**WHAT IS ECMA SCRIPT :**

JavaScript is a superset of ECMAScript scripting language. ECMAScript forms the language base for JavaScript, JScript and ActionScript

ECMAScript6 is also called as “ES6″, “Harmony” and “ES.next”.

**ECMAScript** is a Standard for a scripting languages. Languages like Javascript are based on the **ECMAScript** standard. ECMA Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). ECMA means European Computer Manufacturer's Association

**ECMAScript 6**, also known as **ECMAScript** 2015, is the latest version of the**ECMAScript** standard. ES6 is a significant update to the language, and the first update to the language since ES5 was standardized in 2009. Implementation of these features in major JavaScript engines is underway now.

**ECMAscript is a standard .**

**A standard guarantees that if somebody writes a code following the rules of the standard then the code would run fine in a browser(in case of javascript) if the browser implements the same standard.**

*Why do we need a standard?*

When Javascript was first created by Netscape then there was a war going on between all the browser vendors in the market . Microsoft implemented its own version of javascript in internet explorer . Similarly other browser vendors implemented their own versions.

All this created a huge problem for the developers.One code ran fine on Netscape but was a total waste on Internet Explorer . **To solve cross browser compatibilty Javascript was standardised by ECMA international ; hence the name ECMAscript.**

All browers agreed to implement ECMAscript(though it took a lot of time).

The result is that now the same code runs fine in all browsers.

The exception is DOM(Document object model) . DOM is still interpreted in different ways in different browsers . However there is good news . Slowly all browsers are adhering to the same DOM standard .

Categorizing ES6 Features

All the new features of ES6 can be categorized in 7 categories. Here is the list of categories:

1. Variables and Parameters
2. Classes
3. Proxies
4. Build-in Objects and Prototype based features
5. Promises
6. Reflect API
7. Modules
8. Iterators and Generators

You can also use caniuse.com to find the support of various ES6 features on various different browsers.

### Running ECMAScript 6 in an Incompatible Browser

If you are writing ES6 for your website on development phase then you can embed the [Traceur](https://github.com/google/traceur-compiler" \t "_blank) compiler in your webpages which will compile the ES6 to simple browser supportable JavaScript code on the fly in the browser.

Here is how to embed Traceur in your website

<!doctype html>  
<[html](http://december.com/html/4/element/html.html)>  
    <[head](http://december.com/html/4/element/head.html)>...</[head](http://december.com/html/4/element/head.html)>  
    <[body](http://december.com/html/4/element/body.html)>  
        ...  
  
        <!-- Load Traceur Compiler -->  
        <[script](http://december.com/html/4/element/script.html) src="https://google.github.io/traceur-compiler/bin/traceur.js"></[script](http://december.com/html/4/element/script.html)>  
        <!-- Bootstrap.js finds script tags with type module and compiler them using the interfaces provided by traceur.js -->  
        <[script](http://december.com/html/4/element/script.html) src="https://google.github.io/traceur-compiler/src/bootstrap.js"></[script](http://december.com/html/4/element/script.html)>  
        <[script](http://december.com/html/4/element/script.html) type="module">  
            //Place ES6 code here

if(true)  
{  
    let x = 12;  
    alert(x); *//alert's 12*  
}  
  
alert(x); *//x is undefined here*

        </[script](http://december.com/html/4/element/script.html)>  
    </[body](http://december.com/html/4/element/body.html)>  
</[html](http://december.com/html/4/element/html.html)>

<https://babeljs.io/> USE THIS TO CHECK THE CONVERSION FROM ES5 TO ES6

BABEL IS JAVA SCRIPT COMPILER

“let” keyword was introduced in ES6. It lets you define block scope(bracket scope) variables in JavaScript. Initially JavaScript only supported function scope and global scope variables.

Here is an example code

if(true)  
{  
    let x = 12;  
    alert(x); *//alert's 12*  
}  
  
alert(x); *//x is undefined here*

“let” keyword limits the variables accessibility upto a block, statement or expression.

“const” keyword was introduced in ES6. It lets you define read only variables using JavaScript. Variables created using “const” are block scoped(or bracket scoped). Redeclaring a “const” variable in the same scope throws an error.

Here is an code example:

const x = 12;  
  
*//an constant 'x' is already available in this scope therefore the below line throws an error when you are try to create a new x variable.*  
const x = 13;  
  
if(true)  
{  
    *//an constant 'x' is available in this scope but not defined in this scope therefore the below line will not throw error instead define a new "x" inside this scope.*  
    const x = 13;  
      
    *//here 'y' is available inside this scope not outside this scope*  
    const y = 11;  
}  
  
*//here creating a new 'y' will not throw an error because no other 'y' is available in this scope(i.e., global scope)*  
const y = 12;

Just remember that in a scope you cannot redeclare or change value of an “const” variable if a variable with same name is already available for access in that scope.

