Проаспэт А.

ДКО-122б

**Об ECMAScript 5**

Объекты

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   
  
\* This source code was highlighted with Source Code Highlighter.

Свойства и дескрипторы

Свойства были полностью пересмотрены. Они теперь не являются простыми значениями, связанными с объектом — теперь у вас есть полный контроль над тем, как они могут действовать. Большая сила связана, тем не менее, с повышенной сложностью.  
  
Свойства обьектов разбиты на две части.  
  
Содержание свойства может быть определено двумя вариантами: значение (свойство-данные — это традиционные свойства, которые мы знаем и любим в ECMAScript 3) или геттер и сеттер (свойства-«аксессоры» — мы знаем их по некоторым современным браузерам, таким, как WebKit и Gecko.

* Value. Хранит значение свойства.
* Get. Эта функция вызывается каждый раз, когда происходит доступ к значению свойства.
* Set. Эта функция вызывается каждый раз, когда значение свойства изменяется.

К тому же свойства могут быть…

* Изменяемыми. Если ложь, то значение этого свойства не может быть изменено.
* Конфигурируемыми. Если ложь, любые попытки удалить свойство или изменить его атрибуты (записываемость, конфигурируемость или перечислимость) не получатся.
* Перечислимыми. Если истина, то свойство будет итерировано, когда пользователь делает for (var prop in obj) (или что-то в этом духе).

В общей сложности эти различные атрибуты составляют дескриптор свойства. Например, простой дескриптор может выглядеть примерно следующим:

{  
 value: "test",  
 writable: true,  
 enumerable: true,  
 configurable: true  
}  
  
\* This source code was highlighted with Source Code Highlighter.

Три атрибута (writable, enumerable и configurable) являются факультативными и все по умолчанию истинны. Таким образом, для свойства вам обязательно нужно предоставить только либо значение, либо геттер и сеттер.  
  
Вы можете использовать новый метод Object.getOwnPropertyDescriptor для получения этой информации для уже существующего свойства обьекта.

Object.getOwnPropertyDescriptor( obj, prop )

Этот метод позволяет получить доступ к дескриптору свойства. Он является единственным способом получить эту информацию (иначе говоря, дескрипторы не в распоряжении пользователя — они не существуют как видимые атрибуты свойства, они хранятся внутри в движке ECMAScript).  
  
Пример использования:

var obj = { foo: "test" };  
print(JSON.stringify(   
 Object.getOwnPropertyDescriptor( obj, "foo" )  
));  
// {"value": "test", "writable": true,   
// "enumerable": true, "configurable":true}   
  
\* This source code was highlighted with Source Code Highlighter.

Метод Object.defineProperty( obj, prop, desc )

Этот метод позволяет определить новое свойство для объекта (или изменить дескриптор существующего свойства). Этот метод принимает дескриптор свойства и использует его для инициализации (или обновления) свойства.  
  
Пример использования:

var obj = {};  
Object.defineProperty( obj, "value", {  
 value: true,  
 writable: false,  
 enumerable: true,  
 configurable: true  
});  
  
(function(){  
 var name = "John";  
   
 Object.defineProperty( obj, "name", {  
  get: function(){ return name; },  
  set: function(value){ name = value; }  
 });  
})();  
  
print( obj.value )  
// true  
  
print( obj.name );  
// John  
  
obj.name = "Ted";  
print( obj.name );  
// Ted  
  
for ( var prop in obj ) {  
 print( prop );  
}  
// value  
// name  
  
obj.value = false; // Exception if in strict mode  
  
Object.defineProperty( obj, "value", {  
 writable: true,  
 configurable: false  
});  
  
obj.value = false;  
print( obj.value );  
// false  
  
delete obj.value; // Exception   
  
\* This source code was highlighted with Source Code Highlighter.

Object.defineProperty — базовый метод новой версии ECMAScript. Фактически все остальные важные возможности зависят от существования этого метода.

Object.defineProperties( obj, props )

Определяет несколько свойств для обьекта за один раз (вместо определения каждого индивидуально).  
  
Примерная реализация:

Object.defineProperties = function( obj, props ) {  
 for ( var prop in props ) {  
  Object.defineProperty( obj, prop, props[prop] );  
 }  
};  
  
\* This source code was highlighted with Source Code Highlighter.

Пример использования:

var obj = {};  
Object.defineProperties(obj, {  
 "value": {  
  value: true,  
  writable: false  
 },  
 "name": {  
  value: "John",  
  writable: false  
 }  
});  
  
\* This source code was highlighted with Source Code Highlighter.

Дескрипторы свойств (и связанные с ними методы) является, пожалуй, наиболее важной новой особенностью ECMAScript 5. Это дает разработчикам возможности очень чётко контролировать свои объекты, предотвратить нежелательные модификации, а также поддерживать единый веб-совместимый API.

