**JavaScript** был создан в 1995 году в компании Netscape.

JavaScript является интерпретируемым языком. Когда браузер встречает на веб-странице элемент <script> с кодом javascript, то вступает в действие встроенный интерпретатор javascript. И пока он не закончит свою работу, дальше интерпретация веб-страницы не идет.

# Основы синтаксиса javascript

* Код javascript состоит из инструкций, каждая из которых завершается точкой запятой
* Однострочный комментарий // и многострочный комментарий /\* ...\*/

В отличие от определения кода javascript вынесение его во внешние файлы имеет ряд преимуществ:

* Мы можем повторно использовать один и тот же код на нескольких веб-страницах
* Внешние файлы javascript бразуер может кэшировать, за счет этого при следующем обращении к странице браузер снижает нагрузка на сервер, а браузеру надо загрузить меньший объем информации
* Код веб-страницы становится "чище". Он содержит только html-разметку, а код поведения хранится во внешних файлах. В итоге можно отделить работу по созданию кода html-страницы от написания кода javascript

Поэтому, как правило, предпочтительнее использовать код javascript во внешних файлах, а не в прямых вставках на веб-страницу с помощью элемента script.

### Метод console.log()

Консоль браузера, которая позволяет производить отладку программы. Для вывода различного рода информации в консоли браузера используется **console.log()**

console.log("The result of operation (5+5): ")

### Метод document.write()

Программа выводит результат операции напрямую на веб-страницу:

var a = 5 + 8;

document.write("Результат операции ");

document.write(a);

# **Основы javascript**

## **Переменные и константы**

Для создания переменных применяются ключевые слова **var** и **let**.

var myIncome;

let myIncome2;

Каждая переменная имеет имя, произвольный набор алфавитно-цифровых символов, знака подчеркивания (\_) или знака доллара ($).

$commision someVariable product\_Store income2

**Список зарезервированных слов в JavaScript:**

abstract, boolean, break, byte, case, catch, char, class, const, continue, debugger, default, delete, do, double, else, enum, export, extends, false, final, finally, float, for, function, goto, if, implements, import, in, instanceof, int, inteface, long, native, new, null, package, private, protected, public, return, short, static, super, switch, synchronized, this, throw, throws, transient, true, try, typeof, var, volatile, void, while, with

## **Константы**

Константа хранит значение, которое **не может быть изменено и должна быть инициализаирована сразу**:

const rate = 10;

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

rate = 23;  // ошибка, rate - константа, поэтому мы не можем изменить ее значение

const rate; // ошибка, rate не инициализирована

## **Типы данных**

Number, String, Boolean, undefined, null. Все остальные относятся к типу object.

### Числовые данные

Числа могут быть: целые числа от -253 до 253 (полож. и отр.) и дробные числа (числа с плавающей точкой положит. и отр.) и тот же диапазон: от -253 до 253 В качестве разделителя между целой и дробной частями используется точка.

var x = 45;

var y = 23.897;

### Строки

Тип string представляет строки, т.е. данные заключеные в кавычки двойные или одинарные.

Если внутри строки встречаются кавычки, то мы их должны экранировать слешем:

var companyName = "Бюро \"Рога и копыта\"";

var companyName1 = "Бюро 'Рога и копыта'";

### Тип Boolean (true и false):

var isAlive = true;

var isDead = false;

### null (неопределенное значение) и undefined (значение не установлено)

var isAlive; // undefined

isAlive = undefined;  // undefined

isAlive = null; // null

### object

Простейшее определение объекта представляют фигурные скобки:

var user = {};

Объект может иметь различные свойства и методы. В данном случае объект называется user, и он имеет два свойства: name и age.

var user = {name: "Tom", age:24};

console.log(user.name);

### Слабая типизация

JavaScript является языком со слабой типизацией (переменные могут динамически менять тип).

var xNumber;  // тип undefined

var xNumber = 45;  // тип number

var yNumber = xNumber + 5; // 50

xNumber = "45"; // тип string

var zNumber = xNumber + 5 // 455

### Оператор typeof (получить тип переменной):

var name = "Tom"; console.log(typeof name); // string

var income = 45.8; console.log(typeof income); // number

var isEnabled = true; console.log(typeof isEnabled); // boolean

var undefVariable; console.log(typeof undefVariable); // undefined

# **Операции с переменными**

## **Математические операции**

Сложение: var x = 10; var y = x + 50;

Вычитание: var x = 100; var y = x - 50;

Умножение: var x = 4; var y = 5; var z = x \* y;

Деление: var x = 40; var y = 5; var z = x / y;

**Деление по модулю** (оператор %) возвращает остаток от деления:

var x = 40; var y = 7; var z = x % y; console.log(z); // 5

**Инкремент**:

|  |  |
| --- | --- |
| 1  2 | var x = 5;  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:

|  |  |
| --- | --- |
|  | // префиксный инкремент  var x = 5;  var z = ++x;  console.log(x); // 6  console.log(z); // 6    // постфиксный инкремент  var a = 5;  var b = a++;  console.log(a); // 6  console.log(b); // 5 |

Постфиксный инкремент аналогичен операции: a = a + 1; // a++

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

|  |  |
| --- | --- |
|  | // префиксный декремент  var x = 5;  var z = --x;  console.log(x); // 4  console.log(z); // 4  // постфиксный декремент  var a = 5;  var b = a--;  console.log(a); // 4  console.log(b); // 5 |

Как и принято в математике, все операции выполняются слева направо и различаются по приоритетам: сначала операции инкремента и декремента, затем выполняются умножение и деление, а потом сложение и вычитание. Чтобы изменить стандартный ход выполнения операций, часть выражений можно поместить в скобки:

|  |  |
| --- | --- |
|  | var x = 10;  var y = 5 + (6 - 2) \* --x;  console.log(y); //41 |

**Операции присваивания**

* = Приравнивает переменной определенное значение: var x = 5;
* += Сложение с последующим присвоением результата.

var a = 23; a += 5; // 28

* -= Вычитание с последующим присвоением результата. var a = 28; a -= 10; // 18
* \*= Умножение с последующим присвоением результата: var x = 20; x \*= 2; // 40
* /= Деление с последующим присвоением результата: var x = 40; x /= 4; // 10
* %= Получение остатка от деления с последующим присвоением результата:

|  |  |
| --- | --- |
|  | var x = 10;x %= 3; // аналогично x = x % 3  console.log(x); // 1, так как 10 - 3\*3 = 1 |

**Операторы сравнения**

Как правило, 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 больше или равно второму, 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 <= 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. Например:

|  |  |
| --- | --- |
|  | var income = 100;  var strIncome = "100";  var result = income == strIncome;  console.log(result); //true |

Переменная result здесь будет равна true, так как фактически и income, и strIncome представляют число 100. Но оператор тождественности возвратит в этом случае false, так как данные имеют разные тип:

|  |  |
| --- | --- |
|  | var income = 100;  var strIncome = "100";  var result = income === strIncome;  console.log(result); // false |

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:

|  |  |
| --- | --- |
|  | var income = 100;  var percent = 10;  var result = income > 50 && percent < 12;  console.log(result); //true |

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

|  |  |
| --- | --- |
|  | var income = 100;  var isDeposit = true;  var result = income > 50 || isDeposit == true;  console.log(result); //true |

* ! Возвращает true, если операция сравнения возвращает false:

|  |  |
| --- | --- |
|  | var income = 100;  var result1 = !(income > 50);  console.log(result1); // false, так как income > 50 возвращает true  var isDeposit = false;  var result2 = !isDeposit;  console.log(result2); // true |

**Операции со строками**

Строки могут использовать оператор + для объединения. Например:

|  |  |
| --- | --- |
|  | var name = "Том";  var surname = "Сойер"  var fullname = name + " " + surname;  console.log(fullname); //Том Сойер |

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

|  |  |
| --- | --- |
|  | var name = "Том";  var fullname = name + 256;  console.log(fullname); //Том256 |

В конце напишем небольшую программу, которая продемонстрирует работу с операциями над переменными. Для этого определим следующую веб-страницу index.html:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>JavaScript</title>  </head>  <body>  <script>      var sum = 500; // сумма вклада      var percent = 10;  // процент по вкладу      var income = sum \* percent / 100;  // доход по вкладу      sum = sum + income; // определяем новую сумму      console.log("Доход по вкладу: " + income);      console.log("Сумма вклада после первого года: " + sum);  </script>  </body>  </html> |

В скрипте объявляются три переменных: sum, percent и income. Переменная income вычисляется по остальным двум переменным с помощью операций умножения и деления. И в конце ее значение суммируется с значением переменной sum.

## Преобразования данных

var number1 = "46"; var number2 = "4";

var result = number1 + number2;

console.log(result); //464

Обе переменных представляют строки, а точнее строковые представления чисел. И в итоге мы получим не число 50, а строку 464. Но было бы неплохо, если бы их тоже можно было бы складывать, вычитать, в общем работать как с обычными числами. В этом случае мы можем использовать операции преобразования. Для преобразования строки в число применяется функция **parseInt()**:

|  |  |
| --- | --- |
|  | var number1 = "46"; var number2 = "4";  var result = parseInt(number1) + parseInt(number2);  console.log(result); // 50 |

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

|  |  |
| --- | --- |
|  | var number1 = "46.07"; var number2 = "4.98";  var result = parseFloat(number1) + parseFloat(number2);  console.log(result); //51.05 |

При этом строка может иметь смешанное содержимое, например, "123hello", то есть в данном случае есть цифры, но есть и обычные символы. Но метод parseInt() все равно попытается выполнить преобразование:

|  |  |
| --- | --- |
|  | var num1 = "123hello"; var num2 = parseInt(num1);  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:

|  |  |
| --- | --- |
|  | var num1 = "javascript"; var num2 = "22"; var result = isNaN(num1);  console.log(result); // true - num1 не является числом  result = isNaN(num2);  console.log(result); //  false - num2 - это число |

Выше мы рассматривали перевод строк в числа в десятичной системе. Однако мы можем переводить числа в любую систему. По умолчанию интерпретатор JavaScript сам отгадывает, в число из какой системы исчисления мы хотим преобразовать строку (как правило, выбирается десятичная система). Но мы можем с помощью второго параметра явно указать, что хотим преобразовать строку в число в определенной системе. Например, преобразование в число в двоичной системе:

|  |  |
| --- | --- |
|  | var num1 = "110"; var num2 = parseInt(num1, 2);  console.log(num2); // 6 |

Результатом будет 6, так как 110 в двоичной системе - это число 6 в десятичной. Теперь напишем небольшую программу, в которой используем операции с переменными:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>JavaScript</title>  </head>  <body>  <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>  </body>  </html> |

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. Однако функция prompt() возвращает строку. Поэтому эту строку нам надо преобразовать в число, чтобы выполнить с ней операции. После открытия страницы в браузере мы увидим приглашение к вводу суммы вклада: Затем подобное сообщение отобразится и для ввода процента. И в конце программа получит данные, преобразует их в числа и выполнит подсчет:

## **Массивы**

Для создания массива применяется выражение **new Array()**: 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:

|  |  |
| --- | --- |
|  | var people = ["Tom", "Alice", "Sam"];  console.log(people); |

В этом случае в массиве myArray будет три элемента. Графически его можно представить так:

|  |  |
| --- | --- |
| **Индекс** | **Элемент** |
| 0 | Tom |
| 1 | Alice |
| 2 | Sam |

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

|  |  |
| --- | --- |
|  | var people = ["Tom", "Alice", "Sam"];  console.log(people[0]); // Tom  var person3 = people[2]; // Sam  console.log(person3); // Sam |

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

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

Также по индексу осуществляется установка значений для элементов массива:

|  |  |
| --- | --- |
|  | var people = ["Tom", "Alice", "Sam"];  console.log(people[0]); // Tom  people[0] = "Bob";  console.log(people[0]); // Bob |

Причем в отличие от других языков, как C# или Java, можно установить элемент, который изначально не установлен:

|  |  |
| --- | --- |
|  | var people = ["Tom", "Alice", "Sam"];  console.log(people[7]); // undefined - в массиве только три элемента  people[7] = "Bob";  console.log(people[7]); // Bob |

Также стоит отметить, что в отличие от ряда языков программирования в JavaScript массивы не являются строго типизированными, один массив может хранить данные разных типов:

|  |  |
| --- | --- |
| 1  2 | var objects = ["Tom", 12, true, 3.14, false];  console.log(objects); |

### spread-оператор

spread-оператор **...** позволяет взять значения из массива по отдельности:

|  |  |
| --- | --- |
|  | let numbers = [1, 2, 3, 4];  console.log(...numbers);    // 1 2 3 4  console.log(numbers);       // [1, 2, 3, 4] |

spread-оператор указывается перед массивом. В результате выражение ...numbers возвратит набор чисел, но это будет не массив, а именно отедльные значения.

### Многомерные массивы

Массивы могут быть одномерными и многомерными. Каждый элемент в многомерном массиве может представлять собой отдельный массив. Выше мы рассматривали одномерный массив, теперь создадим многомерный массив:

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

Поскольку массив numbers2 двухмерный, он представляет собой простую таблицу. Каждый его элемент может представлять отдельный массив. Рассмотрим еще один двумерный массив:

|  |  |
| --- | --- |
|  | var people = [          ["Tom", 25, false],          ["Bill", 38, true],          ["Alice", 21, false]  ];  console.log(people[0]); // ["Tom", 25, false]  console.log(people[1]); // ["Bill", 38, true] |

Чтобы получить отдельный элемент массива, также используется индекс:

|  |  |
| --- | --- |
| 1 | var tomInfo = people[0]; |

Только теперь переменная tomInfo будет представлять массив. Чтобы получить элемент внутри вложенного массива, нам надо использовать его вторую размерность:

|  |  |
| --- | --- |
| 1  2 | console.log("Имя: " + people[0][0]); // Tom  console.log("Возраст: " + people[0][1]); // 25 |

То есть если визуально двумерный массив мы можем представить в виде таблицы, то 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). Также мы можем выполнить присвоение:

|  |  |
| --- | --- |
|  | var people = [          ["Tom", 25, false],          ["Bill", 38, true],          ["Alice", 21, false]  ];  people[0][1] = 56; // присваиваем отдельное значение  console.log(people[0][1]); // 56    people[1] = ["Bob", 29, false]; // присваиваем массив  console.log(people[1][0]); // Bob |

При создании многомерных массивов мы не ограничены только двумерными, но также можем использовать массивы больших размерностей:

|  |  |
| --- | --- |
|  | var numbers = [];  numbers[0] = []; // теперь numbers - двумерный массив  numbers[0][0]=[]; // теперь numbers - трехмерный массив  numbers[0][0][0] = 5; // первый элемент трехмерного массива равен 5  console.log(numbers[0][0][0]); |

## **Условные конструкции**

### Выражение if

if(условие)действия;

Пример:

var income = 100;

if(income > 50) alert("доход больше 50");

Если необходимо выполнить по условию набор инструкций, то они помещаются в блок из фигурных скобок:

|  |  |
| --- | --- |
|  | var income = 100;  var age = 19;  if(income < 150 && age > 18){       var message = "доход больше 50";      alert(message);  } |

Конструкция if позволяет проверить наличие значения. Например:

|  |  |
| --- | --- |
|  | var myVar = 89;  if(myVar){      // действия  } |

Если переменная myVar имеет значение, то в условной конструкции она возвратит значение true. Но нередко для проверки значения переменной используют альтернативный вариант - проверяют на значение undefined:

|  |  |
| --- | --- |
|  | if (typeof myVar != "undefined") {      // действия  } |

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

|  |  |
| --- | --- |
|  | var age = 17;  if(age >= 18){        alert("Вы допущены к программе кредитования");  }  else{      alert("Вы не можете участвовать в программе, так как возраст меньше 18");  } |

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

|  |  |
| --- | --- |
|  | var income = 300;  if(income < 200){       alert("Доход ниже среднего");  }  else if(income >= 200 && 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:

|  |  |
| --- | --- |
|  | 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");  } |

### True или false

В javascript любая переменная может применяться в условных выражениях, но не любая переменная представляет тип boolean. И в этой связи возникает вопрос, что возвратит та или иная переменная - true или false? Много зависит от типа данных, который представляет переменная:

* **undefined** Возвращает false
* **null** Возвращает false
* **Boolean** Если переменная равна false, то возвращается false. Соответствено 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 возращать false:

|  |  |
| --- | --- |
|  | var x = NaN;  if(x){  // false  } |

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

|  |  |
| --- | --- |
|  | var y = ""; // false - так как пустая строка  var z = "javascript";   // true - строка не пустая |

* **Object** Всегда возвращает true

|  |  |
| --- | --- |
|  | var user = {name:"Tom"};    // true  var isEnabled = new Boolean(false)  // true  var car = {}    // true |

### Конструкция switch..case

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

|  |  |
| --- | --- |
|  | var income = 300;  switch(income){       case 100 :          console.log("Доход равен 100");          break;      case 200 :          console.log("Доход равен 200");          break;      case 300 :          console.log("Доход равен 300");          break;  } |

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

|  |  |
| --- | --- |
|  | var income = 300;  switch(income){       case 100 :          console.log("Доход равен 100");          break;      case 200 :          console.log("Доход равен 200");          break;      case 300 :          console.log("Доход равен 300");          break;      default:          console.log("Доход неизвестной величины");          break;  } |

### 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. Например:

|  |  |
| --- | --- |
|  | var a = 1;  var b = 2;  var result = a < b ? a + b : a - b;  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.

## Циклы

В JavaScript имеются следующие виды циклов: **for, for..in, for..of, while, do..while**

### Цикл for

for ([инициализация счетчика]; [условие]; [изменение счетчика]){

     // действия

}

Например, используем цикл for для перебора элементов массива:

|  |  |
| --- | --- |
|  | var people = ["Tom", "Alice", "Bob", "Sam"];  for(var i = 0; i<people.length; i++){      console.log(people[i]);  } |

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 браузере. Необязательно увеличивать счетчик на единицу, можно производить с ним другие действия, например, уменьшать на единицу:

|  |  |
| --- | --- |
|  | var people = ["Tom", "Alice", "Bob", "Sam"];  for(var i = people.length - 1; i >= 0; i--){     console.log(people[i]);  } |

В данном случае массив выводится с конца, а перебор массива начинается с i = 3 до i = 0.

### for..in

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

|  |  |
| --- | --- |
|  | for (индекс in массив) {      // действия  } |

Например, переберем элементы массива:

|  |  |
| --- | --- |
|  | var people = ["Tom", "Alice", "Bob", "Sam"];  for(var index in people){      console.log(people[index]);  } |

### Цикл for...of

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

|  |  |
| --- | --- |
|  | let users = ["Tom", "Bob", "Sam"];  for(let val of users)      console.log(val); |

Текущий перебираемый элемент коллекции помещается в переменную val, значение которой затем выводится на консоль.

### Цикл while

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

while(условие){

   // действия

}

Опять же выведем с помощью while элементы массива:

|  |  |
| --- | --- |
|  | var people = ["Tom", "Alice", "Bob", "Sam"];  var index = 0;  while(index < people.length){      console.log(people[index]);      index++;  } |

Цикл while здесь будет выполняться, пока значение index не станет равным длине массива.

### 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:

|  |  |
| --- | --- |
|  | var x = 1;  do{      console.log(x \* x);      x++;  }while(x < 10) |

Здесь код цикла сработает 9 раз, пока x не станет равным 10. При этом цикл do гарантирует хотя бы однократное выполнение действий, даже если условие в инструкции while не будет истинно.

### Операторы continue и break

Выйти из цикла до его завершения - **break**:

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

Данный цикл перебирает все элементы массива, однако последние четыре элемента не будут выведены в браузере, поскольку поверка 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**:

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

В этом случае, если программа встретит в массиве число, больше 10, то это число не будет выводиться в браузере.

# Функциональное программирование

## Функции

Синтаксис определения функции:

|  |  |
| --- | --- |
|  | function имя\_функции([параметр [, ...]]){       // Инструкции  } |

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.

|  |  |
| --- | --- |
|  | function display(){       document.write("функция в JavaScript");  } |

Данная функция называется display(). Она не принимает никаких параметров и все, что она делает, это пишет на веб-страницу строку. Однако простого определения функции еще недостаточно, чтобы она заработала. На надо еще ее вызвать:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <script>  function display(){       document.write("функция в JavaScript");  }  display();  </script>  </body>  </html> |

Необязательно давать функциям определенное имя. Можно использовать анонимные функции:

|  |  |
| --- | --- |
|  | var display = function(){  // определение функции       document.write("функция в JavaScript");  }  display(); |

Фактически мы определяем переменную display и присваиваем ей ссылку на функцию. А затем по имени переменной функция вызывается. Также мы можем динамически присваивать функции для переменной:

|  |  |
| --- | --- |
|  | function goodMorning(){       document.write("Доброе утро");  }  function goodEvening(){       document.write("Добрый вечер");  }  var message = goodMorning;  message(); // Доброе утро  message = goodEvening;  message(); // Добрый вечер |

**Параметры функции**

|  |  |
| --- | --- |
|  | function display(x){  // определение функции       var z = x \* x;      document.write(x + " в квадрате равно " + z);  }  display(5); // вызов функции |

Функция display принимает один параметр - x. Поэтому при вызове функции мы можем передать для него значение, например, число 5, как в данном случае. Если функция принимает несколько параметров, то с помощью spread-оператора ... мы можем передать набор значений для этих параметров из массива:

|  |  |
| --- | --- |
|  | function sum(a, b, c){      let d = a + b + c;      console.log(d);  }  sum(1, 2, 3);  let nums = [4, 5, 6];  sum(...nums); |

Во втором случае в функцию передается числа из массива nums. Но чтобы передавался не просто массив, как одно значение, а именно числа из этого массива, применяется spread-оператор (многоточие ...).

**Необязательные параметры**

Функция может принимать множество параметров, но при этом часть или все параметры могут быть необязательными. Если для параметров не передается значение, то по умолчанию они имеют значение "undefined".

|  |  |
| --- | --- |
|  | function display(x, y){       if(y === undefined) y = 5;      if(x === undefined) x = 8;      let z = x \* y;      console.log(z);  }  display();  // 40  display(6); // 30  display(6, 4) // 24 |

Здесь функция display принимает два параметра. При вызове функции мы можем проверить их значения. При этом, вызывая функцию, необязательно передавать для этих параметров значения. Для проверки наличия значения параметров используется сравнение со значением undefined. Есть и другой способ определения значения для параметров по умолчанию:

|  |  |
| --- | --- |
|  | function display(x = 5, y = 10){      let z = x \* y;      console.log(z);  }  display();      // 50  display(6);     // 60  display(6, 4)   // 24 |

Если параметрам x и y не передаются значения, то они получаются в качестве значений числа 5 и 10 соответствено. Такой способ более лаконичен и интуитивен, чем сравнение с undefined. При этом значение параметра по умолчанию может быть производным, представлять выражение:

|  |  |
| --- | --- |
|  | function display(x = 5, y = 10 + x){      let z = x \* y;      console.log(z);  }  display();      // 75  display(6);     // 96  display(6, 4)   // 24 |

В данном случае значение параметра y зависит от значения x. При необходимости мы можем получить все переданные параметры через глобально доступный массив **arguments**:

|  |  |
| --- | --- |
|  | function display(){      var z = 1;      for(var i=0; i<arguments.length; i++)          z \*= arguments[i];      console.log(z);  }  display(6); // 6  display(6, 4) // 24  display(6, 4, 5) // 120 |

При этом даже не важно, что при определении функции мы не указали никаких параметров, мы все равно можем их передать и получить их значения через массив arguments.

**Неопределенное количество параметров**

С помощью spread-оператора мы можем указать, что с помощью параметра можно передать переменное количество значений:

|  |  |
| --- | --- |
|  | function display(season, ...temps){      console.log(season);      for(index in temps){          console.log(temps[index]);      }  }  display("Весна", -2, -3, 4, 2, 5);  display("Лето", 20, 23, 31); |

В данном случае второй параметр ...temps указывает, что вместо него можно передать разное количество значений. В самой функции temps фактически представляет массив переданных значений, которые мы можем получить. При этом несмотря на это, при вызове функции в нее передается не массив, а именно отдельные значения. Консольный вывод:

Весна

-2

-3

4

2

5

Лето

20

23

31

**Результат функции**

Функция может возвращать результат. Для этого используется оператор **return**:

|  |  |
| --- | --- |
|  | var y = 5;  var z = square(y);  document.write(y + " в квадрате равно " + z);  function square(x) {    return x \* x;  } |

После оператора return идет значение, которое надо возвратить из метода. В данном случае это квадрат числа х. После получения результата функции мы можем присвоить его какой-либо другой переменной: var z = square(y);

**Функции в качестве параметров**

Функции могут выступать в качестве параметров других функций:

|  |  |
| --- | --- |
|  | function sum(x, y){      return x + y;  }   function subtract(x, y){      return x - y;  }  function operation(x, y, func){      var result = func(x, y);      console.log(result);  }  console.log("Sum");  operation(10, 6, sum);  // 16  console.log("Subtract");  operation(10, 6, subtract); // 4 |

Функция operation принимает три параметра: x, y и func. func - представляет функцию, причем на момент определения operation не важно, что это будет за функция. Единственное, что известно, что функция func может принимать два параметра и возвращать значение, которое затем отображается в консоли браузера. Поэтому мы можем определить различные функции (например, функции sum и subtract в данном случае) и передавать их в вызов функции operation.

**Возвращение функции из функции**

Одна функция может возвращать другую функцию:

|  |  |
| --- | --- |
|  | function menu(n){      if(n==1) return function(x, y){ return x+y;}      else if(n==2) return function(x, y){ return x - y;}      else if(n==3) return function(x, y){ return x \* y;}      return undefined;  }  for(var i=1; i < 5; i++){      var action = menu(i);      if(action!==undefined){          var result = action(5, 4);          console.log(result);      }  } |

В данном случае функция menu в зависимости от переданного в нее значения возвращает одну из трех функций или undefined.

## Область видимости переменных

Все переменные в JavaScript имеют определенную область видимости, в пределах которой они могут действовать.

### Глобальные переменные

Все переменные, которые объявлены вне функций, являются глобальными:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <script>  var x = 5;  let d = 8;  function displaySquare(){       var z = x \* x;      console.log(z);  }  </script>  </body>  </html> |

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:

|  |  |
| --- | --- |
|  | function displaySquare(){       var z = 10;      console.log(z);      let b = 8;      console.log(b);  } |

Переменные z и b являются локальными, они существуют только в пределах функции. Вне функции их нельзя использовать:

|  |  |
| --- | --- |
|  | function displaySquare(){       var z = 10;      console.log(z);  }  console.log(z); //ошибка, так как z не определена |

Когда функция заканчивает свою работу, то все переменные, определенные в функции, уничтожаются.

### Сокрытие переменных

Что если у нас есть две переменных - одна глобальная, а другая локальная, которые имеют одинаковое имя:

|  |  |
| --- | --- |
|  | var z = 89;  function displaySquare(){       var z = 10;      console.log(z); // 10  }  displaySquare(); // 10 |

В этом случае в функции будет использоваться та переменная z, которая определена непосредственно в функции. То есть локальная переменная скроет глобальную.

### var или let

При использовании оператора **let** каждый блок кода определяет новую область видимости, в которой существует переменная. Например, мы можем одновременно определить переменную на уровне блока и на уровне функции:

|  |  |
| --- | --- |
|  | let z = 10;  function displayZ(){       let z = 20;       {          let z = 30;          console.log("Block:", z);      }      console.log("Function:", z);  }  displayZ();  console.log("Global:", z); |

Здесь внутри функции displayZ определен блок кода, в котором определена переменная z. Она скрывает глобальную переменную и переменную z, определенную на уровне функции. В реальной программе блок мог быть предеставлять вложенную функцию, блок цикла for или конструкции if. Но в любом случае такой блок определяет новую область видимости, вне которого переменная не существует. И в данном случае мы получим следующий консольный вывод: Block: 30 Function: 20 Global: 10

С помощью оператора var мы не можем определить одновременно переменную с одним и тем же именем и в функции, и в блоке кода в этой функции:

|  |  |
| --- | --- |
|  | function displaySquare(){       var z = 20;      {          var z = 30; // Ошибка ! Переменная z уже определена          console.log("Block:", z);      }      console.log("Function:", z);  } |

То есть с помощью var мы можем определить переменную с одним именем либо на уровне функции, либо на уровне блока кода.

