

Frontend & Backend разработка

Лекция 4

Основы языка программирования JavaScript

Р.В. Шамин

профессор кафедры индустриального программирования

Подключаем JavaScript

JavaScript - полноценный язык программирования и программы на этом языке могут выполняться вне зависимости от HTML5, но для наших задач мы будем использовать JavaScript внутри HTML5 документа.

Создадим три файла index.html, styles.css и script.js.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <link rel="stylesheet" href="styles.css">
  <script src="script.js"></script>
</head>
<body>

</body>
</html>
```

index.html

```
const a = 7;
const b = 8;

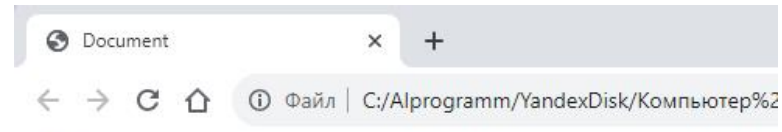
const c = a * b;
```

```
document.write("<div>Hello, World!</div>");
document.write("<div>");
document.write("Результат = " + c.toString());
document.write("</div>");
```

script.js

```
body {
  font-family: Verdana, Tahoma, sans-serif;
}
```

styles.css



Hello, World!
Результат = 56

Константы и переменные

В современном JavaScript переменные перед использованием должны быть объявлены. Хотя язык JS является типизированным и каждая переменная и константа имеют фиксированный тип данных, при объявлении переменной тип не указывается. Тип переменной (константы) определяется при инициализации этой переменной.

В стародавние времена переменные можно было объявлять с помощью оператора “var”. Сейчас следует использовать оператор “let”. Там, где это возможно используйте константы, которые создаются с помощью конструкции “const”.

Изменять значение константы в дальнейшем нельзя, но это верно только для примитивных типов данных. Для объектов (в том числе и массивов) значения полей (элементов) можно менять. Поэтому для таких типов данных желательно использовать константы.

Типы данных в JS:

- **String** - строковый тип.
- **Number** - числовой тип (как целочисленные значения, так и дробные).
- **BigInt** - числовой тип (целочисленный) для очень больших значений.
- **Boolean** - логический тип (false или true).
- **Undefined** - специальный тип, который может иметь только одно значение undefined, которое означает, что значение не установлено.
- **Null** - специальный тип, который может иметь только одно значение null, которое означает отсутствие значения.
- **Object** - тип объекта (об этом поговорим позже).
- **Symbol** - тип для уникальных значений.

```
let a = "Мама мыла раму"; // String
let b = 1024; // Number
let c = 3.1415; // Number
let d = 123456789n; // BigInt
let e = true; // Boolean
let f; // Undefined
let g = null; // Null
let h = Symbol(); // Symbol
```

Операции

В JavaScript существуют все основные операции, которые есть в С-подобных языках программирования. Мы остановимся на наиболее отличительных операциях.

Операторы сравнения `==` и `===`. Разница в том, что оператор `==` сравнивает значения, а оператор `===` еще и типы.

```
let a = 5;
let b = "5";
let res;
if (a == b) {
  res = "Равно";
}
else {
  res = "Неравно";
}
```

```
let a = 5;
let b = "5";
let res;
if (a === b) {
  res = "Равно";
}
else {
  res = "Неравно";
}
```

```
let a = 5;
let b = 5n;
let res;
if (a == b) {
  res = "Равно";
}
else {
  res = "Неравно";
}
```

```
let a = 5;
let b = 5n;
let res;
if (a === b) {
  res = "Равно";
}
else {
  res = "Неравно";
}
```

Оператор `??` позволяет проверить значение на `null` и `undefined`. В выражении «`c = a ?? b`» переменная `c` будет иметь значение переменной `a`, если `a` не равно `null` или `undefined`. В противном случае переменной `c` будет присвоено значение переменной `b`.

