**Примитивные типы:**

**EcmaScript 5**

Всего стандарт определяет девять типов, причём непосредственно доступны в ECMAScript программе, всего шесть:

* Undefined
* Null
* Boolean
* String
* Number
* Object

Остальные три типа доступны лишь на уровне реализации (никакой из объектов ECMASCript программы не может обладать подобным типом) и используются спецификацией для хранения промежуточных значений вычислений, а также, разъяснения поведения некоторых конструкций.

**Тип Undefined** Неопределенный имеет только одно значение – undefined. Всякая переменная, которой не было присвоено значения, имеет значение undefined.

Тип **Nul**l имеет только одно значение – null.

Тип **Boolean** представляет собой логическую сущность, имеющую два значения – true и false.

Тип **String** представляет собой множество всех конечных упорядоченных последовательностей, состоящих из 16-битовых беззнаковых целочисленных значений ("элементов") в количестве ноль или более. В целом тип String используется для представления текстовых данных в активной программе на ECMAScript, при этом каждый элемент в строке String рассматривается как значение кодовой единицы.

Тип **Number** имеет ровно 18437736874454810627 (то есть, 264−253+3) значений, представляющих собой 64-битовые значения с удвоенной точностью в формате IEEE 754 в соответствии со Стандартом IEEE для двоичной арифметики с плавающей точкой, за исключением того, что 9007199254740990 (то есть, 253−2) индивидуальных значений "не число" по стандарту IEEE представлены в языке ECMAScript как одно специальное значение NaN.

**EcmaScript 6**

**let**

объявление переменной (аналог var), локальной для блока (внутри if, switch и т.п.).

var a = 5;

var b = 10;

if (a === 5) {

let a = 4; // видна только внутри if

var b = 1; // видна везде внутри функции

console.log(a); // 4

console.log(b); // 1

}

console.log(a); // 5

console.log(b); // 1

**const**

объявляет переменную и присваивает ей значение, которое потом не может быть изменено.

const a = 15;

console.log(a); // 15

a = 1;

console.log(a); // 15

const b;

console.log(b); // undefined

b = 1;

console.log(b); // undefined

**Примитивные объекты**

**EcmaScript 5**

Объект Object представляет собой набор свойств. Каждое свойство является либо именованным свойством данных, либо именованным свойством-аксессором, либо внутренним свойством.

Именованное свойство данных named data property ассоциирует имя со значением в языке ECMAScript и с набором булевых атрибутов.

Именованное свойство-аксессор named accessor property ассоциирует имя с одной или двумя функциями доступа accessor functions и с набором булевых атрибутов. Функции доступа используются для хранения или извлечения значения в языке ECMAScript, связанного с этим свойством.

Внутреннее свойство internal property не имеет имени и не является напрямую доступным через операторов ECMAScript. Внутренние свойства существуют исключительно в целях спецификации.

Именованные (не внутренние) свойства имеют два типа доступа: get и put, обозначающие извлечение и присвоение соответственно.

Новая возможность ECMAScript 5 — расширяемость объектов теперь можно переключать. Отключение расширяемости может предотвратить добавление новых свойств к объекту.  
  
ES5 предусматривает два способа манипулирования и проверки расширяемости объектов: Object.preventExtensions( obj ), Object.isExtensible( obj )  
preventExtensions блокирует обьект и предотвращает создание любых новых свойств обьекта в будущем. isExtensible — возможность определить, расширяем в настоящее время обьект или нет.  
  
Пример использования:  
var obj = {};  
obj.name = "John";  
print( obj.name );  
// John  
  
print( Object.isExtensible( obj ) );  
// true  
  
Object.preventExtensions( obj );  
  
obj.url = "http://ejohn.org/"; // Exception in strict mode  
  
print( Object.isExtensible( obj ) );  
// false 

**EcmaScript 6**

**Выражение деструктурирования**

«Разворачивает» объект или массив в последовательность переменных. При этом при разворачивании объекта нужно указать только ключи, которые нужны.

var a, b, c = [1, 2];

[a, b] = c;

console.log(a, b); // 1, 2

var pt = {x:2, y:-5, z: 1};

var {x, y1, z} = pt;

console.log(x, y1, z); // 2, undefined, 1

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

var {x: n1, y1: n2, z: n3} = pt;

console.log(n1, n2, n3); // 2, undefined, 1

**WeakMap**

Объект, ключами которого могут быть объекты. Вроде как не боится циклических ссылок внутри себя.