### Константы

Все, что относится к оператору let, относится и к оператору const, который позволяет определить константы. Блоки кода задают область видимости констант, а константы, определенные на вложенных блоках кода, скрывают внешние константы с тем же именем:

|  |  |
| --- | --- |
|  | const z = 10;  function displayZ(){       const z = 20;      {          const z = 30;          console.log("Block:", z);   // 30      }      console.log("Function:", z);    // 20  }  displayZ();  console.log("Global:", z);  // 10 |

### Необъявленные переменные

Если мы не используем это ключевое слово при определении переменной в функции, то такая переменная будет глобальной. Например:

|  |  |
| --- | --- |
|  | function bar(){      foo = "25";  }  bar();  console.log(foo);   // 25 |

Несмотря на то, что вне функции bar переменная foo нигде не определяется, тем не менее она доступна вне функции во внешнем контексте. Иначе было бы если бы мы не только присваивали значение переменной, но и определяли бы ее:

|  |  |
| --- | --- |
|  | function bar(){      var foo = "25";  }    bar();  console.log(foo);   // ошибка |

### strict mode

Определение глобальных переменных в функциях может вести к потенциальным ошибкам. Чтобы их избежать используется строгий режим или strict mode:

|  |  |
| --- | --- |
|  | "use strict";  function bar(){      foo = "25";  }  bar();  console.log(foo); |

В этом случае мы получим ошибку SyntaxError: Unexpected identifier, которая говорит о том, что переменная foo не определена. Установить режим strict mode можно двумя способами:

* добавить выражение "use strict" в начало кода JavaScript, тогда strict mode будет применяться для всего кода
* добавить выражение "use strict" в начало тела функции, тогда strict mode будет применяться только для этой функции

## Замыкания и функции IIFE

### 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
* вложенная функция, которая использует эти переменные

|  |  |
| --- | --- |
|  | function outer(){       // внешняя функция      var n;              // некоторая переменная      return inner(){     // вложенная функция          // действия с переменной n      }  } |

Рассмотрим замыкания на простейшем примере:

|  |  |
| --- | --- |
|  | function outer(){      let x = 5;      function inner(){          x++;          console.log(x);      };      return inner;  }  let fn = outer();   // fn = inner, так как функция outer возвращает функцию inner  // вызываем внутреннюю функцию inner  fn();   // 6  fn();   // 7  fn();   // 8 |

Здесь функция outer задает область видимости, в которой определены внутренняя функция inner и переменная x. Переменная x представляет лексическое окружение для функции inner. В самой функции inner инкрементируем переменную x и выводим ее значение на консоль. В конце функция outer возвращает функцию inner. Далее вызываем функцию outer: let fn = outer();

Поскольку функция outer возвращает функцию inner, то переменная fn будет хранить ссылку на функцию inner. При этом эта функция запомнила свое окружение - то есть внешнюю переменную x. Далее мы фактически три раза вызываем функцию Inner, и мы видим, что переменная x, которая определена вне функции inner, инкрементируется:

|  |  |
| --- | --- |
|  | fn();   // 6  fn();   // 7  fn();   // 8 |

То есть несмотря на то, что переменная x определена вне функции inner, эта функция запомнила свое окружение и может его использовать, несомотря на то, что она вызывается вне функции outer, в которой была определена. В этом и суть замыканий. Рассмотрим еще один пример:

|  |  |
| --- | --- |
|  | function multiply(n){      var x = n;      return function(m){ return x \* m;};  }  var fn1 = multiply(5);  var result1 = fn1(6); // 30  console.log(result1); // 30    var fn2= multiply(4);  var result2 = fn2(6); // 24  console.log(result2); // 24 |

Итак, здесь вызов функции multiply() приводит к вызову другой внутренней функции. Внутренняя же функция:

|  |  |
| --- | --- |
| 1 | function(m){ return x \* m;}; |

запоминает окружение, в котором она была создана, в частности, значение переменной x. В итоге при вызове функции multiply определяется переменная fn1, которая и представляет собой замыкание, то есть объединяет две вещи: функцию и окружение, в котором функция была создана. Окружение состоит из любой локальной переменной, которая была в области действия функции multiply во время создания замыкания. То есть fn1 — это замыкание, которое содержит и внутреннюю функцию function(m){ return x \* m;}, и переменную x, которая существовала во время создания замыкания. При создании двух замыканий: fn1 и fn2, для каждого из этих замыканий создается свое окружение. При этом важно не запутаться в параметрах. При определении замыкания:

|  |  |
| --- | --- |
|  | var fn1 = multiply(5); |

Число 5 передается для параметра n функции multiply. При вызове внутренней функции:

|  |  |
| --- | --- |
|  | var result1 = fn1(6); |

Число 6 передается для параметра m во внутреннюю функцию function(m){ return x \* m;};. Также мы можем использовать другой вариант для вызова замыкания:

|  |  |
| --- | --- |
|  | function multiply(n){      var x = n;      return function(m){ return x \* m;};  }  var result = multiply(5)(6); // 30  console.log(result); |

### Самовызывающиеся функции

Обычно определение функции отделяется от ее вызова: сначала мы определяем функцию, а потом вызываем. Но это необязательно. Мы также можем создать такие функции, которые будут вызываться сразу при определении. Такие функции еще называют Immediately Invoked Function Expression (IIFE).

|  |  |
| --- | --- |
|  | (function(){      console.log("Привет мир");  }());    (function (n){      var result = 1;      for(var i=1; i<=n; i++)          result \*=i;      console.log("Факториал числа " + n + " равен " + result);  }(4)); |

Подобные функции заключаются в скобки, и после определения функции идет в скобках передача параметров.

## Паттерн Модуль

Паттерн "Модуль" базируется на замыканиях и состоит из двух компонентов: внешняя функция, которая определяет лексическое окружение, и возвращаемый набор внутренних функций, которые имеют доступ к этому окружению. Определим простейший модуль:

|  |  |
| --- | --- |
|  | let foo = (function(){      let obj = {greeting: "hello"};        return {          display: function(){              console.log(obj.greeting);          }      }  })();  foo.display();  // hello |

Здесь определена переменная foo, которая представляет результат анонимной функции. Внутри подобной функции определен объект obj с некоторыми данными. Сама анонимная функция возвращает объект, который определяет функцию display. Возвращаемый объект определяет общедоступый API, через который мы можем обращаться к данным, определенным внутри модуля.

|  |  |
| --- | --- |
|  | return {          display: function(){              console.log(obj.greeting);          }      } |

Такая конструкция позволяет закрыть некоторый набор данных в рамках функции-модуля и опосредовать доступ к ним через определенный API - возвращаемые внутренние функции. Рассмотрим чуть более сложный пример:

|  |  |
| --- | --- |
|  | let calculator = (function(){      let data = { number: 0};        return {          sum: function(n){              data.number += n;          },          subtract: function(n){              data.number -= n;          },          display: function(){              console.log("Result: ", data.number);          }      }  })();  calculator.sum(10);  calculator.sum(3);  calculator.display();   // Result: 13  calculator.subtract(4);  calculator.display();   // Result: 9 |

Данный модуль представляет примитивный калькулятор, который выполняет три операции: сложение, вычитание и вывод результата. Все данные инкапсулированы в объекте data, который хранит результат операции. Все операции представлены тремя возвращаемыми функциями: sum, subtract и display. Через эти функции мы можем управлять результатом калькулятора извне.

## Рекурсивные функции

Среди функций отдельно можно выделить рекурсивные функции. Их суть состоит в том, что функция вызывает саму себя. Например, рассмотрим функцию, определяющую факториал числа:

|  |  |
| --- | --- |
|  | function getFactorial(n){      if (n === 1){          return 1;      }      else{            return n \* getFactorial(n - 1);      }  }  var result = getFactorial(4);  console.log(result); // 24 |

Функция getFactorial() возвращает значение 1, если параметр n равен 1, либо возвращает результат опять же функции getFactorial, то в нее передается значение n-1. Например, при передаче числа 4, у нас образуется следующая цепочка вызовов:

|  |  |
| --- | --- |
|  | var result = 4 \* getFactorial(3);  var result = 4 \* 3 \* getFactorial(2);  var result = 4 \* 3 \* 2 \* getFactorial(1);  var result = 4 \* 3 \* 2 \* 1; // 24 |

Рассмотрим другой пример - определение чисел Фибоначчи:

|  |  |
| --- | --- |
|  | function getFibonachi(n)  {      if (n == 0){          return 0;      }      if (n == 1){          return 1;      }      else{          return getFibonachi(n - 1) + getFibonachi(n - 2);      }  }  var result = getFibonachi(8); //21  console.log(result); // 21 |

## Переопределение функций

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

|  |  |
| --- | --- |
|  | function display(){      console.log("Доброе утро");      display = function(){          console.log("Добрый день");      }  }  display(); // Доброе утро  display(); // Добрый день |

При первом срабатывании функции действует основной блок операторов функции, в частности, в данном случае выводится сообщение "Доброе утро". И при первом срабатывании функции display также происходит ее переопределение. Поэтому при всех последующих вызовах функции срабатывает ее переопределенная версия, а на консоль будет выводиться сообщение "Добрый день". Но при переопределении функции надо учитывать некоторые нюансы. В частности, попробуем присвоить ссылку на функцию переменной и через эту переменную вызвать функцию:

|  |  |
| --- | --- |
|  | function display(){      console.log("Доброе утро");      display = function(){          console.log("Добрый день");      }  }  // присвоение ссылки на функцию до переопределения  var displayMessage = display;  display(); // Доброе утро  display(); // Добрый день  displayMessage(); // Доброе утро  displayMessage(); // Доброе утро |

Здесь переменная displayMessage получает ссылку на функцию display до ее переопределения. Поэтому при вызове displayMessage() будет вызываться непереопределенная версия функции display. Но допустим, мы определили переменную displayMessage уже после вызова функции display:

|  |  |
| --- | --- |
|  | display(); // Доброе утро  display(); // Добрый день  var displayMessage = display;  displayMessage(); // Добрый день  displayMessage(); // Добрый день |

В этом случае переменная displayMessage будет указывать на переопределенную версию функции display.

## Hoisting

Hoisting представляет процесс доступа к переменным до их определения. Возможно, данная концепция выглядит немного странно, но она связана с работой компилятора JavaScript. Компиляция кода происходит в два прохода. При первом проходе компилятор получает все объявления переменных, все идентификаторы. При этом никакой код не выполняется, методы не вызываются. При втором проходе собственно происходит выполнение. И даже если переменная определена после непосредственного использования, ошибки не возникнет, так как при первом проходе компилятору уже известны все переменные. То есть как будто происходит поднятие кода с определением переменных и функций вверх до их непосредственного использования. Поднятие на английский переводится как hoisting, сообственно поэтому данный процесс так и называется. Переменные, которые попадают под hoisting, получают значение undefined. Например, возьмем следующий простейший код:

|  |  |
| --- | --- |
|  | console.log(foo); |

Его выполнение вызовет ошибку ReferenceError: foo is not defined

Добавим определение переменной:

|  |  |
| --- | --- |
|  | console.log(foo);   // undefined  var foo = "Tom"; |

В этом случае консоль выведет значение "undefined". При первом проходе компилятор узнает про существование переменной foo. Она получает значение undefined. При втором проходе вызывается метод console.log(foo). Возьмем другой пример:

|  |  |
| --- | --- |
|  | var c = a \* b;  var a = 7;  var b = 3;  console.log(c); // NaN |

Здесь та же ситуация. Переменные a и b используются до опеределения. По умолчанию им присваиваются значения undefined. А если умножить undefined на undefined, то получим Not a Number (NaN). Все то же самое относится и к использованию функций. Мы можем сначала вызвать функцию, а потом уже ее определить:

|  |  |
| --- | --- |
|  | display();  function display(){      console.log("Hello Hoisting");  } |

Здесь функция display благополучно отработает, несмотря на то, что она определена после вызова. Но от этой ситуации надо отличать тот случай, когда функция определяется в виде переменной:

|  |  |
| --- | --- |
|  | display();  var display = function (){      console.log("Hello Hoisting");  } |

В данном случае мы получим ошибку TypeError: display is not a function. При первом проходе компилятор также получит переменную display и присвоет ей значение undefined. При втором проходе, когда надо будет вызывать функцию, на которую будет ссылаться эта переменная, компилятор увидит, что вызывать то нечего: переменная display пока еще равна undefined. И будет выброшена ошибка. Поэтому при определении переменных и функций следует учитывать перепетии такого аспекта как hoisting.

## Передача параметров по значению и по ссылке

### Передача параметров по значению

Строки, числа, логические значения передаются в функцию по значению. Иными словами при передаче значения в функцию, эта функция получает копию данного значения. Рассмотрим, что это значит в практическом плане:

|  |  |
| --- | --- |
|  | function change(x){      x = 2 \* x;      console.log("x in change:", x);  }  var n = 10;  console.log("n before change:", n); // n before change: 10  change(n);                          // x in change: 20  console.log("n after change:", n);  // n after change: 10 |

Функция change получает некоторое число и увеличивает его в два раза. При вызове функции change ей передается число n. Однако после вызова функции мы видим, что число n не изменилось, хотя в самой функции произошло увеличение значения параметра. Потому что при вызове функция change получает копию значения переменной n. И любые изменения с этой копией никак не затрагивают саму переменную n.

**Передача по ссылке**

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

|  |  |
| --- | --- |
|  | function change(user){      user.name = "Tom";  }  var bob ={      name: "Bob"  };  console.log("before change:", bob.name);    // Bob  change(bob);  console.log("after change:", bob.name);     // Tom |

В данном случае функция change получает объект и меняет его свойство name. В итоге мы увидим, что после вызова функции изменился оригинальный объект bob, который передавался в функцию. Однако если мы попробуем переустановить объект или массив полностью, оригинальное значение не изменится.

|  |  |
| --- | --- |
|  | function change(user){      // полная переустановка объекта      user= {          name:"Tom"      };  }  var bob ={      name: "Bob"  };  console.log("before change:", bob.name);    // Bob  change(bob);  console.log("after change:", bob.name);     // Bob |

То же самое касается массивов:

|  |  |
| --- | --- |
|  | function change(array){      array[0] = 8;  }  function changeFull(array){      array = [9, 8, 7];  }    var numbers = [1, 2, 3];    console.log("before change:", numbers);     // [1, 2, 3]  change(numbers);  console.log("after change:", numbers);      // [8, 2, 3]  changeFull(numbers);  console.log("after changeFull:", numbers);  // [8, 2, 3] |

## 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. Например:

|  |  |
| --- | --- |
|  | let sum = (x, y) => x + y;  let a = sum(4, 5);      // 9  let b = sum(10, 5);     // 15 |

В данном случае функция (x, y) => x + y осуществляет сложение двух чисел и присваивается переменной sum. Функция принимает два параметра - x и y. Ее тело составляет сложение значений этих параметров. И поскольку после стрелки фактически идет конкретное значение, которое представляет сумму чисел, то функция возвращает это значение. И мы можем через переменную sum вызвать данную функцию и получить ее результат в переменные a и b. Если после стрелки идет операция или выражение, которое возвращает значение, то это значение фактически возвращается из стрелочной функции. Но также в качестве тела функции может примяться выражение, которое ничего не возвращает и просто выполняет некоторое действие:

|  |  |
| --- | --- |
|  | let sum = (x, y) => console.log(x + y);  sum(4, 5);      // 9  sum(10, 5);     // 15 |

В данном случае функция console.log() ничего не возвращает, и соответственно функция sum также не возвращает никакого результата. Если функция принимает один параметр, то скобки вокруг него можно опустить:

|  |  |
| --- | --- |
|  | var square = n => n \* n;  console.log(square(5));     // 25  console.log(square(6));     // 36  console.log(square(-7));    // 49 |

Если тело функции представляет набор выражений, то они облекаются в фигурные скобки:

|  |  |
| --- | --- |
|  | var square = n => {      let result = n \* n;      return result;  }  console.log(square(5));     // 25 |

Для возвращения результата из функции в таком случае применяется стандартный оператор **return**. Особо следует остановиться на случае, когда стрелочная функция возвращает объект:

|  |  |
| --- | --- |
|  | let user = (userName, userAge) => ({name: userName, age: userAge});  let tom = user("Tom", 34);  let bob = user("Bob", 25);  console.log(tom.name, tom.age);     // "Tom", 34  console.log(bob.name, bob.age);     // "Bob", 25 |

Объект также определяется с помощью фигурных скобок, но при этом он заключается в круглые скобки.

Если стрелочная функция не принимает никаких параметров, то ставятся пустые скобки:

|  |  |
| --- | --- |
|  | var hello = ()=> console.log("Hello World");  hello();    // Hello World  hello();    // Hello World |

# Объектно-ориентированное программирование

## Объекты

Например, если в нашей программе нам надо описать сущность человека, у которого есть имя, возраст, пол и так далее, то естественно мы не сможем представить сущность человека в виде числа или строки. Нам потребуется несколько строк или чисел, чтобы должным образом описать человека. В этом плане человек будет выступать как сложная комплексная структура, у которого будут отдельные свойства - возраст, рост, имя, фамилия и т.д. Для работы с подобными структурами в JavaScript используются **объекты**. Каждый объект может хранить свойства, которые описывают его **состояние**, и методы, которые описывают его **поведение**

### Создание нового объекта

Есть несколько способов создания нового объекта. Первый способ заключается в использовании конструктора Object:

|  |  |
| --- | --- |
|  | var user = new Object(); |

В данном случае объект называется user. Он определяется также, как и любая обычная переменная с помощью ключевого слова var. Выражение new Object() представляет вызов конструктора - функции, создающей новый объект. Для вызова конструктора применяется оператор **new**. Вызов конструктора фактически напоминает вызов обычной функции. Второй способ создания объекта представляет использование фигурных скобок: var user = {};

На сегодняшний день более распространенным является второй способ.

### 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:

|  |  |
| --- | --- |
|  | var user = {}; user.name = "Tom"; user.age = 26; |

В данном случае объявляются два свойства name и age, которым присваиваются соответствующие значения. После этого мы можем использовать эти свойства, например, вывести их значения в консоли:

|  |  |
| --- | --- |
|  | console.log(user.name); console.log(user.age); |

Также можно определить свойства при определении объекта:

|  |  |
| --- | --- |
|  | var user = {      name: "Tom",      age: 26  }; |

В этом случае для присвоения значения свойству используется символ двоеточия, а после определения свойства ставится запятая (а не точка с запятой). Кроме того, доступен сокращенный способ определения свойств:

|  |  |
| --- | --- |
|  | var name = "Tom";  var age = 34;  var user = {name, age};  console.log(user.name);     // Tom  console.log(user.age);      // 34 |

В данном случае названия переменных также являются и названиями свойств объекта. И таким образом можно создавать более сложные конструкции:

|  |  |
| --- | --- |
|  | var name = "Tom";  var age = 34;  var user = {name, age};  var teacher = {user, course: "JavaScript"};  console.log(teacher.user);      // {name: "Tom", age: 34}  console.log(teacher.course);    // JavaScript |

### Методы объекта

Методы объекта определяют его поведение или действия, которые он производит. Методы представляют собой функции. Например, определим метод, который бы выводил имя и возраст человека:

|  |  |
| --- | --- |
|  | var user = {};  user.name = "Tom";  user.age = 26;  user.display = function(){      console.log(user.name);      console.log(user.age);  };  // вызов метода  user.display(); |

Как и в случае с функциями методы сначала определяются, а потом уже вызываются. Также методы могут определяться непосредственно при определении объекта:

|  |  |
| --- | --- |
|  | var user = {      name: "Tom",      age: 26,      display: function(){          console.log(this.name);          console.log(this.age);      }  }; |

Как и в случае со свойствами, методу присваивается ссылка на функцию с помощью знака двоеточия. Чтобы обратиться к свойствам или методам объекта внутри этого объекта, используется ключевое слово **this**. Оно означает ссылку на текущий объект. Также можно использовать сокращенный способ определения методов, когда двоеточие и слово function опускаются:

|  |  |
| --- | --- |
|  | var user = {      name: "Tom",      age: 26,      display(){             console.log(this.name, this.age);      },      move(place){          console.log(this.name, "goes to", place);      }  };  user.display(); // Tom 26  user.move("the shop");  // Tom goes to the shop |

### Синтаксис массивов

Существует также альтернативный способ определения свойств и методов с помощью синтаксиса массивов:

|  |  |
| --- | --- |
|  | var user = {};  user["name"] = "Tom";  user["age"] = 26;  user["display"] = function(){      console.log(user.name);      console.log(user.age);  };    // вызов метода  user["display"](); |

Название каждого свойства или метода заключается в кавычки и в квадратные скобки, затем им также присваивается значение. Например, user["age"] = 26. При обращении к этим свойствам и методам можно использовать либо нотацию точки (user.name), либо обращаться так: user["name"]

### Строки в качестве свойств и методов

Также следует отметить, что названия свойств и методов объекта всегда представляют строки. То есть мы могли предыдущее определение объекта переписать так:

|  |  |
| --- | --- |
|  | var user = {      "name": "Tom",      "age": 26,      "display": function(){            console.log(user.name);          console.log(user.age);      }  };  // вызов метода  user.display(); |

С одной стороны, разницы никакой нет между двумя определениями. С другой стороны, бывают случаи, где заключение названия в строку могут помочь. Например, если название свойства состоит из двух слов, разделенных пробелом:

|  |  |
| --- | --- |
|  | var user = {      name: "Tom",      age: 26,      "full name": "Tom Johns",      "display info": function(){            console.log(user.name);          console.log(user.age);      }  };  console.log(user["full name"]);  user["display info"](); |

Только в этом случае для обращении к подобным свойствам и методам мы должны использовать синтаксис массивов.

### Удаление свойств

Выше мы посмотрели, как можно динамически добавлять новые свойства к объекту. Однако также мы можем удалять свойства и методы с помощью оператора **delete**. И как и в случае с добавлением мы можем удалять свойства двумя способами. Певый способ - использование нотации точки: delete объект.свойство. Либо использовать синтаксис массивов: delete объект["свойство"]. Например, удалим свойство:

|  |  |
| --- | --- |
|  | var user = {};  user.name = "Tom";  user.age = 26;  user.display = function(){      console.log(user.name);      console.log(user.age);  };  console.log(user.name); // Tom  delete user.name; // удаляем свойство  // альтернативный вариант  // delete user["name"];  console.log(user.name); // undefined |

После удаления свойство будет не определено, поэтому при попытке обращения к нему, программа вернет значение undefined.

## Вложенные объекты и массивы в объектах

Одни объекты могут содержать в качестве свойств другие объекты. Например, есть объект страны, у которой можно выделить ряд свойств. Одно из этих свойств может представлять столицу. Но у столицы мы также можем выделить свои свойства, например, название, численность населения, год основания:

|  |  |
| --- | --- |
|  | var country = {       name: "Германия",      language: "немецкий",      capital:{          name: "Берлин",          population: 3375000,          year: 1237      }  };  console.log("Столица: " + country.capital.name); // Берлин  console.log("Население: " + country["capital"]["population"]); // 3375000  console.log("Год основания: " + country.capital["year"]); // 1237 |

Для доступа к свойствам таких вложенных объектов мы можем использовать стандартную нотацию точки: country.capital.name

Либо обращаться к ним как к элементам массивов: country["capital"]["population"]

Также допустим смешанный вид обращения: country.capital["year"]

В качестве свойств также могут использоваться массивы, в том числе массивы других объектов:

|  |  |
| --- | --- |
|  | var country = {      name: "Швейцария",      languages: ["немецкий", "французский", "итальянский"],      capital:{           name: "Берн",          population: 126598      },      cities: [          { name: "Цюрих", population: 378884},          { name: "Женева", population: 188634},          { name: "Базель", population: 164937}      ]  };    // вывод всех элементов из country.languages  document.write("<h3>Официальные языки Швейцарии</h3>");  for(var i=0; i < country.languages.length; i++)      document.write(country.languages[i] + "<br/>");  // вывод всех элементов из country.cities  document.write("<h3>Города Швейцарии</h3>");  for(var i=0; i < country.cities.length; i++)      document.write(country.cities[i].name + "<br/>"); |

В объекте country имеется свойство languages, содержащее массив строк, а также свойство cities, хранящее массив однотипных объектов. С этими массивами мы можем работать также, как и с любыми другими, например, перебрать с помощью цикла for. При переборе массива объектов каждый текущий элемент будет представлять отдельный объект, поэтому мы можем обратиться к его свойствам и методам:

|  |  |
| --- | --- |
| 1 | country.cities[i].name |

В итоге браузер выведет содержимое этих массивов:

## Проверка наличия и перебор методов и свойств

При динамическом определении в объекте новых свойств и методов перед их использованием бывает важно проверить, а есть ли уже такие методы и свойства. Для этого в javascript может использоваться оператор **in**:

|  |  |
| --- | --- |
|  | var user = {};  user.name = "Tom";  user.age = 26;  user.display = function(){      console.log(user.name);      console.log(user.age);  };  var hasNameProp = "name" in user;  console.log(hasNameProp); // true - свойство name есть в user  var hasWeightProp = "weight" in user;  console.log(hasWeightProp); // false - в user нет свойства или метода под названием weight |

Оператор in имеет следующий синтаксис: "свойство|метод" in объект - в кавычках идет название свойства или метода, а после in - название объекта. Если свойство или метод с подобным именем имеется, то оператор возвращает true. Если нет - то возвращается false. Альтернативный способ заключается на значение **undefined**. Если свойство или метод равен undefined, то эти свойство или метод не определены:

|  |  |
| --- | --- |
|  | var hasNameProp = user.name!==undefined;  console.log(hasNameProp); // true  var hasWeightProp = user.weight!==undefined;  console.log(hasWeightProp); // false |

И так как объекты представляют тип Object, а значит, имеет все его методы и свойства, то объекты также могут использовать метод**hasOwnProperty()**, который определен в типе Object:

|  |  |
| --- | --- |
|  | var hasNameProp = user.hasOwnProperty('name');  console.log(hasNameProp); // true  var hasDisplayProp = user.hasOwnProperty('display');  console.log(hasDisplayProp); // true  var hasWeightProp = user.hasOwnProperty('weight');  console.log(hasWeightProp); // false |

**Перебор свойств и методов**

С помощью цикла for мы можем перебрать объект как обычный массив и получить все его свойства и методы и их значения:

|  |  |
| --- | --- |
|  | var user = {};  user.name = "Tom";  user.age = 26;  user.display = function(){        console.log(user.name);      console.log(user.age);  };  for(var key in user) {      console.log(key + " : " + user[key]);  } |

И при запуске консоль браузера отобразит следующий вывод:

name : Tom

age : 26

display : function (){

console.log(user.name);

console.log(user.age);

}

## Объекты в функциях

Функции могут возвращать значения. Но эти значения не обязательно должны представлять примитивные данные - числа, строки, но также могут быть сложными объектами. Например, вынесем создание объекта user в отдельную функцию:

|  |  |
| --- | --- |
|  | function createUser(pName, pAge) {      return {          name: pName,          age: pAge,          displayInfo: function() {              document.write("Имя: " + this.name + " возраст: " + this.age + "<br/>");          }      };  };  var tom = createUser("Tom", 26);  tom.displayInfo();  var alice = createUser("Alice", 24);  alice.displayInfo(); |

Здесь функция createUser() получает значения pName и pAge и по ним создает новый объект, который является возвращаемым результатом. Преимуществом вынесения создания объекта в функцию является то, что далее мы можем создать несколько однотипных объектов с разными значениями. Кроме того объект может передаваться в качестве параметра в функцию:

|  |  |
| --- | --- |
|  | function createUser(pName, pAge) {      return {          name: pName,          age: pAge,          displayInfo: function() {              document.write("Имя: " + this.name + " возраст: " + this.age + "<br/>");          },          driveCar: function(car){              document.write(this.name + " ведет машину " + car.name + "<br/>");          }      };  };    function createCar(mName, mYear){      return{          name: mName,          year: mYear      };  };  var tom = createUser("Том", 26);  tom.displayInfo();  var bently = createCar("Бентли", 2004);  tom.driveCar(bently); |

Здесь используются две функции для создания пользователей и объекта машины. Метод driveCar() объекта user в качестве параметра принимает объект car.

В итоге браузер нам выведет:

Имя: Том возраст: 26

Том ведет машину Бентли

## Конструкторы объектов

Кроме создания новых объектов JavaScript предоставляет нам возможность создавать новые типы объектов с помощью **конструкторов**. Так, одним из способов создания объекта является применение конструктора типа Object:

|  |  |
| --- | --- |
|  | var tom = new Object(); |

После создания переменной tom она будет вести себя как объект типа Object. Конструктор позволяет определить новый тип объекта. Тип представляет собой абстрактное описание или шаблон объекта. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке - наличие двух рук, двух ног, головы, пищеварительной, нервной системы и т.д. Есть некоторый шаблон - этот шаблон можно назвать типом. Реально же существующий человек является объектом этого типа. Определение типа может состоять из функции конструктора, методов и свойств. Для начала определим конструктор:

|  |  |
| --- | --- |
|  | function User(pName, pAge) {      this.name = pName;      this.age = pAge;      this.displayInfo = function(){          document.write("Имя: " + this.name + "; возраст: " + this.age + "<br/>");      };  } |

Конструктор - это обычная функция за тем исключением, что в ней мы можем установить свойства и методы. Для установки свойств и методов используется ключевое слово **this**:

|  |  |
| --- | --- |
|  | this.name = pName; |

В данном случае устанавливаются два свойства name и age и один метод displayInfo. Как правило, названия конструкторы в отличие от названий обычных функций начинаются с большой буквы. После этого в программе мы можем определить объект типа User и использовать его свойства и методы:

|  |  |
| --- | --- |
|  | var tom = new User("Том", 26);  console.log(tom.name); // Том  tom.displayInfo(); |

Чтобы вызвать конструктор, то есть создать объект типа User, надо использовать ключевое слово **new**. Подобным образом мы можем определить и другие типы и использовать их вместе:

|  |  |
| --- | --- |
|  | // конструктор типа Car  function Car(mName, mYear){      this.name = mName;      this.year = mYear;      this.getCarInfo = function(){          document.write("Модель: " + this.name + "  Год выпуска: " + this.year + "<br/>");      };  };  // конструктор типа User  function User(pName, pAge) {      this.name = pName;      this.age = pAge;      this.driveCar = function(car){          document.write(this.name + " ведет машину " + car.name + "<br/>");      };      this.displayInfo = function(){          document.write("Имя: " + this.name + "; возраст: " + this.age + "<br/>");      };  };    var tom = new User("Том", 26);  tom.displayInfo();  var bently = new Car("Бентли", 2004);  tom.driveCar(bently); |

**Оператор instanceof**

Оператор **instanceof** позволяет проверить, с помощью какого конструктора создан объект. Если объект создан с помощью определенного конструктора, то оператор возвращает true:

|  |  |
| --- | --- |
|  | var tom = new User("Том", 26);  var isUser = tom instanceof User;  var isCar = tom instanceof Car;  console.log(isUser);    // true  console.log(isCar);     // false |

## Расширение объектов. Prototype

Кроме непосредственного определения свойств и методов в конструкторе мы также можем использовать свойство **prototype**. Каждая функция имеет свойство **prototype**, представляющее прототип функции. То есть свойство User.prototype представляет прототип объектов User. И любые свойства и методы, которые будут определены в User.prototype, будут общими для всех объектов User. Например, после определения объекта User нам потребовалось добавить к нему метод и свойство:

|  |  |
| --- | --- |
|  | function User(pName, pAge) {      this.name = pName;      this.age = pAge;      this.displayInfo = function(){          document.write("Имя: " + this.name + "; возраст: " + this.age + "<br/>");      };  };    User.prototype.hello = function(){      document.write(this.name + " говорит: 'Привет!'<br/>");  };  User.prototype.maxAge = 110;    var tom = new User("Том", 26);  tom.hello();  var john = new User("Джон", 28);  john.hello();  console.log(tom.maxAge); // 110  console.log(john.maxAge); // 110 |

Здесь добавлены метод hello и свойство maxAge, и каждый объект User сможет их использовать. Но важно заметить, что значение свойства maxAge будет одно и то же для всех объектов, это разделяемое статическое свойство. В отличие, скажем, от свойстваthis.name, которое хранит значение для определенного объекта. В то же время мы можем определить в объекте свойство, которое будет назваться также, как и свойство прототипа. В этом случае собственное свойство объекта будет иметь приоритет перед свойством прототипа:

|  |  |
| --- | --- |
|  | User.prototype.maxAge = 110;  var tom = new User("Том", 26);  var john = new User("Джон", 28);  tom.maxAge = 99;  console.log(tom.maxAge); // 99  console.log(john.maxAge); // 110 |

И при обращении к свойству maxAge javascript сначала ищет это свойство среди свойств объекта, и если оно не было найдено, тогда обращается к свойствам прототипа. То же самое касается и методов.