```
const a = 7;
const b = "плохое значение";
const c = a ?? b;
```

↑
Будет равно 7

```
let a;
const b = "плохое значение";
const c = a ?? b;
```

↑
Будет равно «плохое значение»

Управляющие конструкции: условные операторы и циклы

В JavaScript поддерживает C-подобные и Python-подобные управляющие конструкции:

- условные: if, if-else, switch-case;
- циклы: for, for-in, for-of, while, do-while.

```
let msg = "Допустимая скорость";
let v = 80;
if (v > 60) {
  msg = "Превышение скорости!";
}
```

```
let value;
if(!value){
  console.log("Значение переменной не определено!");
}
```

```
const e = 2.71, pi = 3.14;
if (pi > e){
  console.log("Пи больше e");
} else {
  console.log("Пи не больше e");
}
```

```
let line = "";
if(!line){
  console.log("Пустая строка");
}
```

```
let rnd = "ножницы";
switch(rnd) {
  case "камень":
    console.log("камень");
    break;
  case "ножницы":
    console.log("ножницы");
    break;
  case "бумага":
    console.log("бумага");
    break;
  default:
    console.log("что-то не то");
}
```

```
const N = 100;
for (let i = 0; i < N; i++) {
  if (i == 6) {
    continue;
  }
  if (i == 13) {
    break;
  }
  console.log(i);
}
```

```
let n = 0;
while(n < 10) {
  console.log(n);
  n++;
}
```

```
let j = 0;
const S = "Forward";
do {
  console.log(S[j]);
  j++;
} while (S[j] != 'w')
```

Цикл for-in используется для перебора свойств объекта, а for-of для перебора данных (например, в массиве).

```
let s = "";
const Item = {id: 1, name: "вещь", cost: 37};
for (prop in Item) {
  s += prop + ": " + Item[prop] + "<br/>";
}
```

```
document.write("<div>");
document.write(s);
document.write("</div>");
```

id: 1
name: вещь
cost: 37

```
const line = "Mama";
let s = "";
for (c of line) {
  s += c + "<br/>";
}
```

```
document.write("<div>");
document.write(s);
document.write("</div>");
```

M
a
m
a

Управляющие конструкции: функции

Язык JavaScript поддерживает много конструкций для создания и использования функций.

Функция может принимать параметры, а также возвращать значения.

```
function mean(x, y) {  
  let res;  
  res = (x + y) / 2.0;  
  return res;  
}
```

← типы параметров не указываем
← можно создавать локальные переменные
← возвращаем значение

```
function fact(n) {  
  if (n == 1) {  
    return 1;  
  }  
  else {  
    return n * fact(n - 1);  
  }  
}
```

← возможна рекурсия

Переменная может иметь тип функции.

```
function sum(x, y) {  
  return x + y;  
}
```

```
function mul(x, y) {  
  return x * y;  
}
```

```
let doing;  
doing = sum;  
console.log(doinḡ(7, 8)); // 15  
doing = mul;  
console.log(doinḡ(7, 8)); // 56
```

← это функция sum
← а теперь это функция mul

```
let func = function (x) {  
  return x**3;  
}
```

← анонимная функция

```
console.log(func(3)); // 27
```

Параметры функций

Функции в языке JavaScript имеют достаточно гибкий механизм параметров.

Если какие-либо параметры не были переданы, то в функции они будут иметь значение "undefined". Можно указать значения по умолчанию.

```
function power(x, y) {  
  if (x === undefined) {  
    x = 1;  
  }  
  if (y === undefined) {  
    y = 2;  
  }  
  return x**y;  
}  
console.log(power(2, 10)); // 1024  
console.log(power(6)); // 36  
console.log(power()); // 1
```

```
function power(x = 1, y = 2) {  
  return x**y;  
}  
console.log(power(2, 10)); // 1024  
console.log(power(6)); // 36  
console.log(power()); // 1
```

Неопределенное количество параметров. В теле функции доступен массив arguments, который содержит все параметры. Также можно воспользоваться оператором "...".

