- Класс представляет описание объекта, его состояния и поведения, а объект является конкретным воплощением или экземпляром класса. Для определения класса используется ключевое слово **class**:

```
1 class Person{  
2 }
```

- Также мы можем определить в классе свои конструкторы. Также класс может содержать свойства и методы:

```
class Person{  
  constructor(name, age){  
    this.name = name;  
    this.age = age;  
  }  
  display(){  
    console.log(this.name, this.age);  
  }  
}
```

```
let tom = new Person("Tom", 34);  
tom.display();           // Tom 34  
console.log(tom.name);  // Tom
```

# Операторы сравнения

- Для проверки условия используются операторы сравнения. Операторы сравнения сравнивают два значения и возвращают значение **true** или **false**:
- **==** Оператор равенства сравнивает два значения, и если они равны, возвращает true, иначе возвращает false: `x == 5`
- **===** Оператор тождественности также сравнивает два значения и их тип, и если они равны, возвращает true, иначе возвращает false: `x === 5`
- **!=** Сравнивает два значения, и если они не равны, возвращает true, иначе возвращает false: `x != 5`
- **!==** Сравнивает два значения и их типы, и если они не равны, возвращает true, иначе возвращает false: `x !== 5`
- **>** Сравнивает два значения, и если первое больше второго, то возвращает true, иначе возвращает false: `x > 5`

- < Сравнивает два значения, и если первое меньше второго, то возвращает true, иначе возвращает false:  $x < 5$
- <= Сравнивает два значения, и если первое меньше или равно второму, то возвращает true, иначе возвращает false:  $x \leq 5$
- >= Сравнивает два значения, и если первое больше или равно второму, то возвращает true, иначе возвращает false:  $x \geq 5$
- Все операторы довольно просты, наверное, за исключением оператора равенства и оператора тождественности. Они оба сравнивают два значения, но оператор тождественности также принимает во внимание и тип значения.

```
1 var income = 100;
2 var strIncome = "100";
3 var result = income == strIncome;
4 console.log(result); //true
```

```
1 var income = 100;
2 var strIncome = "100";
3 var result = income === strIncome;
4 console.log(result); // false
```

- Но оператор тождественности возвратит в этом случае false, так как данные имеют разные тип. Аналогично работают операторы неравенства != и !==.

# Логические операции

- Логические операции применяются для объединения результатов двух операций сравнения. В JavaScript есть следующие логические операции:
- **&&** Возвращает true, если обе операции сравнения возвращают true, иначе возвращает false:

```
1 var income = 100;  
2 var percent = 10;  
3 var result = income > 50 && percent < 12;  
4 console.log(result); //true
```

- `||` Возвращает `true`, если хотя бы одна операция сравнения возвращают `true`, иначе возвращает `false`:

```
1 var income = 100;
2 var isDeposit = true;
3 var result = income > 50 || isDeposit == true;
4 console.log(result); //true
```

- `!` Возвращает `true`, если операция сравнения возвращает `false`.

```
1 var income = 100;
2 var result1 = !(income > 50);
3 console.log(result1); // false, так как income > 50 возвращает true
```

# Операции со строками

- Строки могут использовать оператор + для объединения.

```
1 var name = "Том";  
2 var surname = "Сойер"  
3 var fullname = name + " " + surname;  
4 console.log(fullname); //Том Сойер
```

- Если одно из выражений представляет строку, а другое - число, то число преобразуется к строке и выполняется операция объединения строк.

```
1 var name = "Том";  
2 var fullname = name + 256;  
3 console.log(fullname); //Том256
```

# Тернарная операция

- Тернарная операция состоит из трех операндов и имеет следующее определение: [первый операнд - условие] ? [второй операнд] : [третий операнд]. В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно true, то возвращается второй операнд; если условие равно false, то третий.

```
1 var a = 1;
2 var b = 2;
3 var result = a < b ? a + b : a - b;
4 console.log(result); // 3
```

- Если значение переменной *a* меньше значения переменной *b*, то переменная *result* будет равняться *a + b*. Иначе значение *result* будет равняться *a - b*.