## 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.

|  |  |
| --- | --- |
|  | function User(pName, pAge) {      this.name = pName;      this.age = pAge;      this.displayInfo = function(){          document.write("Имя: " + this.name + "; возраст: " + this.age);      };  };  var tom = new User("Том", 26);  tom.name=34;  console.log(tom.name); |

Но мы можем их скрыть от доступа извне, сделав свойства локальными переменными:

|  |  |
| --- | --- |
|  | function User (name, age) {      this.name = name;      var \_age = age;      this.displayInfo = function(){          document.write("Имя: " + this.name + "; возраст: " + \_age + "<br>");      };      this.getAge = function() {          return \_age;      }      this.setAge = function(age) {          if(typeof age === "number" && age >0 && age<110){              \_age = age;          } else {              console.log("Недопустимое значение");          }      }  }  var tom = new User("Том", 26);  console.log(tom.\_age); // undefined - \_age - локальная переменная  console.log(tom.getAge()); // 26  tom.setAge(32);  console.log(tom.getAge()); // 32  tom.setAge("54"); // Недопустимое значение |

В конструкторе User объявляется локальная переменная \_age вместо свойства age. Как правило, названия локальных переменных в конструкторах начинаются со знака подчеркивания. Для того, чтобы работать с возрастом пользователя извне, определяются два метода. Метод getAge() предназначен для получения значения переменной \_age. Этот метод еще называется геттер (getter). Второй метод - setAge, который еще называется сеттер (setter), предназначен для установки значения переменной \_age. Плюсом такого подхода является то, что мы имеем больший контроль над доступом к значению \_age. Например, мы можем проверить какие-то сопутствующие условия, как в данном случае проверяются тип значение (он должен представлять число), само значение (возраст не может быть меньше 0).

## Функция как объект. Методы call и apply

В JavaScript функция тоже является объектом - объектом **Function** и тоже имеет прототип, свойства, методы. Все функции, которые используются в программе, являются объектами Function и имеют все его свойства и методы. Например, мы можем создать функцию с помощью конструктора Function:

|  |  |
| --- | --- |
|  | var square = new Function('n', 'return n \* n;');  console.log(square(5)); |

В конструктор Function может передаваться ряд параметров. Последний параметр представляет собой само тело функции в виде строки. Фактически строка содержит код javascript. Предыдущие аргументы содержат названия параметров. В данном случае определяется функция возведения числа в квадрат, которая имеет один параметр n. Среди свойств объекта Function можно выделить следующие:

* **arguments**: массив аргументов, передаваемых в функцию
* **length**: определяет количество аргументов, которые ожидает функция
* **caller**: определяет функцию, вызвавшую текущую выполняющуюся функцию
* **name**: имя функции
* **prototype**: прототип функции

С помощью прототипа мы можем определить дополнительные свойства:

|  |  |
| --- | --- |
|  | function display(){      console.log("привет мир");  }  Function.prototype.program ="Hello";  console.log(display.program); // Hello |

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

|  |  |
| --- | --- |
|  | function add(x, y){     return x + y;  }  var result = add.call(this, 3, 8);  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 ссылаться на него через ключевое слово this:

|  |  |
| --- | --- |
|  | function User (name, age) {      this.name = name;      this.age = age;  }  var tom = new User("Том", 26);  function display(){      console.log("Ваше имя: " + this.name);  }  display.call(tom); // Ваше имя: Том |

В данном случае передается только одно значение, поскольку функция display не принимает параметров. То есть функция будет вызываться для объекта tom. Если нам не важен объект, для которого вызывается функция, то можно передать значение null:

|  |  |
| --- | --- |
|  | function add(x, y){      return x + y;  }  var result = add.call(null, 3, 8);  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:

|  |  |
| --- | --- |
|  | function add(x, y){      return x + y;  }  var result = add.apply(null, [3, 8]);  console.log(result); // 11 |

## Наследование

JavaScript поддерживает наследование, то позволяет нам при создании новых типов объектов при необходимости унаследовать их функционал от уже существующих. Например, у нас может быть объект User, представляющий отдельного пользователя. И также может быть объект Employee, который представляет работника. Но работник также может являться пользователем и поэтому должен иметь все его свойства и методы. Например:

|  |  |
| --- | --- |
|  | // конструктор пользователя  function User (name, age) {      this.name = name;      this.age = age;      this.go = function(){document.write(this.name + " идет <br/>");}      this.displayInfo = function(){          document.write("Имя: " + this.name + "; возраст: " + this.age + "<br/>");      };  }  User.prototype.maxage = 110;   // конструктор работника  function Employee(name, age, comp){      User.call(this, name, age);      this.company = comp;      this.displayInfo = function(){          document.write("Имя: " + this.name + "; возраст: " + this.age + "; компания: " + this.company + "<br/>");      };  }  Employee.prototype = Object.create(User.prototype);  var tom = new User("Том", 26);  var bill = new Employee("Билл", 32, "Google");  tom.go();  bill.go();  tom.displayInfo();  bill.displayInfo();  console.log(bill.maxage); |

Здесь в начале определяет конструктор User и к его прототипу добавляется свойство maxage. Затем определяется тип Employee. В конструкторе Employee происходит обращение к конструктору User с помощью вызова:

|  |  |
| --- | --- |
| 1 | User.call(this, name, age); |

Передача первого параметра позволяет вызвать функцию конструктора User для объекта, создаваемого конструктором Employee. Благодаря этому все свойства и методы, определенные в конструкторе User, также переходят на объект Employee. Кроме того, необходимо унаследовать также и прототип User. Для этого служит вызов:

|  |  |
| --- | --- |
| 1 | Employee.prototype = Object.create(User.prototype); |

Метод Object.create() позволяет создать объект прототипа User, который затем присваивается прототипу Employee. При этом при необходимости в прототипе Employee мы также можем определить дополнительные свойства и методы. При наследовании мы можем переопределять наследуемый функционал. Например, Employee переопределяет метод displayInfo(), унаследованный от User, чтобы включить в вывод этого метода новое свойство company.

## Ключевое слово this

Поведение ключевого слова **this** зависит от контекста, в котором оно используется, и от того, в каком режиме оно используется - строгом или нестрогом.

**Глобальный контекст**

В глобальном контексте this ссылается на глобальный объект. В данном случае поведение не зависит от режима (строгий или нестрогий):

|  |  |
| --- | --- |
|  | this.alert("global alert");  this.console.log("global console");  var currentDocument = this.document; |

Но в данном случае использвание this избыточно.

**Контекст функции**

Если скрипт запускается в строгом режиме (директива "use strict"), то this ссылается непосредственно на контекст функции. Иначе this ссылается на внешний контекст. Например:

|  |  |
| --- | --- |
|  | function foo(){      var bar = "bar2";      console.log(this.bar);  }  var bar = "bar1";  foo();  // bar1 |

Вообщем-то смысла использования this в данном случае нет, так как мы могли бы напрямую использовать локальную переменную. Если бы мы использовали строгий режим, то this в этом случае имело бы значение undefined:

|  |  |
| --- | --- |
|  | "use strict";  function foo(){      var bar = "bar2";      console.log(this.bar);  }  var bar = "bar1";  foo();  // ошибка - this - undefined |

**Контекст объекта**

В контексте объекта, в том числе в его методах, ключевое слово this ссылается на этот же объект:

|  |  |
| --- | --- |
|  | var o = {      bar: "bar3",      foo: function(){          console.log(this.bar);      }  }  var bar = "bar1";  o.foo();    // bar3 |

**Примеры**

Рассмотрим более сложный пример:

|  |  |
| --- | --- |
|  | function foo(){      var bar = "bar2";      console.log(this.bar);  }  var o3 = {bar:"bar3", foo: foo};  var o4 = {bar:"bar4", foo: foo};  var bar = "bar1";  foo();  // bar1  o3.foo();   // bar3  o4.foo();   // bar4 |

Здесь определена глобальная переменная bar. И также в функции foo определена локальная переменная bar. Значение какой переменной будет выводиться в функции foo? Функция foo выводит значение глобальной переменной, так как данный скрипт запускается в нестрогом режиме, а значит ключеое слово this в функции foo ссылается на внешний контекст. Иначе дело обстоит с объектами. Они определяют свой собственный контекст, в котором существует свое свойство bar. И при вызове метода foo внешним контекстом по отношению к функции будет контекст объектов o3 и o4. Подобное поведение может привести к некоторому непонимаю в отдельных случаях. Так, рассмотрим другую ситуацию:

|  |  |
| --- | --- |
|  | var o1 = {      bar: "bar1",      foo: function(){          console.log(this.bar);      }  }  var o2 = {bar: "bar2", foo: o1.foo};  var bar = "bar3";  var foo = o1.foo;  o1.foo();   // bar1  o2.foo();   // bar2  foo();      // bar3 |

Несмотря на то, что объект o2 использует метод foo из объекта o1, тем не менее функция o1.foo также будет искать значение для this.bar во внешнем котексте, то есть в контексте объекта o2. А в объекте o2 это значение равно bar: "bar2". То же самое с глобальной переменной foo, которая ссылается на ту же функцию, что и метод o1.foo. В этом случае также будет происходить поиск значения для this.bar во внешним контексте, то есть в глобальном контексте, где определена переменная var bar = "bar3". Однако если мы вызываем функцию из другой функции, вызываемая функция также будет использовать внешний контекст:

|  |  |
| --- | --- |
|  | var bar = "bar2";  function daz(){      var bar = "bar5";      function maz(){          console.log(this.bar);      }      maz();  }  daz();  // bar2 |

Здесь функция daz в качестве this.bar использует значение переменной bar из внешнего контекста, то есть значение глобальной переменной bar. Функция maz также в качестве this.bar использует значение переменной bar из внешнего контекста, а это значение this.bar из внешней функции daz, которое в свою очередь представляет значение глобальной переменной bar. Поэтому в итоге консоль выведет "bar2", а не "bar5".

**Явная привязка**

С помощью методов call() и apply() можно задать явную привязку функции к определенному контексту:

|  |  |
| --- | --- |
|  | function foo(){      console.log(this.bar);  }  var o3 = {bar: "bar3"}  var bar = "bar1";  foo();  // bar1  foo.apply(o3);  // bar3  // или  // foo.call(o3); |

Во втором случае функция foo привязывается к объекту o3, который и определяет ее контекст. Поэтому во втором случае консоль выведет "bar3".

**Метод bind**

Метод f.bind(o) позволяет создать новую функцию с тем же телом и областью видимости, что и функция f, но с привязкой к объекту o:

|  |  |
| --- | --- |
|  | function foo(){      console.log(this.bar);  }  var o3 = {bar: "bar3"}  var bar = "bar1";  foo();  // bar1  var func = foo.bind(o3);  func(); // bar3 |

**this и стрелочные функции**

При работе с несколькими контекстами мы вынуждены учитывать, в каком контексте определяется переменная. Например, возьмем следующий код:

|  |  |
| --- | --- |
|  | var school ={      title: "Oxford",      courses: ["JavaScript", "TypeScript", "Java", "Go"],      printCourses: function(){          this.courses.forEach(function(course){              console.log(this.title, course);          })      }  }  school.printCourses(); |

Функция printCourses проходит по всем курсам из массива и при их выводе предваряет их значением свойства title. Однако на консоли при запуске программы мы увидим следующее:

undefined "JavaScript"

undefined "TypeScript"

undefined "Java"

undefined "Go"

Мы видим, что значение this.title не определено, так как this как контекст объекта замещается глобальным контекстом. В этом случае нам надо передать подобное значение this.title или весь контекст объекта.

|  |  |
| --- | --- |
|  | var school ={      title: "Oxford",      courses: ["JavaScript", "TypeScript", "Java", "Go"],      printCourses: function(){          var that = this;          this.courses.forEach(function(course){              console.log(that.title, course);          })      }  }  school.printCourses(); |

Стрелочные функции также позволяют решить данную проблему:

|  |  |
| --- | --- |
|  | var school ={      title: "Oxford",      courses: ["JavaScript", "TypeScript", "Java", "Go"],      printCourses: function(){          this.courses.forEach((course)=>console.log(this.title, course))      }  }  school.printCourses(); |

Контекстом для стрелочной функции в данном случае будет выступать контекст объекта school. Соответственно, нам недо определять дополнительные переменые для передачи данных в функцию.

## Декомпозиция

Декомпозиция (destructuring) позволяет извлечь из объекта отдельные значения в переменные:

|  |  |
| --- | --- |
|  | let user = {      name: "Tom",      age: 24,      phone: "+367438787",      email: "tom@gmail.com"  };  let {name, email} = user;  console.log(name);      // Tom  console.log(email);     // tom@gmail.com |

Для декомпозиции объекта переменные помещаются в фигурные скобки и им присваивается объект. Сопоставление между свойствами объекта и переменными идет по имени. Мы можем указать, что мы хотим получить значения свойств объекта в переменные с другим именем:

|  |  |
| --- | --- |
|  | let user = {      name: "Tom",      age: 24,      phone: "+367438787",      email: "tom@gmail.com"  };  let {name: userName, email: mailAddress} = user;  console.log(userName);      // Tom  console.log(mailAddress);       // tom@gmail.com |

В данном случае свойство name сопоставляется с переменной userName, а поле email - с переменной mailAddress.

**Декомпозиция массивов**

Также можно декомпозировать массивы:

|  |  |
| --- | --- |
|  | let users = ["Tom", "Sam", "Bob"];  let [a, b, c] = users;    console.log(a);     // Tom  console.log(b);     // Sam  console.log(c);     // Bob |

Для декомпозиции массива переменные помещаются в квадратные скобки и последовательно получают значения элементов массива. При эом мы можем пропустить ряд элементов массива, оставив вместо имен переменных пропуски:

|  |  |
| --- | --- |
|  | let users = ["Tom", "Sam", "Bob", "Ann", "Alice", "Kate"];  let [first,,,,fifth] = users;    console.log(first);     // Tom  console.log(fifth);     // Alice |

Выражение first,,,,fifth указывает, что мы хотим получить первый элемент массива в переменную first, затем пропустить три элемента и получить пятый элемент в переменную fifth. Подобным образом можно получить, например, второй и четвертый элементы:

|  |  |
| --- | --- |
|  | let users = ["Tom", "Sam", "Bob", "Ann", "Alice", "Kate"];  let [,second,,forth] = users;  console.log(second);        // Sam  console.log(forth);         // Ann |

Можно совместить получение данных из массива и объекта:

|  |  |
| --- | --- |
|  | let people = [      {name: "Tom", age: 34},      {name: "Bob", age: 23},      {name: "Sam", age: 32}  ];  let [,{name}] = people;  console.log(name);      // Bob |

В данном случае получаем значение свойства name второго объекта в массиве.

**Декомпозиция параметров**

Если в функцию в качестве параметра передается массив или объект, то его также можно подобным образом декомпозировать:

|  |  |
| --- | --- |
|  | function display({name:userName, age:userAge}){      console.log(userName, userAge);  }  function sum([a, b, c]){      var result = a + b + c;      console.log(result);  }  let user = {name:"Alice", age:33, email: "alice@gmail.com"};  let numbers = [3, 5, 7, 8];  display(user);  // Alice 33  sum(numbers);   // 15 |

## Классы

С внедреием стандарта ES2015 (ES6) в JavaScript появился новый способ определения объектов - с помощью классов. Класс представляет описание объекта, его состояния и поведения, а объект является конкретным воплощением или экземпляром класса. Для определения класса используется ключевое слово **class**:

|  |  |
| --- | --- |
|  | class Person{  } |

После слова class идет название класса (в данном случае класс называется Person), и затем в фигурных скобках определяется тело класса. Также можно определить анонимный класс и присвоить его переменной:

|  |  |
| --- | --- |
|  | let Person = class{} |

После этого мы можем создать объекты класса с помощью конструктора:

|  |  |
| --- | --- |
| 1  2  3  4 | class Person{}    let tom = new Person();  let bob = new Person(); |

Для создания объекта с помощью конструктора сначала ставится ключевое слово **new**. Затем собственно идет вызов конструктора - по сути вызов функции по имени класса. По умолчанию классы имеют один конструктор без параметров. Поэтому в данном случае при вызове конструктора в него не передается никаких аргументов. 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 |

Конструктор определяется с помощью метода с именем **constructor**. По сути это обычный метод, который может принимать параметры. Основная цель конструктора - инициализировать объект начальными данными. И в данном случае в конструктор передаются два значения - для имени и возраста пользователя. Для хранения состояния в классе определяются свойства. Для их определения используется ключевое слово **this**. В данном случае в классе два свойства: name и age. Поведение класса определяют методы. В данном случае определен метод display(), который выводит значения свойств на консоль. Поскольку мы определили конструктор, который принимает два параметра, то соответственно мы можем передать в конструктор два значения для инициализации объекта: new Person("Tom", 34).

**Наследование**

Одни классы могут наследоваться от других. Наследование позволяет сократить объем кода в классах-наследниках. Например, определим следеющие классы:

|  |  |
| --- | --- |
|  | class Person{      constructor(name, age){          this.name = name;          this.age = age;      }      display(){          console.log(this.name, this.age);      }  }  class Employee extends Person{      constructor(name, age, company){          super(name, age);          this.company = company;      }      display(){          super.display();          console.log("Employee in", this.company);      }      work(){          console.log(this.name, "is hard working");      }  }  let tom = new Person("Tom", 34);  let bob = new Employee("Bob", 36, "Google");  tom.display();  bob.display();  bob.work(); |

Для наследования одного класса от другого в определении класса применяется оператор **extends**, после которого идет название базового класса. То есть в данном случае класс Employee наследуется от класса Person. Класс Person еще называется базовым классом, классом-родителем, суперклассом, а класс Employee - классом-наследником, подклассом, производным классом. Производный класс, как и базовый, может определять конструкторы, свойства, методы. Вместе с тем с помощью слова **super**производный класс может ссылаться на функционал, определенный в базовом. Например, в конструкторе Employee можно вручную не устанавливать свойства name и age, а с помощью выражения super(name, age); вызвать конструктор базового класса и тем самым передать работу по установке этих свойств базовому классу. Подобным образом в методе display в классе Employee через вызов super.display() можно обратиться к реализации метода display класса Person. Консольный вывод программы:

Tom 34

Bob 36

Employee in Google

Bob is hard working

**Статические методы**

Статические методы вызываются для всего класса вцелом, а не для отедельного объекта. Для их определения применяется оператор **static**. Например:

|  |  |
| --- | --- |
|  | class Person{      constructor(name, age){          this.name = name;          this.age = age;      }      static nameToUpper(person){          return person.name.toUpperCase();      }      display(){          console.log(this.name, this.age);      }  }  let tom = new Person("Tom Soyer", 34);  let personName = Person.nameToUpper(tom);  console.log(personName);        // TOM SOYER |

В данном случае определен статический метод nameToUpper(). В качестве параметра он принимает объект Person и переводит его имя в верхний регистр. Поскольку статический метод относится классу вцелом, а не к объекту, то мы НЕ можем использовать в нем ключевое слово this и через него обращаться к свойствам объекта.

# **Встроенные объекты**

## Объект Date. Работа с датами

Объект **Date** позволяет работать с датами и временем в JavaScript. Существуют различные способы создания объекта Date. Первый способ заключается в использовании пустого конструктора без параметров:

var currentDate = new Date(); document.write(currentDate);

Второй способ заключается в передаче в конструктор Date количества миллисекунд, которые прошли с начала эпохи Unix, то есть с 1 января 1970 года 00:00:00 GMT:

|  |  |
| --- | --- |
| 1  2 | var myDate = new Date(1359270000000);  document.write(myDate); // Sun Jan 27 2013 10:00:00 GMT+0300 (RTZ 2 (зима)) |

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

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

Если мы используем полное название месяца, то оно пишется в по-английски, если используем сокращенный вариант, тогда используется формат месяц/день/год.

Четвертый способ состоит в передаче в конструктор Date всех параметров даты и времени:

|  |  |
| --- | --- |
|  | var myDate = new Date(2012,11,25,18,30,20,10); // Tue Dec 25 2012 18:30:20 GMT+0300 (RTZ 2 (зима)) |

В данном случае используются по порядку следующие параметры: new Date(год, месяц, число, час, минуты, секунды, миллисекунды). При этом надо учитывать, что отсчет месяцев начинается с нуля, то есть январь - 0, а декабрь - 11.

### 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
* **getHours()**: возвращает час (от 0 до 23)
* **getMinutes()**: возвращает минуты (от 0 до 59)
* **getSeconds()**: возвращает секунды (от 0 до 59)
* **getMilliseconds()**: возвращает миллисекунды (от 0 до 999)
* **toTimeString()**: возвращает полное время в виде строки

Получим текущую дату:

|  |  |
| --- | --- |
|  | var days = ["Воскресенье", "Понедельник", "Вторник", "Среда", "Четверг", "Пятница", "Суббота"];  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 |

Перевести из числовых значений в более привычные названия для дней недели и месяцев используются массивы. Получив индекс дня недели (myDate.getDay()) и индекс месяца (myDate.getMonth()) можно получить нужный элемент из массива. Теперь получим текущее время:

|  |  |
| --- | --- |
|  | var welcome;  var myDate = new Date();  var hour = myDate.getHours();  var minute = myDate.getMinutes();  var second = myDate.getSeconds();  if (minute < 10) {      minute = "0" + minute;  }  if (second < 10) {      second = "0" + second;  }  if (hour < 12) {      welcome = "Доброе утро";  } else if (hour < 17) {      welcome = "Добрый день";  } else {      welcome = "Добрый вечер";  }  document.write(welcome + ", текущее время: " + hour +                  ":" + minute + ":" + second); // Добрый вечер, текущее время: 22:50:39 |

### Установка даты и времени

Коме задания параметров даты в конструкторе для установки мы также можем использовать дополнительные методы объекта Date:

* **setDate()**: установка дня в дате
* **setMonth()**: уставовка месяца (отсчет начинается с нуля, то есть месяц с номер 0 - январь)
* **setFullYear()**: устанавливает год
* **setHours()**: установка часа
* **setMinutes()**: установка минут
* **setSeconds()**: установка секунд
* **setMilliseconds()**: установка миллисекунд

Установим дату:

|  |  |
| --- | --- |
|  | var days = ["Воскресенье", "Понедельник", "Вторник", "Среда", "Четвег", "Пятница", "Суббота"];  var months = ["Январь", "Февраль", "Март", "Апрель", "Май", "Июнь",              "Июль", "Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"];  var myDate = new Date();  myDate.setDate(15);  myDate.setMonth(6);  myDate.setYear(2013);  var fullDate = myDate.getDate() + " " + months[myDate.getMonth()] +                  " " + myDate.getFullYear() + ", " + days[myDate.getDay()];  document.write(fullDate); // 15 Июль 2013, Понедельник |

При установке значений мы можем передать величину, большую, чем максимальное допустимое значение. Например, установить для часа число 54:

|  |  |
| --- | --- |
| 1 | myDate.setHour(54); |

В этом случае значение часа будет равно 54 - 24 \* 2 = 6, а оставшиеся часы будут составлять два дня (24 \* 2), что прибавит к дате два дня. То же самое действует и в отношении дней, минут, секунд, миллисекунд и месяцев.

## **Объект Math. Математические операции**

### abs() - возвращает абсолютное значение числа:

var x = -25; var y = 34;

document.write(Math.abs(x)); // 25

document.write(Math.abs(y)); // 34

### min() и max() - возвращают миним. и максим. значение из набора чисел:

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

var min = Math.min(33, 24); // 24

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

var x = Math.ceil(9.2); // 10

var y = Math.ceil(-5.9); // -5

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

var x = Math.floor(9.2); // 9

var y = Math.floor(-5.9); // -6

### round() - округляет число до следующего наименьшего целого, если его десятичная часть меньше 0.5 и если больше 0.5, то 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

### random() - возвращает случайное число с плавающей точкой их диапазона от 0 до 1:

var x = Math.random();

### pow() - возвращает число в определенной степени.

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

### sqrt() - возвращает квадратный корень числа:

|  |  |
| --- | --- |
|  | var y = Math.sqrt(9); // 3  var z = Math.sqrt(20); // 4.47213595499958 |

### log() - возвращает натуральный логарифм числа:

var x = Math.log(1); // 0

var z = Math.log(10); // 2.302585092994046

### Тригонометрические функции

var x = Math.sin(90); // 0.8939966636005579

var y = Math.cos(0); // 1

var z = Math.tan(45); // 1.6197751905438615

var x = Math.asin(0.9); // 1.1197695149986342

var y = Math.acos(1); // 1

var z = Math.atan(1); // 0.7853981633974483

### Набор встроенных констант

* Math.PI (число PI): 3.141592653589793
* Math.SQRT2 (квадратный корень из двух): 1.4142135623730951
* Math.SQRT1\_2 (половина от квадратного корня из двух): 0.7071067811865476
* Math.E (число e или число Эйлера): 2.718281828459045
* Math.LN2 (натуральный логарифм числа 2): 0.6931471805599453
* Math.LN10 (натуральный логарифм числа 10): 2.302585092994046
* Math.LOG2E (двоичный логарифм числа e): 1.4426950408889634
* Math.LOG10E (десятичный логарифм числа e): 0.4342944819032518

Используем константы в вычислениях:

|  |  |
| --- | --- |
|  | var x = Math.log(Math.E); // 1  var z = Math.tan(Math.PI/4); // 0.9999999999999999 |

## Объект Array. Работа с массивами

Объект Array представляет массив и предоставляет ряд свойств и методов, с помощью которых мы можем управлять массивом.

**Инициализация массива**

Можно создать пустой массив, используя квадратные скобки или конструктор Array:

|  |  |
| --- | --- |
|  | var users = new Array();  var people = [];  console.log(users); // Array[0]  console.log(people); // Array[0] |

Можно сразу же инициализировать массив некоторым количеством элементов:

|  |  |
| --- | --- |
|  | var users = new Array("Tom", "Bill", "Alice");  var people = ["Sam", "John", "Kate"];  console.log(users); // ["Tom", "Bill", "Alice"]  console.log(people); // ["Sam", "John", "Kate"] |

Можно определить массив и по ходу определять в него новые элементы:

|  |  |
| --- | --- |
|  | var users = new Array();  users[1] = "Tom";  users[2] = "Kate";  console.log(users[1]); // "Tom"  console.log(users[0]); // undefined |

При этом не важно, что по умолчанию массив создается с нулевой длиной. С помощью индексов мы можем подставить на конкретный индекс в массиве тот или иной элемент.

**length -** узнать длину массива:

|  |  |
| --- | --- |
|  | var fruit = new Array();  fruit[0] = "яблоки"; fruit[1] = "груши"; fruit[2] = "сливы";  document.write("В массиве fruit " + fruit.length + " элемента: <br/>");  for(var i=0; i < fruit.length; i++)      document.write(fruit[i] + "<br/>"); |

По факту длиной массива будет индекс последнего элемента с добавлением единицы.

|  |  |
| --- | --- |
|  | var users = new Array(); // в массиве 0 элементов  users[0] = "Tom"; users[1] = "Kate"; users[4] = "Sam";  for(var i=0; i<users.length;i++)      console.log(users[i]); |

Вывод браузера:

Tom

Kate

undefined

undefined

Sam

Несмотря на то, что для индексов 2 и 3 мы не добавляли элементов, но длиной массива в данном случае будет число 5. Просто элементы с индексами 2 и 3 будут иметь значение undefined.

