## Блочная область видимости (block scope)

В текущей версии JavaScript присутствует функциональная область видимости. Это означает, что все переменные, объявленные c помощью ключевого слова var, будут видны в любом месте функции (даже если они объявлены внутри блока):

function f(a) {

if (a < 0) {

var i = 3;

}

console.log(i); // 3

}

f(-1)

В новой версии появится ключевое слово let, которое позволит объявлять переменные с блочной областью видимости:

function f(a) {

if (a < 0) {

let i = 3;

}

console.log(i); // ReferenceError: i is not defined

}

f(-1)

## Значения параметров по умолчанию

В функциях добавилась возможность объявлять у параметров значения по умолчанию:

function setLevel(newLevel = 0) {

...

}

setLevel(); // newLevel = 0

setLevel(5); // newLevel = 5

setLevel(undefined); // newLevel = 0

## Именованные параметры функций

В функциях также появилась возможность указывать именованные параметры:

function foo({ from, to = 10 }) {

...

}

foo({ from: 1, to: 5 });

foo({ to: 5, from: 1 });

foo({ from: 1 });

Именованные параметры можно комбинировать с обычным (позиционными параметрами):

function foo(positional, { named1, named2 }) {

...

}

foo(123, { named1: 'abc', named2: 'def' })

foo(123, { named2: 'def', named1: 'abc' })

## Destructuring assignment

ECMAScript 6 позволит деструктуризировать при присваивании:

let { first: f, last: l } = { first: 'Jane', last: 'Doe' };

console.log(f); // 'Jane'

console.log(l); // 'Doe'

## Классы

В ECMAScript 6 появятся классы:

// Supertype

class Person {

constructor(name) {

this.name = name;

}

describe() {

return "Person called " + this.name;

}

}

// Subtype

class Employee extends Person {

constructor(name, title) {

super.constructor(name);

this.title = title;

}

describe() {

return super.describe() + " (" + this.title + ")";

}

}

Теперь можно использовать эти классы:

let jane = new Employee("Jane", "CTO");

jane instanceof Person; // true

jane instanceof Employee; // true

jane.describe(); // 'Person called Jane (CTO)'

Всего того же можно было добиться с помощью прототипов:

// Supertype

function Person(name) {

this.name = name;

}

Person.prototype.describe = function () {

return "Person called " + this.name;

};

// Subtype

function Employee(name, title) {

Person.call(this, name);

this.title = title;

}

Employee.prototype = Object.create(Person.prototype);

Employee.prototype.constructor = Employee;

Employee.prototype.describe = function () {

return Person.prototype.describe.call(this) + " (" + this.title + ")";

};

Классы в ECMAScript 6 — это просто синтаксический сахар над конструкторами и функциями.  
  
Классы могут иметь статические методы:

class Point {

constructor(x, y) {

this.x = x;

this.y = y;

}

static zero() {

return new Point(0, 0);

}

}

Приватных полей и методов не будет (по крайней мере, в ECMAScript 6). Однако некоторое сокрытие данных все же появится. Через модули.

## Модули

В JavaScript наконец-то появятся модули:

module Math {

export function sum(x, y) {

return x + y;

}

export var pi = 3.141593;

// Не видна снаружи

function internal() {

...

}

}

Импортирование модуля:

import Math.{sum, pi};

alert("2π = " + sum(pi, pi));

Можно использовать \*, чтобы импортировать всё:

import Math.\*;

alert("2π = " + sum(pi, pi));

Модули можно вкладывать друг в друга:

module Widgets {

module Button { ... }

module Alert { ... }

module TextArea { ... }

...

}

import Widgets.Alert.{messageBox, confirmDialog};

...

Модули можно подгружать из веба или через файловую систему:

module JSON = require('http://json.org/modules/json2.js'); // web

import JSON.\*;

module File = require('io/File'); // file system

import require("bar.js").y; // file system

Все глобальные переменные в модули являются глобальными только в этом модуле.  
  
Возможны циклические зависимости между модулями.

