ES6 (JavaScript)

Оглавление:

Транспиляция кода с помощью Babel

1. Создайте файл **package.json**

```
npm init -y
```

2. Установите **babel**

```
npm install --save-dev babel-cli babel-core babel-preset-es2015
```

3. Установите и запуск простого сервера **http-server**

```
npm install -g http-server
```

запуск сервера (чтобы остановить сервер нажмите Ctrl+C)

```
http-server
```

4. Выполнить транспиляцию кода

```
babel src -d dist –presets es2015
```

5. Упрощенная сборка билда (если правило прописано в package.json)

```
“script”: {
     “build”: “babel src -d dist –presets es2015”
     “watch”: “babel src -d dist –presets es2015 -w”
}
```

команда в консоли вызывает транспиляцию

```
npm run build
```

команда в консоли вызывает постоянную транспиляцию

```
npm run watch
```

Все скрипты находящиеся в папке **src** будут транспилироваться в файлы под теми же названиями в папку **dist**

Структура проекта:

[project]
|---- **[dist]**
     |---- **index.html**
     |---- **...**
|---- **[src]**
     |---- **index.js**
     |---- **...**
|---- **package.json**

Описание файла **package.json**

```
{
  "name": "es6",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "build": "babel src -d dist --presets es2015",
    "watch": "babel src -d dist --presets es2015 -w"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "babel-cli": "^6.26.0",
    "babel-core": "^6.26.0",
    "babel-preset-es2015": "^6.24.1"
  },
  "dependencies": {
    "http-server": "^0.11.1"
  }
}
```

Описание файла **index.html** (папка **dist**)

```
<!DOCTYPE html>
<html>
<head>
	<meta charset="utf-8">
	<title>Test ES6</title>
</head>
<body>
	<button></button><button></button><button></button><button></button>
	<script type="text/javascript" src="./index.js"></script>
</body>
</html>
```

Описание файла **index.js** (папка **src**)

```
let message = 'Hello World!';

if (true) {
	let version = 'ES6';
	console.log(version, message);
}

var buttons = document.querySelectorAll('button');

for (let i = 0; i < buttons.length; i++){
	var button = buttons[i];
	button.innerText = i;
	button.onclick = function(e){
		console.log(i);
	}
}
```

Блочные привязки

```
let value = "Hello";
let count = 10;
const MAX = 100;
const NAME = 'name';

const = person = {
      name: "John"
};
person.name = "Greg";

for (let i = 0; i < 10; i++){
      console.log(i);
}

for(let key in Obj){ }
for(const key in Obj){ }
```

Строки

```
let text = "英a";
text.codePointAt(0);            // 134071
String.fromCodePoint(134071);   // 英
text.normalize();

let msg = "Hello World!!!";
msg.startsWith("Hello");        // true (присутствует в начале)
msg.endsWith("!");              // true (присутствует в конце)
msg.includes("o");              // true (присутствует в тексте)

console.log("xxx".repeat(3));   // xxxxxxxxx (повторится три раза)
```

Литералы шаблотов

```
// Многострочный текст
let message = `Строка 1
Строка2
Строка3
`;

let name = "John",
    msg = `Hello, ${name}`;           // Подстановка значения
```

Теги шаблонов

```
function passthre(literals, ...substitutions) { }

let count = 10;
let price = 0.25;
let msg = passthre`${count} items cost ${(count * price).toFixed(2)}`;
```

Функции

```
function make(url, time=2000, callback) { }
make("/foo", undefined, function(){

});
```

Остаточные параметры (используется троеточие **...**)

```
function pick(obj, ...keys){
      let result = Object.create(null);
      for(let i = 0, len = keys.length; i < len; i++){
            result[key[i]] = obj[key[i]];
      }
      return result;
}
```

arguments — остаточные параметры ES5 и ES6

```
function checkArgs(...args){
      console.log('ES5', arguments.length, arguments[0], arguments[1]);
      console.log('ES6', args.length, args[0], args[1]);
}
checkArgs("a", "b");
```

Конструктор **Function**

```
let add = new Function("first", "second=first", "return first+second");
add(1, 1);   // 2
add(1);      // 2

let pick = new Function("...args", "return args[0]");
pick(1, 2);  // 2
```

Оператор расширения (**...**)

```
let values = [25, 50, 75, 100];
console.log('ES5', Math.max.apply(Math, values));
console.log('ES6', Math.max(...values));
```

Свойство функции **name**

```
function doSomething(){}
let doAnotherThing = function(){}
let doSomething2 = function doAnotherThing2(){}

console.log(doSomething.name);         // doSomething
console.log(doAnotherThing.name);      // doAnotherThing
console.log(doSomething2.name);        // doAnotherThing2
console.log(doSomething.bind().name);  // bound doSomething
console.log((new Function()).name);    // anonymous
```

Создание функций с помощью **new** и без него

- new – создает экземпляр функции и выполняет его.
- без new – просто выполняет функцию как статичную.