**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:

|  |  |
| --- | --- |
|  | var users = ["Tom", "Sam", "Bill"];  console.log(users);     //  ["Tom", "Sam", "Bill"]  var people = users;     //  неглубокое копирование  people[1] = "Mike";     //  изменяем второй элемент  console.log(users);     //  ["Tom", "Mike", "Bill"] |

В данном случае переменная people после копирования будет указывать на тот же массив, что и переменная users. Поэтому при изменении элементов в people, изменятся элементы и в users, так как фактически это один и тот же массив. Такое поведение не всегда является желательным. Например, мы хотим, чтобы после копирования переменные указывали на отдельные массивы. И в этом случае можно использовать глубокое копирование с помощью метода **slice()**:

|  |  |
| --- | --- |
|  | var users = ["Tom", "Sam", "Bill"];  console.log(users);             //  ["Tom", "Sam", "Bill"]  var people = users.slice();     //  глубокое копирование  people[1] = "Mike";             //  изменяем второй элемент  console.log(users);             //  ["Tom", "Sam", "Bill"]  console.log(people);            //  ["Tom", "Mike", "Bill"] |

В данном случае после копирования переменные будут указывать на разные массивы, и мы сможем изменять их отдельно друг от друга. Также метод slice() позволяет скопировать часть массива:

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

В метод slice() передается начальный и конечный индексы, которые используются для выборки значений из массива. То есть в данном случае выборка в новый массив идет начиная с 1 индекса по индекс 4 не включая. И поскольку индексация массивов начинается с нуля, то в новом массиве окажутся второй, третий и четвертый элемент.

**push() -** добавляет элемент в конец массива

var fruit = [];

fruit.push("яблоки");

fruit.push("груши");

fruit.push("сливы");

fruit.push("вишня","абрикос");

document.write("В массиве fruit " + fruit.length + " элемента: <br/>");

document.write(fruit); // яблоки,груши,сливы,вишня,абрикос

**pop() -** удаляет последний элемент из массива

var fruit = ["яблоки", "груши", "сливы"];

var lastFruit = fruit.pop(); // извлекаем из массива последний элемент

document.write(lastFruit + "<br/>");

document.write("В массиве fruit " + fruit.length + " элемента: <br/>");

for(var i=0; i <fruit.length; i++)

    document.write(fruit[i] + "<br/>");

Вывод браузера:

сливы

В массиве fruit 2 элемента:

яблоки

груши

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

|  |  |
| --- | --- |
|  | var fruit = ["яблоки", "груши", "сливы"];  var firstFruit = fruit.shift();  document.write(firstFruit + "<br/>");  document.write("В массиве fruit " + fruit.length + " элемента: <br/>");  for(var i=0; i <fruit.length; i++)      document.write(fruit[i] + "<br/>"); |

Вывод браузера:

яблоки

В массиве fruit 2 элемента:

груши

сливы

**unshift() -**  добавляет новый элемент в начало массива:

|  |  |
| --- | --- |
|  | var fruit = ["яблоки", "груши", "сливы"];  fruit.unshift("абрикосы");  document.write(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:

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

Метод slice возвращает удаленные элементы. В данном случае удаление идет с начала массива. Если передать отрицательный индекс, то удаление будет производиться с конца массива. Например, удалим последний элемент:

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

Дополнительная версия метода позволяет задать конечный индекс для удаления. Например, удалим с первого по третий индекс:

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

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

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

В данном случае удаляем три элемента с 1-й по 3-й индексы и вместо них вставляем два элемента.

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

|  |  |
| --- | --- |
|  | var fruit = ["яблоки", "груши", "сливы"];  var vegetables = ["помидоры", "огурцы", "картофель"];  var products = fruit.concat(vegetables);  for(var i=0; i < products.length; i++)      document.write(products[i] + "<br/>"); |

Объединять можно и разнотипные:

var fruit = ["яблоки", "груши", "сливы"];

var prices = [20, 30, 80];

var products = fruit.concat(prices);

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

|  |  |
| --- | --- |
|  | var fruit = ["яблоки", "груши", "сливы", "абрикосы", "персики"];  var fruitString = fruit.join(", ");  document.write(fruitString); |

В метод join() передается разделитель между элементами массива. В данном случае в качестве разделителя будет использоваться запятая и пробел (", ").

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

var fruit = ["яблоки", "груши", "сливы", "абрикосы", "персики"];

fruit.sort();

for(var i=0; i < fruit.length; i++)

    document.write(fruit[i] + "<br/>");

**reverse() -** вывод массив задом наперед

var fruit = ["яблоки", "груши", "сливы", "абрикосы", "персики"];

fruit.reverse();

for(var i=0; i < fruit.length; i++)

    document.write(fruit[i] + "<br/>");

В сочетании с методом sort() можно отсортировать массив по убыванию:

|  |  |
| --- | --- |
|  | var fruit = ["яблоки", "груши", "сливы", "абрикосы", "персики"];  fruit.sort().reverse();  for(var i=0; i < fruit.length; i++)      document.write(fruit[i] + "<br/>"); |

Вывод в браузере:

яблоки

сливы

персики

груши

абрикосы

**Поиск индекса элемента**

Методы **indexOf()** и **lastIndexOf()** возвращают индекс первого и последнего включения элемента в массиве. Например:

|  |  |
| --- | --- |
|  | var fruit = ["яблоки", "груши", "сливы", "яблоки", "груши"];  var firstIndex = fruit.indexOf("яблоки");  var lastIndex = fruit.lastIndexOf("яблоки");  var otherIndex = fruit.indexOf("вишни");  document.write(firstIndex); // 0  document.write(lastIndex);  // 3  document.write(otherIndex); // -1 |

firstIndex имеет значение 0, так как первое включение стоки "яблоки" в массиве приходится на индекс 0, а последнее на индекс 3. Если же элемент отсутствует в массиве, то в этом случае методы indexOf() и lastIndexOf() возвращают значение -1.

**every() -** проверяет, все ли элементы соответствуют определенному условию:

|  |  |
| --- | --- |
|  | var numbers = [ 1, -12, 8, -4, 25, 42 ];  function condition(value, index, array) {      var result = false;      if (value > 0) {          result = true;      }      return result;  };  var passed = numbers.every(condition);  document.write(passed); // false |

В метод every() в качестве параметра передается функция, представляющая условие. Эта функция принимает три параметра:

|  |  |
| --- | --- |
|  | function condition(value, index, array) {  } |

Параметр **value** представляет текущий перебираемый элемент массива, параметр **index** представляет индекс этого элемента, а параметр **array** передает ссылку на массив. В этой функции мы можем проверить переданное значение элемента на соответствие какому-нибудь условию. Например, в данном примере мы проверяем каждый элемент массива, больше ли он нуля. Если больше, то возвращаем значение true, то есть элемент соответствует условию. Если меньше, то возвращаем false - элемент не соответствует условию. В итоге, когда происходит вызов метода numbers.every(condition) он перебирает все элементы массива numbers и по очереди передает их в функцию condition. Если эта функция возвращает значение true для всех элементов, то и метод every() возвращает true. Если хотя бы один элемент не соответствует условию, то метод every() возвращает значение false.

**some()**

Проверяет, соответствует ли хотя бы один элемент условию и тогда возвращает true иначе false:

|  |  |
| --- | --- |
|  | var numbers = [ 1, -12, 8, -4, 25, 42 ];  function condition(value, index, array) {      var result = false;      if (value === 8) {          result = true;      }      return result;  };  var passed = numbers.some(condition); // true |

**filter()**

Метод filter(), как some() и every(), принимает функцию условия. Но при этом возвращает массив тех элементов, которые соответствуют этому условию:

|  |  |
| --- | --- |
|  | var numbers = [ 1, -12, 8, -4, 25, 42 ];  function condition(value, index, array) {      var result = false;      if (value > 0) {          result = true;      }      return result;  };  var filteredNumbers = numbers.filter(condition);    for(var i=0; i < filteredNumbers.length; i++)      document.write(filteredNumbers[i] + "<br/>"); |

Вывод в браузере:

1

8

25

42

**forEach() и map()**

Методы **forEach()** и **map()** осуществляют перебор элементов и выполняют с ними определенный операции. Например, для вычисления квадратов чисел в массиве можно использовать следующий код:

|  |  |
| --- | --- |
|  | var numbers = [ 1, 2, 3, 4, 5, 6];  for(var i = 0; i<numbers.length; i++){      var result = numbers[i] \* numbers[i];      document.write("Квадрат числа " + numbers[i] + " равен " + result + "<br/>");  } |

Но с помощью метода forEach() можно упростить эту конструкцию:

|  |  |
| --- | --- |
|  | var numbers = [ 1, 2, 3, 4, 5, 6];  function square(value, index, array) {       var result = value \* value;      document.write("Квадрат числа " + value + " равен " + result + "<br/>");  };    numbers.forEach(square); |

Метод forEach() в качестве параметра принимает все ту же функцию, в которую при переборе элементов передается текущий перебираемый элемент и над ним выполняются операции. Метод **map()** похож на метод forEach, он также в качестве параметра принимает функцию, с помощью которой выполняются операции над перебираемыми элементами массива, но при этом метод map() возвращает новый массив с результатами операций над элементами массива. Например, применим метод map к вычислению квадратов чисел массива:

|  |  |
| --- | --- |
|  | var numbers = [ 1, 2, 3, 4, 5, 6];  function square(value, index, array) {      return result = value \* value;  };  var squareArray = numbers.map(square);  document.write(squareArray); |

Функция, которая передается в метод map() получает текущий перебираемый элемент, выполняет над ним операции и возвращает некоторое значение. Это значение затем попадает в результирующий массив squareArray

## Объект Number

Объект Number представляет числа. Чтобы создать число, надо передать в конструктор Number число или стоку, представляющую число:

|  |  |
| --- | --- |
|  | var x = new Number(34);  var y = new Number('34');  document.write(x+y); // 68 |

Определения x и y в данном случае будут практически аналогичны. Однако создавать объект Number можно и просто присвоив переменной определенное число: var z = 34;

Объект Number предоставляет ряд свойств и методов. Некоторые его свойства:

* Number.MAX\_VALUE: наибольшее возможное целое число. Приблизительно равно 1.79E+308. Числа, которые больше этого значения, рассматриваются как Infinity
* Number.MIN\_VALUE: наименьшее возможное положительное целое число. Приблизительно равно 5e-324 (где-то около нуля)
* Number.NaN: специальное значение, которое указывает, что объект не является числом
* Number.NEGATIVE\_INFINITY: значение, которое обозначает отрицательную неопределенность и которое возникает при переполнении. Например, если мы складываем два отрицательных числа, которые по модулю равны Number.MAX\_VALUE. Например:

|  |  |
| --- | --- |
|  | var x = -1 \* Number.MAX\_VALUE  var y = -1 \* Number.MAX\_VALUE  var z = x + y;  if(z===Number.NEGATIVE\_INFINITY)      document.write("отрицательная неопределенность");  else      document.write(z); |

* Number.POSITIVE\_INFINITY: положительная неопределенность. Также, как и отрицательная неопределенность, возникает при переполнении, только теперь в положительную сторону:

|  |  |
| --- | --- |
|  | var x = Number.MAX\_VALUE  var y = Number.MAX\_VALUE  var z = x \* y;  if(z===Number.POSITIVE\_INFINITY)      document.write("положительная неопределенность");  else      document.write(z); |

Некоторые основные методы:

* isNaN(): определяет, является ли объект числом. Если объект не является числом, то возвращается значение true:

|  |  |
| --- | --- |
|  | var a = Number.isNaN(Number.NaN); // true  var b = Number.isNaN(true); // false - new Number(true) = 1  var c = Number.isNaN(null);  // false - new Number(null) = 0  var d = Number.isNaN(25);  // false  var e = Number.isNaN("54"); // false |

* Но следующее выражение вернет false, хотя значение не является числом:

|  |  |
| --- | --- |
|  | var f = Number.isNaN("hello"); // false |

* Чтобы избежать подобных ситуаций, лучше применять глобальную функцию **isNaN**:

|  |  |
| --- | --- |
|  | var f = isNaN("hello"); // true |

* parseFloat(): преобразует строку в число с плавающей точкой. Например:

|  |  |
| --- | --- |
|  | var a = Number.parseFloat("34.90"); // 34.9  document.write(a);  var b = Number.parseFloat("hello"); // NaN  document.write(b);  var c = Number.parseFloat("34hello"); // 34  document.write(c); |

* parseInt(): преобразует строку в целое число. Например:

|  |  |
| --- | --- |
|  | var a = Number.parseInt("34.90"); // 34  document.write(a);  var b = Number.parseInt("hello"); // NaN  document.write(b);  var c = Number.parseInt("25hello"); // 25  document.write(c); |

* toFixed(): оставляет в числе с плавающей точкой определенное количество знаков в дробной части. Например:

|  |  |
| --- | --- |
|  | var a =  10 / 1.44;  document.write("До метода toFixed(): " + a + "<br/>");  a = a.toFixed(2); // оставляем два знака после запятой  document.write("После метода toFixed(): " + a + "<br/>"); |

* Вывод браузера:
* До метода toFixed(): 6.944444444444445
* После метода toFixed(): 6.94

# Строки

## Строки и объект String

let name = "Tom";

var name = new String("Tom"); // как правило, используется первый более краткий способ

В первом случае JavaScript при необходимости автоматически преобразует переменную примитивного типа в объект String. Объект String имеет большой набор свойств и методов, с помощью которых мы можем манипулировать строками.

Свойство **length** указывает на длину строки:

|  |  |
| --- | --- |
|  | var hello = "привет мир";  console.log("В строке '" + hello + "' " + hello.length + " символов"); |

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

|  |  |
| --- | --- |
|  | let hello = "hello ";  console.log(hello.repeat(3));   // hello hello hello |

### Шаблоны строк

Шаблоны строк позволяют вставлять в строку различные значения. Для этого строки заключаются в косые кавычки:

|  |  |
| --- | --- |
|  | let name = "Tom";  let hello = `Hello ${name}`;  console.log(hello);     // Hello Tom   let age = 23;  let info = `${name} is ${age} years old`;  console.log(info);      // Tom is 23 years old |

Для вставки значения в строку оно заключается в фигурные скобки, перед которыми ставится знак доллара. Также вместо скалярных значений могут добавляться свойства сложных объектов или результаты выражений:

|  |  |
| --- | --- |
|  | let tom ={      name: "Tom",      age: 25  }  let info = `${tom.name} is ${tom.age} years old`;  console.log(info);  // Tom is 23 years old   function sum(x, y){      return x + y;  }  let a = 5, b = 4;  let result = `${a} + ${b} = ${sum(a, b)}`;  console.log(result);    // 5 + 4 = 9 |

### Поиск в строке

Для поиска в строки некоторой подстроки используются методы **indexOf()** (индекс первого вхождения подстроки) и **lastIndexOf()**(индекс последнего вхождения подстроки). Эти методы принимают два параметра:

* Подстроку, которую надо найти
* Необязательный параметр, который указывает, с какого символа следует проводить поиск подстроки в строке

Оба этих метода возвращают индекс символа, с которого в строке начинается подстрока. Если подстрока не найдена, то возвращается число -1.

|  |  |
| --- | --- |
|  | let hello = "привет мир. пока мир";  let key = "мир";  let firstPos = hello.indexOf(key);  let lastPos = hello.lastIndexOf(key);  console.log("Первое вхождение: ", firstPos);    // 7  console.log("Последнее вхождение: ", lastPos);  // 17 |

Еще один метод - **includes()** возвращает true, если строка содержит определенную подстроку.

|  |  |
| --- | --- |
|  | let hello = "привет мир. пока мир";   console.log(hello.includes("мир")); // true  console.log(hello.includes("миг")); // false |

С помощью второго дополнительного параметра можно определить индекс, с которого будет начинаться поиск подстроки:

|  |  |
| --- | --- |
|  | let hello = "привет мир. пока мир";   console.log(hello.includes("мир", 5));  // true  console.log(hello.includes("привет", 6));   // false |

### Выбор подстроки

Для того, чтобы вырезать из строки подстроку, применяются методы **substr()** и **substring()**. Метод **substring()** принимает два параметра:

* индекс символа в строке, начиная с которого надо проводить обрезку строки
* индекс, до которого надо обрезать строку

|  |  |
| --- | --- |
|  | let hello = "привет мир. пока мир";  let world = hello.substring(7, 10); // с 7-го по 10-й индекс  console.log(world); // мир |

Метод substr() также в качестве первого параметра принимает начальный индекс подстроки, а в качестве второго - длину вырезаемой подстроки:

|  |  |
| --- | --- |
|  | let hello = "привет мир. пока мир";  let bye = hello.substr(12, 4);  console.log(bye); // пока |

Если второй параметр не указывается, то обрезается вся остальная часть строки:

|  |  |
| --- | --- |
|  | let hello = "привет мир. пока мир";  let bye = hello.substr(12);  console.log(bye); // пока мир |

### Управление регистром

Для изменения регистра имеются методы **toLowerCase()** (для перевода в нижний регистр) и **toUpperCase()** (для перевода в верхний регистр).

|  |  |
| --- | --- |
|  | let hello = "Привет Том";  console.log(hello.toLowerCase()); // привет том  console.log(hello.toUpperCase()); // ПРИВЕТ ТОМ |

### Получение символа по индексу

Чтобы получить определенный символ в строке по индексу, можно применять методы **charAt()** и **charCodeAt()**. Оба этих метода в качестве параметра принимают индекс символа:

|  |  |
| --- | --- |
|  | let hello = "Привет Том";  console.log(hello.charAt(2));       // и  console.log(hello.charCodeAt(2));   // 1080 |

Но если в качестве результата метод charAt() возвращает сам символ, то метод charCodeAt() возвращает числовой код этого символа.

### trim() - удаление пробелов в начале и конце

|  |  |
| --- | --- |
|  | let hello = "   Привет Том  ";  let beforeLength = hello.length;  hello = hello.trim();  let afterLength = hello.length;  console.log("Длина строки до: ", beforeLength);     // 15  console.log("Длина строки после: ", afterLength);   // 10 |

### Объединение строк

Метод **concat()** объединяет две строки:

let hello = "Привет ";

let world = "мир";

hello = hello.concat(world); // Привет мир

### Замена подстроки

Метод **replace()** заменяет первое вхождение одной подстроки на другую:

|  |  |
| --- | --- |
|  | let hello = "Добрый день";  hello = hello.replace("день", "вечер");  console.log(hello); // Добрый вечер |

Первый параметр метода указывает, какую подстроку надо заменить, а второй параметр - на какую подстроку надо заменить.

### Разделение строки

Метод **split()** разбивает строку на массив подстрок по определенному разделителю. В качестве разделителя используется строка, которая передается в метод:

|  |  |
| --- | --- |
|  | var message = "Сегодня была прекрасная погода";  var stringArray = message.split(" ");  for(var str in stringArray)      console.log(stringArray[str]); |

### Проверка начала и окончания строки

Метод **startsWith()** возвращает true, если строка начинается с определенной подстроки. А метод **endsWith()** возвращает true, если строка оканчивается на определенную подстроку.

|  |  |
| --- | --- |
|  | let hello = "let me speak from my heart";  console.log(hello.startsWith("let"));       // true  console.log(hello.startsWith("Let"));       // false  console.log(hello.startsWith("lets"));      // false  console.log(hello.endsWith("heart"));       // true  console.log(hello.startsWith("bart"));      // false |

При этом играет роль регистр символов, и из примера выше мы видим, что "let" не эквивалентно "Let". Дополнительный второй параметр позволяет указать индекс (для startsWith - индекс с начала, а для endsWith - индекс с конца строки), относительно которого будет производиться сравнение:

|  |  |
| --- | --- |
|  | let hello = "let me speak from my heart";  console.log(hello.startsWith("me", 4));     // true, "me" - 4 индекс с начала строки  console.log(hello.startsWith("my", hello.length-8));    // true, "my" - 8 индекс с конца |

## Объект RegExp. Регулярные выражения

**Регулярные выражения** представляют шаблон, который используется для поиска или модификации строки. Для работы с регулярными выражениями в JavaScript определен объект **RegExp**. Определить регулярное выражение можно двумя способами:

|  |  |
| --- | --- |
| 1  2 | var myExp = /hello/;  var myExp = new RegExp("hello"); |

Используемое здесь регулярное выражение довольно простое: оно состоит из одного слова "hello". В первом случае выражение помещается между двумя косыми чертами, а во втором случае используется конструктор RegExp, в который выражение передается в виде строки.

### Методы RegExp

Чтобы определить, соответствует ли регулярное выражение строке, в объекте RegExp определен метод **test()**. Этот метод возвращает true, если строка соответствует регулярному выражению, и false, если не соответствует.

|  |  |
| --- | --- |
|  | var initialText = "hello world!";  var exp = /hello/;  var result = exp.test(initialText);  document.write(result + "<br/>"); // true    initialText = "beautifull wheather";  result = exp.test(initialText);  document.write(result); // false - в строке initialText нет "hello" |

Аналогично работает метод **exec** - он также проверяет, соответствует ли строка регулярному выражению, только теперь данный метод возвращает ту часть строки, которая соответствует выражению. Если соответствий нет, то возвращается значение **null**.

|  |  |
| --- | --- |
|  | var initialText = "hello world!";  var exp = /hello/;  var result = exp.exec(initialText);  document.write(result + "<br/>"); // hello  initialText = "beautifull wheather";  result = exp.exec(initialText);  document.write(result); // null |

### Группы символов

Регулярное выражение не обязательно состоит из обычных строк, но также может включать специальные элементы синтаксиса регулярных выражений. Один из таких элементов представляют группы символов, заключенные в квадратные скобки. Например:

|  |  |
| --- | --- |
|  | var initialText = "обороноспособность";  var exp = /[абв]/;  var result = exp.test(initialText);  document.write(result + "<br/>"); // true  initialText = "город";  result = exp.test(initialText);  document.write(result); // false |

Выражение [абв] указывает на то, что строка должна иметь одну из трех букв. Если нам надо определить наличие в строке буквенных символов из определенного диапазона, то можно разу задать этот диапазон:

|  |  |
| --- | --- |
|  | var initialText = "обороноспособность";  var exp = /[а-я]/;  var result = exp.test(initialText);  document.write(result + "<br/>"); // true  initialText = "3di0789";  result = exp.test(initialText);  document.write(result); // false |

В данном случае строка должна содержать хотя бы один символ из диапазона а-я. Если, наоборот, не надо, чтобы строка имела только определенные символы, то необходимо в квадратных скобках перед перечислением символов ставить знак ^:

|  |  |
| --- | --- |
|  | var initialText = "обороноспособность";  var exp = /[^а-я]/;  var result = exp.test(initialText);  document.write(result + "<br/>"); // false  initialText = "3di0789";  exp = /[^0-9]/;  result = exp.test(initialText);  document.write(result); // true |

В первом случае строка не должна иметь только символы из диапазона а-я, но поскольку строка "обороноспособность" состоит только из символов из этого диапазона, то метод test() возвращает false, то есть регулярное выражение не соответствует стоке. Во втором случае ("3di0789") строка не должна состоять только из цифровых символов. Но так как в строке также есть и буквы, то строка соответствует регулярному выражению, поэтому метод test возвращает значение true. При необходимости мы можем собирать комбинации выражений:

|  |  |
| --- | --- |
|  | var initialText = "дома";  var exp = /[дт]о[нм]/;  var result = exp.test(initialText);  document.write(result); // true |

Выражение [дт]о[нм] указывает на те строки, которые могут содержать подстроки "дом", "том", "дон", "тон".

### Свойства выражений

* Свойство **global** позволяет найти все подстроки, которые соответствуют регулярному выражению. По умолчанию при поиске подстрок регулярное выражение выбирает первую попавшуюся подстроку из строки, которая соответствует выражению. Хотя в строке может быть множество подстрок, которые также соответствуют выражению. Для этого применяется данное свойство в виде символа g в выражениях
* Свойство **ignoreCase** позволяет найти подстоки, которые соответствуют регулярному выражению, вне зависимости от регистра символов в строке. Для этого в регулярных выражениях применяется символ i
* Свойство **multiline** позволяет найти подстроки, которые соответствуют регулярному выражению, в многострочном тексте. Для этого в регулярных выражениях применяется символ m

Например:

|  |  |
| --- | --- |
|  | var initialText = "привет Мир";  var exp = /мир/;  var result = exp.test(initialText); // false |

Здесь совпадения строки с выражением нет, так как "Мир" отличается от "мир" по регистру. В этом случае надо изменить регулярное выражение, добавив в него свойство ignoreCase:

|  |  |
| --- | --- |
| 1 | var exp = /мир/i; |

Ну и также мы можем использовать сразу несколько свойств: var exp = /мир/ig;

## Регулярные выражения в методах String

Ряд методов объекта String могут использовать регулярные выражения в качестве параметра.

**Разделение строки. Метод split**

Метод split может использовать регулярные выражения для разделения строк. Например, разделим приложение по словам:

|  |  |
| --- | --- |
|  | var initialText = "Сегодня была прекрасная погода";  var exp = /\s/;  var result = initialText.split(exp);  result.forEach(function(value, index, array){      document.write(value + "<br/>");  }) |

Вывод браузера:

Сегодня

была

прекрасная

погода

**Метод match**

Для поиска всех соответствий в строке применяется метод **match()**:

|  |  |
| --- | --- |
|  | var initialText = "Он пришел домой и сделал домашнюю работу";  var exp = /дом[а-я]\*/gi;  var result = initialText.match(exp);  result.forEach(function(value, index, array){        document.write(value + "<br/>");  }) |

Символ звездочки указывает на возможность наличия после строки "дом" неопределенного количества символов от а до я. В итоге в массиве result окажутся следующие слова:

домой

домашнюю

**Поиск в строке. Метод search**

Метод search находит индекс первого включения соответствия в строке:

|  |  |
| --- | --- |
|  | var initialText = "hello world";  var exp = /wor/;  var result = initialText.search(exp);  document.write(result); // 6 |

**Замена. Метод replace**

Метод replace позволяет заменить все соответствия регулярному выражению определенной строкой:

|  |  |
| --- | --- |
|  | var menu = "Завтрак: каша, чай. Обед: суп, чай. Ужин: салат, чай.";  var exp = /чай/gi;  menu = menu.replace(exp, "кофе");  document.write(menu); |

Вывод браузера:

Завтрак: каша, кофе. Обед: суп, кофе. Ужин: салат, кофе.

## Синтаксис регулярных выражений

Рассмотрим базовые моменты синтаксиса регулярных выражений.

### Метасимволы

Регулярные выражения также могут использовать метасимволы - символы, которые имеют определенный смысл:

* \d: соответствует любой цифре от 0 до 9
* \D: соответствует любому символу, который не является цифрой
* \w: соответствует любой букве, цифре или символу подчеркивания (диапазоны A–Z, a–z, 0–9)
* \W: соответствует любому символу, который не является буквой, цифрой или символом подчеркивания (то есть не находится в следующих диапазонах A–Z, a–z, 0–9)
* \s: соответствует пробелу
* \S: соответствует любому символу, который не является пробелом
* .: соответствует любому символу

Здесь надо заметить, что метасимвол \w применяется только для букв латинского алфавита, кириллические символы для него не подходят.

Так, стандартный формат номера телефона +1-234-567-8901 соответствует регулярному выражению \d-\d\d\d-\d\d\d-\d\d\d\d. Например, заменим числа номера нулями:

|  |  |
| --- | --- |
| 1  2  3  4 | var phoneNumber = "+1-234-567-8901";  var myExp = /\d-\d\d\d-\d\d\d-\d\d\d\d/;  phoneNumber = phoneNumber.replace(myExp, "00000000000");  document.write(phoneNumber); |

### Модификаторы

Кроме выше рассмотренных элементов регулярных выражений есть еще одна группа комбинаций, которая указывает, как символы в строке будут повторяться. Такие комбинации еще называют модификаторами:

* {n}: соответствует n-ому количеству повторений предыдущего символа. Например, h{3} соответствует подстроке "hhh"
* {n,}: соответствует n и более количеству повторений предыдущего символа. Например, h{3,} соответствует подстрокам "hhh", "hhhh", "hhhhh" и т.д.
* {n,m}: соответствует от n до m повторений предыдущего символа. Например, h{2, 4} соответствует подстрокам "hh", "hhh", "hhhh".
* ?: соответствует одному вхождению предыдущего символа в подстроку или его отсутствию в подстроке. Например, /h?ome/соответствует подстрокам "home" и "ome".
* +: соответствует одному и более повторений предыдущего символа
* \*: соответствует любому количеству повторений или отсутствию предыдущего символа
* ^: соответствует началу строки. Например, ^h соответствует строке "home", но не "ohma", так как h должен представлять начало строки
* $: соответствует концу строки. Например, м$ соответствует строке "дом", так как строка должна оканчиваться на букву м

Например, возьмем номер тот же телефона. Ему соответствует регулярное выражение \d-\d\d\d-\d\d\d-\d\d\d\d. Однако с помощью выше рассмотренных комбинаций мы его можем упростить: \d-\d{3}-\d{3}-\d{4}

Также надо отметить, что так как символы ?, +, \* имеют особый смысл в регулярных выражениях, то чтобы их использовать в обычным для них значении (например, нам надо заменить знак плюс в строке на минус), то данные символы надо экранировать с помощью слеша:

|  |  |
| --- | --- |
| 1  2  3  4 | var phoneNumber = "+1-234-567-8901";  var myExp = /\+\d-\d{3}-\d{3}-\d{4}/;  phoneNumber = phoneNumber.replace(myExp, "80000000000");  document.write(phoneNumber); |

Отдельно рассмотрим применение комбинации '\b', которая указывает на соответствие в пределах слова. Например, у нас есть следующая строка: "Языки обучения: Java, JavaScript, C++". Со временем мы решили, что Java надо заменить на C#. Но простая замена приведет также к замене строки "JavaScript" на "C#Script", что недопустимо. И в этом случае мы можем проводить замену, если регуляное выражение соответствует всему слову:

|  |  |
| --- | --- |
| 1  2  3  4 | var initialText = "Языки обучения: Java, JavaScript, C++";  var exp = /Java\b/g;  var result = initialText.replace(exp, "C#");  document.write(result); // Языки обучения: C#, JavaScript, C++ |

Но при использовании '\b' надо учитывать, что в JavaScript отсутствует полноценная поддержка юникода, поэтому применять '\b' мы сможем только к англоязычным словам.