## Цикл for-of

Как вы знаете, цикл for-in в JavaScript итерирует по всем полям объекта (включая наследованных). Т.е. итерироваться по значениям массива можно, но опасно:

let arr = [ "blue", "green" ];

arr.notAnIndex = 123;

Array.prototype.protoProp = 456;

for(var x in arr) {

console.log(x); // Напечатает blue, green, notAnIndex, protoProp

}

В ECMAScript 6 появится цикл for-of, который решит данную проблему:

for(var x of arr) {

console.log(x); // Напечатает только blue, green

}

#### Arrow-функции

В ECMAScript 6 появятся arrow functions:

let squares = [ 1, 2, 3 ].map(x => x \* x);

Код выше эквивалентен этому:

let squares = [ 1, 2, 3 ].map(function (x) { return x \* x });

Arrow-функции немножко отличаются от обычных функций. В первую очередь тем, что в arrow-функциях this привязан к вышестоящему контексту. Т.е.

let jane = {

name: "Jane",

sayHello: function (friends) {

friends.forEach(friend => { console.log(this.name + " says hello to " + friend) });

}

}

jane.sayHello([ 'Mark', 'John' ]);

выведет

Jane says hello to Mark

Jane says hello to John

как и ожидалось. А

let jane = {

name: "Jane",

sayHello: function (friends) {

friends.forEach(function(friend) { console.log(this.name + " says hello to " + friend) });

}

}

выведет:

says hello to Mark

says hello to John

Проблема в том, что this из анонимной функции function(friend) { ... }) перекрывает this из окружающего контекста. Для того, чтобы этого избежать, можно использовать старый прием с var self = this или использовать функцию bind:

var jane = {

name: "Jane",

sayHello: function (friends) {

friends.forEach(function (friend) {

console.log(this.name + " says hello to " + friend)

}.bind(this));

}

}

Т.е. по сути своей arrow functions — опять же синтаксический сахар над существующими анонимными функциями:

(x, y) => x + y + this.z

есть ничто иное как:

function (x, y) { return x + y + this.z }.bind(this)

## Определения методов

В EcmaScript5 методы - свойства, значениями которых является функция.

var obj = {  
  method: function() {  
  
  }  
};

В EcmaScript6 методы также являются функциями, однако вы можете объявлять их более компактно

let obj {  
  method() {  
  
  }  
};

Существует также способ кратко определить свойства, значения которых являются функциями генератора

let obj = {  
  \* myGeneratorMethod() {  
  
  }  
};

Этот код равен этому

let obj = {  
  myGeneratorMethod: function\* () {  
  
  }  
};

**Сокращения значений свойств**

Сокращения позволяют вам сократить объявление свойства в литерале объекта. Если название переменной, которая определяет значение свойства, также ключевое слово, вы можете опустить ключ. Вот, как это выглядит

let x = 4;  
let y = 1;  
let obj = { x, y };

Последняя строчка равна этой

let obj = { x: x, y: y };

Сокращения значений свойств прекрасно работают с деструктором

let obj = { x: 4, y: 1 };  
let {x,y} = obj;  
console.log(x); // 4  
console.log(y); // 1

**Новые методы для объектов**

**Object.assign(target, source\_1, source\_2, ...)**

Этот метод соединяет **sources** в **target**. Это изменяет **target**, во-первых, копирует все перечислимые собственные свойства **source\_1** в **target**, затем то же самое с **source\_2**. В конце возвращается **target**.

let obj = { foo: 123 };  
Object.assign(obj, { bar: true });  
console.log(JSON.stringify(obj));  
// {"foo":123,"bar":true}

**Object.getOwnPropertySymbols(obj)**

В **ES6** ключ свойства может быть либо строкой, либо символом. Теперь есть 5 методов, которые извлекают ключи объекта

* **Object.keys(obj)** - извлекает все строковые перечисляемые ключи свойств
* **Object.getOwnPropertyNames(obj)** - извлекает все строковые ключи свойств
* **Object.getOwnPropertySymbols(obj)** - возвращает все символьные ключи свойств
* **Reflect.ownKeys(obj)** - извлекает все ключи всех собственных свойств
* **Reflect.enumerate(obj)** - извлекает все строчные значения ключей всех перечисляемых свойств

**Object.is(value1, value2)**

Строгий оператор эквивалентности рассматривает два значения иначе, чем можно было ожидать

Первое: **NaN** не равен самому себе. Это очень печально, потому что это часто мешает нам обнаружить **NaN**

[0,NaN,2].indexOf(NaN)  
-1

Второе: в **javascript** 2 нуля, но строгое сравнение относится к них одинаково

-0 === +0  
true

**Object.is()** предоставляет возможность сравнения значений чуть более точно, чем обычное строго сравнение.

Object.is(NaN, NaN)  
true  
Object.is(-0, +0)  
false

Все другое сравнивается так же, как и строгое сравнение.