Сбособ определить каким образом была создана функция, через new или без него

```
function MyFunc(){}

if(this instanceof MyFunc){}                  // ES5
if(typeof new.target !== "undefined") {}      // ES6
if(typeof new.target === MyFunc) {}           // ES6
```

Функции уровня блока

функция поднимается вверх перед условием

```
"use strict"
if(true){
      function pick(){}
      pick();
}
```

Функция не поднимается

```
"use strict"
if(true){
      let pick = function(){}
      pick();
}
```

Стрелочные функции (=>)

- отсутствуют привязки this, super, arguments и new.target
- не могут вызываться с помощью new
- отсутствует prototype
- нельзя изменить this
- отсутствует объект arguments
- не поддерживают повторяющиеся имена пераметров

```
let reflect = velue => value;

let sum = (n1, n2) => n1 + n2;

let getName = () => "John";

let sum = (n1, n2) => {
      return n1 + n2;
};

let myfunc = () => { };
```

```
let reflect = function(value){
      return value;
}

let sum = function(n1, n2){
      return n1 + n2;
}

let getName = function(){
      return "John";
}

let sum = function(n1, n2){
      return n1 + n2;
}

let myfunc = function(){ }
```

Возврат литерала объекта

```
let getTemp = id => ({
	id: id,
	name: "Temp"
});
```

```
let getTemp = function(id){
	return {
		id: id,
		name: "Temp"
	}
}
```

Создание выражений немедленно вызываемых функций

```
let person = ( (name)=>{
	return {
		getName: () => {
			return name;
		}
	}
} )("John");
person.getName();
```

```
let person = function(name){
	return{
		getName: function(){
			return name;
		}
	}
}("John");
person.getName();
```

Без использования bind (this)

```
let Page = {
		id: "12345",
		init: function(){
			document.addEventListener("click",
					event => this.onClick(event.type), false);
		},
		onClick: function(type){
			console.log(type);
		}
};
```

Стрелочные функции для массивов

```
let values = [25, 50, 75, 100];

let res = values.sort( (a,b)=> a-b );
```

```
let values = [25, 50, 75, 100];
let res = values.sort(function(a, b){
	return a - b;
});
```

Стрелочные функции и arguments

```
function test(){
	return () => arguments[0];
}
let tfunc = test(5);
console.log(tfunc);	//5
```

Идентификациястрелочной функции

```
let comparator = (a, b) => a – b;

console.log( typeof comparator);			// "function"
console.log( comparator instanceof Function);	// true
```

Объекты

Сокращенный синтаксис инициализации свойств

```
function createpers(name, age){
	return {
		name,
		age
	};
}
```

```
function createpers(name, age){
	return {
		name: name,
		age: age
	};
}
```

Вычисляемые именя свойств

```
let lastName = "last name";
let person = {
		"first name": "John",
		[lastName]: "Smith"
};
console.log(person["first name"], person[lastName]);
```

Новые методы

- Object.is() - сравнение двух значений. Object.is(5, “5”);
- Object.assign() - используется взамен функции mixin() принимает объекты-поставщиков и присваивает объекту приемнику.

Дубликаты свойств в литералных объектах

```
"use strict"
let person = {
		name: "John",
		name: "Paul"		// корректно в строгом режиме ES6
};
console.log(person.name);		// Paul
```

Порядок перечисления собвтвенных свойств (Reflect.ownkeys)

```
let obj = { a:1, 0:1, c:1, 2:1, b:1, 1:1 };
obj.z = 1;
console.log(Object.getOwnPropertyNames(obj).join(""));	//012acbz
```

Расширения в прототипах. Смена прототипа объекта.

```
let person = {
		getGreeting(){
			return "Hello";
		}
};

let dog = {
		getGreeting(){
			return "Woof";
		}
};
```

```
let friend = Object.create(person);
Object.setPrototypeOf(friend, dog);                    // смена прототипа
if(Object.getPrototypeOf(friend) === dog) {
      console.log("OK!");
}
```

Простой доступ к прототипу с помощью ссылки **super**

return Object.getPrototypeOf(this).getGreeting.call(this);

проще можно записать так

return super.getGreeting() + “, hi!”;

```
let person = {
            getGreeting(){
                  return "Hello";
            }
};
let friend = {
            getGreeting(){
                  return super.getGreeting() + ", hi!";
            }
};
Object.setPrototypeOf(friend, person);

let relative = Object.create(friend);
console.log(relative.getGreeting());                   // Hello, hi!
```

Деструктуризация

Деструктуризация объектов

```
let node = {
      type: "0001",
      name: "bar"
};

let {tyme, name} = node;
```

```
let node = {
      type: "0001",
      name: "bar"
};
let type = node.type;
let name = node.name;
```

Присвоение с диструктуризацией

```
let type = "literal",
      name = "bar";
({type, name} = node);

function outputInfo(value){}
outputInfo({type, name} = node);
```

Значение по умолчанию

```
let {type, name, value = true} = node;
```

Присваивание локальным переменным с другими именами

```
let {type: localType, name: localName = "foo"} = node;
```

Деструктуризация вложенных объектов

```
let node = {
      type: "literal",
      name: "foo",
      loc: {
            start: { line:1, col:1 },
            start: { line:1, col:2 },
      }
};

let { loc: {start} } = node;
console.log(start.line);
console.log(start.col);
```