### Использование групп в регулярных выражениях

Для поиска в строке более сложных соответствий применяются группы. В регулярных выражениях группы заключаются в скобки. Например, у нас есть следующий код html, который содержит тег изображения: '<img src="picture.png" />'. И допустим, нам надо вычленить из этого кода пути к изображениям:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var initialText = '<img src= "picture.png" />';  var exp = /[a-z]+\.(png|jpg)/i;  var result = initialText.match(exp);  result.forEach(function(value, index, array){        document.write(value + "<br/>");  }) |

Вывод браузера:

picture.png

png

Первая часть до скобок ([a-z]+\.) указывает на наличие в строке от 1 и более символов из диапазона a-z, после которых идет точка. Так как точка является специальным символом в регулярных выражениях, то она экранируется слешем. А дальше идет группа: (png|jpg). Эта группа указывает, что после точки может использоваться как "png", так и "jpg".

# Работа с браузером и BOM

## Browser Object Model и объект window

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.

**Объект window**

Объект **window** представляет собой окно веб-браузера, в котором размещаются веб-страницы. window является глобальным объектом, поэтому при доступе к его свойствам и методам необязательно использовать его имя. Например, window имеет метод **alert()**, который отображает окно сообщения. Но нам необязательно писать:

|  |  |
| --- | --- |
| 1 | window.alert("Привет мир!"); |

window можно не использовать:

|  |  |
| --- | --- |
| 1 | alert("Привет мир!"); |

Но так как данный объект глобальный, то это накладывает некоторые ограничения. Например:

|  |  |
| --- | --- |
|  | var alert = function(message){        document.write("Сообщение: " + message);  };  window.alert("Привет мир!"); |

Все объявляемые в программе глобальные переменные или функции автоматически добавляются к объекту window. И поскольку название новой функции будет совпадать с названием метода alert(), то произойдет переопределение этого метода в объекте window новой функцией.

И если мы объявим в программе какую-нибудь глобальную переменную, то она нам доступна как свойство в объекте window:

|  |  |
| --- | --- |
| 1  2 | var message = "hello";  document.write(window.message); |

## Управление окнами

### Диалоговые окна

Для взаимодействия с пользователем в объекте window определен ряд методов, которые позволяют создавать диалоговые окна. Метод **alert()** выводит окно с сообщением:

|  |  |
| --- | --- |
| 1 | alert("hello world"); |

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

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

И метод **prompt()** позволяет с помощью диалогового окна запрашивать у пользователя какие-либо данные. Данный метод возвращает введенное пользователем значение:

|  |  |
| --- | --- |
| 1  2 | var age = prompt("Введите свой возраст:");  document.write("Вам " + age + " лет"); |

Если пользователь откажется вводить значение и нажмет на кнопку отмены, то метод возвратит значение null.

### Открытие, закрытие и позиционирование окон

Объект window также предоставляет ряд методов для управления окнами браузера. Так, метод **open()** открывает определенный ресурс в новом окне браузера:

|  |  |
| --- | --- |
| 1 | var popup = window.open('[https://microsoft.com](https://microsoft.com/)', 'Microsoft', 'width=400, height=400, resizable=yes'); |

Метод open() принимает ряд параметров: путь к ресурсу, описательное название для окна и в качестве третьего параметра набор стилевых значений окна. Метод возвращает ссылку на объект нового окна.

Мы можем установить следующие стилевые характеристики:

* width: ширина окна в пикселях. Например, width=640
* height: высота окна в пикселях. Например, height=480
* left: координата X относительно начала экрана в пикселях. Например, left=0
* top: координата Y относительно начала экрана в пикселях. Например, top=0
* titlebar: будет ли окно иметь строку с заголовком. Например, titlebar=no
* menubar: будет ли окно иметь панель меню. Например, menubar=yes
* toolbar: будет ли окно иметь панели инструментов. Например, toolbar=yes
* location: будет ли окно иметь адресную строку. Например, location=no
* scrollbars: допускается ли наличие полос прокрутки. Например, scrollbars=yes
* status: наличие статусной строки. Например, status=yes
* resizable: может ли окно изменять размеры. Например, resizable=no

С помощью метода **close()** можно закрыть окно. Например, откроем новое окно и через 10 секунд закроем его:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var popup = window.open('[https://microsoft.com](https://microsoft.com/)', 'Microsoft', 'width=400, height=400, resizable=yes');  function closeWindow(){      popup.close();  }  setTimeout(closeWindow, 10000); |

Метод **moveTo()** позволяет переместить окно на новую позицию:

|  |  |
| --- | --- |
| 1  2 | var popup = window.open('[https://microsoft.com](https://microsoft.com/)', 'Microsoft', 'width=400, height=400, resizable=yes');  popup.moveTo(50,50); |

В данном случае окно перемещается на позицию с координатами x=50, y=50 относительно левого верхнего угла экрана.

Метод **resizeTo()** позволяет изменить размеры окна:

|  |  |
| --- | --- |
| 1  2 | var popup = window.open('[https://microsoft.com](https://microsoft.com/)', 'Microsoft', 'width=400, height=400, resizable=yes');  popup.resizeTo(500,350); // 500 - ширина и 350 - высота |

## История браузера. Объект history

Объект **history** предназначен для хранения истории посещений веб-страниц в браузере. Нам этот объект доступен через объект window. Все сведения о посещении пользователя хранятся в специальном стеке (history stack). С помощью свойства **length** можно узнать, как много веб-станиц хранится в стеке:

|  |  |
| --- | --- |
| 1 | document.write("В истории " + history.length + " станиц"); |

Для перемещения по страницам в истории в объекте history определены методы **back()** (перемещение к прошлой посмотренной странице) и **forward()** (перемещение к следующей просмотренной странице)

|  |  |
| --- | --- |
| 1 | history.back(); // перемещение назад |

Также в объекте history определен специальный метод **go()**, который позволяет перемещаться вперед и назад по истории на определенное число страниц. Например, переместимся на 2 страницы назад:

|  |  |
| --- | --- |
| 1 | history.go(-2); |

Соответственно если надо переместиться на несколько страниц вперед, то в метод передается положительное значение. Например, переместимся вперед на три страницы:

|  |  |
| --- | --- |
| 1 | history.go(3); |

## Объект location

Объект **location** содержит информацию о расположении текущей веб-страницы: URL, информацию о сервере, номер порта, протокол. С помощью свойств объекта мы можем получить эту информацию:

* **href**: полная строка запроса к ресурсу
* **pathname**: путь к ресурсу
* **origin**: общая схема запроса
* **protocol**: протокол
* **port**: порт, используемый ресурсом
* **host**: хост
* **hostname**: название хоста
* **hash**: если строка запроса содержит символ решетки (#), то данное свойство возвращает ту часть строки, которая идет после этого символа
* **search**: если строка запроса содержит знак вопроса (?), например, то данное свойство возвращает ту часть строки, которая идет после знака вопроса

Например, пусть есть следующая веб-страница *test.html*, которая лежит на локальном веб-сервере:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <script>  document.write("Строка запроса: " + location.href + "<br />");  document.write("Путь к ресурсу: " + location.pathname + "<br />");  document.write("Схема: " + location.origin + "<br />");  document.write("Протокол: " + location.protocol + "<br />");  document.write("Порт: " + location.port + "<br />");  document.write("Хост: " + location.host + "<br />");  document.write("Имя хоста: " + location.hostname + "<br />");  document.write("Хэш: " + location.hash + "<br />");  document.write("Поиск: " + location.search + "<br />");  </script>  </body>  </html> |

Также объект location предоставляет ряд методов, которые можно использовать для управления путем запроса:

* **assign(url)**: загружает ресурс, который находится по пути url
* **reload(forcedReload)**: перезагружает текущую веб-страницу. Параметр forcedReload указывает, надо ли использовать кэш браузера. Если параметр равен true, то кэш не используется
* **replace(url)**: заменяет текущую веб-станицу другим ресурсом, который находится по пути url. В отличие от метода assign, который также загружает веб-станицу с другого ресурса, метод replace не сохраняет предыдущую веб-страницу в стеке истории переходов history, поэтому мы не сможем вызвать метод history.back() для перехода к ней.

Для перенаправления на другой ресурс мы можем использовать как свойства, так и методы location:

|  |  |
| --- | --- |
|  | location = "[http://google.com](http://google.com/)";  // аналогично  // location.href = "[http://google.com](http://google.com/)";  // location.assign("[http://google.com](http://google.com/)"); |

Переход на другой локальный ресурс:

|  |  |
| --- | --- |
| 1 | location.replace("index.html"); |

## Объект navigator

Объект navigator содержит информацию о браузере и операционной системе, в которой браузер запущен. Он определяет ряд свойств и методов, основным из которых является свойство **userAgent**, представляющее браузер пользователя:

|  |  |
| --- | --- |
| 1 | document.write(navigator.userAgent); |

Данное свойство хранит полную стоку юзер-агента, например, Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.157 Safari/537.36

Чтобы вычленить из этой информации непосредственно браузер, можно попробовать найти в этой информации название браузера:

|  |  |
| --- | --- |
|  | var browser, uAgent = navigator.userAgent;    if(uAgent.indexOf("Chrome") > -1) {      browser = "Google Chrome";  } else if (uAgent.indexOf("Safari") > -1) {      browser = "Apple Safari";  } else if (uAgent.indexOf("Opera") > -1) {      browser = "Opera";  } else if (uAgent.indexOf("Firefox") > -1) {      browser = "Mozilla Firefox";  } else if (uAgent.indexOf("MSIE") > -1) {      browser = "Microsoft Internet Explorer";  }  document.write(browser); |

### Объект geolocation

Объект navigator хранит свойство geolocation, с помощью которого можно получить географическое положение пользователя. Для получения положения используется метод **getCurrentPosition()**. Этот метод принимает два параметра: функцию, которая срабатывает при удачном запуске, и функцию, которая срабатывает при ошибке запроса геоданных:

|  |  |
| --- | --- |
|  | function success(position) {      var latitude = position.coords.latitude;      var longitude = position.coords.longitude;      var altitude = position.coords.altitude;      var speed = position.coords.speed;      document.write("Широта: " + latitude + "<br/>");      document.write("Долгота: " + longitude + "<br/>");      document.write("Высота: " + altitude + "<br/>");      document.write("Скорость перемещения: " + speed + "<br/>");  };    function error(obj) {      document.write("Ошибка при определении положения");  };  navigator.geolocation.getCurrentPosition(success, error); |

В функцию, которая выполняется при удачном определении геоданных, передается позиция пользователя в виде параметра position. Передаваемый объект имеет вложенный объект **coords**, с помощью свойство которого можно получить непосредственные координаты пользователя:

* latitude: географическая широта
* longitude: географическая долгота
* altitude: высота
* speed: скорость, с которой перемещается пользователь (например, если он идет или перемещается на транспорте)

При этом надо учитывать, что в браузерах действует политика безопасности, которая при обращении к методу geolocation.getCurrentPosition() отображает пользователю сообщение, в котором пользователь может подтвердить отправку географических координат. Если же пользователь откажется, то будет срабатывать функция error().

## Таймеры

Для выполнения действий через определенные промежутки времени в объекте window предусмотрены функции таймеров. Есть два типа таймеров: одни выполняются только один раз, а другие постоянно через промежуток времени.

**Функция setTimeout**

Для одноразового выполнения действий через промежуток времени предназначена функция **setTimeout()**. Она может принимать два параметра:

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

Параметр period указывает на промежуток, через который будет выполняться функция из параметра someFunction. А в качестве результата функция возвращает id таймера.

|  |  |
| --- | --- |
|  | function timerFunction() {      document.write("выполнение функции setTimeout");  }  setTimeout(timerFunction, 3000); |

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

Для остановки таймера применяется функция **clearTimeout()**.

|  |  |
| --- | --- |
|  | function timerFunction() {      document.write("выполнение функции setTimeout");  }  var timerId = setTimeout(timerFunction, 3000);  clearTimeout(timerId); |

**Функция setInterval**

Функции **setInterval()** и **clearInterval()** работают аналогично функциям setTimeout() и clearTimeout() с той лишь разницей, что **setInterval()** постоянно выполняет определенную функцию через промежуток времени.

Например, напишем небольшую программу для вывода текущего времени:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <div id="time"></div>  <script>  function updateTime() {      document.getElementById("time").innerHTML = new Date().toTimeString();  }  setInterval(updateTime, 1000);  </script>  </body>  </html> |

Здесь через каждую секунду (1000 миллисекунд) вызывается функция updateTime(), которая обновляет содержимое поля <div id="time" >, устанавливая в качестве его кода html текущее вемя.

**requestAnimationFrame()**

Метод **requestAnimationFrame()** действует аналогично setInterval() за тем исключением, что он больше заточен под анимации, работу с графикой и имеет ряд оптимизаций, которые улучшают его производительность.

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <style>      #rect {          margin: 100px;          width: 100px;          height: 100px;          background: #50c878;      }      </style>  </head>  <body>  <div id="rect"></div>  <script>  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);  </script>  </body>  </html> |

В метод window.requestAnimationFrame() передается функция, которая будет вызываться определенное количество раз (обычно 60) в секунду. В данном случае в этот метод передается функция rotate, которая изменяет угол поворота блока на странице и затем обращается опять же к методу window.requestAnimationFrame(rotate). В качестве возвращаемого результата метод window.requestAnimationFrame() возвращает уникальный id, который может потом использоваться для остановки анимации:

|  |  |
| --- | --- |
| 1 | window.cancelAnimationFrame(id); |

# Работа с DOM

## Введение в DOM

Для JavaScript веб-страница доступна в виде объектной модели документа (document object model) или сокращенно DOM. DOM описывает структуру веб-станицы в виде древовидного представления и предоставляет разработчикам способ получить доступ к отдельным элементам веб-станицы. 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:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <title>Page Title</title>  </head>  <body>      <h2>Page Header</h2>      <div>          <h3>Block Header</h3>          <p>Text</p>      </div>  </body>  </html> |

Таким образом, все компоненты упорядочены в DOM иерархическим образом, где каждый компонент представляет отдельный узел. То есть каждый элемент, например, элемент div, представляет собой узел. Но также и текст внутри элемента представляет отдельный узел.

Существует следующие виды узлов:

* **Element**: html-элемент
* **Attr**: атрибут html-элемента
* **Document**: корневой узел html-документа
* **DocumentType**: DTD или тип схемы XML-документа
* **DocumentFragment**: место для временного хранения частей документа
* **EntityReference**: ссылка на сущность XML-документа
* **ProcessingInstruction**: инструкция обработки веб-страницы
* **Comment**: элемент комментария
* **Text**: текст элемента
* **CDATASection**: секция CDATA в документе XML
* **Entity**: необработанная сущность DTD
* **Notation**: нотация, объявленная в DTD

Несмотря на такое большое количество типов узлов, как правило, мы будем работать только с некоторыми из них.

## 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
* **getElementsByClassName(value)**: выбирает все элементы, которые имеют класс value
* **querySelector(value)**: выбирает первый элемент, который соответствует css-селектору value
* **querySelectorAll(value)**: выбирает все элементы, которые соответствуют css-селектору value

Например, найдем элемент по id:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <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>  </html> |

С помощью вызова document.getElementById("header") находим элемент, у которого id="header". А с помощью свойства **innerText**можно получить текст найденного элемента. Поиск по определенному тегу:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <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>  </html> |

С помощью вызова document.getElementsByTagName("p") находим все элементы параграфов. Этот вызов возвращает массив найденных элементов. Поэтому, чтобы получить отдельные элементы массива, необходимо пробежаться по ним в цикле. Если нам надо получить только первый элемент, то можно к первому элементу найденной коллекции объектов:

|  |  |
| --- | --- |
| 1  2 | var pElement = document.getElementsByTagName("p")[0];  document.write("Текст параграфа: " + pElement.innerText); |

Получение элемента по классу:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>      <div class="article">          <h3>Заголовок статьи</h3>          <p class="text">Первый абзац</p>          <p class="text">Второй абзац</p>      </div>  <script>  var articleDiv = document.getElementsByClassName("article")[0];      console.log(articleDiv);  var textElems = document.getElementsByClassName("text");  for (var i = 0; i < textElems.length; i++) {      console.log(textElems[i]);  }  </script>  </body>  </html> |

Выбор по селектору css:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <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>  </html> |

Выражение document.querySelector(".annotation p") находит элемент, который соответствует селектору .annotation p. Если на странице несколько элементов, соответствующих селектору, то метод выберет первый из них. В итоге браузер выведет:

Аннотация статьи

Первый абзац

Второй абзац

Текст селектора: Аннотация статьи

Чтобы получить все элементы по селектору, можно подобным образом использовать метод **document.querySelectorAll**, который возвращает массив найденных элементов:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>      <div class="annotation">          <p>Аннотация статьи</p>      </div>      <div class="text">          <p>Первый абзац</p>          <p>Второй абзац</p>      </div>  <script>  var elems = document.querySelectorAll(".text p");    for (var i = 0; i < elems.length; i++) {      document.write("Текст селектора " + i + ": " + elems[i].innerText + "<br/>");  }  </script>  </body>  </html> |

Вывод браузера:

Аннотация статьи

Первый абзац

Второй абзац

Текст селектора 0: Первый абзац

Текст селектора 1: Второй абзац

## Свойства объекта document

Кроме ранее рассмотренных методов объект document позволяет обратиться к определенным элементам веб-страницы через свойства:

* **documentElement**: предоставляет доступ к корневому элементу <html>
* **body**: предоставляет доступ к элементу <body> на веб-странице
* **images**: содержит коллекцию всех объектов изображений (элементов img)
* **links**: содержит коллекцию ссылок - элементов <a> и <area>, у которых определен атрибут href
* **anchors**: предоставляет доступ к коллекции элементов <a>, у которых определен атрибут name
* **forms**: содержит коллекцию всех форм на веб-странице

Эти свойства не предоставляют доступ ко всем элементам, однако позволяют получить наиболее часто используемые элементы на веб-странице. Например, получим корневой узел документа:

|  |  |
| --- | --- |
|  | var container = document.documentElement; |

Получим все изображения на странице:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>      <img src="picure1.png" alt="Картинка 1" />      <img src="picure2.png" alt="Картинка 2" />      <img src="picure3.png" alt="Картинка 3" />  <script>  var images = document.images;  // изменим первое изображение  images[0].src="pics/picture\_4.jpg";  images[0].alt="Новая картинка";  // перебирем все изображения  for(var i=0; i<images.length;i++){        document.write("<br/>" + images[i].src);      document.write("<br/>" + images[i].alt);  }  </script>  </body>  </html> |

Подобно тому, как в коде html мы можем установить атрибуты у элемента img, так и в коде javascript мы можем через свойства src и alt получить и установить значения этих атрибутов. Рассмотрим получение всех ссылок на странице:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>      <a href="article1.html">Статья 1</a>      <a href="article2.html">Статья 2</a>      <a href="article3.html">Статья 3</a>  <script>  var links = document.links;  for(var i=0; i<links.length;i++){        document.write("<br/>" + links[i].innerText);      document.write("<br/>" + links[i].href);  }  </script>  </body>  </html> |

Опять же, так как у ссылки определен атрибут href, то при переборе ссылок мы можем получить его значение.

## Объект Node. Навигация по DOM

Каждый отдельный узел, будь то html-элемент, его атрибут или текст, в структуре DOM представлен объектом **Node**. Этот объект предоставляет ряд свойств, с помощью которых мы можем получить информацию о данном узле:

* **childNodes**: содержит коллекцию дочерних узлов
* **firstChild**: возвращает первый дочерний узел текущего узла
* **lastChild**: возвращает последний дочерний узел текущего узла
* **previousSibling**: возвращает предыдущий элемент, который находится на одном уровне с текущим
* **nextSibling**: возвращает следующий элемент, который находится на одном уровне с текущим
* **ownerDocument**: возвращает корневой узел документа
* **parentNode**: возвращает элемент, который содержит текущий узел
* **nodeName**: возвращает имя узла
* **nodeType**: возвращает тип узла в виде числа
* **nodeValue**: возвращает или устанавливает значение узла в виде простого текста

Перебирем все дочерние узлы элемента:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>      <div class="article">          <h3>Заголовок статьи</h3>          <p>Первый абзац</p>          <p>Второй абзац</p>      </div>  <script>  var articleDiv = document.querySelector("div.article");  var nodes = articleDiv.childNodes;  for(var i=0; i<nodes.length;i++){       var type = "";      if(nodes[i].nodeType===1)          type="элемент";      else if(nodes[i].nodeType===2)          type="атрибут";      else if(nodes[i].nodeType===3)          type="текст";      console.log(nodes[i].nodeName + ": " + type);  }  </script>  </body>  </html> |

С помощью метода document.querySelector("div.article") выбираем элемент div с классом article и пробегаемся по его дочерним узлам. И в цикле выводим имя узла и его тип с помощью свойств nodeName и nodeType. Каждому типу соответствует определенное число:

|  |  |
| --- | --- |
| **nodeType** | **Тип узла** |
| 1 | элемент |
| 2 | атрибут |
| 3 | текст |

Но несмотря на то, что в блоке div на странице только три узла: h3 и 2 параграфа, но консоль отобразит нам пять узлов. Дело в том, что пробелы между узлами также считаются за отдельные текстовые узлы. Если бы пробелов не было:

|  |  |
| --- | --- |
| 1 | <div class="article"><h3>Заголовок статьи</h3><p>Первый абзац</p><p>Второй абзац</p></div> |

То при переборе мы бы обнаружили только три дочерних узла, как и ожидалось.

### Навигация по узлам

Используя свойства nextSibling и previousSibling можно также пройтись по узлам в прямом или обратном порядке. Например, для прохода по элементам на той же станице мы могли бы использовать следующий код javascript:

|  |  |
| --- | --- |
|  | var articleDiv = document.querySelector("div.article");  // получаем первый дочерний элемент  var node = articleDiv.firstChild;  console.log(node.nodeName);  // обращаемся к следующему узлу, пока он определен  while((node=node.nextSibling)!==null){      console.log(node.nodeName);  } |

Тут все аналогично, только в цикле мы обращаемся к следующему узлу, который находится на одном уровне с текущим (или сестринскому узлу). И результат будет такой же. Также можно перебрать узлы в обратном порядке: сначала получаем последний узел, а затем обращаемся к предыдущему сестринскому узлу:

|  |  |
| --- | --- |
|  | var articleDiv = document.querySelector("div.article");   // получаем первый дочерний элемент  var node = articleDiv.lastChild;  console.log(node.nodeName);  // обращаемся к предыдующему узлу, пока он определен  while((node=node.previousSibling)!==null){        console.log(node.nodeName);  } |

## Создание, добавление и удаление элементов веб-станицы

Для создания элементов объект document имеет следующие методы:

* **createElement(elementName)**: создает элемент html, тег которого передается в качестве параметра. Возвращает созданный элемент
* **createTextNode(text)**: создает и возвращает текстовый узел. В качестве параметра передается текст узла.

|  |  |
| --- | --- |
|  | var elem = document.createElement("div");  var elemText = document.createTextNode("Привет мир"); |

Таким образом, переменная elem будет хранить ссылку на элемент div. Однако одного создания элементов недостаточно, их еще надо добавить на веб-страницу. Для добавления элементов мы можем использовать один из методов объекта Node:

* **appendChild(newNode)**: добавляет новый узел newNode в конец коллекции дочерних узлов
* **insertBefore(newNode, referenceNode)**: добавляет новый узел newNode перед узлом referenceNode

Используем метод appendChild:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>      <div class="article">          <h3>Заголовок статьи</h3>          <p>Первый абзац</p>          <p>Второй абзац</p>      </div>  <script>  var articleDiv = document.querySelector("div.article");  // создаем элемент  var elem = document.createElement("h2");  // создаем для него текст  var elemText = document.createTextNode("Привет мир");  // добавляем текст в элемент в качестве дочернего элемента  elem.appendChild(elemText);  // добавляем элемент в блок div  articleDiv.appendChild(elem);  </script>  </body>  </html> |

Сначала создаем обычный элемент заголовка h2 и текстовый узел. Затем текстовый узел добавляем в элемент заголовка. Затем заголовок добавляем в один из элементов веб-страницы: Однако нам необязательно для определения текста внутри элемента создавать дополнительный текстовый узел, так как мы можем воспользоваться свойством **textContent** и напрямую ему присвоить текст:

|  |  |
| --- | --- |
|  | var elem = document.createElement("h2");  elem.textContent = "Привет мир"; |

В этом случае текстовый узел будет создан неявно при установке текста. Теперь рассмотрим, как аналогичный элемент добавить в начало коллекции дочерних узлов блока div:

|  |  |
| --- | --- |
|  | var articleDiv = document.querySelector("div.article");  // создаем элемент  var elem = document.createElement("h2");  // создаем для него текст  var elemText = document.createTextNode("Привет мир");  // добавляем текст в элемент в качестве дочернего элемента  elem.appendChild(elemText);  // получаем первый элемент, перед которым будет идти добавление  var firstElem = articleDiv.firstChild.nextSibling;  // добавляем элемент в блок div перед первым узлом  articleDiv.insertBefore(elem, firstElem); |

Если нам надо вставить новый узел на второе, третье или любое другое место, то нам надо найти узел, перед которым надо вставлять, с помощью комбинаций свойств firstChild/lastChild и nextSibling/previousSibling.

### Копирование элемента

Иногда элементы бывают довольно сложными по составу, и гораздо проще их скопировать, чем с помощью отдельных вызовов создавать из содержимое. Для копирования уже имеющихся узлов у объекта Node можно использовать метод **cloneNode()**:

|  |  |
| --- | --- |
|  | var articleDiv = document.querySelector("div.article");  // клонируем элемент articleDiv  var newArticleDiv = articleDiv.cloneNode(true);  // добавляем в конец элемента body  document.body.appendChild(newArticleDiv); |

В метод cloneNode() в качестве параметра передается логическое значение: если передается true, то элемент будет копироваться со всеми дочерними узлами; если передается false - то копируется без дочерних узлов. То есть в данном случае мы копируем узел со всем его содержимым и потом добавляем в конец элемента body.