# Функция как объект. Методы call и apply

- Среди методов надо отметить методы **call()** и **apply()**.
- Метод **call()** вызывает функцию с указанным значением **this** и аргументами:

```
1 function add(x, y){
2
3     return x + y;
4 }
5 var result = add.call(this, 3, 8);
6
7 console.log(result); // 11
```

- **this** указывает на объект, для которого вызывается функция - в данном случае это глобальный объект **window**. После **this** передаются значения для параметров.



- При передаче объекта через первый параметр, мы можем сослаться на него через ключевое слово `this`:

```
1 function User (name, age) {
2     this.name = name;
3     this.age = age;
4 }
5 var tom = new User("Том", 26);
6 function display(){
7     console.log("Ваше имя: " + this.name);
8 }
9 display.call(tom); // Ваше имя: Том
```

- Если нам не важен объект, для которого вызывается функция, то можно передать значение `null`:

```
1 function add(x, y){
2
3     return x + y;
4 }
5 var result = add.call(null, 3, 8);
6
7 console.log(result); // 11
```

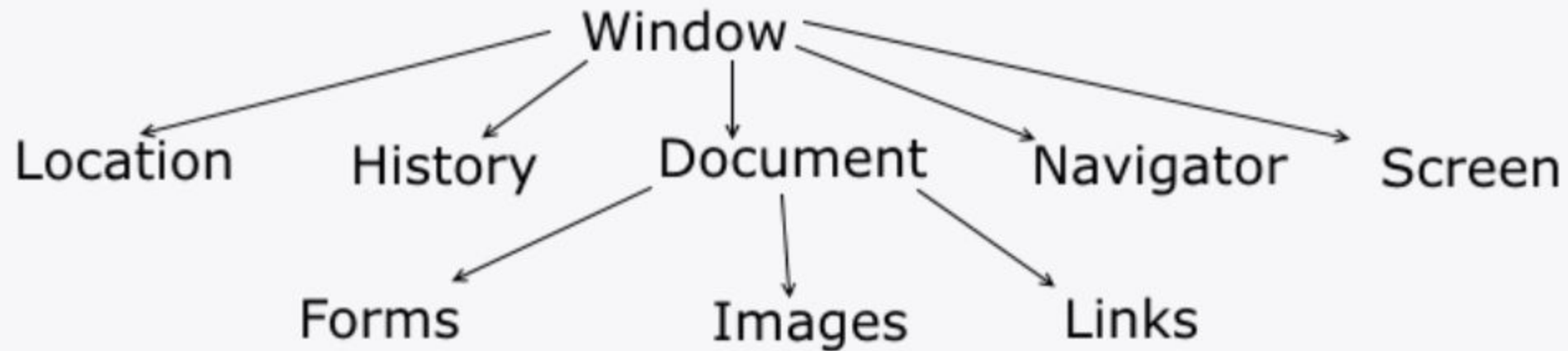
- На метод `call()` похож метод `apply()`, который также вызывает функцию и в качестве первого параметра также получает объект, для которого функция вызывается. Только теперь в качестве второго параметра передается массив аргументов:

```
1 function add(x, y){  
2  
3     return x + y;  
4 }  
5 var result = add.apply(null, [3, 8]);  
6  
7 console.log(result); // 11
```

# Работа с браузером и BOM

## Browser Object Model

- Большое значение в JavaScript имеет работа с веб-браузером и теми объектами, которые он предоставляет. Например, использование объектов браузера позволяет манипулировать элементами html, которые имеются на странице, или взаимодействовать с пользователем.
- Все объекты, через которые JavaScript взаимодействует с браузером, описываются таким понятием как **Browser Object Model** (Объектная Модель Браузера).
- Browser Object Model можно представить в виде следующей СХЕМЫ:



- В вершине находится главный объект - объект **window**, который представляет собой браузер. Этот объект в свою очередь включает ряд других объектов, в частности, объект **document**, который представляет отдельную веб-страницу, отображаемую в браузере.