Деструктуризация массивов

```
let colors = ["red", "green", "blue"];
let [firstColor, secondColor] = colors;
let [, , thridColor] = colors;
```

Присвоение с деструктуризацией массива

```
let colors = ["red", "green", "blue"],
      firstColor = "black",
      secondColor = "white";
[firstColor, secondColor] = colors;
```

Обмен значениями

```
let a = 1, b = 2;
[a, b] = [b, a];                                    // a=2, b=1
```

Значение по умолчанию

```
let colors = ["red"];
let [firstColor, secondColor = "green"] = colors;
```

Деструктуризация вложенных массивов

```
let colors = ["red", ["green", "blue"], "black"];
let [firstColor, [secondColor]] = colors;           // red, green
```

Остаточные элементы (...)

```
let colors = ["red", "green", "blue"];
let [firstColor, ...restColors] = colors;
console.log(firstColor);                            // red
console.log(restColors.length);                     // 2
console.log(restColors[0]);                         // green
console.log(restColors[1]);                         // blue
```

Деструктурированные параметры

```
function setCookie(options){
	options = options || {};
	let {name, value} = options;
}
setCookie({name: "site", value: 1000});
```

Явное указание ожидаемых параметров

```
function setCookie({name, value}){ }
```

Значение по умолчанию

```
function setCookie({type = 1, flag = true} = {}){ }
```

Символы. Символьные свойства.

Создание символов

```
let firstName = Symbol("first name");
let person = {};
person[firstName] = "Paul";
```

Использование символов. Object.defineProperty() и Object.defineProperties()

```
let firstName = Symbol("first name");
let person = {
	[firstName]: "Paul"
};
Object.defineProperty(person, firstName, {writable: false});

let lastName = Symbol("last name");
Object.defineProperties(person, {
	[lastName]: {
		value: "Zakas",
		writable: false
	}
});
```

Совместное использование символов. Symbol.for()

```
let uid = Symbol.for("uid");
let object = {
	[uid]: "12345"
};

let uid2 = Symbol.for("uid");

console.log(uid2 === uid);                                  // true
console.log(object[uid2]);                                  // 12345
```

Приведение типов для символов

```
let uid = Symbol.for("uid");
let str = String(uid);
console.log(str);                                    // Symbol(uid)
```

Извлечение символьных свойств. Object.keys() и Object.getOwnPropertySymbols()

```
let uid = Symbol.for("uid");
let object = {
      [uid]: "12345"
};
let symbol = Object.getOwnPropertySymbols(object);
console.log(symbol[0]);                              // Symbol(uid)
console.log(Object[symbol[0]]);                      // 12345
```

Экспортирование внутренних операций в виде стандартных символов

- Symbol.hasInstance – метод используется оператором instance
- Symbol.isConcatSpreadable – указывает на метод Array.prototype.concat()
- Symbol.iterator – метод возвращает итератор
- Symbol.match – метод String.prototype.match()
- Symbol.replace – метод String.prototype.replace()
- Symbol.search – метод String.prototype.search()
- Symbol.species – метод конструктор для создания производных классов
- Symbol.split – метод String.prototype.split()
- Symbol.toPrimitive – возвращает элементарное представление объекта
- Symbol.toStringTag – метод String.prototype.toString()
- Symbol.unscopables – инструкция with

Множества. Ассоциативные массивы.

Имитация множества и ассоциативных массивов в ES5

Множество

```
var set = Object.create(null);
set.foo = true;
set.bar = false;
```

Ассоциативные массивы

```
var map = Objcet.create(null);
map[5] = "foo";

var map = {};
map['name'] = "Bill";
```

Множества в ES6.
`Set()` - Создание множества и добавление элементов. Методы **`add`**, **`size`**, **`has`**, **`delete`**.

```
let set = new Set();
set.add(5);
set.add("5");
console.log(set.size);                                  // 2

let set = new Set([1,2,3,4,5,5,5,5]);
console.log(set.size);                                  // 5 (1,2,3,4,5)
console.log(set.has(5));                                // true

set.delete(5);                                          // удаление элемента
```

Метод **`forEach()`** для множеств

```
let set = new Set([1,2]);
set.forEach(function(value, key, ownerSet){
      console.log(key, value);
      console.log(ownerSet === set);
});
```

Передача **`this`** в метод **`forEach()`** для функции обратного вызова

```
let set = new Set([1,2]);
let processor = {
            output(value){
                  console.log(value);
            },
            process(dataset){
                  dataset.forEach(function(value, key, ownerSet){
                        this.output(value);
                  }, this);
            }
};
processor.process(set);
```

Преобразование множества в массив

```
let set = new Set([1,2,3,4,5]);
let array = [...set];
```

Множество со слабыми ссылками (только объекты)

```
let set = new Set([1,2,3,4,5]);
let array = [...set];

let set = new WeakSet();
let key = {};
set.add(key);
console.log(set.has(key));
set.delete(key);
key = null;                           //ссылка будет удалена из множества
```

Ассоциативные массивы в ES6
`Map()` - Создание ассоциативного массива. Методы **`set()`** и **`get()`**.

```
let map = new Map();
map.set("title", "This is test");
map.set("year", 2018);

console.log(map.get("title"), map.get("year"));
```

