# Блочная область видимости (block scope)

В ECMAScript5 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 ECMAScript6 появилось 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 позволяет деструктуризировать при присваивании:

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

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

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

# Классы

В ECMAScript 6 появились 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 могут иметь статические методы:

**class** **Point** {

constructor(x, y) {

**this**.x = x;

**this**.y = y;

}

**static** **zero**() {

**return** **new** Point(0, 0);

}

}

# Модули

В JavaScript появились 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

Цикл for-in в JavaScript итерирует по всем полям объекта:

**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 появился цикл for-of:

**for**(**var** x **of** arr) {

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

}

# Стрелочные функции

В ECMAScript 6 появились 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**

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. 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**)

Другие отличия arrow-функций от обычных функций:

* Нельзя использовать arrow-функции как конструкторы (new (() => {}) кинет ошибку)
* Arrow-функции не могут обратиться к переменной arguments (да и незачем)

В остальном arrow-функции не отличаются от обычных функций. Они поддерживают значения по умолчанию, переменное количество параметров, операторы typeof и instanceof:

typeof () => {}; *// 'function'*

() => {} **instanceof** **Function**; *// true*