### Удаление элемента

Метод **removeChild()** объекта Node удаляет один из дочерних узлов:

|  |  |
| --- | --- |
|  | var articleDiv = document.querySelector("div.article");  // находим узел, который будем удалять - первый параграф  var removableNode = document.querySelectorAll("div.article p")[0];  // удаляем узел  articleDiv.removeChild(removableNode); |

В данном случае удаляется первый параграф из блока div

### Замена элемента

Для замены элемента применяется метод **replaceChild(newNode, oldNode)** объекта Node. Этот метод в качестве первого параметра принимает новый элемент, который заменяет старый элемент oldNode, передаваемый в качестве второго параметра.

|  |  |
| --- | --- |
|  | var articleDiv = document.querySelector("div.article");  // находим узел, который будем заменять - первый параграф  var oldNode = document.querySelectorAll("div.article p")[0];  // создаем элемент  var newNode = document.createElement("h2");  // создаем для него текст  var elemText = document.createTextNode("Привет мир");  // добавляем текст в элемент в качестве дочернего элемента  newNode.appendChild(elemText);  // заменяем старый узел новым  articleDiv.replaceChild(newNode, oldNode); |

В данном случае заменяем первый параграф заголовком h2:

## Объект Element. Управление элементами

Кроме методов и свойств объекта Node в JavaScript мы можем использовать свойства и методы объектов Element. Важно не путать эти два объекта: Node и Element. Node представляет все узлы веб-станицы, в то время как объект Element представляет непосредственно только html-элементы. То есть объекты Element - это фактически те же самые узлы - объекты Node, у которых тип узла (свойство nodeType) равно 1. Одним из ключевых свойств объекта Element является свойство **tagName**, которое возвращает тег элемента. Например, получим все элементы, которые есть на странице:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <div class="article">  <h3>Заголовок статьи</h3>  <p>Первый абзац</p>  <p>Второй абзац</p>  </div>  <script>  function getChildren(elem){      for(var i in elem.childNodes){           if(elem.childNodes[i].nodeType===1){              console.log(elem.childNodes[i].tagName);              getChildren(elem.childNodes[i]);          }      }  }  var root = document.documentElement;  console.log(root.tagName);  getChildren(root);  </script>  </body>  </html> |

Здесь вначале получаем те корневого элемента <html> и затем с помощью рекурсивной функции getChildren получаем все вложенные элементы:

### Свойства innerText и innerHTML

Для получения или установки текстового содержимого элемента мы можем использовать свойство **innerText**, а для получения или установки кода html - свойство **innerHTML**:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <div class="article">  <h3>Заголовок статьи</h3>  <p>Первый абзац</p>  <p>Второй абзац</p>  </div>  <script>  var articleDiv = document.querySelector("div.article");  console.log(articleDiv.innerText);  console.log("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");  console.log(articleDiv.innerHTML);  </script>  </body>  </html> |

Надо отметить, что свойство innerText во многом аналогично свойству **textContent**. То есть следующие вызовы будут равноценны:

|  |  |
| --- | --- |
|  | var pElement = document.querySelectorAll("div.article p")[0];  pElement.innerText = "hello";  pElement.textContent = "hello"; |

Установка кода html у элемента:

|  |  |
| --- | --- |
|  | var articleDiv = document.querySelector("div.article");  articleDiv.innerHTML ="<h2>Hello World!!!</h2><p>bla bla bla</p>"; |

### Методы объекта Element

Среди методов объекта Element можно отметить методы управления атрибутами:

* **getAttribute(attr)**: возвращает значение атрибута attr
* **setAttribute(attr, value)**: устанавливает для атрибута attr значение value. Если атрибута нет, то он добавляется
* **removeAttribute(attr)**: удаляет атрибут attr и его значение

Работа с атрибутами:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <div class="article" style="color:red;">  <h3>Заголовок статьи</h3>  <p>Первый абзац</p>  <p>Второй абзац</p>  </div>  <script>  var articleDiv = document.querySelector("div.article");  // получаем атрибут style  var styleValue = articleDiv.getAttribute("style");  console.log("До изменения атрибута: " + styleValue);  // удаляем атрибут  articleDiv.removeAttribute("style");  // добавляем заново атрибут style  articleDiv.setAttribute("style", "color:blue;");  styleValue = articleDiv.getAttribute("style");  console.log("После изменения атрибута: " + styleValue);  </script>  </body>  </html> |

### Размеры и позиция элементов

Элементы имеют ряд свойств, которые позволяют определить размер элемента. Но важно понимать разницу между всеми этими свойствами. Свойства **offsetWidth** и **offsetHeight** определяют соответственно ширину и высоту элемента в пикселях. В ширину и высоту включается граница элемента. Свойства **clientWidth** и **clientHeight** также определяют ширину и высоту элемента в пикселях, но уже без учета границы:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <style>      #rect {          width: 100px;          height: 100px;          background: #50c878;          border: 3px solid silver;      }      </style>  </head>  <body>  <div id="rect"></div>  <script>  var rect = document.getElementById("rect");  console.log("offsetHeight: " + rect.offsetHeight);  console.log("offsetWidth: " + rect.offsetWidth);  console.log("clientHeight: " + rect.clientHeight);  console.log("clientWidth: " + rect.clientWidth);  </script>  </body>  </html> |

Поскольку у блока div определена граница в 3 пикселя, то по сравнению с clientHeight/clientWidth к offsetHeight/offsetWidth добавляет по 6 пикселей. Для определения позиции элемента наиболее эффективным способом является метод **getBoundingClientRect()**. Этот метод возвращает объект со свойствами top, bottom, left, right, которые указывают на смещение элемента относительно верхнего левого угла браузера:

|  |  |
| --- | --- |
|  | var rect = document.getElementById("rect");  var clientRect  = rect.getBoundingClientRect();  console.log("top: " + clientRect.top);  console.log("bottom: " + clientRect.bottom);  console.log("left: " + clientRect.left);  console.log("right: " + clientRect.right); |

## Изменение стиля элементов

Для работы со стилевыми свойствами элементов в JavaScript применяются, главным образом, два подхода:

* Изменение свойства **style**
* Изменение значения атрибута **class**

### Свойство style

Свойство **style** представляет сложный объект для управления стилем и напрямую сопоставляется с атрибутом style html-элемента. Этот объект содержит набор свойств CSS: element.style.свойствоCSS. Например, установим цвет шрифта:

|  |  |
| --- | --- |
|  | var root = document.documentElement;  // устанавливаем стиль  root.style.color = "blue";  document.write(root.style.color); // blue |

В данном случае название свойства color совпадает со свойством css. Аналогично мы могли бы установить цвет с помощью css:

|  |  |
| --- | --- |
|  | html{      color:blue;  } |

Однако ряд свойств css в названиях имеют дефис, например, font-family. В JavaScript для этих свойств дефис не употребляется. Только первая буква, которая идет после дефиса, переводится в верхний регистр:

|  |  |
| --- | --- |
|  | var root = document.documentElement;  root.style.fontFamily = "Verdana"; |

### Свойство className

С помощью свойства **className** можно установить атрибут class элемента html. Например:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <style>      .blueStyle{          color:blue;          font-family:Verdana;      }      .article{          font-size:20px;      }      </style>  </head>  <body>  <div class="article">  <h3>Заголовок статьи</h3>  <p>Первый абзац</p>  <p>Второй абзац</p>  </div>  <script>  var articleDiv = document.querySelector("div.article");  // установка нового класса  articleDiv.className = "blueStyle";  // получаем название класса  document.write(articleDiv.className);  </script>  </body>  </html> |

Благодаря использованию классов не придется настраивать каждое отдельное свойство css с помощью свойства style. Но при этом надо учитывать, что прежнее значение атрибута class удаляется. Поэтому, если нам надо добавить класс, надо объединить его название со старым классом:

|  |  |
| --- | --- |
|  | articleDiv.className = articleDiv.className + " blueStyle"; |

И если надо вовсе удалить все классы, то можно присвоить свойству пустую строку:

|  |  |
| --- | --- |
|  | articleDiv.className = ""; |

### Свойство classList

Выше было рассмотрено, как добавлять классы к элементу, однако для управления множеством классов гораздо удобнее использовать свойство **classList**. Это свойство представляет объект, реализующий следующие методы:

* **add(className)**: добавляет класс className
* **remove(className)**: удаляет класс className
* **toggle(className)**: переключает у элемента класс на className. Если класса нет, то он добавляется, если есть, то удаляется

Например:

|  |  |
| --- | --- |
|  | var articleDiv = document.querySelector("div.article");  // удаляем класс  articleDiv.classList.remove("article");  // добавляем класс  articleDiv.classList.add("blueStyle");  // переключаем класс  articleDiv.classList.toggle("article"); |

# События

## Введение в обработку событий

ля взаимодействия с пользователем в JavaScript определен механизм событий. Например, когда пользователь нажимает кнопку, то возникает событие нажатия кнопки. В коде JavaScript мы можем определить возникновение события и как-то его обработать. В JavaScript есть следующие типы событий:

* События мыши (перемещение курсора, нажатие мыши и т.д.)
* События клавиатуры (нажатие или отпускание клавиши клавиатуры)
* События жизненного цикла элементов (например, событие загрузки веб-станицы)
* События элементов форм (нажатие кнопки на форме, выбор элемента в выпадающем списке и т.д.)
* События, возникающие при изменении элементов DOM
* События, возникающие при касании на сенсорных экранах
* События, возникающие при возникновении ошибок

Рассмотрим простейшую обработку событий. Например, на веб-странице у нас есть следующий элемент div:

|  |  |
| --- | --- |
| 1 | <div id="rect" onclick="alert('Нажато')" style="width:50px;height:50px;background-color:blue;"></div> |

Здесь определен обычный блок div, который имеет атрибут **onclick**, который задает **обработчик события** нажатия на блок div. То есть, чтобы обработать какое-либо событие, нам надо определить для него обработчик. Обработчик представляет собой код на языке JavaScript. В данном случае обработчик выглядит довольно просто:

|  |  |
| --- | --- |
| 1 | alert('Нажато') |

И при нажатии на кнопку будет выскакивать сообщение: Также можно было бы вынести все действия по обработке события в отдельную функцию:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <div id="rect" onclick="displayMessage()" style="width:50px;height:50px;background-color:blue;"></div>  <script>  function displayMessage(){        alert('Нажато');  }  </script>  </body>  </html> |

Теперь обработчиком события будет выступать функция displayMessage.

### Передача параметров в обработчик события

В обработчик можно передавать параметры. Например, мы можем передать текущий объект, на котором возникает событие:

|  |  |
| --- | --- |
|  | <a href="page1.html" onclick="return handler(this)">Станица 1</a>  <script>  function handler(obj){        alert(obj.href);      return false;  }  </script> |

Ключевое слово this указывает на текущий объект ссылки, на которую производится нажатие. И в коде обработчика мы можем получить этот объект и обратиться к его свойствам, например, к свойству href.

Кроме того, надо отметить, что здесь обработчик возвращает результат. Хотя в первом примере с блоком div от обработчика не требовалось возвращения результата. Дело в том, что для некоторых обработчиков можно подтвердить или остановить обработку события. Например, нажатие на ссылку должно привести к переадресации. Но возвращая из обработчика false, мы можем остановить стандартный путь обработки события, и переадресации не будет. Если же возвращать значение true, то событие обрабатывается в стандартном порядке. Если же мы вовсе уберем возвращении результата, то событие будет обрабатываться, как будто возвращается значение true:

|  |  |
| --- | --- |
|  | <a href="page1.html" onclick="handler(this)">Станица 1</a>  <script>  function handler(obj){        alert(obj.href);  }  </script> |

Кроме непосредственно элемента-источника события в обработчик мы можем передавать объект event. Этот объект не определяется разработчиком, это просто аргумент функции обработчика, который хранит всю информацию о событии. Например:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <style>      #rect{          width:50px;          height:50px;          background-color:blue;      }      </style>  </head>  <body>  <div id="rect" onclick="handler(event)"></div>  <script>  function handler(e){        alert(e.type); // получаем тип события  }  </script>  </body>  </html> |

В данном случае с помощью свойства type объекта event получаем тип события (в данном случае тип click).

## Обработчики событий

### Встроенные обработчики

В прошлой теме были рассмотрены встроенные обработчики (inline event handler), которые определяются в коде элемента с помощью атрибутов:

|  |  |
| --- | --- |
| 1 | <div id="rect" onclick="handler(event)"></div> |

Хотя этот подход прекрасно работает, но он имеет кучу недостатков:

* Код html смешивается с кодом JavaScript, в связи с чем становится труднее разрабатывать, отлаживать и поддерживать приложение
* Обработчики событий можно задать только для уже созданных на веб-странице элементов. Динамически создаваемые элементы в этом случае лишаются возможности обработки событий
* К элементу для одного события может быть прикреплен только один обработчик
* Нельзя удалить обработчик без изменения кода

### Свойства обработчиков событий

Проблемы, которые возникают при использовании встроенных обработчиков, были призваны решить свойства обработчиков. Подобно тому, как у html-элементов есть атрибуты для обработчиков, так и в коде javascript у элементов DOM мы можем получить свойства обработчиков, которые соответствуют атрибутам:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <style>      #rect{          width:50px;          height:50px;          background-color:blue;      }      </style>  </head>  <body>  <div id="rect"></div>  <script>  function handler(e){        alert(e.type);  }  document.getElementById("rect").onclick = handler;  </script>  </body>  </html> |

В итоге нам достаточно взять свойство onclick и присвоить ему функцию, используемую в качестве обработчика. За счет этого код html отделяется от кода javascript. Стоит также отметить, что в обработчик события браузер автоматически передает объект Event, хранящий всю информацию о событии. Поэтому также мы можем получить этот объект в функции обработчика в качестве параметра.

### Слушатели событий

Несмотря на то, что свойства обработчиков решают ряд проблем, которые связаны с использованием атрибутов, в то же время это также не оптимальный подход. Еще один способ установки обработчиков событий представляет использование слушателей. Для работы со слушателями событий в JavaScript есть объект **EventTarget**, который определяет методы **addEventListener()** (для добавления слушателя) и **removeEventListener()** для удаления слушателя. И поскольку html-элементы DOM тоже являются объектами EventTarget, то они также имеют эти методы. Фактически слушатели представляют те же функции обработчиков. Метод addEventListener() принимает два параметра: название события без префикса on и функцию обработчика этого события. Например:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <style>      #rect{          width:50px;          height:50px;          background-color:blue;      }      </style>  </head>  <body>  <div id="rect"></div>  <script>  var rect = document.getElementById("rect");    rect.addEventListener("click", function (e) {      alert(e.type);  });  </script>  </body>  </html> |

То есть в данном случае опять же обрабатывается событие click. И также можно было бы в качестве второго параметра название функции:

|  |  |
| --- | --- |
|  | function handler(e){     alert(e.type);  }  var rect = document.getElementById("rect");   rect.addEventListener("click", handler); |

Удаление слушателя аналогично добавлению:

|  |  |
| --- | --- |
| 1 | rect.removeEventListener("click", handler); |

Преимуществом использования слушателей является и то, что мы можем установить для одного события несколько функций:

|  |  |
| --- | --- |
|  | var clicks = 0;  function handlerOne(e){      alert(e.type);  }  function handlerTwo(e){      clicks++;      var newNode = document.createElement("p");      newNode.textContent = "произошло нажатие " + clicks;      document.body.appendChild(newNode);  }  var rect = document.getElementById("rect");  // прикрепляем первый обработчик  rect.addEventListener("click", handlerOne);  // прикрепляем второй обработчик  rect.addEventListener("click", handlerTwo); |

## Объект Event

При обработке события браузер автоматически передает в функцию обработчика в качестве параметра объект **Event**, который инкапсулирует всю информацию о событии. И с помощью его свойств мы можем получить эту информацию:

* **bubbles**: возвращает true, если событие является восходящим. Например, если событие возникло на вложенном элементе, то оно может быть обработано на родительском элементе.
* **cancelable**: возвращает true, если можно отменить стандартную обработку события
* **currentTarget**: определяет элемент, к которому прикреплен обработчик события
* **defaultPrevented**: возвращает true, если был вызван у объекта Event метод preventDefault()
* **eventPhase**: определяет стадию обработки события
* **target**: указывает на элемент, на котором было вызвано событие
* **timeStamp**: хранит время возникновения события
* **type**: указывает на имя события

Например:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <style>      #rect{          width:50px;          height:50px;          background-color:blue;      }      </style>  </head>  <body>  <div id="rect"></div>  <script>  function handler(event){      console.log("Тип события: " + event.type);      console.log(event.target);  }  var rect = document.getElementById("rect");  rect.addEventListener("click", handler);  </script>  </body>  </html> |

Причем в данном случае свойство target представляет собой элемент, поэтому мы можем манипулировать им как и любыми другими узлами и элементами DOM. Например, изменим фоновый цвет:

|  |  |
| --- | --- |
|  | function handler(e){      e.target.style.backgroundColor = "red";  } |

### Остановка выполнения события

С помощью метода **preventDefault()** объекта Event мы можем остановить дальнейшее выполнение события. В ряде случаев этот метод не играет большой роли. Однако в некоторых ситуаций он может быть полезен. Например, при нажатии на ссылку мы можем с помощью дополнительной обработки определить, надо ли переходить по ссылке или надо запретить переход. Или другой пример: пользователь отправляет данные формы, но в ходе обработки в обработчике события мы определили, что поля формы заполнены неправильно, и в этом случае мы также можем запретить отправку.

Например, запретим переход по ссылке после 12 часов:

|  |  |
| --- | --- |
|  | <a href="[http://google.com](http://google.com/)" id="link">Поиск</a>  <script>  function linkHandler(e){        var date = new Date();      var hour = date.getHours();      console.log(hour);      if(hour>12){            e.preventDefault();          document.write("После 12 переход запрещен");      }  }  var link = document.getElementById("link");  link.addEventListener("click", linkHandler);  </script> |

## Распространение событий

Когда мы нажимаем на какой-либо элемент на станице и генерируется событие нажатия, то это событие может распространяться от элемента к элементу. Например, если мы нажимаем на блок div, то также мы нажимаем и на элемент body, в котором блок div находится. То есть происходит распространение события. Есть несколько форм распространения событий:

* Восходящие: событие распространяется вверх по дереву DOM от дочерних узлов к родительским
* Нисходящие: событие распространяется вниз по дереву DOM от родительских узлов к дочерним, пока не достигнет того элемента, на котором это событие и возникло

### Восходящие события

Рассмотрим восходящие (bubbling) события, которые распространяются в верх по дереву DOM. Допустим, у нас есть следующая веб-страница:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <style>      #blueRect{          width:100px;          height:100px;          background-color:blue;      }      #redRect{          width:50px;          height:50px;          background-color:red;      }      </style>  </head>  <body>  <div id="blueRect">      <div id="redRect"></div>  </div>    <script>  var redRect = document.getElementById("redRect");  redRect.addEventListener("click", function(){      console.log("Событие на redRect");  });    var blueRect = document.getElementById("blueRect");  blueRect.addEventListener("click", function(){      console.log("Событие на blueRect");  });    document.body.addEventListener("click", function(){      console.log("Событие на body");  });  </script>  </body>  </html> |

Если мы нажмем на вложенный div, то событие пойдет к родительскому элементу div и далее к элементу body:

Надо сказать, что подобное поведение не всегда является желательным. И в этом случае мы можем остановить распространение событие с помощью метода **stopPropagation()** объекта Event:

|  |  |
| --- | --- |
|  | var redRect = document.getElementById("redRect");  redRect.addEventListener("click", function(e){      console.log("Событие на redRect");      e.stopPropagation();  }); |

И в результате нажатия событие будет обработано только обработчиком для redRect.

### Нисходящие события

События также могут быть нисходящими (capturing). Для их использования в метод **addEventListener()** в качестве третьего необязательного параметра передается логическое значение true или false, которое указывает, будет ли событие нисходящим. По умолчанию все события восходящие. Возьмем ту же веб-станицу, только изменим ее код javascript:

|  |  |
| --- | --- |
|  | var redRect = document.getElementById("redRect");  redRect.addEventListener("click", function(){      console.log("Событие на redRect");  }, true);    var blueRect = document.getElementById("blueRect");  blueRect.addEventListener("click", function(){      console.log("Событие на blueRect");  }, true);    document.body.addEventListener("click", function(){      console.log("Событие на body");  }, true); |

Теперь события будут распространяться в обратном порядке:

## События мыши

Одну из наиболее часто используемых событий составляют события мыши:

* click: возникает при нажатии указателем мыши на элемент
* mousedown: возникает при нахождении указателя мыши на элементе, когда кнопка мыши находится в нажатом состоянии
* mouseup: возникает при нахождении указателя мыши на элементе во время отпускания кнопки мыши
* mouseover: возникает при вхождении указателя мыши в границы элемента
* mousemove: возникает при прохождении указателя мыши над элементом
* mouseout: возникает, когда указатель мыши выходит за пределы элемента

Например, обработаем события mouseover и mouseout:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <style>      #blueRect{          width:100px;          height:100px;          background-color:blue;      }      </style>  </head>  <body>  <div id="blueRect"></div>    <script>  function setColor(e){        if(e.type==="mouseover")          e.target.style.backgroundColor = "red";      else if(e.type==="mouseout")          e.target.style.backgroundColor = "blue";  }  var blueRect = document.getElementById("blueRect");  blueRect.addEventListener("mouseover", setColor);  blueRect.addEventListener("mouseout", setColor);  </script>  </body>  </html> |

Теперь при наведении указателя мыши на блок blueRect он будет окрашиваться в красный цвет, а при уходе указателя мыши - блок будет обратно окрашиваться в синий цвет. Объект Event является общим для всех событий. Однако для разных типов событий существуют также свои объекты событий, которые добавляют ряд своих свойств. Так, для работы с событиями указателя мыши определен объект **MouseEvent**, который добавляет следующие свойства:

* **altKey**: возвращает true, если была нажата клавиша Alt во время генерации события
* **button**: указывает, какая кнопка мыши была нажата
* **clientX**: определяет координату Х окна браузера, на которой находился указатель мыши во время генерации события
* **clientY**: определяет координату Y окна браузера, на которой находился указатель мыши во время генерации события
* **ctrlKey**: возвращает true, если была нажата клавиша Ctrl во время генерации события
* **metaKey**: возвращает true, если была нажата во время генерации события метаклавиша клавиатуры
* **relatedTarget**: определяет вторичный источник возникновения события
* **screenX**: определяет координату Х относительно верхнего левого угла экрана монитора, на которой находился указатель мыши во время генерации события
* **screenY**: определяет координату Y относительно верхнего левого угла экрана монитора, на которой находился указатель мыши во время генерации события
* **shiftKey**: возвращает true, если была нажата клавиша Shift во время генерации события

Определим координаты клика:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <style>      #blueRect{          width:100px;          height:100px;          background-color:blue;      }      </style>  </head>  <body>  <div id="blueRect"></div>    <script>  function handleClick(e){        console.log("screenX: " + e.screenX);      console.log("screenY: " + e.screenY);      console.log("clientX: " + e.clientX);  console.log("clientY: " + e.clientY);  }  var blueRect = document.getElementById("blueRect");  blueRect.addEventListener("click", handleClick);  </script>  </body>  </html> |

## События клавиатуры

Другим распространенным типом событий являются события клавиатуры.

* keydown: возникает при нажатии клавиши клавиатуры и длится, пока нажата клавиша
* keyup: возникает при отпускании клавиши клавиатуры
* keypress: возникает при нажатии клавиши клавиатуры, но после события keydown и до события keyup. Надо учитывать, что данное событие генерируется только для тех клавиш, которые формируют вывод в виде символов, например, при печати символов. Нажатия на остальные клавиши, например, на Alt, не учитываются.

Для работы с событиями клавиатуры определен объект **KeyboardEvent**, который добавляет к свойствам объекта Event ряд специфичных для клавиатуры свойств:

* **altKey**: возвращает true, если была нажата клавиша Alt во время генерации события
* **charCode**: возвращает символ Unicode для нажатой клавиши (используется для события keypress)
* **keyCode**: возвращает числовое представление нажатой клавиши клавиатуры
* **ctrlKey**: возвращает true, если была нажата клавиша Ctrl во время генерации события
* **metaKey**: возвращает true, если была нажата во время генерации события метаклавиша клавиатуры
* **shiftKey**: возвращает true, если была нажата клавиша Shift во время генерации события

Например, мы можем с помощью клавиш клавиатуры перемещать элемент на веб-странице:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <style>      html, body{          margin:0;          overflow:hidden;      }      #blueRect{          width:100px;          height:100px;          background-color:blue;      }      </style>  </head>  <body>  <div id="blueRect"></div>    <script>  function moveRect(e){        var blueRect = document.getElementById("blueRect");      // получаем стиль для blueRect      var cs = window.getComputedStyle(blueRect);        var left = parseInt(cs.marginLeft);      var top = parseInt(cs.marginTop);        switch(e.keyCode){            case 37:  // если нажата клавиша влево              if(left>0)                  blueRect.style.marginLeft = left - 10 + "px";              break;          case 38:   // если нажата клавиша вверх              if(top>0)                  blueRect.style.marginTop = top - 10 + "px";              break;          case 39:   // если нажата клавиша вправо              if(left < document.documentElement.clientWidth - 100)                  blueRect.style.marginLeft = left + 10 + "px";              break;          case 40:   // если нажата клавиша вниз              if(top < document.documentElement.clientHeight - 100)                  blueRect.style.marginTop = top + 10 + "px";              break;      }  }    addEventListener("keydown", moveRect);  </script>  </body>  </html> |

В данном случае обрабатывается событие keydown. В обработчике moveRect с помощью метода window.getComputedStyle() получаем стиль элемента blueRect. А затем из этого стиля выбираем значения свойств marginLeft и marginTop. С помощью свойства e.keyCode получаем нажатую клавишу. Список кодов клавиш клавиатуры можно посмотреть на сайте <http://www.javascriptkeycode.com/>. Здесь нам интересуют четыре клавиши: вверх, вниз, влево, вправо. Если одна из них нажата, производим действия: увеличение или уменьшение отступа элемента от верхней или левой границы. Ну и чтобы элемент не выходил за границы окна, проверяем предельные значения с помощью document.documentElement.clientWidth (ширина корневого элемента) и document.documentElement.clientHeight (высота корневого элемента).

# Работа с формами

## Формы и их элементы

Один из способов взаимодействия с пользователями представляют html-формы. Например, если нам надо получить от пользователя некоторую информацию, мы можем определить на веб-странице формы, которая будет содержать текстовые поля для ввода информации и кнопку для отправки. И после ввода данных мы можем обработать введенную информацию.

Для создания формы используется элемент <form>:

|  |  |
| --- | --- |
| 1  2 | <form name="search">  </form> |

В JavaScript форма представлена объектом **HtmlFormElement**. И после создания формы мы можем к ней обратиться различными способами. Первый способ заключается в прямом обращении по имени формы:

|  |  |
| --- | --- |
| 1 | var searchForm = document.search; |

Второй способ состоит в обращении к коллекции форм документа и поиске в ней нужной формы:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <form name="search"></form>  <form name="settings"></form>  <script>  var searchForm;  for (var i = 0; i < document.forms.length; i++) {        if(document.forms[i].name==="search")          searchForm = document.forms[i];  }  document.write(searchForm.name);  </script>  </body>  </html> |

С помощью свойства name объекта формы мы можем получить значение атрибута name у соответствующего элемента формы в коде html. Еще один способ сочетает оба подхода:

|  |  |
| --- | --- |
| 1 | var searchForm = document.forms["search"]; |

И также можно применять стандартные способы для поиска элемента формы, например, по id, по тегу или по селектору. Например:

|  |  |
| --- | --- |
| 1 | var searchForm = document.getElementsByTagname("form")[0] |

Форма имеет ряд свойств, из которых наиболее важными являются вышерассмотренное свойство name, а также свойство **elements**, которое содержит коллекцию элементов формы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <form name="search">      <input type="text" name="key"></input>      <input type="submit" name="send"></input>  </form>  <script>  var searchForm = document.forms["search"];  for(var i=0; i<searchForm.elements.length;i++)      document.write(searchForm.elements[i].name + "<br/>");  </script> |

Среди методов формы надо отметить метод **submit()**, который отправляет данные формы на сервер, и метод **reset()**, который очищает поля формы:

|  |  |
| --- | --- |
| 1  2  3 | var searchForm = document.forms["search"];  searchForm.submit();  searchForm.reset(); |

**Элементы форм**

Форма может содержать различные элементы ввода html: input, textarea, button, select и т.д. Но все они имеют ряд общих свойств и методов. Также, как и форма, элементы форм имеют свойство **name**, с помощью которого можно получить значение атрибута name:

|  |  |
| --- | --- |
|  | <form name="search">      <input type="text" name="key" value="hello world"></input>      <input type="submit" name="send"></input>  </form>  <script>  var searchForm = document.forms["search"];  // выведем имя всех элементов  for(var i=0; i<searchForm.elements.length;i++)      document.write(searchForm.elements[i].name + "<br/>");    // получим по имени текстовое поле  var keyBox = searchForm.elements["key"];  document.write(keyBox.name); // key  </script> |

Другим важным свойством является свойство **value**, которое позволяет получить или изменить значение поля:

|  |  |
| --- | --- |
|  | var searchForm = document.forms["search"];  var keyBox = searchForm.elements["key"];  document.write(keyBox.value); // hello world  // установка значения  keyBox.value = "Привет мир"; |

С помощью свойства **form** можно получить родительский объект формы:

|  |  |
| --- | --- |
|  | var searchForm = document.forms["search"];  var keyBox = searchForm.elements["key"];  document.write(keyBox.form.name); // search |

Данное свойство может быть полезно, например, при отправке формы, когда перед непосредственной отправкой формы необходимо провести валидацию всех полей формы. Свойство **type** позволяет получить тип поля ввода. Это либо название тега элемента (например, textarea), либо значение атрибута type у элементов input. Из методов можно выделить методы **focus()** (устанавливает фокус на элемент) и **blur()** (убирает фокус с элемента):

|  |  |
| --- | --- |
| 1  2  3 | var searchForm = document.forms["search"];  var keyBox = searchForm.elements["key"];  keyBox.focus(); |

## Кнопки

Для отправки введенных данных на форме используются кнопки. Для создания кнопки используется либо элемент button:

|  |  |
| --- | --- |
| 1 | <button name="send">Отправить</button> |

Либо элемент input:

|  |  |
| --- | --- |
| 1 | <input type="submit" name="send" value="Отправить" /> |

С точки зрения функциональности в html эти элементы не совсем равноценны, но в данном случае они нас интересуют с точки зрения взаимодействия с кодом javascript. При нажатии на любой из этих двух вариантов кнопки происходит отправка формы по адресу, который указан у формы в атрибуте action, либо по адресу веб-страницы, если атрибут action не указан. Однако в коде javascript мы можем перехватить отправку, обрабатывая событие **click**

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <form name="search">      <input type="text" name="key"></input>      <input type="submit" name="send" value="Отправить" />  </form>  <script>  function sendForm(e){        // получаем значение поля key      var keyBox = document.search.key;      var val = keyBox.value;      if(val.length>5){          alert("Недопустимая длина строки");          e.preventDefault();      }      else          alert("Отправка разрешена");  }    var sendButton = document.search.send;  sendButton.addEventListener("click", sendForm);  </script>  </body>  </html> |

При нажатии на кнопку происходит событие click, и для его обработки к кнопке прикрепляем обработчик sendForm. В этом обработчике проверяем введенный в текстовое поле текст. Если его длина больше 5 символов, то выводим сообщение о недостимой длине и прерываем обычный ход события с помощью вызова **e.preventDefault()**. В итоге форма не отправляется. Если же длина текста меньше шести символов, то также выводится сообщение, и затем форма отправляется. Также мы можем при необходимости при отправке изменить адрес, на который отправляются данные:

|  |  |
| --- | --- |
|  | function sendForm(e){        // получаем значение поля key      var keyBox = document.search.key;      var val = keyBox.value;      if(val.length>5){          alert("Недопустимая длина строки");          document.search.action="PostForm";      }      else          alert("Отправка разрешена");  } |

В данном случае, если длина текста больше пяти символов, то текст отправляется, только теперь он отправляется по адресу PostForm, поскольку задано свойство action:

|  |  |
| --- | --- |
| 1 | document.search.action="PostForm"; |

Для очистки формы предназначены следующие равноценные по функциональности кнопки:

|  |  |
| --- | --- |
| 1  2 | <button type="reset">Очистить</button>  <input type="reset" value="Очистить" /> |

При нажатию на кнопки произойдет очистка форм. Но также функциональность по очистке полей формы можно реализовать с помощью метода reset():

|  |  |
| --- | --- |
|  | function sendForm(e){        // получаем значение поля key      var keyBox = document.search.key;      var val = keyBox.value;      if(val.length>5){          alert("Недопустимая длина строки");          document.search.reset();          e.preventDefault();      }      else          alert("Отправка разрешена");  } |

Кроме специальных кнопок отправки и очистки на форме также может использоваться обычная кнопка:

|  |  |
| --- | --- |
| 1 | <input type="button" name="send" value="Отправить" /> |

При нажатии на подобную кнопку отправки данных не происходит, хотя также генерируется событие click:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <form name="search">      <input type="text" name="key" placeholder="Введите ключ"></input>      <input type="button" name="print" value="Печать" />  </form>  <div id="printBlock"></div>  <script>  function printForm(e){        // получаем значение поля key      var keyBox = document.search.key;      var val = keyBox.value;      // получаем элемент printBlock      var printBlock = document.getElementById("printBlock");      // создаем новый параграф      var pElement = document.createElement("p");      // устанавливаем у него текст      pElement.textContent = val;      // добавляем параграф в printBlock      printBlock.appendChild(pElement);  }    var printButton = document.search.print;  printButton.addEventListener("click", printForm);  </script>  </body>  </html> |

При нажатии на кнопку получаем введенный в текстовое поле текст, создаем новый элемент параграфа для этого текста и добавляем параграф в элемент printBlock.