# Default Function Arguments Values in JavaScript

ES6 provides a new syntax that can be used to define default values to function parameters:

function myFunction(x = 1, y = 2, z = 3)  
{  
     console.log(x, y, z); *// Output "6 7 3"*  
}  
myFunction(6,7);

Also, passing undefined is considered as missing an argument. Here is an example  
to demonstrate this:

   function myFunction(x = 1, y = 2, z = 3)  
   {  
     console.log(x, y, z); *// Output "1 7 9"*  
   }  
   myFunction(undefined,7,9);

Defaults can also be expressions. Here is an example to demonstrate this:

   function myFunction(x = 1, y = 2, z = 3 + 5)  
   {  
     console.log(x, y, z); *// Output "6 7 8"*  
   }  
   myFunction(6,7);

# JavaScript “…” Operator

ES6 introduced “…” operator which is also called as spread operator. When “…” operator is applied on an array it expands the array into multiple variables in syntax wise. And when its applied to an function argument it makes the function argument behave like array of arguments.

We can use spread operator to take indefinite number of arguments.

Here is an example code of how to use this operator

*//args variable is an array holding the passed function arguments*  
function function\_one(...args)  
{     
    console.log(args);  
    console.log(args.length);  
}  
  
function\_one(1, 4);  
function\_one(1, 4, 7);  
function\_one(1, 4, 7, 0);  
  
  
function function\_two(a, b, ...args)  
{  
    console.log(args);  
    console.log(args.length);  
}  
  
*//"args" holds only 7 and 9*  
function\_two(1, 5, 7, 9);

# JavaScript “class” Keyword

ECMAScript 6 introduced “class” keyword to define classes in JavaScript. Earlier to ES6, we had to use [constructor function](http://qnimate.com/understanding-object-oriented-javascript/#Creating_Objects_In_JavaScript).

Here is an example code on how to define classes and then how to create objects of those class types i.e., instances of classes.

class Student  
{  
    *//constructor of the class*  
    constructor(name, age)  
    {  
        *//"this" points to the current object*  
        this.name = name;  
  
        this.\_age = age;  
    }  
  
    *//member function*  
    getName()  
    {  
        return this.name;  
    }  
  
    setName(name)  
    {  
        this.name = name;  
    }  
  
    *//getters and setters make a function accessible like a variable. They are used as wrappers around other variables.*  
    set age(value)  
    {  
        this.\_age = value;  
    }  
  
    get age()  
    {  
        return this.\_age;  
    }  
}  
  
*//class person inherits student class*  
class Person extends Student  
{  
    constructor(name, age, citizen)  
    {  
        *//this points to the person class*  
        this.citizen = citizen;  
  
        *//call constructor of super class. "super" is an pointer to the super class object*  
        super(name, age);  
    }  
  
    getCitizen()  
    {  
        return this.citizen;  
    }  
  
    *//overriding*  
    getName()  
    {  
        *//we are calling the super class getName function*  
        return super.getName();  
    }  
}  
  
*//instance of student class*  
var stud = new Student("Narayan", 21);  
  
*//instance of person class*  
var p = new Person("Narayan Prusty", 21, "India");  
  
stud.age = 12; *//executes setter*  
console.log(stud.age); *//executes getter*

# JavaScript Arrow “=>” Function

ECMAScript 6 provides a new way to create functions which just contain one line of statement. This new type of function is called lambda or arrow function.

Here is how to create a arrow function

*//sum is the function name*  
*//x and y are function parameters*  
var sum = (x, y) => x + y;  
  
console.log(sum(2, 900)); *//902*

You can also write multiple statements in an arrow function but arrow functions are mostly used in replacement of single statement functions. Here is code example of multiple statements in an arrow function

var sum = (x, y) => {  
    x = x + 10;  
    y = y + 10;  
    return x + y;  
}  
  
console.log(sum(10, 10)); *//40*

As arrow function actually returns a regular JavaScript function object so they can be used wherever we use regular JavaScript function object. For example, they can be used as callback.

function sum(p, q)  
{  
    console.log(p() + q()); *//87*  
}  
  
sum(a => 20 + 10, b => 1 + 56); *//here we are passing two function objects*

One last and most important feature about arrow function is that the “this” pointer inside an asynchronously executed arrow function points to the scope inside which it was passed as callback. A regular function’s this pointer points to global scope when executed asynchronously.

window.age = 12;  
  
function Person(){  
  this.age = 34;  
  
  setTimeout(() => {  
    console.log(this.age); *//34*  
  }, 1000);  
  
  setTimeout(function(){  
    console.log(this.age); *//12*  
  }, 1000);    
}  
  
var p = new Person();

# JavaScript “for of” Loop

for of loop was introduced in ES6 which allows you to easily iterate over elements of an collection.for of iterates over the values of elements of a collection not the keys. A collection can be an array, set, list, custom collection object etc.

Earlier to ES6 we had to use for loop or Array’s foreach loop to walkthrough elements of an collection. ES6 introduced a new way for iteration.