Имеет методы get, set, has, delete.

var wm1 = new WeakMap(),

wm2 = new WeakMap();

var o1 = {},

o2 = function(){},

o3 = window;

wm1.set(o1, 37);

wm1.set(o2, "azerty");

wm2.set(o1, o2); // любое значение, включая функции и объекты

wm2.set(o3, undefined);

wm2.set(wm1, wm2); // ключами и значениями могут быть любые объекты, даже WeakMaps

wm1.get(o2); // "azerty"

wm2.get(o2); // undefined, т.к. нет такого ключа

wm2.get(o3); // undefined, т.к. там хранится именно это значение

wm1.has(o2); // true

wm2.has(o2); // false

wm2.has(o3); // true (несмотря на значение "undefined")

wm1.has(o1); // true

wm1.delete(o1);

wm1.has(o1); // false

**Итераторы**

Объект, который знает, в каком порядке нужно обходить его свойства, и хранит текущую позицию в обходе. Сводится к методу next.

var lang = { name: 'JavaScript', birthYear: 1995 };

var it = Iterator(lang);

// однажды инициализированный, позволяет получить последовательный доступ к парам ключ-значение объекта

var pair = it.next(); // пара ["name", "JavaScript"]

pair = it.next(); // пара ["birthYear", 1995]

pair = it.next(); // Возбуждается исключение StopIteration

// удобно использовать для обхода объекта

var it = Iterator(lang);

for (var pair in it)

console.log(pair); // каждый раз выводит пару [key, value]

**Функции**

**EcmaScript 5**

Попытка переписать объект аргументов приведёт к ошибке:

arguments = [...]; // not allowed

Определение нескольких аргументов с одинаковым названием приведет к ошибке 

function(Foo, Foo) {} // error

Доступ к arguments.caller и arguments.callee сейчас выбросит исключение. Таким образом, любые анонимные функции, на которые вы хотите сделать ссылку, необходимо будет именовать, например, так:

setTimeout(function later(){  
 // do stuff...  
 setTimeout( later, 1000 );  
}, 1000 );  
  
\* This source code was highlighted with [Source Code Highlighter](http://virtser.net/blog/post/source-code-highlighter.aspx).

Свойства arguments и caller других функций больше не существуют — и способность определить их запрещена.  
function test(){  
 function inner(){  
  // Don't exist, either  
  test.arguments = ...; // Error  
  inner.caller = ...; // Error  
 }  
}  
  
Наконец, давняя (и очень раздражающая) ошибка была исправлена: Случаи, когда null или undefined принуждены становиться глобальным объектом. Строгий режим теперь препятствует тому, чтобы это случалось, и бросает исключение вместо этого.  
(function(){ ... }).call( null ); // Exception

**EcmaScript 6**

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' })

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)

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

**Массивы**

**EcmaScript 5**

Новые функции для работы с массивами в ECMAScript 5:

**forEach()**

Последовательно проходим каждый элемент массива и каждый раз вызываем функцию, которая передаётся первым и единственным параметром. В свою очередь эта передаваемая функция принимает три параметра: значение текущего элемента, его индекс и ссылку на сам массив.

Простейший пример:

var ar = [1,2,3];

ar.forEach(

function(element, index, array){

console.log(element, index, array);

});

Вывод:

1 0 [1, 2, 3]

2 1 [1, 2, 3]

3 2 [1, 2, 3]

К сожалению, если нам понадобится преждевременно выйти из цикла, то break здесь не сработает и мы получим ошибку "unlabeled break must be inside loop or switch", то есть break только в циклах или switch, а forEach не является ни тем ни другим, а методом Array.

Предлагается такая конструкция через возбуждение исключения, если функция, которая передается в forEach() возбуждает исключение foreach.break, то выполнение прекращается.

var ar = [1,2,3];

function foreach(array, func){

try{

array.forEach(func);

} catch(e){

if(e == foreach.break) return;

else throw e;

}}

foreach.break = new Error('StopIteration');

foreach(ar, function(value) {

console.log(value);

if(value == 2)

throw foreach.break;

})

Вывод: 1,2. До третьего элемента мы уже не дойдем.