# Управление окнами

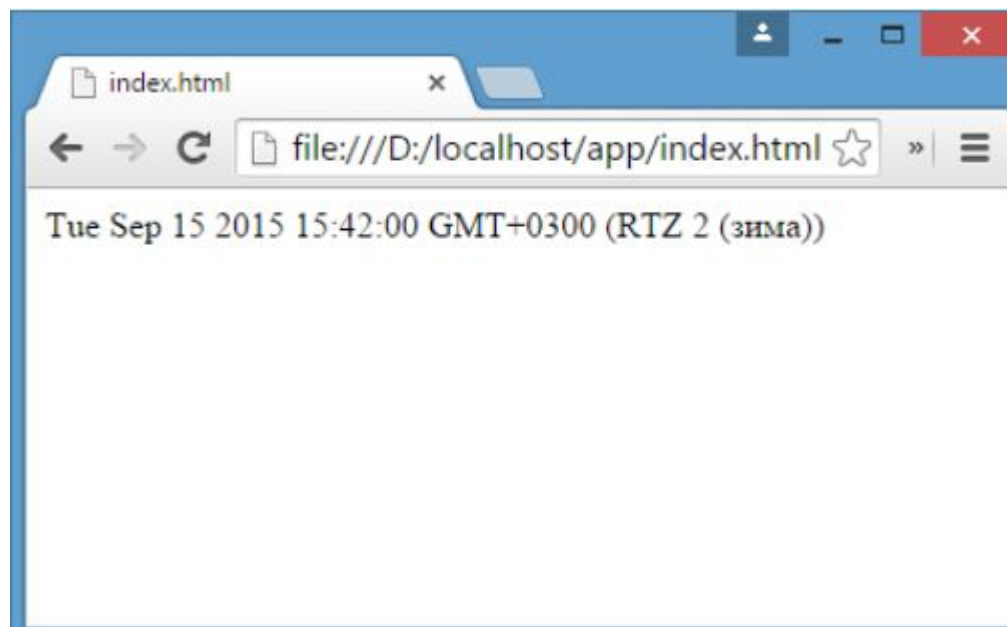
- Метод `confirm()` отображает окно с сообщением, в котором пользователь должен подтвердить действие двух кнопок ОК и Отмена. В зависимости от выбора пользователя метод возвращает `true` (если пользователь нажал ОК) или `false` (если пользователь нажал кнопку Отмены):

```
1 var result = confirm("Завершить выполнение программы?");
2 if(result===true)
3     document.write("Работа программы завершена");
4 else
5     document.write("Программа продолжает работать");
```

# Встроенные объекты

- **Объект Date. Работа с датами**
- Объект **Date** позволяет работать с датами и временем в JavaScript.

```
1 var currentDate = new Date();  
2 document.write(currentDate);
```



- способ состоит в передаче в конструктор Date дня, месяца и года:

```
1 var myDate = new Date("27 March 2008");  
2 // или так  
3 // var myDate = new Date("3/27/2008");  
4 document.write(myDate); // Thu Mar 27 2008 00:00:00 GMT+0300 (RTZ 2 (зима))
```

- **Получение даты и времени**

- Для получения различных компонентов даты применяется ряд методов:
- **getDate()**: возвращает день месяца
- **getDay()**: возвращает день недели (отсчет начинается с 0 - воскресенье, и последний день - 6 - суббота)
- **getMonth()**: возвращает номер месяца (отсчет начинается с нуля, то есть месяц с номер 0 - январь)
- **getFullYear()**: возвращает год
- **toDateString()**: возвращает полную дату в виде строки

```
var days = ["Воскресенье", "Понедельник", "Вторник", "Среда", "Четверг", "Пятница",  
var months = ["Январь", "Февраль", "Март", "Апрель", "Май", "Июнь",  
              "Июль", "Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"];  
  
var myDate = new Date();  
var fullDate = "Сегодня: " + myDate.getDate() + " " + months[myDate.getMonth()] +  
              " " + myDate.getFullYear() + ", " + days[myDate.getDay()];  
document.write(fullDate); // Сегодня: 18 Август 2015, Вторник
```