Новые возможности

Некоторые интересные новые возможности были введены в язык, используя вышеописанные расширения.  
  
Следующие два метода являются очень полезными для сбора массивов всех свойств объекта.

Object.keys( obj )

Возвращает массив строк, представляющих имена всех перечислимых свойств объекта. Его поведение совпадает с поведением метода из библиотеки Prototype.js.  
  
Примерная реализация:

Object.keys = function( obj ) {  
 var array = new Array();  
 for ( var prop in obj ) {  
  array.push( prop );  
 }  
 return array;  
};  
  
\* This source code was highlighted with Source Code Highlighter.

Пример использования:

var obj = { name: "John", url: "http://ejohn.org/" };  
print( Object.keys(obj).join(", ") );  
// name, url   
  
  
\* This source code was highlighted with Source Code Highlighter.

Object.getOwnPropertyNames( obj )

Почти идентичен Object.keys, но возвращает имена всех свойств объекта (а не только перечислимых).  
  
Пример реализации не представляется возможным в ECMAScript 3, так как неперечислимые свойства не могут быть перечислены. Возвращаемое значение и использование идентично Object.keys.

Object.create (proto, props)

Создает новый объект, прототипом которого является proto, и чьи свойства устанавливаются с помощью Object.defineProperties (props).  
  
Простая реализация будет выглядеть так (требует нового метода Object.defineProperties):

Object.create = function( proto, props ) {  
 var obj = new Object();  
 obj.\_\_proto\_\_ = proto;  
 if ( typeof props !== "undefined" ) {  
  Object.defineProperties( obj, props );  
 }  
   
 return obj;  
};  
  
\* This source code was highlighted with Source Code Highlighter.

Примечание: Приведенный выше код использует специфичное для Mozilla свойство \_proto\_. Это свойство дает вам доступ к внутреннему прототипу объекта, — и позволяет также установить его значение. В ES5 метод Object.getPrototypeOf позволяет получить доступ к этому свойству, но не позволяет его устанавливать — следовательно, метод выше невозможно реализовать в общей, совместимой с ES, манере.  
  
Я [обсудил Object.getPrototypeOf ранее](http://ejohn.org/blog/objectgetprototypeof/), поэтому я не буду обсуждать здесь его еще раз.

Пример использования:

function User(){}  
User.prototype.name = "Anonymous";  
User.prototype.url = "http://google.com/";  
var john = Object.create(new User(), {  
 name: { value: "John", writable: false },  
 url: { value: "http://google.com/" }  
});  
  
print( john.name );  
// John  
  
john.name = "Ted"; // Exception if in strict mode   
  
\* This source code was highlighted with Source Code Highlighter.

Object.seal( obj ), Object.isSealed( obj )

Запечатывание объекта мешает коду удалять или изменять дескрипторы любых свойств обьекта- и запрещает добавлять новые свойства.  
  
Пример реализации:

Object.seal = function( obj ) {  
 var props = Object.getOwnPropertyNames( obj );  
   
 for ( var i = 0; i < props.length; i++ ) {  
  var desc = Object.getOwnPropertyDescriptor( obj, props[i] );  
    
  desc.configurable = false;  
  Object.defineProperty( obj, props[i], desc );  
 }  
   
 return Object.preventExtensions( obj );  
};  
  
\* This source code was highlighted with Source Code Highlighter.

Вы предпочтёте запечатывание объекта, если вы хотите, чтобы набор его существующих свойств оставался неповрежденным, без предоставления новых дополнений, но в то же время позволяя пользователю редактировать их значения.

Object.freeze( obj ), Object.isFrozen( obj )

Замораживание объекта практически идентично запечатыванию, но с тем дополнением, что свойства становятся неизменяемыми.  
  
Пример реализации:

Object.freeze = function( obj ) {  
 var props = Object.getOwnPropertyNames( obj );  
   
 for ( var i = 0; i < props.length; i++ ) {  
  var desc = Object.getOwnPropertyDescriptor( obj, props[i] );  
    
  if ( "value" in desc ) {  
   desc.writable = false;  
  }  
    
  desc.configurable = false;  
  Object.defineProperty( obj, props[i], desc );  
 }  
   
 return Object.preventExtensions( obj );  
};  
  
\* This source code was highlighted with Source Code Highlighter.

Замораживание объекта является ультимативной формой блокировки. После того, как объект был заморожен, он не может быть разморожен — не может быть изменен в любой форме. Это лучший способ убедиться в том, что ваши объекты будут оставаться в точности такими, какими вы оставили их — на неопределенный срок.