## **Текстовые поля**

Для ввода простейшей текстовой информации предназначены элементы <input type="text":

|  |  |
| --- | --- |
| 1 | <input type="text" name="kye" size="10" maxlength="15" value="hello world" /> |

Данный элемент поддерживает ряд событий, в частности:

* focus: происходит при получении фокуса
* blur: происходит при потере фокуса
* change: происходит при изменении значения поля
* select: происходит при выделении текста в текстовом поле
* keydown: происходит при нажатии клавиши клавиатуры
* keypress: происходит при нажатии клавиши клавиатуры для печатаемых символов
* keyup: происходит при отпускании ранее нажатой клавиши клавиатуры

Применим ряд событий:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <form name="search">      <input type="text" name="key" placeholder="Введите ключ"></input>      <input type="button" name="print" value="Печать" />  </form>  <div id="printBlock"></div>  <script>  var keyBox = document.search.key;    // обработчик изменения текста  function onchange(e){      // получаем элемент printBlock      var printBlock = document.getElementById("printBlock");      // получаем новое значение      var val = e.target.value;      // установка значения      printBlock.textContent = val;  }  // обработка потери фокуса  function onblur(e){        // получаем его значение и обрезаем все пробелы      var text = keyBox.value.trim();      if(text==="")          keyBox.style.borderColor = "red";      else          keyBox.style.borderColor = "green";  }  // получение фокуса  function onfocus(e){        // установка цвета границ поля      keyBox.style.borderColor = "blue";  }  keyBox.addEventListener("change", onchange);  keyBox.addEventListener("blur", onblur);  keyBox.addEventListener("focus", onfocus);  </script>  </body>  </html> |

Здесь к текстовому полю прикрепляется три обработчика для событий blur, focus и change. Обработка события change позволяет сформировать что-то вроде привязки: при изменении текста весь текст отображается в блоке printBlock. Но надо учитывать, что событие change возникает не сразу после изменения текста, а после потери им фокуса. Обработка события потери фокуса blur позволяет провести валидацию введенного значения. Например, в данном случае если текст состоит из пробелов или не был введен, то окрашиваем границу поля в красный цвет.

Кроме данного текстового поля есть еще специальные поля ввода. Так, поле <input type="password" предназначено для ввода пароля. По функциональности оно во многом аналогично обычному текстовому полю за тем исключением, что для вводимых символов используется маска: <input type="password" name="password" />

Если нам надо, чтобы на форме было некоторое значение, но чтобы оно было скрыто от пользователя, то для этого могут использоваться скрытые поля: <input type="hidden" name="id" value="345" />

Для скрытого поля обычно не используется обработка событий, но также, как и для других элементов, мы можем в javascript получить его значение или изменить его.

### Элемент textarea

Для создания многострочных текстовых полей используется элемент textarea:

|  |  |
| --- | --- |
| 1 | <textarea rows="15" cols="40" name="textArea"></textarea> |

Данные элемент генерирует все те же самые события, что и обычное текстовое поле:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <form name="search">      <textarea rows="7" cols="40" name="message"></textarea>  </form>  <div id="printBlock"></div>  <script>  var messageBox = document.search.message;   // обработчик ввода символа  function onkeypress(e){      // получаем элемент printBlock      var printBlock = document.getElementById("printBlock");      // получаем введенный символ      var val = String.fromCharCode(e.keyCode);      // добавление символа      printBlock.textContent += val;  }   function onkeydown(e){      if(e.keyCode===8){ // если нажат Backspace            // получаем элемент printBlock          var printBlock = document.getElementById("printBlock"),              length = printBlock.textContent.length;          // обрезаем строку по последнему символу          printBlock.textContent = printBlock.textContent.substring(0, length-1);      }  }   messageBox.addEventListener("keypress", onkeypress);  messageBox.addEventListener("keydown", onkeydown);  </script>  </body>  </html> |

Здесь к текстовому полю прикрепляются обработчики для событий keypress и keydown. В обработчике keypress получаем введенный символ с помощью конвертации числового кода клавиши в строку: var val = String.fromCharCode(e.keyCode);

Затем символ добавляется к содержимому блока printBlock. Событие keypress возникает при нажатии на клавиши для печатаемых символов, то такие символы отображаются в текстовом поле. Однако есть и другие клавиши, которые оказывают влияние на текстовое поле, но они не дают отображаемого символа, поэтому не отслеживаются событием keypress. К таким клавишам относится клавиша Backspace, которая удаляет последний символ. И для ее отслеживания также обрабатываем событие keydown. В обработчике keydown удаляем из строки в блоке printBlock последний символ.

## Флажки и переключатели

Особую группу элементов ввода составляют флажки и переключатели. Флажки представляют поле, в которое можно поставить отметки и которое создается с помощью элемента <input type="checkbox". Отличительную особенность флажка составляет свойство **checked**, которое в отмеченном состоянии принимает значение true:

|  |  |
| --- | --- |
|  | <form name="myForm">      <input type="checkbox" name="enabled" checked><span>Включить</span>  </form>  <div id="printBlock"></div>  <script>  var enabledBox = document.myForm.enabled;  function onclick(e){      var printBlock = document.getElementById("printBlock");      var enabled = e.target.checked;      printBlock.textContent = enabled;  }  enabledBox.addEventListener("click", onclick);  </script> |

Нажатие на флажок генерирует событие click. В данном случае при обработке данного события мы просто выводим информацию, отмечен ли данный флажок, в блок div. Переключатели представляют группы кнопок, из которых мы можем выбрать только одну. Переключатели создаются элементом <input type="radio". Выбор или нажатие на одну из них также представляет событие click:

|  |  |
| --- | --- |
|  | <form name="myForm">      <input type="radio" name="languages" checked="checked" value="Java" /><span>Java</span>      <input type="radio" name="languages" value="C#" /><span>C#</span>      <input type="radio" name="languages" value="C++" /><span>C++</span>  </form>  <div id="printBlock"></div>  <script>  function onclick(e){      var printBlock = document.getElementById("printBlock");      var language = e.target.value;      printBlock.textContent = "Вы выбрали: " + language;  }  for (var i = 0; i < myForm.languages.length; i++) {      myForm.languages[i].addEventListener("click", onclick);  }  </script> |

При создании группы переключателей их атрибут name должен иметь одно и то же значение. В данном случае это - languages. То есть переключатели образуют группу languages. Поскольку переключателей может быть много, то при прикреплении к ним обработчика события нам надо пробежаться по всему массиву переключателей, который можно получить по имени группы:

|  |  |
| --- | --- |
| 1  2  3 | for (var i = 0; i < myForm.languages.length; i++) {      myForm.languages[i].addEventListener("click", onclick);  } |

Значение выбранного переключателя также можно получить через объект Event: e.target.value

Каждый переключатель также, как и флажок, имеет свойство checked, которое возвращает значение true, если переключатель выбран. Например, отметим последний переключатель:

|  |  |
| --- | --- |
| 1 | myForm.languages[myForm.languages.length-1].checked = true; |

## Список select

Для создания списка используется html-элемент select. Причем с его помощью можно создавать как выпадающие списки, так и обычные с ординарным или множественным выбором. Например, стандартный список:

|  |  |
| --- | --- |
|  | <select name="language" size="4">      <option value="JS" selected="selected">JavaScript</option>      <option value="Java">Java</option>      <option value="C#">C#</option>      <option value="C++">C++</option>  </select> |

Атрибут size позволяет установить, сколько элементов будут отображаться одномоментно в списке. Значение size="1" отображает только один элемент списка, а сам список становится выпадающим. Если установить у элемента select атрибут **multiple**, то в списке можно выбрать сразу несколько значений. Каждый элемент списка представлен html-элементом option, у которого есть отображаемая метка и есть значения в виде атрибута value. В JavaScript элементу select соответствует объект **HTMLSelectElement**, а элементу option - объект **HtmlOptionElement** или просто **Option**. Все элементы списка в javascript доступны через коллекцию options. А каждый объект HtmlOptionElement имеет свойства: index(индекс в коллекции options), text (отображаемый текст) и value (значение элемента). Например, получим первый элемент списка и выведем о нем через его свойства всю информацию:

|  |  |
| --- | --- |
|  | <form name="myForm">      <select name="language" size="4">          <option value="JS" selected="selected">JavaScript</option>          <option value="Java">Java</option>          <option value="CS">C#</option>          <option value="CPP">C++</option>      </select>  </form>  <script>  var firstLanguage = myForm.language.options[0];  document.write("Index: " + firstLanguage.index + "<br/>");  document.write("Text: " + firstLanguage.text + "<br/>");  document.write("Value: " + firstLanguage.value + "<br/>");  </script> |

В javascript мы можем не только получать элементы, но и динамически управлять списком. Например, применим добавление и удаление объектов списка:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <form name="myForm">      <select name="language" size="5">          <option value="JS" selected="selected">JavaScript</option>          <option value="Java">Java</option>          <option value="CS">C#</option>          <option value="CPP">C++</option>      </select>      <p><input type="text" name="textInput" placeholder="Введите текст" /></p>      <p><input type="text" name="valueInput" placeholder="Введите значение" /></p>      <p><input type="button" name="addButton" value="Добавить" /><input type="button" name="removeButton" value="Удалить" /></p>  </form>    <script>  var addButton = myForm.addButton,      removeButton = myForm.removeButton,      languagesSelect = myForm.language;  // обработчик добавления элемента  function addOption(){      // получаем текст для элемента      var text = myForm.textInput.value;      // получаем значение для элемента      var value = myForm.valueInput.value;      // создаем новый элемента      var newOption = new Option(text, value);      languagesSelect.options[languagesSelect.options.length]=newOption;  }  // обработчик удаления элемент  function removeOption(){        var selectedIndex = languagesSelect.options.selectedIndex;      // удаляем элемент      languagesSelect.options[selectedIndex] = null;  }    addButton.addEventListener("click", addOption);  removeButton.addEventListener("click", removeOption);  </script>  </body>  </html> |

Для добавления на форме предназначены два текстовых поля (для текстовой метки и значения элемента option) и кнопка. Для удаления выделенного элемента предназначена еще одна кнопка. За добавление в коде javascript отвечает функция addOption, в которой получаем введенные в текстовые поля значения, создаем новый объект Option и добавляем его в массив options объекта списка. За удаление отвечает функция removeOption, в которой просто получаем индекс выделенного элемента с помощью свойства **selectedIndex** и в коллекции options приравниваем по этому индексу значение null. Для добавления/удаления также в качестве альтернативы можно использовать методы элемента select:

|  |  |
| --- | --- |
|  | // вместо вызова  // languagesSelect.options[languagesSelect.options.length]=newOption;  // использовать для добавления вызов метода add  languagesSelect.add(newOption);  // вместо вызова  // languagesSelect.options[selectedIndex] = null;  // использовать для удаления метод remove  languagesSelect.remove(selectedIndex); |

**События элемента select**

Элемент select поддерживает три события: blur (потеря фокуса), focus (получение фокуса) и change (изменение выделенного элемента в списке). Рассмотрим применение события select:

|  |  |
| --- | --- |
|  | <form name="myForm">      <select name="language" size="5">          <option value="JS" selected="selected">JavaScript</option>          <option value="Java">Java</option>          <option value="CS">C#</option>          <option value="CPP">C++</option>      </select>  </form>  <div id="selection"></div>  <script>  var languagesSelect = myForm.language;    function changeOption(){        var selection = document.getElementById("selection");      var selectedOption = languagesSelect.options[languagesSelect.selectedIndex];      selection.textContent = "Вы выбрали: " + selectedOption.text;  }    languagesSelect.addEventListener("change", changeOption);  </script> |

# JSON

JSON (JavaScript Object Notation) это формат хранения данных. JSON описывает структуру и организацию данных JavaScript. Простота JSON привела к тому, что в настоящий момент он является наиболее популярным форматом передачи данных в среде web, вытеснив другой некогда популярный формат xml. Объекты JSON очень похожи на объекты JavaScript, тем более что JSON является подмножеством JavaScript. В то же время важно их различать: JavaScript является языком программирования, а JSON является форматом данных. JSON поддерживает три типа данных: примитивные значения, объекты и массивы. Примитивные значения представляют стандартные строки, числа, значение null, логические значения true и false.

Объекты представляют набор простейших данных, других объектов и массивов. Например, типичный объект JSON:

|  |  |
| --- | --- |
|  | {      "name": "Tom",      "married": true,      "age": 30  } |

В javascript этому объекту соответствовал бы следующий:

|  |  |
| --- | --- |
|  | var user = {      name: "Tom",      married: true,      age: 30  } |

Несмотря на общее сходство, в то же время есть и различия: в JSON названия свойств заключаются в кавычки, как обычные строки. Кроме того, объекты JSON не могут хранить функции, переменные, как объекты javascript. Объекты могут быть сложными:

|  |  |
| --- | --- |
|  | {      "name": "Tom",      "married": true,      "age": 30,      "company": {          "name": "Microsoft",          "address": "USA, Redmond"      }  } |

Массивы в JSON похожи на массивы javascript и также могут хранить простейшие данные или объекты:

|  |  |
| --- | --- |
|  | ["Tom", true, 30] |

Массив объектов:

|  |  |
| --- | --- |
|  | [{      "name": "Tom",      "married": true,      "age": 30  },{      "name": "Alice",      "married": false,      "age": 23  }] |

**Сериализация в JSON и десериализация**

Для сериализации объекта javascript в json применяется функция **JSON.stringify()**:

|  |  |
| --- | --- |
|  | var user = {      name: "Tom",      married: true,      age: 30  };  var serializedUser = JSON.stringify(user);  document.write(serializedUser); // {"name":"Tom","married":true,"age":30} |

Для обратной операции - десериализации или парсинга json-объекта в javascript применяется метод **JSON.parse()**:

|  |  |
| --- | --- |
|  | var user = {      name: "Tom",      married: true,      age: 30  };  // сериализация  var serializedUser = JSON.stringify(user);  // десериализация  var tomUser = JSON.parse(serializedUser);  document.write(tomUser.name); // Tom |

# Хранение данных

## Куки

Одну из возможностей сохранения данных в javascript представляет использование куки. Для работы с куками в объекте **document**предназначено свойство **cookie**. Для установки куков достаточно свойству document.cookie присвоить строку с куками:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>    <script>  document.cookie = "login=tom32;";  </script>  </body>  </html> |

В данном случае устанавливается кука, которая называется "login" и которая имеет значение "tom32". И в большинстве браузеров мы можем посмотреть ее, узнать всю информацию о ней и в дальнейшем ее можно использовать в приложении:

Строка куки принимает до шести различных параметров: имя куки, значение, срок окончания действия (expires), путь (path), домен (domain) и secure. Выше использовались только два параметра: имя куки и значение. То есть в случае со строкой "login=tom32;" куки имеет имя login и значение tom32. Но подобная куки имеет очень ограниченный срок жизни: если явным образом не установить срок действия, то кука будет удалена с закрытием браузера. Подобная ситуация, возможно, идеальна для тех случаев, когда необходимо удалять всю информацию после завершения работы с веб-приложением и закрытия браузера. Однако данное поведение не всегда подходит. И в этом случае нам надо установить параметр **expires**, то есть срок действия куков:

|  |  |
| --- | --- |
| 1 | document.cookie = "login=tom32;expires=Mon, 31 Aug 2015 00:00:00 GMT;"; |

То есть срок действия куки login истекает в понедельник 31 августа 2015 года в 00:00. Формат параметра expires очень важен. Однако его можно сгенерировать программно. Для этого мы можем использовать метод **toUTCString()** объекта Date:

|  |  |
| --- | --- |
|  | var expire = new Date();  expire.setHours(expire.getHours() + 4);  document.cookie = "login=tom32;expires=" + expire.toUTCString() + ";"; |

В данном случае срок действия куки будет составлять 4 часа. Если в друг нам надо установить куки для какого-то определенного пути на сайте, то мы можем использовать параметр **path**. Например, мы хотим установить куки только для пути *www.mysite.com/home*:

|  |  |
| --- | --- |
| 1 | document.cookie = "login=tom32;expires=Mon, 31 Aug 2015 00:00:00 GMT;path=/home;"; |

В этом случае для других путей на сайте, например, *www.mysite.com/shop*, эти куки будут недоступны. Если на нашем сайте есть несколько доменов, и мы хотим установить куки непосредственно для определенного домена, тогда можно использовать параметр **domain**. Например, у нас на сайте есть поддомен *blog.mysite.com*:

|  |  |
| --- | --- |
| 1 | document.cookie = "login=tom32;expires=Mon, 31 Aug 2015 00:00:00 GMT;path=/;domain=blog.mysite.com;"; |

Параметр path=/ указывает, что куки будут доступны для всех директорий и путей поддомена blog.mysite.com. Последний параметр - **secure** задает использование SSL (SecureSockets Layer) и подходит для сайтов, использующих протокол https. Если значение этого параметра равно true, то куки будут использоваться только при установке защищенного соединения ssl. По умолчанию данный параметр равен false.

|  |  |
| --- | --- |
| 1 | document.cookie = "login=tom32;expires=Mon, 31 Aug 2015 00:00:00 GMT;path=/;domain=blog.mysite.com;secure=true;"; |

**Получение куки**

Для простейшего извлечения куки из браузера достаточно обратиться к свойству **document.cookie**:

|  |  |
| --- | --- |
|  | var expire = new Date();  expire.setHours(expire.getHours() + 4);  document.cookie = "city=Berlin;expires="+expire.toUTCString()+";";  document.cookie = "country=Germany;expires="+expire.toUTCString()+";";  document.cookie = "login=tom32;";  document.write(document.cookie); |

Здесь были установлены три куки, и браузер выведет нам все эти куки:

Извлеченные куки не включают параметры expires, path, domain и secure. Кроме того, сами куки разделяются точкой с запятой, поэтому нужно еще провести некоторые преобразования, чтобы получить их имя и значение:

|  |  |
| --- | --- |
|  | var cookies = document.cookie.split(";");  for(var i=0; i<cookies.length; i++){        var parts = cookies[i].split("="),          name = parts[0],          value = parts[1];      document.write("Имя куки: " + name + "<br/>");      document.write("Значение: " + value + "<br/><br/>");  } |

## Web Storage

Хотя куки позволяют сохранять информацию, они имеют ряд ограничений. Например, браузер имеет ограничения на размер куков - каждая кука не может превышать 4 кб. Куки имеют срок действия, после которого удаляются. Куки являются неотъемлемой чертой протокола HTTP и при каждом запросе к серверу передаются вместе с запросом на сервер. Однако для работы с куками на стороне клиента в коде javascript не имеет значения передача куков на сервер. Кроме того, для извлечения сохраненных куков надо написать некоторую порцию кода. Поэтому в HTML5 была внедрена новая концепция для хранения данных - **web storage**. Web storage состоит из двух компонентов:**session storage** и **local storage**. **Session storage** представляет временное хранилище информации, которая удаляется после закрытия браузера. **Local storage** представляет хранилище для данных на постоянной основе. Данные из local storage автоматически не удаляются и не имеют срока действия. Эти данные не передаются на сервер в запросе HTTP. Кроме того, объем local storage составляет в Chrome и Firefox 5 Mб для домена, а в IE - 10 Mб. Все данные в web storage представляют набор пар ключ-значение. То есть каждый объект имеет уникальное имя-ключ и определенное значение. Для работы с local storage в javascript используется объект **localStorage**, а для работы с session storage - объект **sessionStorage**.

Для сохранения данных надо передать в метод **setItem()** объекта localStorage:

|  |  |
| --- | --- |
| 1 | localStorage.setItem("login", "tom32@gmail.com"); |

В этот метод передаются два значения: ключ и значение сохраняемого объекта. Если в localStorage уже есть объект с ключом "login", то его значение заменяется новым. Для получения сохраненных данных надо вызвать метод **getItem()**:

|  |  |
| --- | --- |
| 1 | var login = localStorage.getItem("login"); //tom32@gmail.com |

В этот метод передается ключ объекта. Чтобы удалить объект, применяется метод **removeItem()**, который принимает ключ удаляемого объекта:

|  |  |
| --- | --- |
| 1 | localStorage.removeItem("login"); |

И для полного удаления всех объектов из localStorage можно использовать метод **clear()**:

|  |  |
| --- | --- |
| 1 | localStorage.clear(); |

С сохранением простых объектов все просто, однако при этом надо учитывать, что данные в localStorage сохраняются в виде строки:

|  |  |
| --- | --- |
| 1  2  3  4 | localStorage.setItem("age", 23);  var age = localStorage.getItem("age");  age=parseInt(age)+10;  document.write(age); //33 |

Если в данном случае не преобразовать значение к числу с помощью parseInt(), то age будет действовать как строка. Трудности могут возникнуть с сохранением сложных объектов:

|  |  |
| --- | --- |
|  | var user ={      name: "Tom",      age: 23,      married: false  };    localStorage.setItem("user", user);  var savedUser = localStorage.getItem("user");  document.write(savedUser); //[object Object]  document.write(savedUser.name); // undefined - savedUser - строка, а не объект |

В этом случае нам надо использовать сериализацию в формат JSON:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var user ={      name: "Tom",      age: 23,      married: false  };    localStorage.setItem("user", JSON.stringify(user));  var savedUser = JSON.parse(localStorage.getItem("user"));  document.write(savedUser.name + " " + savedUser.age +" " + savedUser.married); // Tom 23 false |

И в завершении надо сказать, что в некоторых браузерах с помощью специальных инструментов мы можем увидеть сохраненные объекты в local storage. Например, в Google Chrome:

# Коллекции и итераторы

## Итераторы

Итераторы применяются для организации последовательного доступа к элементам коллекции - массивам, объектам Set и Map. Итераторы предоставляют метод **next()**, который возвращает два значения: value и done. value хранит собственно значение текущего перебираемого элемента. А свойство done указывает, есть ли еще в коллекции объекты, доступные для перебора. Некоторые методы коллекций возвращают итераторы. Например, метод **entries()**, который есть у коллекций Array, Set, Map:

|  |  |
| --- | --- |
| 1  2  3 | let users = ["Tom", "Bob", "Sam"];  let items = users.entries();  console.log(items.next()); |

Метод next возвратит следующий объект на консоль:

{value: Array(2), done: false}

done: false

value: Array(2)

0: 0

1: "Tom"

length: 2

\_\_proto\_\_: Array(0)

\_\_proto\_\_: Object

Здесь мы видим, что свойство done имеет значение false, так как мы перебрали только один элемент в множестве, и там еще есть два элемента. Свойство value представляет массив из двух значений. Первое значение представляет ключ или индекс элемента массива, а второй элемент - значение по этому индексу. имеет ключ и значение. Соответственно мы можем организовать и перебор всей коллекции:

|  |  |
| --- | --- |
|  | let users = ["Tom", "Bob", "Sam"];  let items = users.entries();  var result = items.next();    while(result.done === false){      console.log(result.value[0], result.value[1]);      result = items.next();  } |

Консольный вывод:

0 "Tom"

1 "Bob"

2 "Sam"

Но в этом нет смысла, поскольку все коллекции, возвращающие итераторы, поддерживают перебор с помощью цикла **for...of**, который как раз и использует итератор для получения элементов:

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

Если мы хотим извлечь еще и индекс элемента в массиве, то мы можем использовать для перебора итератор из entries():

|  |  |
| --- | --- |
|  | let users = ["Tom", "Bob", "Sam"];  for(let user of users.entries()){      console.log(user[0], user[1]);  } |

Консольный вывод:

0 "Tom"

1 "Bob"

2 "Sam"

## Генераторы

Генераторы используются для генерации значений. Для определения генераторов применяется символ звездочки \*, который ставится после ключевого слова function.

|  |  |
| --- | --- |
|  | function\* getNumber(){      yield 5;  }  let numberGenerator = getNumber();  let next = numberGenerator.next();  console.log(next); |

Функция генератора возвращает итератор. Для получения значения из генератора применяется оператор **yield**. То есть фактически в данном случае генератор генерирует число 5. Далее с помощью вызова этой функции создается объект итератора в виде переменной numberGenerator. Используя этот объект, мы можем получать из генератора значения. Для перехода к следующему значению применяется метод **next()**. Если мы посмотрим на консольный вывод, то мы увидим, что данный метод возвращает следующие данные:

{value: 5, done: false}

То есть по сути возвращается объект, свойство **value** которого содержит собственно сгенерированное значение. А свойство **done**указывает, достигли ли мы конца генератора.

Теперь изменим код:

|  |  |
| --- | --- |
|  | function\* getNumber(){      yield 5;  }  let numberGenerator = getNumber();  let next = numberGenerator.next();  console.log(next);  next = numberGenerator.next();  console.log(next); |

Здесь обращение к методу next() происходит два раза:

{value: 5, done: false}

{value: undefined, done: true}

Но функция генератора getNumber генерирует только одно значение - число 5. Поэтому при повторном вызове свойство value будет иметь значение undefined, а свойство done - true, то есть работа генератора завершена. Генератор может создавать множество значений:

|  |  |
| --- | --- |
|  | function\* getNumber(){      yield 5;      yield 25;      yield 125;  }  let numberGenerator = getNumber();  console.log(numberGenerator.next());  console.log(numberGenerator.next());  console.log(numberGenerator.next());  console.log(numberGenerator.next()); |

Консольный вывод:

{value: 5, done: false}

{value: 25, done: false}

{value: 125, done: false}

{value: undefined, done: true}

То есть при первом вызове метода next() из итератора извлекается значение, которое идет после первого оператора yield, при втором вызове метода next() - значение после второго оператора yield и так далее. Поскольку для получения значений применяется итератор, то мы можем использовать цикл **for...of**:

|  |  |
| --- | --- |
|  | function\* getNumber(){      yield 5;      yield 25;      yield 125;  }  let numberGenerator = getNumber();    for(let num of numberGenerator){      console.log(num);  } |

Консольный вывод:

5

25

125

Генератор необязательно содержит только определение операторов yield. Он также может содержать более сложную логику.

С помощью генераторов удобно создавать бесконечные последовательности:

|  |  |
| --- | --- |
|  | function\* points(){        let x = 0;      let y = 0;      while(true){          yield {x:x, y:y};          x += 2;          y += 1;      }  }  let pointGenerator = points();    console.log(pointGenerator.next().value);  console.log(pointGenerator.next().value);  console.log(pointGenerator.next().value); |

Консольный вывод:

{x: 0, y: 0}

{x: 2, y: 1}

{x: 4, y: 2}

**Передача данных в генератор**

С помощью next() можно передать в генератор данные.

|  |  |
| --- | --- |
|  | function\* getNumber(){      let n = yield 5;      console.log("n:", n);      let m = yield 25 \* n;      console.log("m:", m);      yield 125 \* m;  }  let numberGenerator = getNumber();    console.log(numberGenerator.next().value);  console.log(numberGenerator.next(2).value);  console.log(numberGenerator.next(3).value); |

Консольный вывод:

5

n: 2

50

m: 3

375

При втором вызове метода next():

|  |  |
| --- | --- |
| 1 | numberGenerator.next(2).value |

Мы можем получить переданные через него данные, присвоив результат **первого** оператора yield:

|  |  |
| --- | --- |
| 1 | let n = yield 5; |

То есть здесь переменная n будет равна 2, так как в метод next() передается число 2.

Далее мы можем использовать это значение, например, для генерации нового значения:

|  |  |
| --- | --- |
| 1 | let m = yield 25 \* n; |

Соответственно, переменная m получить значение, переданное через третий вызов метода next(), то есть число 3.