## Установка даты и времени

- Кроме задания параметров даты в конструкторе для установки мы также можем использовать дополнительные методы объекта Date:
- **setDate()**: установка дня в дате
- **setMonth()**: установка месяца (отсчет начинается с нуля, то есть месяц с номер 0 - январь)
- **setFullYear()**: устанавливает год



```
1 var days = ["Воскресенье", "Понедельник", "Вторник", "Среда", "Четверг", "Пятница", "Суббота"];
2 var months = ["Январь", "Февраль", "Март", "Апрель", "Май", "Июнь",
3              "Июль", "Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"];
4
5 var myDate = new Date();
6 myDate.setDate(15);
7 myDate.setMonth(6);
8 myDate.setYear(2013);
9
10 var fullDate = myDate.getDate() + " " + months[myDate.getMonth()] +
11               " " + myDate.getFullYear() + ", " + days[myDate.getDay()];
12 document.write(fullDate); // 15 Июль 2013, Понедельник
```

## Объект Math. Математические операции

- `min()` и `max()`
- Функции `min()` и `max()` возвращают соответственно минимальное и максимальное значение из набора чисел:

```
1 var max = Math.max(19, 45); // 45
2 var min = Math.min(33, 24); // 24
```

- Эти функции необязательно должны принимать два числа, в них можно передавать и большее количество чисел:

```
1 | var max = Math.max(1, 2, 3, -9, 46, -23); // 46
```

- **ceil()**

- Функция `ceil()` округляет число до следующего наибольшего целого числа:

```
1 | var x = Math.ceil(9.2); // 10  
2 | var y = Math.ceil(-5.9); // -5
```

- **floor()**

- Функция `floor()` округляет число до следующего наименьшего целого числа:

```
1 | var x = Math.floor(9.2); // 9  
2 | var y = Math.floor(-5.9); // -6
```

- **round()**

- Функция `round()` округляет число до следующего наименьшего целого числа, если его десятичная часть меньше 0.5. Если же десятичная часть равна или больше 0.5, то округление идет до ближайшего наибольшего целого

```
var x = Math.round(5.5); // 6
var y = Math.round(5.4); // 5
var z = Math.round(-5.4); // -5
var n = Math.round(-5.5); // -5
var m = Math.round(-5.6); // -6
console.log(x);
console.log(y);
console.log(z);
console.log(n);
```

- **pow()**

- Функция `pow()` возвращает число в определенной степени. Например, возведем число 2 в степень 3:

```
1 | var x = Math.pow(2, 3); // 8
```

# Объект Array. Работа с массивами

- **Копирование массива. slice()**
- Копирование массива может быть поверхностным или неглубоким (shallow copy) и глубоким (deep copy).
- При неглубоком копировании достаточно присвоить переменной значение другой переменной, которая хранит массив:

```
1 var users = ["Tom", "Sam", "Bill"];
2 console.log(users); // ["Tom", "Sam", "Bill"]
3 var people = users; // неглубокое копирование
4
5 people[1] = "Mike"; // изменяем второй элемент
6 console.log(users); // ["Tom", "Mike", "Bill"]
```

```
1 var users = ["Tom", "Sam", "Bill"];
2 console.log(users); // ["Tom", "Sam", "Bill"]
3 var people = users.slice(); // глубокое копирование
4
5 people[1] = "Mike"; // изменяем второй элемент
6 console.log(users); // ["Tom", "Sam", "Bill"]
7 console.log(people); // ["Tom", "Mike", "Bill"]
```

- Также метод `slice()` позволяет скопировать часть массива:

```
1 var users = ["Tom", "Sam", "Bill", "Alice", "Kate"];
2 var people = users.slice(1, 4);
3 console.log(people);           // ["Sam", "Bill", "Alice"]
```

- **push()**

- Метод `push()` добавляет элемент в конец массива:

```
1 var fruit = [];
2 fruit.push("яблоки");
3 fruit.push("груши");
4 fruit.push("сливы");
5 fruit.push("вишня", "абрикос");
6
7 console.log("В массиве fruit " + fruit.length + " элемента: <br/>");
8 console.log(fruit); // яблоки,груши,сливы,вишня,абрикос
```