О ECMAScript 6

## Переменные

### LET

Вы привыкли к объявлению переменных с помощью var. Теперь вы можете использовать также и let. Тонкое различие заключается в области видимости. В то время как var в качестве области видимости дает в результате тело функции, в которой объявлена переменная, то область видимости переменной, объявленной с помощью let, – это только блок, в котором выполнено объявление.

if(true) {

let x = 1;

}

console.log(x); // undefined

Это поможет сделать код более чистым, в результате уменьшения количества висящих переменных в области видимости. Например, классическая итерация массива:

for(let i = 0, l = list.length; i < l; i++) {

// do something with list[i]

}

console.log(i); // undefined

Часто можно было бы использовать, например, переменную j для другой итерации в этой же области видимости. Но сlet, вы можете смело объявлять i снова, так как переменные объявлены и доступны только внутри области видимости их собственного блока.

### CONST

Существует еще один способ для объявления переменных с областью видимости, ограниченной блоком. С const вы объявляете ссылку на значение, доступную только для чтения. Значение переменной вы должны задать непосредственно при объявлении. Если вы попытаетесь изменить значение переменной или сразу не зададите значение, то получите сообщение об ошибке:

const MY\_CONSTANT = 1;

MY\_CONSTANT = 2 // Error

const SOME\_CONST; // Error

Обратите внимание, что вы все еще можете изменять значения свойств объекта или членов массива:

const MY\_OBJECT = {some: 1};

MY\_OBJECT.some = 'body'; // Cool

## Стрелочные функции

Стрелочные функции – отличное дополнение к языку JavaScript. Они создают короткий и лаконичный код. В этой статье мы рассказываем о стрелочных функциях пораньше, чтобы позже воспользоваться ими в других примерах. Следующий фрагмент кода показывает стрелочную функцию в сравнении аналогичной функцией, написанной в знакомом стиле ES5:

let books = [{title: 'X', price: 10}, {title: 'Y', price: 15}];

let titles = books.map( item => item.title );

// ES5 equivalent:

var titles = books.map(function(item) {

return item.title;

});

Если мы посмотрим на синтаксис стрелочных функций, то не увидим ключевого слова function. Остается только ноль или более аргументов, «жирная стрелка» (=>) и выражение функции. Возвращаемое выражение добавляется неявно.

При нуле или более одном аргументе, вы должны добавить круглые скобки:

// без аргументов

books.map( () => 1 ); // [1, 1]

// несколько аргументов

[1,2].map( (n, index) => n \* index ); // [0, 2]

Заключите выражение функции в блок ({...}), если вам необходима более сложная логика или больше пробельных символов (например, символ новой строки):

let result = [1, 2, 3, 4, 5].map( n => {

n = n % 3;

return n;

});

Стрелочные функции означают не только меньшее количество напечатанных символов, но их поведение отличается от обычных функций. Выражение стрелочной функции наследует this и аргументы из окружающего контекста. Это означает, что вы можете избавиться от уродливых выражений, подобных var that = this, и вам не нужно связывать функции с правильным контекстом. Пример кода (обратите внимание на this.title в сравнении с that.title в версии ES5):

let book = {

title: 'X',

sellers: ['A', 'B'],

printSellers() {

this.sellers.forEach(seller => console.log(seller + ' sells ' + this.title));

}

}

// ES5 equivalent:

var book = {

title: 'X',

sellers: ['A', 'B'],

printSellers: function() {

var that = this;

this.sellers.forEach(function(seller) {

console.log(seller + ' sells ' + that.title)

})

}

}

## Строки

### Методы

К прототипу String была добавлена пара удобных методов. Большинство из них служат просто устранения обходных путей с методом indexOf() для получения того же результата:

'my string'.startsWith('my'); //true

'my string'.endsWith('my'); // false

'my string'.includes('str'); // true

Просто, но эффективно. Другой удобный метод был добавлен для создания повторяющейся строки:

'my '.repeat(3); // 'my my my '

### Шаблонные литералы

Шаблонные литералы обеспечивают чистый способ для создания строк и интерполяции выражений. Возможно, вы уже знакомы с синтаксисом; он основан на знаке доллара и фигурных скобках ${...}. Шаблонные литералы заключаются в обратные кавычки. Краткий пример:

let name = 'John',

apples = 5,

pears = 7,

bananas = function() { return 3; }

console.log(`This is ${name}.`);

console.log(`He carries ${apples} apples, ${pears} pears, and ${bananas()} bananas.`);

// ES5 equivalent:

console.log('He carries ' + apples + ' apples, ' + pears + ' pears, and ' + bananas() +' bananas.');

Как видно из кода выше, шаблонные литералы просто удобны для конкатенации по сравнению с ES5. Однако шаблонные литералы также можно использовать и для многострочных литералов. Имейте в виду, что пробельные символы (например, символ новой строки) являются частью строки:

let x = `1...

2...