Инициализация ассоциативных массивов

```
let map = new Map([ ["name", "Bill"], ["age", 25] ]);
```

Метод **`forEach()`** для перебора ассоциативного массива

```
map.forEach(function(value, key, ownerMap){
	console.log(key, value);
	console.log(ownerMap === Map);
});
```

Ассоциативные массивы со слабыми ссылками

```
let map = new WeakMap();
let element = document.querySelector(".element");

map.set(element, "Original");
console.log(map.get(element));

element.parentNode.removeChild(element);
element = null;                                    // элемент удален из массива
console.log(map.has(element));                     // false
```

Инициализация ассоциативного массива со слабыми ссылками

```
let key1 = {},
    key2 = {},
    map = new WeakMap([ [key1, "Hello"], [key2, 42] ]);
```

Приватные данные объекта. (ассоциативный массив со слабыми ссылками)

```
let Person = (function() {
	let privateData = new WeakMap();

	function Person(name){
		privateData.set(this, {name: name});
	}

	Person.prototype.getName = function(){
		return privateData.get(this).name;
	};

	return Person;
}());
```

Итераторы и генераторы.

- **Итераторы** — это объекты со специальным интерфейсом, спроектированным для итераций.
- **Геренатор** — это функция, возвращающая итератор. Эти функции отмечаются знаком звезды * после слова function и используют ключевое слово **yield**.

Итераторы. Обработка в циклах.

```
let xmen = ['Cyclops', 'Wolverine', 'Rogue'];
for(let i = 0; i < xmen.length; i++){
    console.log('Цикл FOR', i, xmen[i]);
}
xmen.forEach(element => {
    console.log('Цикл FOREACH', element);
});
for(let key in xmen){
    console.log('Цикл FOR...IN', key ,xmen[key]);
}
for(let value of xmen){
    console.log('Цикл FOR...OF', value);
}

console.log('Итераторы (Symbol.iterator)', typeof xmen[Symbol.iterator]);
console.log('Итераторы (Symbol.iterator)', xmen[Symbol.iterator]());
```

Итераторы.

```
let iterator = xmen[Symbol.iterator]();
console.log('Итераторы (Next)', iterator.next());

let next = iterator.next();
while(!next.done){
    console.log('Итераторы (Next)', next.value);
    next = iterator.next();
}

// Последовательность чисел от 1 до 100
let idGenerator = {
    [Symbol.iterator](){
        let id = 1;
        return {
            next(){
                let value = id > 100 ? undefined: id++;
                let done = !value;
                return { value, done };
            }
        }
    }
};

for(let id of idGenerator){
    console.log('Итераторы в объекте ID', id);
}
```

```
// Генератор десяти случайных чисел
let randomGenerator = {
    generate(){
        return this[Symbol.iterator]();
    },

    [Symbol.iterator](){
        let count = 0;
        return {
            next(){
                let value = Math.ceil(Math.random() * 100);
                let done = count > 10;
                count++;
                return { value, done };
            }
        }
    }
};
for(let random of randomGenerator){
    console.log('Итераторы в объекте RANDOM', random);
}

//let random = randomGenerator.[Symbol.iterator]();
let random = randomGenerator.generate();
console.log('Итераторы в объекте (Symbol.iterator)', random.next().value);

// Класс
class TaskList {
    constructor(){
        this.tasks = [];
    }

    addTasks(...tasks){
        this.tasks = this.tasks.concat(tasks);
    }

    [Symbol.iterator](){
        let tasks = this.tasks;
        let index = 0;
        return {
            next() {
                let result = {value: undefined, done: true};
                if(index < tasks.length){
                    result.value = tasks[index];
                    result.done = false;
                    index++;
                }
                return result;
            }
        }
    }
}

let taskList = new TaskList();
taskList.addTasks('Задача №1', 'Задача №2', 'Задача №3')

for (let task of taskList){
    console.log('Итераторы в объекте (Class)', task);
}
```

Генераторы.

```
function *generate() {
    console.log('Генератор:', 'Start');
    yield 1;
    yield 2;
    yield 3;
    console.log('Генератор:', 'Finish');
}

let iteratorGenerator = generate();
console.log('Генератор', iteratorGenerator.next());  // Start, 1
console.log('Генератор', iteratorGenerator.next());  // 2
console.log('Генератор', iteratorGenerator.next());  // 3
console.log('Генератор', iteratorGenerator.next());  // Finish

// создание итератора
function generate2() {
    let currunt = 1;
    console.log('Генератор:', 'Start');
    return {
        [Symbol.iterator]() {
            return {
                next() {
                    let result = { value: undefined, done: true };
                    if (currunt <= 3) {
                        result.value = this.current;
                        result.done = false;
                        currunt++;
                    } else {
                        console.log('Генератор:', 'Finish');
                    }
                    return result;
                }
            }
        }
    }
}

let iteratorGenerator2 = generate2()[Symbol.iterator]();
console.log('Генератор', iteratorGenerator2.next());  // Start, 1
console.log('Генератор', iteratorGenerator2.next());  // 2
console.log('Генератор', iteratorGenerator2.next());  // 3
console.log('Генератор', iteratorGenerator2.next());  // Finish


let numbersGenerator = {
    *range(start, end) {
        let current = start;
        while (current <= end) {
            yield current++;
        }
    }
};

for (let num of numbersGenerator.range(1, 10)) {
    console.log('Генератор', num);
}
```

```
// делегирование генератора
function *dGenerator(){
    try {
        yield 42;
        yield* [1, 2, 3];
        yield 43;
    } catch (error) {
        console.error(error);
    }

}

let dIterator = dGenerator();
console.log('Генератор (делегирование)', dIterator.next().value);
console.log('Генератор (делегирование)', dIterator.next().value);
console.log('Генератор (делегирование)', dIterator.next().value);
console.log('Генератор (делегирование)', dIterator.next().value);
console.log('Генератор (делегирование)', dIterator.next().value);
// остановить генератор
console.log('Генератор (делегирование)', dIterator.return());
console.log('Генератор (делегирование)',
                                dIterator.throw(new Error('This is error')));
```