**map()**

Порядок вызова аналогичен forEach, за исключением, что callback-функция должна возвратить значение. То есть мы проходимся по каждому элементу, наша функция проделывает некие операции и возвращает новое значение, но не для этого массива, а нового, старый массив не изменяется!

Если массив был с дырками, то есть индексы с разрывами (a[0], a[5], a[78]), то результирующий массив будет иметь точно такой же вид (b[0], b[5], b[78])

Массив, все элементы которого на 1 больше, чем у исходного

var arA = [1,2,3];

var arB = arA.map(function(value){ return value + 1;

});

console.log(arA);

console.log(arB);

Вывод:

[1,2,3] - исходный массив не изменился

[2,3,4] - новый массив, после вызова map()

**filter()**

Полезна для фильтрации элементов, callback-функция (та которую мы передаём в filter, forEach, map и т.д.) должна возвращать true или false. В первом случае текущий элемент помещается в новый массив, в лругом мы идем дальше по списку.

Как и в случае с map() мы получаем на выходе новый массив, старый не меняется.

Оставить элементы меньшие 5

var arA = [1,2,3,4,5,6,7,8,9];

var arB = arA.filter(function(value){ return value < 5 ? true : false;

});

console.log(arA);

console.log(arB);

Вывод:

[1,2,3,4,5,6,7,8,9] - исходный массив не изменился

[1,2,3,4] - новый массив, после вызова filter()

**every() и some()**

Эти два метода возвращают не массив, а true/false.

every() возвращает true тогда и только тогда, когда callback-функция вернула true после всех вызовов дял каждого элемента.

var ar = [1,2,3,4,5,6,7,8,9];

var result = ar.every(function(value){ return value < 10;

});

console.log(result);

Вывод: true - все элементы массива меньше 10.

var ar = [1,2,3,4,5,6,7,8,9];

var result = ar.every(function(value){ return value < 5;

});

console.log(result);

Вывод: false - не все элементы массива меньше 5.

some() же возвращает true если хотя бы один раз callback-функция вернула true.

var ar = [1,2,3,4,5,6,7,8,9];

var result = ar.some(function(value){ return value < 5;

});

console.log(result);

Вывод: true - есть хотя бы один элемент меньше 5.

var ar = [1,2,3,4,5,6,7,8,9];

var result = ar.some(function(value){ return value > 50;

});

console.log(result);

Вывод: false - ни одного элемента больше 50.

**reduce() и reduceRight()**

Два параметра: первый - callback-функция, второй - начальное значение переменной.

callback-функция принимает два значения: первое это результат предыдущей работы функции, второе - очередной элемент массива.

Продемонстрировать лучше на примере:

var ar = [4,3,2,1];

var result = ar.reduce(function(x, y){ return x + y;

}, 0);

console.log(result);

Изначально первый параметр x = 0.

Шаг 1: x = 0, y = ar[0] = 4; x + y = 4;

Шаг 2: x = 4, y = ar[1] = 3; x + y = 7;

Шаг 3: x = 7, y = ar[2] = 2; x + y = 9;

Шаг 2: x = 9, y = ar[1] = 1; x + y = 10;

reduceRight идет с конца массива.

**indexOf() и lastIndexOf()**

Возвращает индекс первого и последнего вхождения элемента в массив.

var ar = [4,3,2,1,4];

console.log(ar.indexOf(4));

console.log(ar.lastIndexOf(4));

Вывод: 0 и 4

**EcmaScript 6**

**Spread**

Распаковывание массива в параметры вызова функции

let a = [0,1,2,3], o = func(...a);

**Циклы**

**EcmaScript 5**

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

}

**EcmaScript 6**

В ECMAScript 6 появится цикл for-of, который решит данную проблему:  
for(var x of arr) {

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

}

Также, возможно, в язык добавится оператор yield, с помощью которого можно легко и красиво писать кастомные итераторы

**ООП**

**EcmaScript 5**

Использование 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 + ")";

};

**EcmaScript 6**

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)'

Как видите, 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.