3 lines long!`; // Yay

// ES5 equivalents:

var x = "1...\n" +

"2...\n" +

"3 lines long!";

var x = "1...\n2...\n3 lines long!";

## Массивы

Появилось несколько новых статических методов класса у объекта Array и несколько новых методов у прототипа Array.

Во-первых, Array.from создает экземпляры Array из массивоподобных и итерируемых объектов. Примеры массивоподобных объектов включают в себя:

* arguments внутри функции;
* nodeList, возвращенный методом document.getElementsByTageName();
* данные новых структур Map и Set.

let itemElements = document.querySelectorAll('.items');

let items = Array.from(itemElements);

items.forEach(function(element) {

console.log(element.nodeType)

});

// часто используемый в ES5 обходной путь:

let items = Array.prototype.slice.call(itemElements);

В приведенном выше примере, вы можете увидеть, что у массива элементов есть метод forEach, который недоступен в коллекции itemElements.

Интересной особенностью Array.from является второй необязательный аргумент mapFunction, который позволяет создать новый распределяемый массив в одном вызове.

let navElements = document.querySelectorAll('nav li');

let navTitles = Array.from(navElements, el => el.textContent);

Далее у нас есть Array.of, который ведет себя так же, как конструктор Array. Он исправляет особый случай, когда в качестве аргумента передается одно число. Это делает Array.of более предпочтительным по сравнению с new Array(). Тем не менее, в большинстве случаев вы захотите использовать литералы array.

let x = new Array(3); // [undefined, undefined, undefined]

let y = Array.of(8); // [8]

let z = [1, 2, 3]; // Array literal

Последняя, но не менее важная, пара методов была добавлена для прототипа Array. Я думаю, что методы поиска будут очень кстати для большинства разработчиков JavaScript.

* find возвращает первый элемент, для которого функция обратного вызова вернет true;
* findIndex возвращает индекс первого элемента, для которого функция обратного вызова вернет true;
* fill «перезаписывает» элементы массива переданным аргументом.

[5, 1, 10, 8].find(n => n === 10) // 10

[5, 1, 10, 8].findIndex(n => n === 10) // 2

[0, 0, 0].fill(7) // [7, 7, 7]

[0, 0, 0, 0, 0].fill(7, 1, 3) // [0, 7, 7, 7, 0]

## Math

В объект Math была добавлена пара новых методов.

* Math.sign возвращает знак числа как 1, -1 или 0;
* Math.trunc возвращает переданное число без цифр после запятой;
* Math.cbrt возвращает кубический корень числа.

Math.sign(5); // 1

Math.sign(-9); // -1

Math.trunc(5.9); // 5

Math.trunc(5.123); // 5

Math.cbrt(64); // 4

Если вы хотите узнать больше о новых числовых и математических возможностях в ES6, [Dr. Axel Rauschmayer расскажет вам](http://www.2ality.com/2015/04/numbers-math-es6.html).

## Оператор распространения

Оператор распространения (...) – это очень удобный синтаксис для разворачивания элементов массива в определенных местах, например, в качестве аргументов в вызовах функций. Показать несколько примеров – это, вероятно, лучший способ продемонстрировать, насколько они полезны.

Во-первых, посмотрим, как развернуть элементы массива внутри другого массива:

let values = [1, 2, 4];

let some = [...values, 8]; // [1, 2, 4, 8]

let more = [...values, 8, ...values]; // [1, 2, 4, 8, 1, 2, 4]

// ES5 equivalent:

let values = [1, 2, 4];

// Перебрать, вставить, вспотеть, повторить...

// Перебрать, вставить, вспотеть, повторить...

Синтаксис распространения также удобен при вызове функций с аргументами:

let values = [1, 2, 4];

doSomething(...values);

function doSomething(x, y, z) {

// x = 1, y = 2, z = 4

}

// ES5 equivalent:

doSomething.apply(null, values);

Как можете видеть, это спасает нас от часто используемого обхода fn.apply(). Синтаксис очень гибок, поэтому оператор распространения может использоваться в любом месте в качестве списка аргументов. Это означает, что следующий вызов дает тот же результат:

let values = [2, 4];

doSomething(1, ...values);

Мы применяем оператор распространения к массивам и аргументам. На самом деле, он может быть применен ко всем итерируемым объектам, таким как NodeList:

let form = document.querySelector('#my-form'),

inputs = form.querySelectorAll('input'),

selects = form.querySelectorAll('select');

let allTheThings = [form, ...inputs, ...selects];

Теперь allTheThings – это одномерный массив, содержащий элементы <form> и их дочерние элементы <input> и<select>.