Классы.

Структура класса в ES5

```
function MyClass(name){
      this.name = name;
}

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

var mclass = new MyClass("John");
console.log(mclass.getName());                          // John
```

Объявление класса в ES6

```
class MyClass {
      constructor(name){
            this.name = name;
      }

      getname(){
            return this.name;
      }
}

let mclass = new MyClass("Bill");
console.log(mclass.getName());                          // Bill
```

Именованные классы-выражения

```
let mclass = class MyClass {
	// ...
}
console.log(typeof mclass);                    // function
console.log(typeof MyClass);                   // undefined
```

Свойства с методами доступа (**getter** и **setter**)

```
class MyClass {
	constructor(element){
		this.element = element;
	}

	get html(){
		return this.element.innerHTML;
	}

	set html(value){
		this.element.innerHTML = value;
	}
}
```

Вычисляемые имена членов

```
let methodName = "getName";
class MyClass {
	constructor(element){
		this.element = element;
	}
	[methodName](){
		return this.name;
	}
}

let mclass = new MyClass("Paul");
console.log(mclass.getName());                 // Paul
```

Методы-генераторы

```
class MyClass {
	*createIterator(){
		yield 1;
		yield 2;
		yield 3;
	}
}

let mclass = new MyClass();
let iterator =
mclass.createIterator();
```

```
class Collection {
	constructor(){
		this.items = [];
	}
	*[Symbol.iterator](){
		yield *this.items.values();
	}
}

let collection = new Collection();
collection.items.push(1);
```

Статические члены

```
class MyClass {
	constructor(element){
		this.element = element;
	}
	static create(name){
		return new MyClass(name);
	}
}

let mclass = MyClass.create("John");
```

Наследование и производный класс.

Наследование в ES5

```
// Класс Rectangle
function Rectangle(height, width){
	this.height = height;
	this.width = width;
}
Rectangle.prototype.getArea = function(){
	return this.height * this.width;
}

//Класс Square
function Square(height){
	Rectangle.call(this, height, height);
}
Square.prototype = Object.create(Rectangle.prototype, {
	constructor: {
		value: Square,
		enumerable: true,
		writable: true,
		configurable: true
	}
});

var square = new Square(3);

console.log(square.getArea());                  // 9
console.log(square instanceof Square);          // true
console.log(square instanceof Rectangle);       // true
```

Наследование в ES6

```
class Rectangle {                                       // Класс Rectangle
      constructor(height, width){
            this.height = height;
            this.width = width;
      }

      getArea(){
            return this.height * this.width;
      }
}

class Square extends Rectangle {                        //Класс Square
      constructor(height){
            super(height);
      }
}

let square = new Square(3);
console.log(square.getArea());                          // 9
console.log(square instanceof Square);                  // true
console.log(square instanceof Rectangle);               // true
```

Наследование встроенных объектов

```
class MyArray extends Array {
      // пусто
}

let colors = new MyArray();
colors[0] = "red";
console.log(colors.length);
```

Сравнение классов ES5 и ES6.

Класс ES5	Класс ES6

`// конструктор`
`function MyClass(title) {`
`  this._title = title;`
`  this._done+ false;`
`  MyClass.count += 1;`
`}`
`// геттер и сеттер`
`Object.defineProperty(MyClass, 'title', {`
`  get: function() {`
`  return this._title;`
`  },`
`  set: function(value) {`
`  this._title = value;`
`  }`
`});`
`// метод класса`
`MyClass.prototype.complete = function() {`
`  this._done = true;`
`}`
`// статический метод`
`MyClass.getDefaultTitle = function() {`
`  return 'Test class';`
`}`
`MyClass.count = 0; // статическое свойство`
	`class MyClass {`
	`// конструктор`
	`  constructor(title) {`
	`  this._title = title;`
	`  this._done = false;`
	`  MyClass.count += 1;`
	`  }`
	`// геттер и сеттер`
	`  get title(){`
	`  return this._title;`
	`  }`
	`  set title(value){`
	`  this._title = value;`
	`  }`
	`// метод класса`
	`  complete() {`
	`  this._done = true;`
	`  }`
	`// статический метод`
	`  static getDefaultTitel(){`
	`  return "Test class";`
	`  }`
	`}`
	`Task.count = 0;  // статическое свойство`