```
function mean() {  
  let sum = 0;  
  for(const x of arguments) {  
    sum += x;  
  }  
  return sum / arguments.length;  
}  
console.log(mean(1, 2, 3, 4)); // 2.5
```

```
function meanp(p, ...val) {  
  let sum = 0;  
  for(const x of val) {  
    sum += x**p;  
  }  
  return sum;  
}  
console.log(meanp(3, 1, 2, 3, 4)); // 100
```


Стрелочные функции

В JavaScript активно используются т.н. стрелочные функции, которые вообще не имеют имени.

```
const f = (a, b) => a**b;  
console.log(f(2, 10)); // 1024
```

Стрелочная функция может иметь полноценное тело функции и возвращать значения.

```
const fact = (n) => {  
  if (n <= 1) {  
    return 1;  
  }  
  let res = 1;  
  for (let i = 0; i < n; i++) {  
    res *= (n - i);  
  }  
  return res;  
}
```

Стрелочные функции используются при функциональном преобразовании массивов. О массивах мы поговорим чуть дальше, но сейчас приведем пример использования стрелочной функции.

```
const A = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
const S = A.map(n => n * n);  
console.log(S); // [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```



Стрелочная функция используется для преобразования массива


Объекты

Язык JavaScript поддерживает концепцию объектно-ориентированного программирования лишь частично. Однако в веб-программировании объекты играют принципиально важную роль.

Создадим объекты разными способами.

```
const Item = {}; // создаем пустой объект
Item.id = 2; // добавляем свойства
Item.Name = "Вещь"; // добавляем свойства
Item.Cost = 37; // добавляем свойства
```

```
console.log(Item);
```



```
▼ {id: 2, Name: 'Вещь', Cost: 37} ⓘ
  Cost: 37
  Name: "Вещь"
  id: 2
  ► [[Prototype]]: Object
```

```
const Item = {
  id: 2,
  Name: "Вещь",
  Cost: 37
};
```

```
console.log(Item);
```



Методы объекта создаются как переменные-функции. При этом методы имеют доступ к полям (свойствам) объекта доступ через ключевое слово this.

```
const Item = {
  id: 2,
  Name: "Вещь",
  Cost: 37,
  Print: function() {
    return this.id.toString() + "|" + this.Name.toString() + "|" + this.Cost.toString();
  }
};
document.write("<div>");
document.write(Item.Print());
document.write("</div>");
```



```
→ 2|Вещь|37
```

Объекты

На объекты можно смотреть как на массивы и осуществлять доступ к полям и методам объекта.

```
document.write("<div>");  
document.write(Item["Name"]); —————> Вещь  
document.write("</div>");
```

```
document.write("<div>");  
document.write(Item["Print"]()); —————> 2|Вещь|37  
document.write("</div>");
```

Можно обойти все поля и методы объекта.

```
for(const a in Item) {  
  console.log(Item[a]);  
}
```

```
2  
Вещь  
37  
f () {  
    return this.id.toString() + "/" + this.Name.toString() + "/" +  
    this.Cost.toString();  
}
```

Ператается исходный текст метода

Объекты: копирование и сравнение

В отличие от примитивных типов таких, как Number или String, переменные и константы типа Object хранят лишь адрес (ссылку) на область памяти, где расположены свойства, поэтому их нельзя так просто скопировать.

```
const Item = {
  id: 2,
  Name: "Вещь",
  Cost: 37,
  Print: function() {
    return this.id.toString() + "|" + this.Name.toString() + "|" + this.Cost.toString();
  }
};

let Item2 = Item; // копируем!

// меняем данные в "копии"
Item2.id = 3;
Item2.Name = "Бириюлька";
Item2.Cost = 17;

document.write("<div>");
document.write("Item = " + Item["Print"]()); // данные в исходном объекте также изменились!
document.write("<br/>Item2 = " + Item2["Print"]());
document.write("</div>");
```