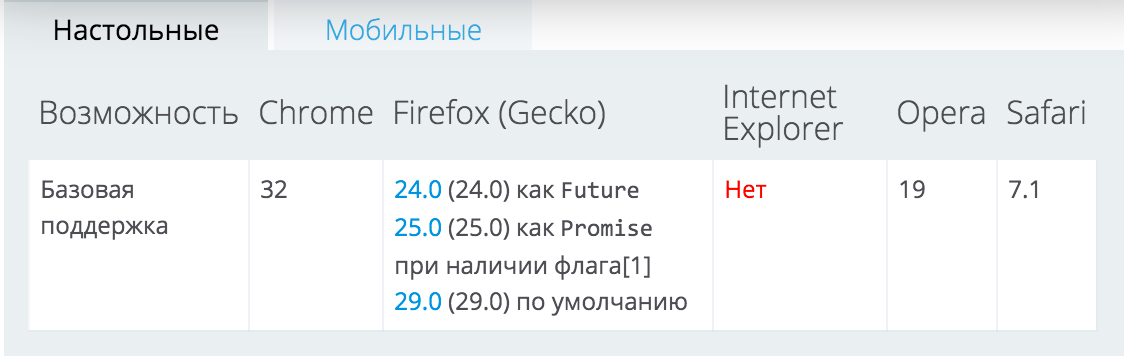
Если мы совместим **Object.is()** с новым **ES6** методом массивов **findIndex()**, то получим **NaN**.

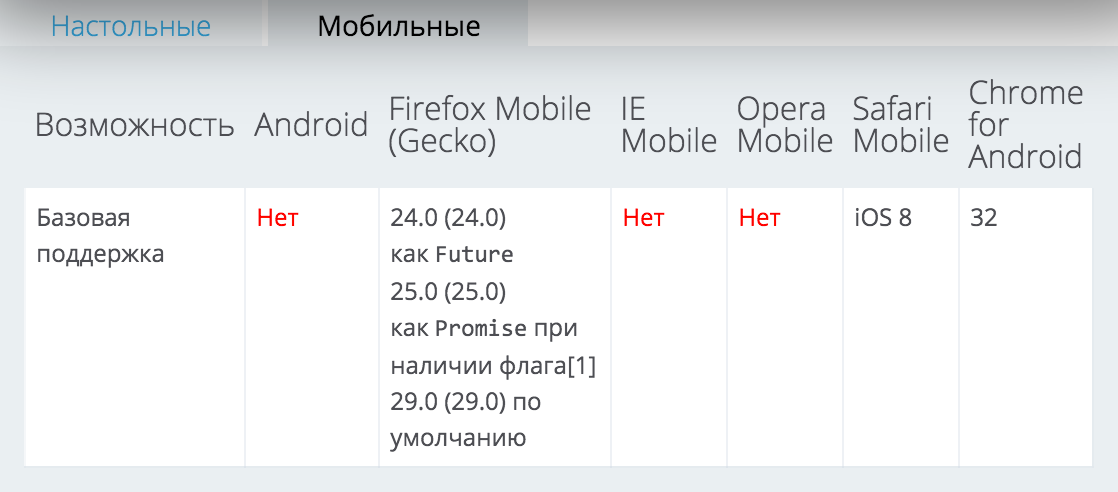
[0,NaN,2].findIndex(x => Object.is(x, NaN))  
1

**Object.setPrototypeOf(obj, proto)**

Этот метод устанавливает прототип **proto** для объекта **obj**. В **ES5** мы должны были делать это через **Object.create()**, теперь же мы можем делать это через удобный метод.

**Совместимость с браузерами**





Получить наиболее актуальную информацию по поддержке ES6 тем или иным браузером удобнее всего по ссылке: <https://kangax.github.io/compat-table/es6/>, которую должно открыть теми браузерами, которые вас интересуют.