Подкласс ES5	**Подкласс ES6**

`function MySubClass(title, parent) {`
`  MyClass.call(this, title);`
`  this._parent = parent;`
`}`
 	
`MySubClass.prototype =`
`  Object.create(MyClass.prototype);`
`MySubClass.prototype.constructor =`
`  MySubClass;`
 	
`// Проверяем`
`var mc = new MyClass('Основной класс');`
`var msc = new MySubClass('Подкласс', mc);`
`console.log(mc);`
`console.log(msc);`
	`class MySubClass extends MyClass {`
	`  constructor(title, parent) {`
	`  super(title);`
	`  this._parent = parent;`
	`  }`
	`}`
	
	`// Проверяем`
	`let mc = new MyClass('Основной класс');`
	`let msc = new MySubClass('Подкласс', mc);`
	`console.log(mc);`
	`console.log(msc);`

Расширенные возможности массивов.

Метод **`Array.of()`** - создает массив, содержащий аргументы метода независимо от их количества или типов.

```
let items = Array.of(1, 2);
console.log(items.length);                              // 2
console.log(items[0]);                                  // 1

items = Array.of("3");
console.log(items[0]);                                  // 3
```

Метод **`Array.from()`** - преобразование объектов в массивы

```
function translate() {
      return Array.from(arguments, (value) => value + 1);
}

let numbers = translate(1,2,3);                         // 2,3,4
```

Метод **`find()`** и **`findIndex()`** - поиск элементов

```
let numbers = [25, 30, 35, 40, 45];
numbers.find(n => n > 33);                              // значение 35
numbers.findIndex(n => n > 33);                         // индекс 2
```

Метод **`fill()`** - заполняет один или несколько элементов массива

```
let numbers = [1, 2, 3, 4];
numbers.fill(1);                        // 1,1,1,1 все элементы 1

let numbers = [1, 2, 3, 4];
numbers.fill(1, 2);                     // 1,2,1,1 начиная с индекса 2 вствить 1

let numbers = [1, 2, 3, 4];
numbers.fill(0, 1, 3);                  // 1,0,0,3
```

Метод **`copyWithin()`** - позволяет копировать элементы в массив

```
let numbers = [1, 2, 3, 4];
numbers.copyWithin(2, 0);               // 1,2,1,2 (копирует с 0 вставляет с 2)
```

Типизированные массивы.

Буферы массивов.

```
let buffer = new ArrayBuffer(10);       // выделяем 10 байт
console.log(buffer.byteLength);         // 10

let view = new DataView(buffer);        // представление
console.log(view.byteOffset);           // 0
console.log(view.buffer);
console.log(view.byteLength);
```

```
view.setInt8(0, 5);                    // записать 5
view.setInt8(1, -1);                   // записать -1

view.пуеInt8(0);                       // чтение 5
view.пуеInt8(1);                       // чтение -1

view.пуеInt16(0);                      // чтение 1535
```

Типизированные массивы это представления: Int8Array, Uint8Array, Uint8ClampedArray, Int16Array, Uint16Array, Int32Array, Uint32Array, Float32Array, Float64Array

```
let buffer = new ArrayBuffer(10);
let view1 = new Int8Array(buffer);
let view2 = new Int8Array(buffer, 5, 2);
```

Итераторы

```
let ints = new Int16Array([25, 50]);
let intsArray = [...ints];
console.log(intsArray instanceof Array);     // true
console.log(intsArray[0]);      // 25
console.log(intsArray[1]);      // 50
```

Объект Promise и асинхронное программирование.

Модель событий

```
let button = document.getElementById("my-btn");
button.onclick = function(event){
      console.log("Click!");
};
```

Обратный вызов

NodeJS продвигает улучшенную модель асинхронного программирования основанного на обратных вызовах.

```
readFile("example.txt", function(err, contents){
      if(err) throw err;

      writeFile("example.txt", function(err){
            if(err)throw err;
            console.log("File was written!");
      });
});
```

Но при таком методе можно попасть в callback hell (в ад) обратных вызовов.

Жизненный цикл объекта **Promise**

- pending – ожидание
- fulfilled – выполнено
- rejected – отклонено
- then() – метод выполнения действий (resolve, reject)
- catch() – метод определяет обработчик ошибок.

```
let promise = new Promise(function(resolve, reject){
      resolve("OK");
});
```

resolve — функция вызывается когда исполнение завершено успешно
reject — функция сообщает что исполнение потерпело неудачу.

```
let promise = readFile("example.txt");
promise.then(function(contents){
      console.log(contents);
}, function(err){
      console.error(err.message);
});
```

Пример:

```
      let promise = new Promise(function(resolve, reject){
            resolve();
      });
      promise.then(function(){
            console.log("Resolved")
      });
```

Создание установившихся объектов Promise

Метод **`resolve(value)`** – возвращает объект Promise в состоянии «Выполнено»

```
let promise = Promise.resolve(42);
promise.then(function(value){
      console.log(value);                                  // 42
});
```