## Деструктурирование

Деструктурирование обеспечивает удобный способ для извлечения данных из объектов или массивов. Для начала, хороший пример может быть показан на массиве:

let [x, y] = [1, 2]; // x = 1, y = 2

// ES5 equivalent:

var arr = [1, 2];

var x = arr[0];

var y = arr[1];

С этим синтаксисом нескольким переменным могут быть присвоены значения за один раз. Приятный побочный эффект заключается в том, что вы можете легко поменять местами значения переменных:

let x = 1,

y = 2;

[x, y] = [y, x]; // x = 2, y = 1

Деструктурирование также работает и с объектами. Убедитесь в наличии соответствующих ключей:

let obj = {x: 1, y: 2};

let {x, y} = obj; // x = 1, y = 2

Также вы можете использовать этот механизм, чтобы изменить имена переменных:

let obj = {x: 1, y: 2};

let {x: a, y: b} = obj; // a = 1, b = 2

Еще один интересный образец имитации нескольких возвращаемых значений:

function doSomething() {

return [1, 2]

}

let [x, y] = doSomething(); // x = 1, y = 2

Деструктурирование может быть использовано, чтобы задать значения по умолчанию для объектов аргументов. С литералом объекта вы можете имитировать именованные параметры.

function doSomething({y = 1, z = 0}) {

console.log(y, z);

}

doSomething({y: 2});

## Параметры

### Значения по умолчанию

Значения по умолчанию для параметров функций доступны в ES6 со следующим синтаксисом:

function doSomething(x, y = 2) {

return x \* y;

}

doSomething(5); // 10

doSomething(5, undefined); // 10

doSomething(5, 3); // 15

Выглядит довольно чисто, не так ли? Я уверен, что раньше вы были вынуждены заполнять некоторые аргументы в ES5 похожим способом:

function doSomething(x, y) {

y = y === undefined ? 2 : y;

return x \* y;

}

Значение по умолчанию для аргумента вызывается либо undefined, либо отсутствием аргумента.

### Остаточные параметры

Ранее мы рассмотрели оператор распространения. Остаточные параметры очень похожи на него. Они также используют синтаксис ... и позволяют сохранять оставшиеся аргументы в массив:

function doSomething(x, ...remaining) {

return x \* remaining.length;

}

doSomething(5, 0, 0, 0); // 15

## Модули

Модули – это, конечно, долгожданное дополнение к языку JavaScript. Я думаю, что это главное, ради чего стоит копаться в ES6.

Сегодня любой серьезный JavaScript проект использует какой-либо тип модульной системы, возможно что-то вроде шаблона «открытый модуль» или более обширные форматы AMD или CommonJS. Тем не менее, браузеры не располагают каким-либо типом модульной системы. Вам всегда необходим либо этап сборки, либо загрузчик ваших модулей AMD или CommonJS. Инструменты для обработки этого включают в себя RequireJS, Browserify и Webpack.

Спецификация ES6 включает в себя как новый синтаксис, так и механизм загрузчика модулей. Если вы хотите использовать модули и писать на будущее, то должны использовать данный синтаксис. Современные инструменты сборки поддерживают этот формат, возможно через плагин, поэтому всё должно пройти хорошо (не беспокойтесь, мы обсудим это далее в разделе «Транспиляция»).

А теперь о синтаксисе модулей ES6. Модули разработаны вокруг ключевых слов export и import. Рассмотрим пример с двумя модулями сразу:

// lib/math.js

export function sum(x, y) {

return x + y;

}

export var pi = 3.141593;

// app.js

import { sum, pi } from "lib/math";

console.log('2π = ' + sum(pi, pi));

Как можете видеть, возможен экспорт нескольких выражений export. Каждое должно явно указывать тип экспортируемого значения (в этом примере function и var).

Выражение import в этом примере использует синтаксис (подобный деструктуризации), чтобы явно определить, что будет импортировано. Чтобы импортировать модуль целиком, должен быть использован символ \* в сочетании с ключевым словом, чтобы дать модулю локальное имя:

// app.js

import \* as math from "lib/math";

console.log('2π = ' + math.sum(math.pi, math.pi));

В модульной системе есть экспорт default, который может быть функцией. Чтобы импортировать это значение по умолчанию в модуле, вам необходимо только предоставить локальное имя (т.е. без деструктурирования):

// lib/my-fn.js

export default function() {

console.log('echo echo');

}

// app.js

import doSomething from 'lib/my-fn';

doSomething();

Пожалуйста, обратите внимание, что выражения import синхронны, но код модуля не выполняется, пока все зависимости не загрузятся.

## Классы

Классы – широко обсуждаемая особенность ES6. Некоторые считают, что они идут против природы прототипов JavaScript, в то время как другие думают, что они понижают входной барьер для начинающих и людей, приходящих из других языков, и что они помогают людям писать большие приложения. В любом случае, они являются частью ES6. Далее приведено очень краткое введение в классы JavaScript.