- **pop()**

- Метод pop() удаляет последний элемент из массива:

```
1 var fruit = ["яблоки", "груши", "сливы"];
2
3 var lastFruit = fruit.pop(); // извлекаем из массива последний элемент
4 console.log(lastFruit );
5 console.log("В массиве fruit " + fruit.length + " элемента: <br/>");
6 for(var i=0; i <fruit.length; i++)
7     console.log(fruit[i] );
```

сливы

В массиве fruit 2 элемента:

яблоки

груши

- **shift()**

- Метод `shift()` извлекает и удаляет первый элемент из массива:

```
1 var fruit = ["яблоки", "груши", "сливы"];
2
3 var firstFruit = fruit.shift();
4 console.log(firstFruit );
5 console.log("В массиве fruit " + fruit.length + " элемента: <br/>");
6 for(var i=0; i <fruit.length; i++)
7     console.log(fruit[i] );
```

яблоки

В массиве fruit 2 элемента:

груши

сливы

- **unshift()**

- Метод `unshift()` добавляет новый элемент в начало массива:

```
1 var fruit = ["яблоки", "груши", "сливы"];
2 fruit.unshift("абрикосы");
3 console.log(fruit);
```

Вывод браузера:

```
абрикосы,яблоки,груши,сливы
```

- **Удаление элемента по индексу. splice()**

- Метод `splice()` удаляет элементы с определенного индекса. Например, удаление элементов с третьего индекса:

```
1 var users = ["Tom", "Sam", "Bill", "Alice", "Kate"];
2 var deleted = users.splice(3);
3 console.log(deleted);      // [ "Alice", "Kate" ]
4 console.log(users);       // [ "Tom", "Sam", "Bill" ]
```



- В данном случае удаление идет с начала массива. Если передать отрицательный индекс, то удаление будет производиться с конца массива.

```
1 var users = ["Tom", "Sam", "Bill", "Alice", "Kate"];
2 var deleted = users.splice(-1);
3 console.log(deleted); // [ "Kate" ]
4 console.log(users); // [ "Tom", "Sam", "Bill", "Alice" ]
```

- Дополнительная версия метода позволяет задать количество элементов для удаления. Например, удалим с первого индекса три элемента:

```
1 var users = ["Tom", "Sam", "Bill", "Alice", "Kate"];
2 var deleted = users.splice(1,3);
3 console.log(deleted); // [ "Sam", "Bill", "Alice" ]
4 console.log(users); // [ "Tom", "Kate" ]
```

- Еще одна версия метода `splice` позволяет вставить вместо удаляемых элементов новые элементы:

```
1 var users = ["Tom", "Sam", "Bill", "Alice", "Kate"];
2 var deleted = users.splice(1,3, "Ann", "Bob");
3 console.log(deleted);           // [ "Sam", "Bill", "Alice" ]
4 console.log(users);            // [ "Tom", "Ann", "Bob", "Kate" ]
```

- **`concat()`**

- Метод **`concat()`** служит для объединения массивов:

```
1 var fruit = ["яблоки", "груши", "сливы"];
2 var vegetables = ["помидоры", "огурцы", "картофель"];
3 var products = fruit.concat(vegetables);
4
5 for(var i=0; i < products.length; i++)
6     console.log(products[i] );
```

- **join()**

- Метод **join()** объединяет все элементы массива в одну строку:

```
1 var fruit = ["яблоки", "груши", "сливы", "абрикосы", "персики"];
2 var fruitString = fruit.join(", ");
3 console.log(fruitString);
```

- **sort()**

- Метод **sort()** сортирует массив по возрастанию:

```
1 var fruit = ["яблоки", "груши", "сливы", "абрикосы", "персики"];
2 fruit.sort();
3
4 for(var i=0; i < fruit.length; i++)
5     console.log(fruit[i] );
```

абрикосы

груши

персики

сливы

яблоки

- **reverse()**

- Метод **reverse()** переворачивает массив задом наперед:

```
1 var fruit = ["яблоки", "груши", "сливы", "абрикосы", "персики"];
2 fruit.reverse();
3
4 for(var i=0; i < fruit.length; i++)
5     console.log(fruit[i] );
```