Метод **`reject(value)`** – возвращает объект Promise в состоянии «Откланено»

```
let promise = Promise.reject("Ошибка");
promise.catch(function(value){
      console.error(value);                                // Ошибка
});
```

Thenable – объекты, отличаются от Promise

```
let thenable = {
      then: function(resolve, reject){
            resolve(42);
      }
}
```

```
let promise = Promise.resolve(thenable);
promise.then(function(value){
	console.log(value);                                    // 42
});
```

Ошибки исполнения

```
let promise = new Promise(function(resolve, reject){
	throw new Error("Explosion!");
});
promise.catch(function(error){
	console.error(error.message);
});
```

Составление цепочек из объектов Promise

```
let promise = new Promise(function(resolve, reject){
	resolve(45);
});
promise.then(function(value){
	console.log(value);
}).then(function(){
	console.log("Finished");
});
```

аналогично можно записать так

```
let promise1 = new Promise(function(resolve, reject){
	resolve(45);
});
let promise2 = promise1.then(function(value){
	console.log(value);
});
promise2.then(function(){
	console.log("Finished");
});
```

Перехват ошибок. (всегда добавляйте обработчик отказа в кенце цепочки объектов Promise)

```
let promise = new Promise(function(resolve, reject){
	resolve(42);
});
promise.then(function(value){
	throw new Error("Explosion!");
}).then(function(value){
	console.log(value);
}).catch(function(error){
	console.error(error.message);
});
```

Возврат значений в цепочке объектов Promise

```
let promise = new Promise(function(resolve, reject){
	resolve(42);
});
```

```
promise.then(function(value){
	console.log(value);                                    // 42
	return value + 1;
}).then(function(value){
	console.log(value);                                    // 43
});
```

Обработка сразу нескольких объектов Promise

Метод **`Promise.all()`** – переходит в состояние «Выполнено» только если в этом состоянии окажутся все объекты Promise в интерируемом объекте.

```
let p1 = new Promise(function(resolve, reject){ resolve(41); });
let p2 = new Promise(function(resolve, reject){ resolve(42); });
let p3 = new Promise(function(resolve, reject){ resolve(43); });
let promise = Promise.all([p1, p2, p3]);
prommise.then(function(value){
	console.log(value[0], value[1], value[2]);          // 41 42 43
});
```

Метод **`Promise.race()`** – возвращает Promise устанавливается срузу же, как только устанавливается первый из переданных объектов Promise. (Не ждет когда выполнятся все объекты Promise как метод all)

```
let p1 = new Promise(function(resolve, reject){ resolve(41); });
let p2 = new Promise(function(resolve, reject){ resolve(42); });
let p3 = new Promise(function(resolve, reject){ resolve(43); });
let promise = Promise.race([p1, p2, p3]);
prommise.then(function(value){
	console.log(value);                                    // 41
});
```

Наследование Promise

```
class MyPromise extends Promise {
	success(resolve, reject){
		return this.then(resolve, reject);
	}
	failure(reject){
		return this.catch(reject);
	}
}

let promise = new MyPromise(function(resolve, reject){
	resolve(42);
});

promise.success(function(value){
	console.log(value);
}).failure(function(value){
	console.error(value);
});
```

Пример:

```
function applyForVisa(document){
	let promise = new Promise(function(resolve, reject){
		Math.random() > 0.5 ? resolve({}) : reject("В визе отказано");
	});
	return promise;
}

function hotel(visa){
	return new Promise(function(resolve, reject){
		Math.random() > 0.5 ? resolve(visa) : reject("Нет свободных номеров");
	})
}

function tickets(visa){
	Math.random() > 0.5 ? Promise.resolve(visa) : Promise.reject("Нет билетов");
}

applyForVisa({}).then(visa => { return visa;})
			.then(hotel)
			.then(tickets)
			.catch(error => console.error(error));
```

Прокси-объекты и Reflection API.

- Proxy() - создает прокси-объект для использования вместо другого объекта.
- Reflection API — программный интерфейс представлен объектом Reflect — коллекцией методов преобразования

Создание простого прокси-объекта

```
	let target = {};
	let proxy = new Proxy(target, {});
	proxy.name = "proxy_name";
	console.log(proxy.name === target.name);	// true
```

Проверка свойств с помощью ловушки **`set()`**

```
	let target = { name: "target" };
	let proxy = new Proxy(target, {

		set(trapTarget, key, value, receiver){
			// игнорировать существующие свойства
			if(!trapTarget.hasOwnProperty(key)){
				if(isNaN(value)){
					throw new TypeError("Свойство не число");
				}
			}
			return Reflect.set(trapTarget, key, value, receiver);
		}
	});

	proxy.count = 1;		// корректно (это число)
	proxy.name = "proxy";		// корректно (это свойство изначально)
	proxy.anotherName = "abc";	// ошибка (новое свойство не число)
```

Проверка формы объектов с помощью ловушки **`get()`**

```
let target = { };
let proxy = new Proxy(target, {

      get(trapTarget, key, receiver){
            if(!(key in receiver)){
                  throw new TypeError("Propetry " + key + " doesn't exist.");
            }
            return Reflect.get(trapTarget, key, receiver);
      }

});

proxy.name = "proxy";
console.log(proxy.name);  // корректно
console.log(proxy.nme);          // вернет ошибку
```