Классы построены вокруг ключевых слов class и constructor. Небольшой пример:

class Vehicle {

constructor(name) {

this.name = name;

this.kind = 'vehicle';

}

getName() {

return this.name;

}

}

// Создадим экземпляр

let myVehicle = new Vehicle('rocky');

Обратите внимание, что определение класса не является обычным объектом, следовательно, нет запятых между членами класса.

Чтобы создать экземпляр класса, вы должны использовать ключевое слово new. Для наследования базового класса, используйте extend:

class Car extends Vehicle {

constructor(name) {

super(name);

this.kind = 'car'

}

}

let myCar = new Car('bumpy');

myCar.getName(); // 'bumpy'

myCar instanceof Car; // true

myCar instanceof Vehicle; //true

Из любого конструктора или метода производного класса вы можете использовать super, чтобы получить доступ к базовому классу:

* чтобы вызвать родительский конструктор, используйте super();
* чтобы вызвать другой метод, используйте, например, super.getName().

Если вы хотите копать эту тему глубже, я рекомендую [«Классы в ECMAScript 6» от Dr. Axel Rauschmayer](http://www.2ality.com/2015/02/es6-classes-final.html).

## Символы

Символы – новый примитивный тип данных, как, например, Number и String. Вы можете использовать символы, чтобы создать уникальные идентификаторы для свойств объекта, или чтобы создать уникальные константы.

const MY\_CONSTANT = Symbol();

let obj = {};

obj[MY\_CONSTANT] = 1;

Обратите внимание, что набор пар ключ-значение с символами не возвращается методомObject.getOwnPropertyNames(), и они не видны в итерациях for...in, Object.keys() или JSON.stringify(). Этим они отличаются от обычных ключей на основе строк. Вы можете получить массив символов объекта с помощьюObject.getOwnPropertySymbols().

Символы, естественно, работают с const из-за своего неизменяемого характера:

const CHINESE = Symbol();

const ENGLISH = Symbol();

const SPANISH = Symbol();

switch(language) {

case CHINESE:

//

break;

case ENGLISH:

//

break;

case SPANISH:

//

break;

default:

//

break;

}

Вы можете добавить описание к символу. Вы не сможете использовать его для получения доступа к самому символу, но оно может быть полезно при отладке:

const CONST\_1 = Symbol('my symbol');

const CONST\_2 = Symbol('my symbol');

typeof CONST\_1 === 'symbol'; // true

CONST\_1 === CONST\_2; // false

ECMAScript 6: обзор и сравнение

Подробная статья на английском <http://es6-features.org/#Constants>

**Поддержка ECMAScript 5 и ECMAScript 6 Harmony в браузерах уже сейчас**

## **ECMAScript 5**

Это первое большое изменение спецификации, разработка которой шла около 10 лет. И, к сожалению, только недавно браузеры начали активное внедрение нового функционала. Коротко о том, что нового пришло с ECMAScript 5.

### **Array extras**

* Array.[forEach](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/forEach)
* Array.[map](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/map)
* Array.[every](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/every)
* Array.[filter](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/filter)
* Array.[some](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/some)
* Array.[indexOf](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/indexOf)
* Array.[lastIndexOf](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/lastIndexOf)
* Array.[isArray](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/isArray)

### **Object extras**

* Object.[create](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/create)
* Object.[keys](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/keys)
* Object.[defineProperty](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/defineProperty)
* Object.[defineProperties](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/defineProperties)
* Object.[getOwnPropertyDescriptor](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor)
* Object.[getOwnPropertyNames](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames)
* Object.[preventExtensions](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/preventExtensions)
* Object.[isExtensible](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/isExtensible)
* Object.[seal](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/seal)
* Object.[isSealed](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/isSealed)
* Object.[freeze](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/freeze)
* Object.[isFrozen](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/isFrozen)

Так же добавлена поддержка [native JSON](https://developer.mozilla.org/En/Using_native_JSON" \t "_blank), [getters](https://developer.mozilla.org/en/JavaScript/Reference/Operators/get" \t "_blank) и [setters](https://developer.mozilla.org/en/JavaScript/Reference/Operators/set" \t "_blank), метода [bind для Function](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Function/bind" \t "_blank) и [strict режим](https://developer.mozilla.org/en/JavaScript/Reference/Functions_and_function_scope/Strict_mode" \t "_blank).

На данный момент, только самые современные браузеры более-менее полно поддерживают ECMAScript 5. Среди них >= IE9, >= Firefox 4, >= Safari 5.1, >= Chrome 13, >= Opera 12. Полную таблицу поддержки спецификации можно [посмотреть здесь](http://kangax.github.com/es5-compat-table/).