персики

абрикосы

сливы

груши

яблоки

- Метод **repeat()** позволяет создать строку путем многократного повторения другой строки. Количество повторов передается в качестве аргумента:

```
1 let hello = "hello ";
2 console.log(hello.repeat(3)); // hello hello hello
```

## □ Удаление пробелов

- Для удаления начальных и конечных пробелов в строке используется метод `trim()`:

```
1 let hello = "  Привет Том  ";
2 let beforeLength = hello.length;
3 hello = hello.trim();
4 let afterLength = hello.length;
5 console.log("Длина строки до: ", beforeLength); // 15
6 console.log("Длина строки после: ", afterLength); // 10
```

# Таймеры

- **Функция `setTimeout`**

- Для однократного выполнения действий через промежуток времени предназначена функция `setTimeout()`. Она может принимать два параметра:

```
1 | var timerId = setTimeout(someFunction, period)
```

```
1 | function timerFunction() {  
2 |     document.write("выполнение функции setTimeout");  
3 | }  
4 | setTimeout(timerFunction, 3000);
```

- В данном случае через 3 секунды после загрузки страницы произойдет срабатывание функции `timerFunction`.



- Для остановки таймера применяется функция `clearTimeout()`.

```
1 function timerFunction() {  
2     document.write("выполнение функции setTimeout");  
3 }  
4 var timerId = setTimeout(timerFunction, 3000);  
5 clearTimeout(timerId);
```

## • Функция `setInterval`

- Функции `setInterval()` работают аналогично функциям `setTimeout()` что `setInterval()` постоянно выполняет определенную функцию через промежуток времени.

```
function updateTime() {  
    document.getElementById("time").innerHTML = new Date().toLocaleTimeString();  
}  
setInterval(updateTime, 1000);
```

```
<div id="time"></div>
```

- **requestAnimationFrame()**
- Метод **requestAnimationFrame()** работу с графикой и имеет ряд ОПТИМИЗАЦИЙ, которые улучшают его производительность.

```
var square = document.getElementById("rect");
var angle = 0;
function rotate() {
    angle = (angle + 2)%360;
    square.style.transform = "rotate(" + angle + "deg)";
    window.requestAnimationFrame(rotate);
}
var id = window.requestAnimationFrame(rotate);
```

```
<div id="rect"></div>
```

```
<style>
#rect {
    margin: 100px;
    width: 100px;
    height: 100px;
    background: #50c878;
}
</style>
```