- `has()` - сокрытие свойств с помощью `Reflect.has()`
- `deleteProperty()` - предотвращает удаление свойств
- `setPropertyOf()` и `getPropertyOf()` - ловушки операций с прототипами
- `isExtensible()` и `preventExtensible()` - ловушки расширяемости объектов
- `getOwnPropertyDescriptor()` - ловушки операций с дескрипторами
- `ownKeys()` - перехватывает метод [`[OwnPropertyKeys]`]
- `apply()` и `construct()` - перехватывают `[[Call]]` и `[[Construct]]`
- `defineProperty`

Отключение прокси-объектов `Proxy.revocable()`

```
let target = { name: "target" };
let {proxy, revoke} = Proxy.revocable(target, {});

console.log(proxy.name);                                    // target
revoke();                                                   // отключение прокси
console.log(proxy.name);                                    // ошибка
```

Инкапсуляция кода в модули.

- Модуль — фрагмент кода который автоматически действует в строгом режиме без возможности изменений его.
- Сценарий — подключающийся любой код который не является модулем

export - Основы экспортирования

```
export var color = "red";
export let name = "John";
export const MAX_VALUE = 100;
```

Экспорт данных

```
export function sum(n1, n2){
      return n1 + n2;
}
```

Экспорт функции

Экспорт класса

```
export class Rectangle {
	constructor(name){
		this.name = name;
	}
}
```

Экспорт после определения функции

```
function multiply(n1, n2){
	return n1 * n2;
}
export multiply;
```

Экспорт по умолчанию

```
class Rectangle {
	constructor(name){
		this.name = name;
	}
}
export default Rectangle;
```

import - Основы импортирования

Импортирование единственной привязки

```
import { sum } from "./exapmle.js";
console.log(sum(1, 2));
```

Импортирование нескольких привязок

```
import { sum, multiply, MAX_VALUE } from "./exapmle.js";
console.log(sum(1, MAX_VALUE));
console.log(multiply(2, MAX_VALUE));
```

Импортироание всего модуля

```
import * as example from "./exapmle.js";
console.log(example.sum(1, 2));
console.log(example.MAX_VALUE);
```

Тонкая особенность импортирования привязок

```
export var name = "John";
export function setName(newName){
	name = newName;
}

-----------------------------------------------

import { name, setName } from "./exapmle.js";

console.log(name);                                    // John
setName("Bill");
console.log(name);                                    // Bill

name = "Greg";                                        // ошибка
```

Переименование экспортируемых и импортируемых привязок

```
function sum(n1, n2){
	return n1 + n2;
}
export { sum as add };

import { add as sum } from "./exapmle.js";
```

Значение по умолчанию в модулях (**default**)

Экспортирование значений по умолчанию

```
export default function(n1, n2){
	return n1 + n2;
}

function sum(n1, n2){
	return n1 + n2;
}
export default sum;
export { sum as default };
```

Импортирование значений по умолчанию

```
import sum from "./exapmle.js";
console.log(sum(1, 2));
```

Импортирование несколько привязок вместе с привязкой по умолчанию

```
export let color = "red";
export default function(n1, n2){
	return n1 + n2;
}

------------------------------------------------------

import sum, { color } from "./exapmle.js";
console.log(sum(1, 2));
console.log(color);

import { default as sum, color } from "./exapmle.js";
```

Реэкспорт привязок

```
import { sum } from "./exapmle.js";
export { sum }

то же самое можно записать проще
export { sum } from "./exapmle.js";

экспорт значения с другим именем
export { sum as add } from "./exapmle.js";

экспортировать все что находится в импортируемом модуле
export * from "./exapmle.js";
```

Импортирование без привязок (применяется для полифилов и расширений)

```
Array.prototype.pushAll = function(items){
      return this.push(...items);
};
----------------------------------------
import "./exapmle.js";
let color = ["red", "green", "blue"];
let item = [];
item.pushAll(color);
```

Загрузка модулей на странице. (type = “module”)

загрузка модуля из файла

```
<script type="module" src="example.js"></script>
```

встроенный код в качестве модуля

```
<script type="module">
      import { sum } from "./exapmle.js";
      let result = sum(1, 2);
</script>
```

Подключение СЦЕНАРИЯ `type = “text/javascript”`

Подключение МОДУЛЯ `type = “module”`

Асинхронная загрузка модулей в веб-браузерах

```
<script type="module" async src="example1.js"></script>
<script type="module" async src="example2.js"></script>
```

Загрузка модулей в фоновые потоки выполнения (WebWorkers и Service Workers)

```
let worker = new Worker("script.js");                    // загрузка сценарий
let worker = new Worker("script.js", {type: "module"});  // загрузка модуля
```

Расширение спецификаторов модулей

`/`	поиск начинается с корневого каталога	`"/example1.js"`
`./`	поиск в текущем каталоге	`"./example1.js"`
`../`	поиск в родительском каталоге	`"../example1.js"`