## **ECMAScript 6 или Harmony**

В данный момент спецификация не закончена. На мой взгляд это самое большое, структурное и синтаксическое изменение языка за все время. Спецификация довольно объемная, поэтому я остановлюсь только на самом интересном для себя.

* [let](https://developer.mozilla.org/en/JavaScript/Reference/Statements/let) — точно так же как var ограничивает видимость переменных в пределах функции, let ограничивает видимость переменных в пределах блока.
* [const](https://developer.mozilla.org/en/JavaScript/Reference/Statements/const) — создание read-only констант.
* [Maps](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Map) и [WeakMap](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/WeakMap" \t "_blank) — для работы с объектами ключ/значение.

Кстати, CoffeeScript очень помог развитию Harmony, вот изменения в спецификации напрямую перешедшие из него:

* [Реструктуризующее присваивание (destructuring assignment)](http://coffeescript.org/#destructuring).
* Возможность указывать [значения параметров функции по-умолчанию](http://coffeescript.org/#literals).
* Операторы сравнения [is и isnt](http://coffeescript.org/" \l "operators" \t "_blank).

Так же расширение методов Function, String, Number, Object, RegExp, Math и [другие изменения](http://wiki.ecmascript.org/doku.php?id=harmony:proposals).

Поддержка в текущих браузерах достаточно слабая, однако в отдельных браузерах части спецификации уже реализованы. Полную таблицу поддержки спецификации можно [посмотреть здесь](http://kangax.github.com/es5-compat-table/es6/).

## **Все это здорово, но я хочу сейчас!**

Специально для тех кто хочет уже сейчас писать по спецификации и не думать работает это в их браузере или нет созданы два проекта:[es5-shim](https://github.com/kriskowal/es5-shim) и [es6-shim](https://github.com/paulmillr/es6-shim). Оба этих проекта реализуют функционал из спецификации недоступный текущему браузеру. Разумеется речи не идет о полной поддержки, однако большинство полезных новшеств уже работают хорошо.

**Таблица совместимости: ECMAScript 5 и ECMAScript 6**

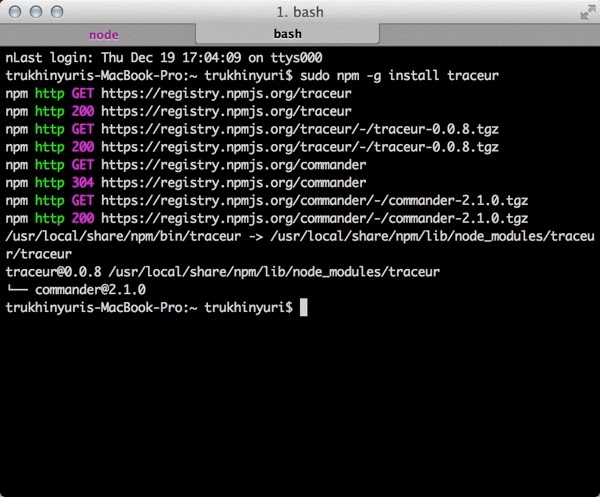
<https://kangax.github.io/compat-table/es6/>

**Способы использовать ECMAScript 6 с устаревшими браузерами:**

1) Можно пойти на REPL [страницу](http://google.github.io/traceur-compiler/demo/repl.html), вставить туда код на ES6 и получить результат компиляции ([что такое REPL](http://ru.wikipedia.org/wiki/REPL)).

2) Можно подключить traceur к вашей странице и компилировать код ES6 на лету (это будет неосознанно делать каждый, загружающий страницу в браузер и мягко говоря - это не самое оптимальное использование вычислительных ресурсов пользователей, хотя для использования с простыми примерами подойдет).

3) Так же можно воспользоваться offline компилятором traceur для node.js. Сначала необходимо установить node.js [к себе на машину](http://nodejs.org/) (или[в облако](http://infoboxcloud.ru/community/blog/cloudblog/7.html)). Далее ставим traceur командой sudo npm -g install traceur

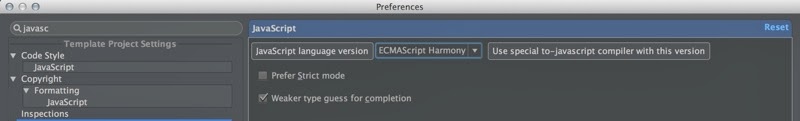


Вообщем-то и все. Теперь мы можем создать ES6 файл и скомпилировать его в ES5. Для компиляции необходимо запустить команду:

./traceur --out out/filename.js filename.js

Однако рановато нам еще компилировать. Для начала нужно написать ES6 код. Я для разработки пользуюсь [IntelliJ IDEA](http://www.jetbrains.com/idea/) и вам рекомендую (JavaScript поддерживает Ultimate версия, или [WebStorm](http://www.jetbrains.com/webstorm/), если вам от IDE нужен только Web).