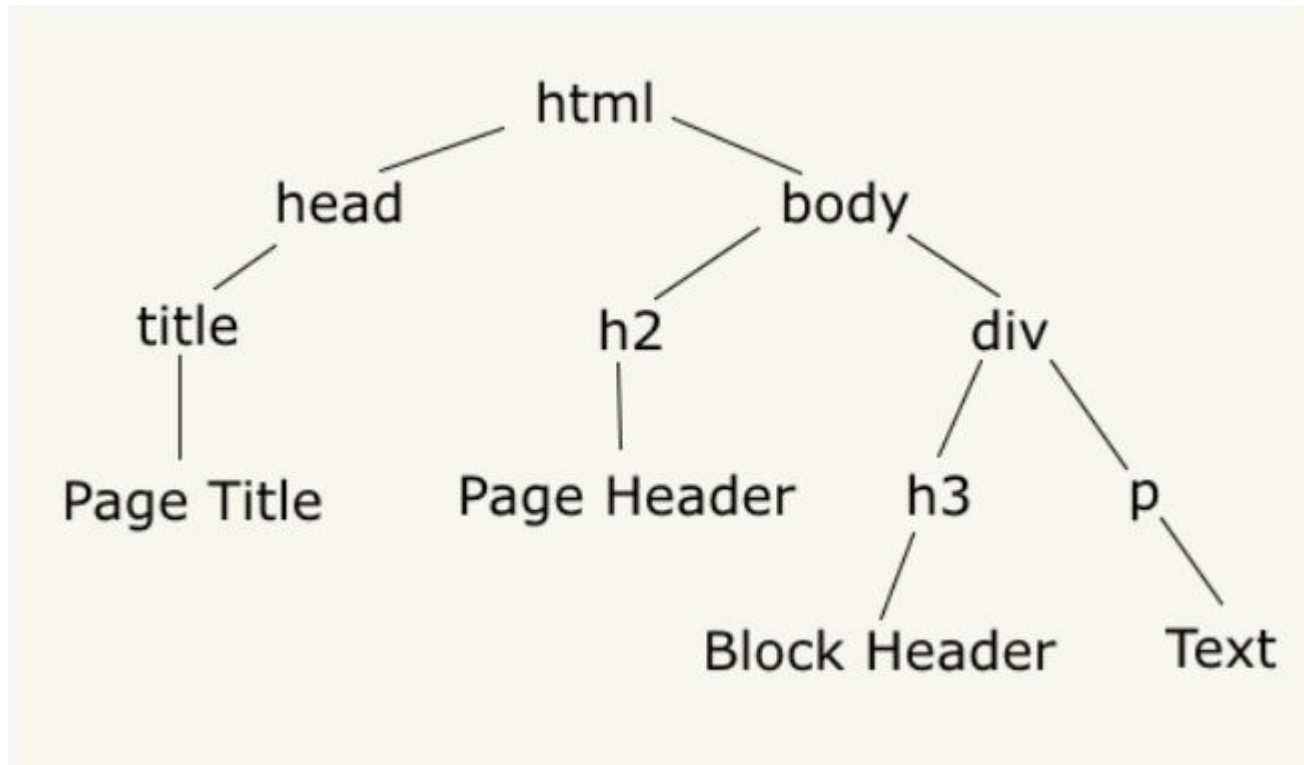


# Работа с DOM

## Введение в DOM

- Одной из ключевых задач JavaScript является взаимодействие с пользователем и манипуляция элементами веб-страницы. Для JavaScript веб-страница доступна в виде объектной модели документа (document object model) или сокращенно DOM. DOM описывает структуру веб-страницы в виде древовидного представления и предоставляет разработчикам способ получить доступ к отдельным элементам веб-страницы.
- Важно не путать понятия BOM (Browser Object Model - объектная модель браузера) и DOM (объектная модель документа). Если BOM предоставляет доступ к браузеру и его свойствам в целом, то DOM предоставляет доступ к отдельной веб-странице или html-документу и его элементам.

- Например, рассмотрим простейшую страницу:
- Дерево DOM для этой страницы будет выглядеть следующим образом:



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Page Title</title>
5 </head>
6 <body>
7     <h2>Page Header</h2>
8     <div>
9         <h3>Block Header</h3>
10        <p>Text</p>
11    </div>
12 </body>
13 </html>
```

# Объект `document`. Поиск элементов

- Для работы со структурой DOM в JavaScript предназначен объект `document`, который определен в глобальном объекте `window`. Объект `document` предоставляет ряд свойств и методов для управления элементами страницы.
- **Поиск элементов**
- Для поиска элементов на странице применяются следующие методы:
- `getElementById(value)`: выбирает элемент, у которого атрибут `id` равен `value`
- `getElementsByTagName(value)`: выбирает все элементы, у которых тег равен `value`

- **querySelector(value):** выбирает первый элемент, который соответствует css-селектору value
- Например, найдем элемент по id:

```
<body>
  <div>
    <h3 id="header">Block Header</h3>
    <p>Text</p>
  </div>
<script>
var headerElement = document.getElementById("header");
document.write("Текст заголовка: " + headerElement.innerText);
</script>
</body>
```

- Поиск по определенному тегу:

```
<body>
  <div>
    <h3>Заголовок</h3>
    <p>Первый абзац</p>
    <p>Второй абзац</p>
  </div>
<script>
var pElements = document.getElementsByTagName("p");

for (var i = 0; i < pElements.length; i++) {
  document.write("Текст параграфа: " + pElements[i].innerText + "<br/>");
}
</script>
</body>
```

- Выбор по селектору css:

```
<body>
  <div class="annotation">
    <p>Аннотация статьи</p>
  </div>
  <div class="text">
    <p>Первый абзац</p>
    <p>Второй абзац</p>
  </div>
<script>
var elem = document.querySelector(".annotation p");
document.write("Текст селектора: " + elem.innerText);
</script>
</body>
```