**Инициализация генератора**

Есть также другой способ передачи данных в генератор, когда мы передаем некоторые данные в саму функцию генератора, то есть фактически инициализируем генератор некоторыми начальными данными:

|  |  |
| --- | --- |
|  | function\* takeItem(arr){      for(var i=0; i < arr.length; i++){          yield arr[i];      }  }  var users = ["Tom", "Bob", "Sam", "Alice", "Kate", "Ann"];  var userGenerator = takeItem(users);  var timer = setInterval(function(){      var user = userGenerator.next();      if(user.done){          clearInterval(timer);          console.log("The End...");      } else{          console.log(user.value);      }  }, 500); |

В данном случае в генератор передается массив, который используется для генерации значений в таймере.

## Множества Set

Данные с уникальными значениями. В JavaScript функционал множества опредляет объект **Set**. Для создания множества применяется конструктор этого объекта:

|  |  |
| --- | --- |
| 1 | let mySet = new Set(); |

Также можно передать в конструктор массив значений, которыми будет инициализировано множество:

|  |  |
| --- | --- |
| 1  2  3 | let arr = [1, 1, 2, 3, 4, 5, 2, 4];  let numbers = new Set(arr);  console.log(numbers);           // Set(5) {1, 2, 3, 4, 5} |

В данном случае в множество передаются данные из массива. Однако поскольку множество может хранить только уникальные значения, то при его создании повторяющиеся значения, которые есть в массиве, удаляются. Для проверки количества элементов можно использовать свойство **size**.

|  |  |
| --- | --- |
|  | let arr = [1, 1, 2, 3, 4, 5, 2, 4];  let numbers = new Set(arr);  console.log(numbers.size);      // 5 |

**Добавление**

Для добавления применяется метод **add()**. Его результатом является измененное множество:

|  |  |
| --- | --- |
|  | let numbers = new Set();  numbers.add(1);  numbers.add(3);  numbers.add(5);  numbers.add(3);     // не добавляется  numbers.add(1);     // не добавляется  console.log(numbers);       // Set(3) {1, 3, 5} |

При этом, поскольку множество хранит только уникальные значения, то добавление элементов, которые уже в нем есть, не имеет смысла. Так как метод add возвращает ссылку на это же множество, то мы можем вызывать методы по цепочке:

|  |  |
| --- | --- |
|  | let numbers = new Set();  numbers.add(1).add(3).add(5);  console.log(numbers);       // Set(3) {1, 3, 5} |

**Удаление**

Для удаления элементов применяется метод **delete()**:

|  |  |
| --- | --- |
|  | let numbers = new Set();  numbers.add(1).add(3).add(5);  numbers.delete(3);  console.log(numbers);       // Set(2) {1, 5} |

Причем данный метод возвращает булевое значение: true - если элемент удален и false - если удаление не произошло (например, когда удаляемого элемента нет в множестве):

|  |  |
| --- | --- |
|  | let numbers = new Set();  numbers.add(1).add(3).add(5);  let isDeleted = numbers.delete(3);  console.log(isDeleted);         // true  isDeleted = numbers.delete(54);  console.log(isDeleted);         // false |

Если необходимо удалить вообще все элементы из множества, то применяется метод **clear()**:

|  |  |
| --- | --- |
|  | let numbers = new Set();  numbers.add(1).add(3).add(5);  numbers.clear();  console.log(numbers);           // Set(0) {} |

**Проверка наличия элемента**

Если нужно проверить, если ли элемент в множестве, то используется метод **has()**. Если элемент есть, то метод возвращает true, иначе возвращает false:

|  |  |
| --- | --- |
|  | let numbers = new Set();  numbers.add(1).add(3).add(5);  console.log(numbers.has(3));    // true  console.log(numbers.has(32));   // false |

**Перебор множества**

Для перебора элементов множества применяется метод **forEach()**:

|  |  |
| --- | --- |
|  | let arr = [1, 2, 3, 5];  let numbers = new Set(arr);  numbers.forEach(function(value1, value2, set){      console.log(value1);  }) |

Для совместимости с массивами, которые тоже имеют метод forEach, в данный метод передается функция обратного вызова, которая принимает три параметра. Непосредственно для множества первый и второй параметры представляют текущий перебираемый элемент, а третий параметр - перебираемое множество. Также для перебора множества можно использовать цикл for...of:

|  |  |
| --- | --- |
|  | let numbers = new Set([1, 2, 3, 5]);  for(n of numbers){      console.log(n);  } |

## Map

Map или карта (отображение, словарь) представляет структуру данных, где каждый элемент имеет ключ и значение. Ключи в рамках карты являются уникальными, то есть с одним ключом может быть сопоставлен только один элемент. Для создания карты применяется конструктор объекта **Map**:

|  |  |
| --- | --- |
|  | let myMap = new Map(); |

Также можно инициализировать карту начальными значениями. Для этого в конструктор передается массив, элементы которого представляют массивы из двух элементов - первый элемент будет выступать в качестве ключа, а второй - в качестве значения:

|  |  |
| --- | --- |
|  | let myMap = new Map([[1, "a"], [2, "b"], [3, "c"]]);  console.log(myMap);     // Map(3) {1 => "a", 2 => "b", 3 => "c"} |

В данном случае числа 1, 2, 3 являются ключами, а строки "a", "b", "c" - значениями. При этом ключи и значения необзательно должны быть одного типа:

|  |  |
| --- | --- |
|  | let myMap = new Map([["a", 1], [2, "b"], ["c", true]]);  console.log(myMap);     // Map(3) {"a" => 1, 2 => "b", "c" => true} |

**Добавление и изменение элементов**

Для добавления или изменения значения применяется метод **set()**:

|  |  |
| --- | --- |
|  | let myMap = new Map([[1, "a"], [2, "b"], [3, "c"]]);  myMap.set(4, "d");      // добавление элемента  myMap.set(2, "v");      // изменение элемента  console.log(myMap);     // Map(4) {1 => "a", 2 => "v", 3 => "c", 4 => "d"} |

Первый параметр метода set() представляет ключ, а второй параметр - значение элемента. Если по такому ключу нет элементов, то добавляется новый элемент. Если ключ уже есть, то уже имеющийся элемент изменяет свое значение.

**Получение элементов**

Для получения элемента по ключу применяется метод **get()**, в который передается ключ элемента:

|  |  |
| --- | --- |
|  | let myMap = new Map([[1, "Tom"], [2, "Bob"], [3, "Sam"]]);  console.log(myMap.get(2));  // Bob  console.log(myMap.get(7));  // undefined |

Если map не содержит элемента по заданному ключу, то метод возвращает undefined. Чтобы избежать возвращения undefined мы можем проверить наличие элемента по ключу с помощью метода **has()**. Если элемент по ключу имеется, то метод возвращает true, иначе возвращается false:

|  |  |
| --- | --- |
|  | let myMap = new Map([[1, "Tom"], [2, "Bob"], [3, "Sam"]]);  console.log(myMap.has(2));  // true  console.log(myMap.has(7));  // false |

**Удаление элементов**

Для удаления одного элемента по ключу применяется метод **delete()**:

|  |  |
| --- | --- |
|  | let myMap = new Map([[1, "Tom"], [2, "Bob"], [3, "Sam"]]);  myMap.delete(2);  console.log(myMap);     // Map(2) {1 => "Tom", 3 => "Sam"} |

Для удаления всех элементов используется метод **clear()**:

|  |  |
| --- | --- |
|  | let myMap = new Map([[1, "Tom"], [2, "Bob"], [3, "Sam"]]);  myMap.clear();  console.log(myMap);     // Map(0) {} |

**Перебор элементов**

Для перебора элементов используется метод **forEach**:

|  |  |
| --- | --- |
|  | let myMap = new Map([[1, "Tom"], [2, "Bob"], [3, "Sam"]]);  myMap.forEach(function(value1, value2, map){      console.log(value2, value1);  }) |

Метод forEach в качестве параметра получает функцию обратного вызова, которая имеет три параметра. Первый и второй параметры - это соответственно значение и ключ текущего перебираемого элемента, а третий параметр - перебираемый объект Map. Консольный вывод данного примера:

1 Tom

2 Bob

3 Sam

Также для перебора объекта Map можно использовать цикл for...of:

|  |  |
| --- | --- |
|  | let myMap = new Map([[1, "Tom"], [2, "Bob"], [3, "Sam"]]);    for(item of myMap){      console.log(item[0], item[1]);  } |

Каждый элемент из Map помещается в переменную item, которая в свою очередь представляет массив. Первый элемент этого массива - ключ, а второй элемент - значение элемента. Также объект Map имеет два дополнительных метода: **keys()** позволяет перебрать только ключи и **values()** позволяет перебирать значения элементов. Оба метода возвращают итераторы, поэтому для перебоа ключей и значений по отдельности также можно использовать цикл for...of:

|  |  |
| --- | --- |
|  | let myMap = new Map([[1, "Tom"], [2, "Bob"], [3, "Sam"]]);  for(item of myMap.keys()){      console.log(item);  }  for(item of myMap.values()){      console.log(item);  } |

## WeakSet и WeakMap

**WeakSet**

Объект **WeakSet** во многом похож на обычное множество. Он также может хранить только уникальные значения, но каждый его элемент должен представлять объект. Для создания объекта WeakSet используется его конструктор, в который можно передать начальные значения:

|  |  |
| --- | --- |
|  | // пустой WeakSet  let weakSet1 = new WeakSet();  // инициализация начальными значениями  let weakSet2 = new WeakSet([{name:"Tom"}, {age: 34}]); |

Для инициализации как в случае с объектом Set в конструктор передается массив, но данный массив содержит именно объекты, а не скалярные значения, типа чисел или строк. Для добавления данных в WeakSet применяется метод **add()**:

|  |  |
| --- | --- |
|  | let weakSet = new WeakSet();  weakSet.add({lang: "JavaScript"});  weakSet.add({lang: "TypeScript"});  // weakSet.add(34); // так нельзя - 34 - число, а не объект  console.log(weakSet);   // {{lang: "JavaScript"}, {lang: "TypeScript"}} |

Причем опять же добавить мы можем только объект, а не скалярные значения типа чисел или строк. Для удаления применяется метод **delete()**, в который передается ссылка на удаляемый объект:

|  |  |
| --- | --- |
|  | let weakSet = new WeakSet();  var js = {lang: "JavaScript"};  var ts = {lang: "TypeScript"};  weakSet.add(js);  weakSet.add(ts);  weakSet.delete(js);  console.log(weakSet);   // {{lang: "TypeScript"}} |

Если надо проверить, имеется ли объект в WeakSet, то можно использовать метод **has()**, который возвращает true при наличии объекта:

|  |  |
| --- | --- |
|  | var js = {lang: "JavaScript"};  var ts = {lang: "TypeScript"};  var java = {lang: "Java"};  let weakSet = new WeakSet([js, ts]);  console.log(weakSet.has(ts));       // true  console.log(weakSet.has(java));     //  false |

**WeakMap**

WeakMap представляет развитие коллекции Map. Особенностью WeakMap является то, что все ее элементы должны представлять объекты. При этом объектами должны быть как ключи, так значения. Создание WeakMap:

|  |  |
| --- | --- |
|  | // пустой WeakMap  let weakMap1 = new WeakMap();  // WeakMap с инициализацией данными  var key1 = {key:1};  var key2 = {key:2};  var value1 = {name: "Tom"};  var value2 = {name: "Sam"};  let weakMap2 = new WeakMap([[key1, value1], [key2, value2]]); |

Для получения объектов по ключу из WeakMap применяется метод **get()**:

|  |  |
| --- | --- |
|  | var key1 = {key:1};  var key2 = {key:2};  var value1 = {name: "Tom"};  var value2 = {name: "Sam"};  let weakMap2 = new WeakMap([[key1, value1], [key2, value2]]);  console.log(weakMap2.get(key1));    // {name: "Tom"} |

Для добавления новых объектов или изменения старых применяется метод **set()**:

|  |  |
| --- | --- |
|  | var key1 = {key:1};  var key2 = {key:2};  var value1 = {name: "Tom"};  var value2 = {name: "Sam"};  let weakMap2 = new WeakMap([[key1, value1]]);  weakMap2.set(key2, value2);  weakMap2.set(key1, {name: "Kate"});  console.log(weakMap2.get(key1));    //{name: "Kate"}  console.log(weakMap2.get(key2));    //{name: "Sam"} |

Чтобы проверить наличие элемента по определенному ключу, применяется метод **has()**, который возвращает true при наличии элемента:

|  |  |
| --- | --- |
|  | var key1 = {key:1},      key2 = {key:2};  var value1 = {name: "Tom"},      value2 = {name: "Sam"};  let weakMap2 = new WeakMap([[key1, value1]]);  console.log(weakMap2.has(key1));    // true  console.log(weakMap2.has(key2));    // false |

Для удаления элемента по ключу применяется метод **delete()**:

|  |  |
| --- | --- |
|  | var key1 = {key:1},      key2 = {key:2};  var value1 = {name: "Tom"},      value2 = {name: "Sam"};  let weakMap2 = new WeakMap([[key1, value1], [key2, value2]]);  console.log(weakMap2.has(key1));    // true  weakMap2.delete(key1);  console.log(weakMap2.has(key1));    // false |

# Ajax

Современные веб-приложение, как правило, разделяются на две части: клиент и сервер. Клиент представляет собой веб-страницу с кодом JavaScript. К серверным технологиям относятся PHP, Ruby, Node.js, ASP.NET и т.д., которые получают запрос от клиента, обрабатывают и отправляют в ответ результат обработки. Ajax представляет технологию для отправки запросов к серверу из клиентского кода JavaScript без перезагрузки страницы. Сам термин расшифровывается как Asynchronous JavaScript And XML. То есть изначально AJAX предполагал асинхронное взаимодействие клиента и сервера посредством данных в формате XML. Хотя сейчас XML во многом вытеснил формат JSON. В любом случае AJAX революционизировал веб-среду, позволив создавать динамичные отзывчивые веб-приложения. Поскольку Ajax предполагает взаимодействие клиента и сервера, то для работы с Ajax и в частности этой главы необходим локальный веб-сервер. Это может быть любой веб-сервер: Apache, IIS и т.д.

## Объект XMLHttpRequest

Для создания приложений, использующих Ajax, применяются различные способы. Но самым распространенным способом является использование объекта **XMLHttpRequest**:

|  |  |
| --- | --- |
|  | var request = new XMLHttpRequest(); |

После создания объекта XMLHttpRequest можно отправлять запросы к серверу. Но для начала надо вызвать метод **open()** для инициализации:

|  |  |
| --- | --- |
|  | request.open("GET", "<http://localhost/hello.txt>", false); |

Метод open() принимает три параметра: тип запроса (GET, POST, HEAD, PUT), адрес запроса и третий необязательный параметр - логическое значение true или false, указывающее, будет ли запрос осуществляться в асинхронном режиме. То есть в данном случае запрос будет иметь тип GET, он будет направляться по адресу *"http://localhost/hello.txt* в синхронном режиме, так как стоит значение false (для асинхронного режима указывается значение true). Синхронный и асинхронный режим отличаются тем, что запрос в синхронном режиме пока запрос не выполнится, остальной код javascript не может выполняться. По умолчанию, если третий параметр не используется, то запрос отправляется в асинхронном режиме, что позволяет параллельно с выполнением запроса выполнять также и другой код javascript. И в большинстве случаев, как правило, используется именно асинхронный режим. Кроме того, метод open() может принимать еще два параметра: логин и пароль пользователя, если для выполнения запроса нужна аутентификация.

|  |  |
| --- | --- |
|  | request.open("GET", "<http://localhost/home.php>", true, "login", "password"); |

После инициализации запроса методом open() необходимо отправить запрос с помощью метода **send()**: request.send();

### Свойства XMLHttpRequest

Объект XMLHttpRequest имеет ряд свойств, которые позволяют проконтролировать выполнение запроса:

* **status**: содержит статусный код ответа HTTP, который пришел от сервера. С помощью статусного кода можно судить об успешности запроса или об ошибках, которые могли бы возникнуть при его выполнении. Например, статусный код 200 указывает на то, что запрос прошел успешно. Код 403 говорит о необходимости авторизации для выполнения запроса, а код 404 сообщает, что ресурс не найден и так далее.
* **statusText**: возвращает текст статуса ответа, например, "200 OK"
* **responseType**: возвращает тип ответа. Есть следующие типы:
  + ""
  + "arraybuffer"
  + "blob"
  + "document"
  + "json"
  + "text"
* **response**: возвращает ответ сервера
* **responseText**: возвращает текст ответа сервера
* **responseXML**: возвращает xml, если ответ от сервера в формате xml

Например, выполним запрос к текстовому файлу, который находится на локальном веб-сервере. Для выполнения ajax-запросов потребуется запущенный локальный веб-сервер, на котором будет лежать файл hello.txt, в котором будет содержаться одна строка: "Привет мир". Код веб-страницы (пусть она называется *test.html*) будет следующим:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>    <script>  var request = new XMLHttpRequest();  request.open("GET", "<http://localhost:8080/hello.txt>", false);  request.send();  var status = request.status;  if(status==200)      document.write("Текст ответа: " + request.responseText)  else if(status==404)      document.write("Ресурс не найден")  else      document.write(request.statusText)  </script>  </body>  </html> |

И после загрузки страницы выполнится ajax-запрос к ресурсу *http://localhost:8080/hello.txt*. Но важно отметить, что получение статуса сразу после вызова метода request.send() будет работать только для синхронного запроса.

### Асинхронные запросы

Хотя синхронные запросы вполне работают и их можно использовать, но в то же время их рекомендуется избегать. Поскольку нередко запрос может занять продолжительное время, то это может заблокировать выполнение остального кода и работу с html-страницей до окончания выполнения запроса. Поэтому рекомендуется использовать преимущественно асинхронные запросы. Работа с асинхронными запросами чуть более сложна, чем с синхронными, поскольку нам надо еще обработать событие **readystatechange** объекта XMLHttpRequest. При асинхронном запросе объект XMLHttpRequest использует свойство **readyState** для хранения состояния запроса. Состояние запроса представляет собой число:

* 0: объект XMLHttpRequest создан, но метод open() еще не был вызван для инициализации объекта
* 1: метод open() был вызван, но запрос еще не был отправлен методом send()
* 2: запрос был отправлен, заголовки и статус ответа получены и готовы к использованию
* 3: ответ получен от сервера
* 4: выполнение запроса полностью завершено (даже если получен код ошибки, например, 404)

Событие **readystatechange** возникает каждый раз, когда изменяется значение свойства **readyState**. Например, выполним асинхронный запрос:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <script>  var request = new XMLHttpRequest();  function reqReadyStateChange() {      if (request.readyState == 4) {          var status = request.status;          if (status == 200) {              document.write(request.responseText);          } else {              document.write("Ответ сервера " + request.statusText);          }      }  }  request.open("GET", "<http://localhost:8080/hello.txt>");  request.onreadystatechange = reqReadyStateChange;  request.send();  </script>  </body>  </html> |

Кроме обработки события readystatechange для получения ответа сервера можно также обрабатывать событие **load**, которое возникает после выполнения запроса. Его использование аналогично:

|  |  |
| --- | --- |
|  | var request = new XMLHttpRequest();   function responceLoad() {      if (request.readyState == 4) {          var status = request.status;          if (status == 200) {              document.write(request.responseText);          } else {              document.write("Ответ сервера " + request.statusText);          }      }  }   request.open("GET", "<http://localhost:8080/hello.txt>");  request.onload = responceLoad;  request.send(); |

## Отправка данных

Принцип отправки данных может отличаться в различных ситуациях. Рассмотрим эти ситуации.

**Отправка GET-запроса**

GET-запрос характеризуется тем, что данные могут отправляться в строке запроса:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <div id="output"></div>  <script>  // объект для отправки  var user = {      name: "Tom",      age: 23  };    var request = new XMLHttpRequest();  function reqReadyStateChange() {      if (request.readyState == 4) {          var status = request.status;          if (status == 200) {              document.getElementById("output").innerHTML=request.responseText;          }      }  }  // строка с параметрами для отправки  var body = "name=" + user.name + "&age="+user.age;  request.open("GET", "<http://localhost:8080/postdata.php?>"+body);  request.onreadystatechange = reqReadyStateChange;  request.send();  </script>  </body>  </html> |

Для отправки берем свойства объекта user и формируем из их значений сроку с параметрами: "name=" + user.name + "&age="+user.age. Затем эта строка добавляется к строке запроса в методе open("GET", "http://localhost:8080/postdata.php?"+body)

Предполагается, что данные отправляются скрипту на языке php postdata.php, который может иметь, например, следующее содержание:

|  |  |
| --- | --- |
|  | <?php  $name = "Не известно";  $age = "Не известно";  if(isset($\_GET['name'])) $name = $\_GET['name'];  if (isset($\_GET['age'])) $age = $\_GET['age'];  echo "Ваше имя: $name  <br> Ваш возраст: $age";  ?> |

Конкретная технология стороны сервера тут не важна. И в качестве тестирования можно взять любую другую технологию. Например, ASP.NET MVC. Метод контроллера в ASP.NET MVC, который принимает данные, мог бы выглядеть следующим образом:

|  |  |
| --- | --- |
|  | public string PostData(string name, int age)  {      return "Ваше имя: "+name +"; Ваш возраст: "+ age;  } |

**Кодирование параметров**

Все отправляемые в GET-запросе параметры разделяются знаком амперсанда(&). Но что, если какой-нибудь параметр имеет знак амперсанда. Например,

|  |  |
| --- | --- |
|  | var user = {      name: "Tom&Tim",      age: 23  };  // строка с параметрами для отправки  var body = "name=" + user.name + "&age="+user.age; |

В этом случае при получении параметров скрипт на стороне сервера может неправильно обработать данные и неправильно извлечь параметры. Поэтому, чтобы кодировать все передаваемые данные, нужно применять функцию **encodeURIComponent()**:

|  |  |
| --- | --- |
| 1 | var body = "name=" + encodeURIComponent(user.name) + "&age="+encodeURIComponent(user.age); |

При этом строка "Tom&Tim" будет кодирована в следующую строку: "Tom%26Tim". При необходимости мы можем выполнить обратное декодирование с помощью функции **decodeURIComponent()**:

|  |  |
| --- | --- |
| 1  2 | var encodeName = encodeURIComponent(user.name); // Tom%26Tim  var decodeName = decodeURIComponent(encodeName); // Tom&Tim |

**POST-запросы**

Отправка данных в POST-запросах будет немного отличаться:

|  |  |
| --- | --- |
|  | var user = {      name: "Tom",      age: 23  };    var request = new XMLHttpRequest();  function reqReadyStateChange() {      if (request.readyState == 4 && request.status == 200)          document.getElementById("output").innerHTML=request.responseText;  }  var body = "name=" + user.name + "&age="+user.age;  request.open("POST", "<http://localhost:8080/postdata.php>");  request.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');  request.onreadystatechange = reqReadyStateChange;  request.send(body); |

Для отправки данных методом POST надо установить заголовок **Content-Type** с помощью метода **setRequestHeader()**. В данном случае заголовок имеет значение application/x-www-form-urlencoded.

**Отправка форм. FormData**

Начиная со спецификации XMLHttpRequest2 в JavaScript появился новый объект - **FormData**, который позволяет сериализовать данные формы для ее последующей отправки. При этом нам даже необязательно создавать форму в коде html, мы можем создать ее динамически в JavaScript:

|  |  |
| --- | --- |
|  | var formData = new FormData();  formData.append('name', 'Tom');  formData.append('age', 23);  var request = new XMLHttpRequest();  function reqReadyStateChange() {      if (request.readyState == 4 && request.status == 200)          document.getElementById("output").innerHTML=request.responseText;  }  request.open("POST", "<http://localhost:8080/display.php>");  request.onreadystatechange = reqReadyStateChange;  request.send(formData); |

Для добавления данных у объекта FormData используется метод **append('имя\_параметра', значение)**. При этом никакие заголовки указывать не надо. Также мы можем определить форму в html и использовать ее для отправки:

|  |  |
| --- | --- |
|  | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />  </head>  <body>  <div id="output"></div>  <form name="user" action="<http://localhost:8080/postdata.php>">  <input type="text" name="username" placeholder="Введите имя" /><br/>  <input type="text" name="age" placeholder="Введите возраст" /><br/>  <input type="submit" name="submit" value="Отправить" />  </form>  <script>  // получаем объект формы  var form = document.forms.user;  // прикрепляем обработчик кнопки  form.submit.addEventListener("click", sendRequest);    // обработчик нажатия  function sendRequest(event){      event.preventDefault();      var formData = new FormData(form);       var request = new XMLHttpRequest();      request.open("POST", form.action);      request.onreadystatechange = function () {          if (request.readyState == 4 && request.status == 200)              document.getElementById("output").innerHTML=request.responseText;      }      request.send(formData);  }  </script>  </body>  </html> |

Для сериализации всех полей формы нам достаточно передать объект формы в конструктор FormData: var formData = new FormData(form);

**Отправка данных в формате json**

Для отправки данных в формате json нам необходимо установить соответствующий заголовок и сериализовать данные с помощью метода JSON.stringify:

|  |  |
| --- | --- |
|  | // объект для отправки  var user = {      username: "Tom",      age: 23  };  var json = JSON.stringify(user);  var request = new XMLHttpRequest();  request.open("POST", "<http://localhost:8080/postjson.php>");  request.setRequestHeader('Content-type', 'application/json; charset=utf-8');  request.onreadystatechange = function () {          if (request.readyState == 4 && request.status == 200)              document.getElementById("output").innerHTML=request.responseText;  }  request.send(json); |

## Promise в Ajax-запросах

Как видно из примеров прошлых тем для создания ajax-запросов используются фактически повторяющиеся вызовы, отличающиеся лишь деталями - строкой запроса, функциями обработки ответа. И вполне было бы не плохо создать для всех действий, связанных с асинхронным ajax-запросом, создать какую-то общую абстракцию и затем использовать ее при следующих обращениях к серверу. Для создания дополнительного уровня абстракции в данном случае удобно применять объект **Promise**, который обертывает асинхронную операцию в один объект, который позволяет определить действия, выполняющиеся при успешном или неудачном выполнении этой операции. Инкапсулируем асинхронный запрос в объект Promise:

|  |  |
| --- | --- |
|  | function get(url) {    return new Promise(function(succeed, fail) {      var request = new XMLHttpRequest();      request.open("GET", url, true);      request.addEventListener("load", function() {        if (request.status < 400)          succeed(request.response);        else          fail(new Error("Request failed: " + request.statusText));      });      request.addEventListener("error", function() {        fail(new Error("Network error"));      });      request.send();    });  } |

Метод get получает в качестве параметра адрес ресурса сервера и возвращает объект Promise. Конструктор Promise в качестве параметра принимает функцию обратного вызова, которая в свою очередь принимает два параметра - две функции: одна выполняется при успешной обработке запроса, а вторая - при неудачной. Допустим, на сервере будет размещен файл **users.json** со следующим содержимым:

|  |  |
| --- | --- |
|  | [      {          "name": "Tom",          "age": 34      }, {          "name": "Sam",          "age": 32      }, {          "name": "Bob",          "age": 26      }  ] |

Теперь вызовем метод get для осуществления запроса к серверу:

|  |  |
| --- | --- |
|  | get("<http://localhost:8080/users.json>").then(function(text) {    console.log(text);  }, function(error) {    console.log("Error!!!");    console.log(error);  }); |

Для обработки результата объекта Promise вызывается метод **then()**, который принимает два параметра: функцию, вызываемую при успешном выполнении запроса, и функцию, которая вызывается при неудачном выполнении запроса. Метод then() также возвращает объект Promise. Поэтому при необходимости мы можем применить к его результату цепочки вызовов метода then: get().then().then().... Например:

|  |  |
| --- | --- |
|  | get("<http://localhost:8080/users.json>").then(function(response) {      console.log(response);      return JSON.parse(response);  }).then(function(data) {      console.log(data[0]);  }); |

В данном случае функция в первом вызове метода then получает ответ сервера и возвращает распарсенные данные в виде массива с помощью функции JSON.parse(). Функция во втором вызове then получает эти распарсенные данные, то есть массив, в виде параметра (возвращаемое значение предудыщего then является параметром для последующего then). Затем первый элемент массива выводится на консоль. Для обработки ошибок мы мы можем использовать метод **catch()**, в который передается функция обработки ошибок:

|  |  |
| --- | --- |
|  | get("<http://localhost:8080/users.jsn>").then(function(response) {      console.log(response);      return JSON.parse(response);  }).then(function(data) {      console.log(data[0]);  }).catch(function(error){      console.log("Error!!!");      console.log(error);  }); |

Подобным образом через Promise можно было бы отправлять данные на сервер:

|  |  |
| --- | --- |
|  | function post(url, requestuestBody) {    return new Promise(function(succeed, fail) {      var request = new XMLHttpRequest();      request.open("POST", url, true);      request.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');      request.addEventListener("load", function() {        if (request.status < 400)          succeed(request.responseText);        else          fail(new Error("Request failed: " + request.statusText));      });      request.addEventListener("error", function() {        fail(new Error("Network error"));      });      request.send(requestuestBody);    });  }  var user = {      name: "Tom&Tim",      age: 23  };  // данные для отправки  var params = "name=" + user.name + "&age="+user.age;  post("<http://localhost:8080/postdata.php>", params).then(function(text) {    console.log(text);  }, function(error) {    console.log(error);  }); |

# https://metanit.com/web/javascript/2.2.php