Item = 3|Бириюлька|17
Item2 = 3|Бириюлька|17

Объекты: копирование и сравнение

Чтобы скопировать объект (называется глубокое копирование) можно использовать Метод `Object.assign()`

```
const Item = {
  id: 2,
  Name: "Вещь",
  Cost: 37,
  Print: function() {
    return this.id.toString() + "|" + this.Name.toString() + "|" + this.Cost.toString();
  }
};
```

```
let Item2 = {};
Object.assign(Item2, Item); // копируем!
```

```
// меняем данные в "копии"
```

```
Item2.id = 3;
Item2.Name = "Бирюлька";
Item2.Cost = 17;
```

```
document.write("<div>");
document.write("Item = " + Item["Print"]());
document.write("<br/>Item2 = " + Item2["Print"]());
document.write("</div>");
```

```
Item = 2|Вещь|37
Item2 = 3|Бирюлька|17
```

При сравнении двух объектов - сравниваться будут только ссылки, а не значения.

```
console.log(Item == Item2); // false
console.log(Item === Item2); // false
```

Классы: создание с помощью конструкторов

В объектно-ориентированном программировании конструктор - это метод, который вызывается при создании объекта и инициализирует поля объекта.

```
class TItem {  
    constructor(id, Name) {  
        this.id = id;  
        this.Name = Name;  
        this.Print = function () {  
            return this.id.toString() + "|" + this.Name.toString();  
        };  
    }  
}
```

```
const Item1 = new TItem(1, "Коробка");  
const Item2 = new TItem(2, "Корзинка");
```

```
document.write("<div>");  
document.write("Item1 = " + Item1["Print"]());  
document.write("<br/>Item2 = " + Item2["Print"]());  
document.write("</div>");
```

→ Item1 = 1|Коробка
Item2 = 2|Корзинка

Поля и методы объекта можно добавлять следующим образом.

```
Item1.Tag = 10;  
Item1.PrintTag = function () {  
    return "Tag = " + this.Tag.toString();  
}
```

```
console.log(Item1.PrintTag()); // Tag = 10
```

Приватные поля и методы

Концепция инкапсуляции объектно-ориентированного программирования подразумевает наличие частных полей и методов, к которым можно обратиться только из методов самого объекта. В JavaScript частные поля и методы должны иметь имена, начинающиеся с символа #.

```
class TItem {
  #id;
  constructor(id, Name) {
    this.#id = id;
    this.Name = Name;
    this.Print = function () {
      return this.#id.toString() + "|" + this.Name.toString();
    };
  }
}

const Item = new TItem(1, "Коробка");

Item.#id = -1;
```

Поле #id объявлено как частное

Из мет

```
class TItem {
  #id;
  constructor(id, Name) {
    this.#id = id;
    this.#correct(1)
    this.Name = Name;
    this.Print = function () {
      return this.#id.toString() + "|" + this.Name.toString();
    };
  }
  #correct(min) {
    if (this.#id < min) {
      this.#id = min;
    }
  }
}
```


Приватные поля и методы

Концепция инкапсуляции объектно-ориентированного программирования подразумевает наличие частных полей и методов, к которым можно обратиться только из методов самого объекта. В JavaScript частные поля и методы должны иметь имена, начинающиеся с символа #.

```
class TItem {  
  #id;  
  constructor(id, Name) {  
    this.#id = id;  
    this.#correct(1) ← Можно обратиться к частному методу  
    this.Name = Name;  
    this.Print = function () {  
      return this.#id.toString() + "|" + this.Name.toString();  
    };  
  }  
  #correct(min) { ← Метод объявлен как частный  
    if (this.#id < min) {  
      this.#id = min;  
    }  
  }  
}
```

```
const Item = new TItem(-1, "Коробка");  
document.write("<div>");  
document.write(Item.Print()); → 1|Коробка  
document.write("</div>");
```

← Значение было скорректировано

Статические поля

Класс - это определенный тип данных, а объекты - это переменные типа класса. При этом у каждого объекта свои поля, которые не зависят друг от друга. Статическими полями называются поля, которые имеют одно значения для всех объектов класса. Такие поля объявляются с помощью ключевого слова `static`.

```
class TItem {  
    static Count = 0;  
    constructor(Name) {  
        TItem.Count++;  
        this.id = TItem.Count;  
        this.Name = Name;  
        this.Print = function () {  
            return this.id.toString() + "|" + this.Name.toString();  
        };  
    }  
}
```