В настройках IDE установим уровень Javascript – Harmony (кодовое имя ES6).



Напишем простейший пример просто для того, чтобы проверить, работает ли компилятор:

class HelloWorld {

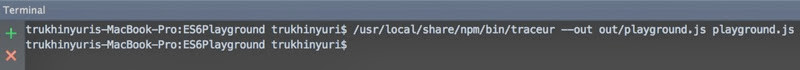
sayHello() { console.**log**('HelloWorld!'); }

}

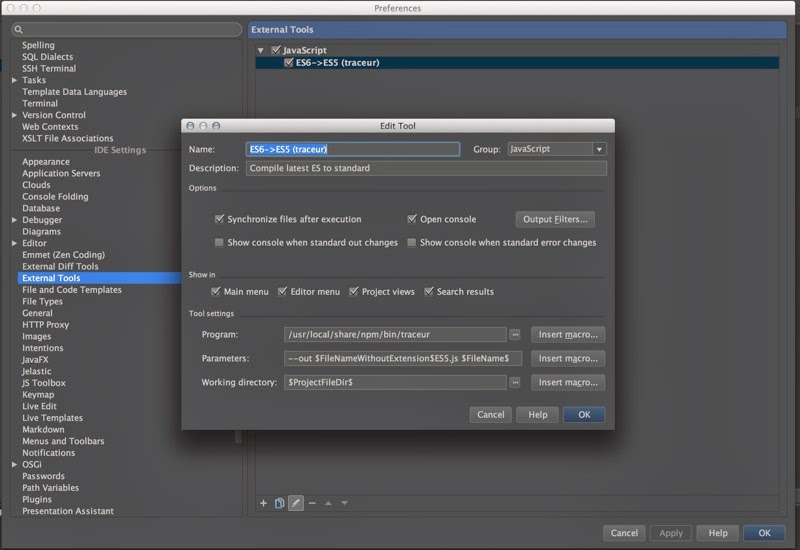
**var** itsMyWorld = **new** HelloWorld();

itsMyWorld.sayHello();

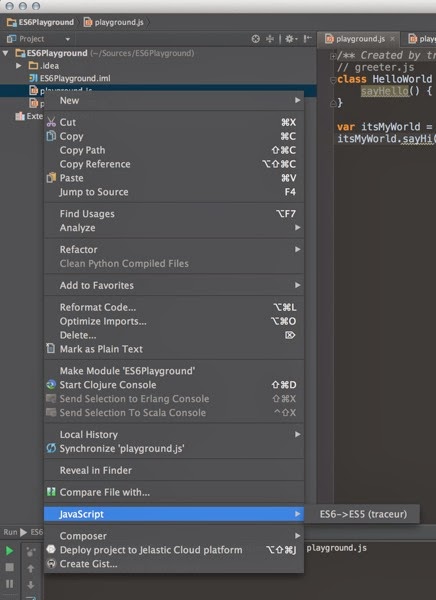
Далее есть несколько путей. Можно просто собрать файл в терминале, встроенном в IntelliJ IDEA:



 или можно написать небольшую автоматизацию, чтобы иметь в IntelliJ IDEA пункт в контекстном меню, собирающий ES6 в ES5. Вот так выглядит простейший способ для сборки 1 файла исходника ES6 в 1 файл результата ES5:



Теперь в контекстном меню IntelliJ появилась возможность компиляции файла ES6 в ES5:



Результат компиляции:

**var** HelloWorld = **function**() {

"use strict";

};

HelloWorld = ($traceurRuntime.createClass)(HelloWorld, {sayHello: **function**() {

"use strict";

console.**log**('HelloWorld!');

}}, {});

**var** itsMyWorld = **new** HelloWorld();

itsMyWorld.sayHello();

Как мы видим - это уже нормальный ES5.

Используя параметр —out с traceur можно собирать в 1 es5 файл не только 1 файл es6, но и множество. Так же можно строить [sourcemap](http://habrahabr.ru/post/148098/).

Таким образом уже сейчас можно учить ES6 и смотреть, как работает написанный код.

## Online ES6-песочницы

* [http://babeljs.io/repl/](http://babeljs.io/repl/#?experimental=true&playground=true&evaluate=true&loose=false&spec=false&code=%0Alet%20foo%20%3D%20(...bar)%20%3D%3E%20console.log(bar.join())%3B%0A%0Afoo(1%2C%202%2C%203)%3B)
* [http://es6.zloirock.ru/](http://es6.zloirock.ru/#%0Alet%20foo%20%3D%20(...bar)%20%3D%3E%20console.log(bar.join())%3B%0A%0Afoo(1%2C%202%2C%203)%3B)
* [http://www.es6fiddle.net/](http://www.es6fiddle.net/i6garqcl/)