Это будет статическое поле, которое принадлежит не объектам, а классу

Обращаемся к статическим полям через имя класса, а не `this`!

```
const Item1 = new TItem("Коробка");  
const Item2 = new TItem("Корзинка");  
const Item3 = new TItem("Картонка");
```

```
document.write("<div>");  
document.write("<b>Количество экземпляров класса = " + TItem.Count.toString() + "</b>");  
document.write("<br/>" + Item1.Print());  
document.write("<br/>" + Item2.Print());  
document.write("<br/>" + Item3.Print());  
document.write("</div>");
```

Количество экземпляров класса = 3

1|Коробка
2|Корзинка
3|Картонка

Поля доступа («сеттеры» и «геттеры»)

Современная концепция объектно-ориентированного программирования включает в себя поля доступа, которые внешне ведут себя как поля, но при доступе к ним вызываются методы класса.

```
class TPerson {  
    #AgeValue; ← Скрытое поле  
    constructor (Name, Age) {  
        this.Name = Name;  
        this.Age = Age;  
    }  
    set Age(value) {  
        if (value < 0) {  
            this.#AgeValue = 1;  
        } else {  
            this.#AgeValue = value;  
        }  
    }  
    get Age() {  
        if (this.#AgeValue > 18) {  
            return "Взрослый";  
        } else {  
            return "Ребенок";  
        }  
    }  
}  
  
const Ivan = new TPerson("Иван", -1);  
  
document.write("<div>");  
document.write(Ivan.Name + " - " + Ivan.Age); → Иван - Ребенок  
document.write("</div>");
```

Наследование

Современный JavaScript поддерживает механизм наследования классов. При создании класса мы можем унаследовать все поля и методы уже существующего другого класса.

```
class TPoint {  
  constructor (x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  Draw() {  
    return "(" + this.x + ", " + this.y + ")";  
  }  
}
```

```
A = new TPoint(2, 3);
```

```
document.write("<div>");  
document.write(A.Draw());  
document.write("</div>");
```

→ (2, 3)

Этот метод переопределяет существующий метод

```
class TCircle extends TPoint {  
  constructor (x, y, R) {  
    super(x, y);  
    this.R = R;  
  }  
  Draw() {  
    let s = super.Draw();  
    s += "; R = " + this.R;  
    return s;  
  }  
  isInCircle(x, y) {  
    const r = Math.sqrt((this.x - x)**2 + (this.y - y)**2);  
    if (r < this.R) {  
      return true;  
    } else {  
      return false;  
    }  
  }  
}
```

```
C = new TCircle (5, 7, 10);
```

```
document.write("<div>");  
document.write(C.Draw());  
document.write("</div>");  
document.write("<div>");  
document.write(C.isInCircle(3, 4).toString());  
document.write("</div>");
```

→ (5, 7); R = 10
true

Массивы

Массивы в JavaScript являются классами, которые имеют развитые методы для работы с ними. Разумеется, в JavaScript массивы являются динамическими и нет необходимости задумываться о памяти, в которой они размещены.

```
const List = []; // Создание массива. Можно создавать: const List = new Array();
```

```
List[2] = "Мама";
```

```
List[3] = "Папа";
```

```
List[5] = "Я";
```

```
List[7] = 512;
```

```
let S = "";
```

```
let i = 0;
```

```
for (const el of List) {  
    S += i.toString() + ": " + el + "<br/>";  
    i++;  
}
```

```
document.write("<div>");
```

```
document.write(S);
```

```
document.write("</div>");
```

0: undefined

1: undefined

2: Мама

3: Папа

4: undefined

5: Я

6: undefined

7: 512

Обратите внимание: элементы можно вставлять с произвольным индексом, при этом пропущенные элементы будут иметь тип Undefined.

Как и в других языках программирования высокого уровня. Массивы JavaScript могут иметь элементы разного типа.

Массивы

Часто в JavaScript массивы инициализируются при создании.

```
const Items = ["Меркурий", "Венера", "Марс"];
```

Массивы имеют свойство length, с помощью которого можно узнать количество элементов в массиве. При этом пропущенные (неинициализированные) элементы тоже считаются.

```
const Items = [];  
Items[1] = 0;  
Items[2] = 1;  
Items[4] = 2;  
Items[8] = 3;  
  
console.log(Items.length); // 9
```

Операция spread позволяет разложить массив на элементы.

```
console.log(...Items); // undefined 0 1 undefined 2 undefined undefined undefined 3
```



С помощью этой операции можно объединять массивы.



```
const A = ["A", "a"];  
const C = ["C", "c"];  
const ABC = [...A, "B", "b", ...C];  
console.log(ABC); // ['A', 'a', 'B', 'b', 'C', 'c']
```

Операции с массивами

Массивы в JavaScript имеют богатый набор методов для работы с ними.

Копирование может быть поверхностным, когда копируется лишь ссылка на массив и глубоким, когда копируются и все элементы массива.

```
let A = [2, 4, 6];  
let B = A;  Поверхностное копирование  
console.log(B); // [2, 4, 6]  
B[1] = -3;  
console.log(A); // [2, -3, 6] 
```

```
let A = [2, 4, 6];  
let B = A.slice();  Глубокое копирование  
B[1] = -3;  
console.log(A); // [2, 4, 6]   
console.log(B); // [2, -3, 6]
```

Метод slice(begin, end) позволяет копировать не полностью массив, а часть массива, при этом параметр begin указывает индекс, начиная с которого будет копирование, а end - окончание, не включая элемент с индексом end.

```
let A = [1, 2, 3, 4, 5, 6];  
let B = A.slice(2, 4);  
console.log(B); // [3, 4]
```

Метод push() добавляет в конец массива элемент, а pop() - возвращает и удаляет последний элемент.
Метод unshift() добавляет в начало массива элемент, а метод shift() - возвращает и удаляет начальный элемент.

```
const A = [1, 2, 3, 4, 5, 6];  
console.log(A.pop()); // 6  
A.push("Конец");  
console.log(A.shift()); // 1  
A.unshift("Начало");  
console.log(A); // ['Начало', 2, 3, 4, 5, 'Конец']
```


Сортировка и поиск в массивах

Массивы в JavaScript имеют методы `sort()` и `toSorted()` - первый сортирует сам массив, меняя его, а второй - возвращает отсортированный массив. Каким образом JavaScript сортирует массивы? По умолчанию - как строки.

```
const A = ["Понедельник", "Вторник", "Среда", "Четверг", "Пятница", "Суббота", "Воскресенье"];
A.sort();
console.log(A); // ['Воскресенье', 'Вторник', 'Понедельник', 'Пятница', 'Среда', 'Суббота', 'Четверг']
const X = [-10, 3, 256, 1024, 77];
console.log(X.toSorted()); // [-10, 1024, 256, 3, 77]
```

Это странно!

Метод `toSorted()` позволяет определить функцию сравнения, которая будет использоваться при сортировке массива.

```
const X = [-10, 3, 256, 1024, 77];
console.log(X.toSorted((x, y) => x - y)); // [-10, 3, 77, 256, 1024]
```

Это стрелочная функция

Если первый параметр меньше второго, возвращаем отрицательное число,
Если первый параметр больше второго, возвращаем положительное число.

Методы `indexOf()` и `lastIndexOf()` возвращают индекс первого и последнего вхождения элемента в массиве.

```
const A = ["Ваня", "Боря", "Миша", "Саша", "Миша", "Гриша"];
console.log(A.indexOf("Миша")); // 2
console.log(A.lastIndexOf("Миша")); // 4
```

Для проверки наличия в массиве какого-либо элемента используется метод `includes()`.

```
const A = ["Ваня", "Боря", "Миша", "Саша", "Миша", "Гриша"];
console.log(A.includes("Саша")); // true
console.log(A.includes("Петя")); // false
```