An iterator is an construct that lets us visit or walkthrough every element of an collection.

Iterating an Array using “for of”

Here is an example code on how to iterate an array using for of loop

var array = [1, 3, 5, 7, 9];  
  
*//'i' references to the values of the array indexes*  
for(var i of array)  
{  
    console.log(i); *//1, 3, 5, 7, 9*  
}

# JavaScript “Set” Object

JavaScript “Set” object is a collection of unique keys. Keys are object references or primitive types.

Arrays can store duplicate values but Sets don’t store duplicate keys, this is what makes it different from arrays.

Here is code example on how to create a set object, add keys, delete keys, find size etc.

*//create a set*  
var set = new Set();  
  
*//add three keys to the set*  
set.add({x: 12});  
set.add(44);  
set.add("text");  
  
*//check if a provided key is present*  
console.log(set.has("text"));  
  
*//delete a key*  
set.delete(44);  
  
*//loop through the keys in an set*  
for(var i of set)  
{  
    console.log(i);  
}  
  
*//create a set from array values*  
var set\_1 = new Set([1, 2, 3, 4, 5]);   
  
*//size of set*  
console.log(set\_1.size); *//5*  
  
*//create a clone of another set*  
var set\_2 = new Set(set.values());

# Creating JavaScript Modules

n nutshell JavaScript modules are just a way of packaging related JavaScript code in its own scope which can be consumed by other JavaScript programs.

In this tutorial I will show the different ways to create JavaScript modules.

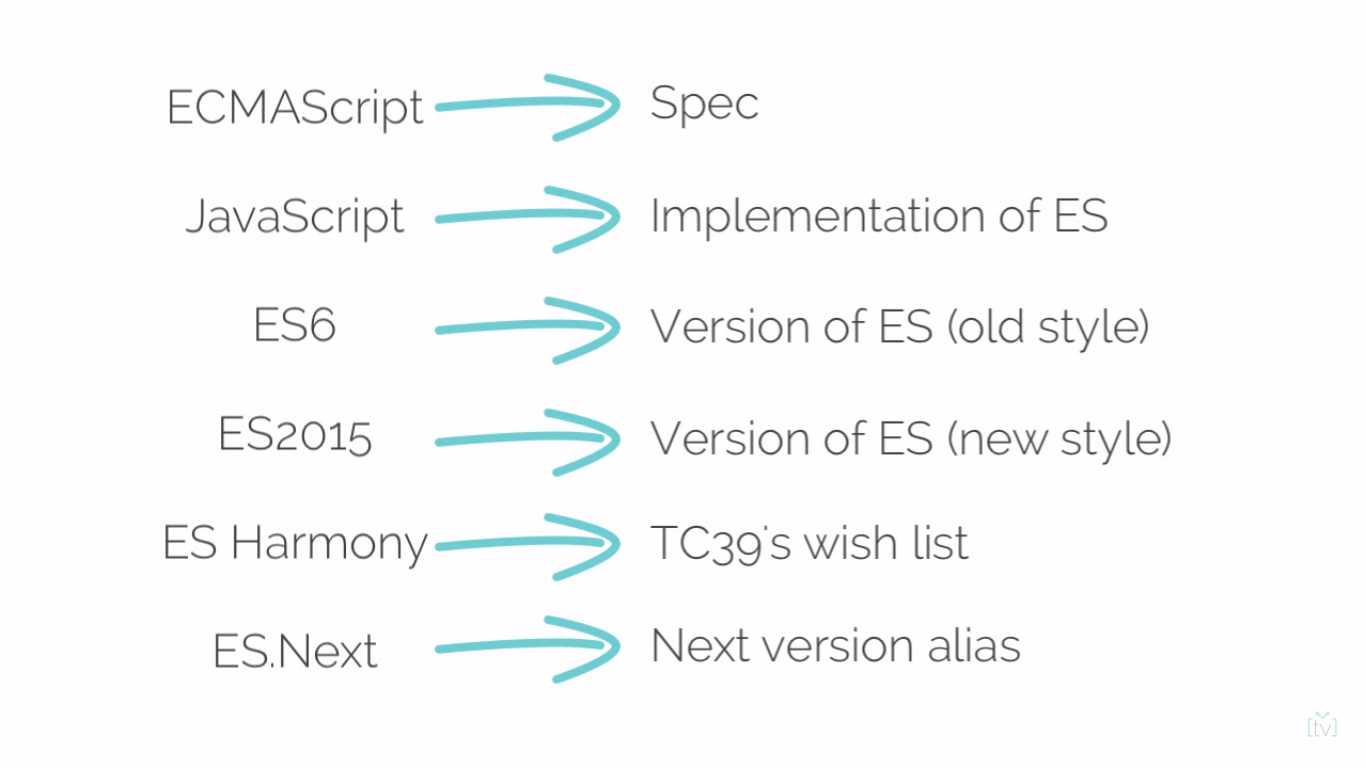
Immediately-Invoked Function Expression

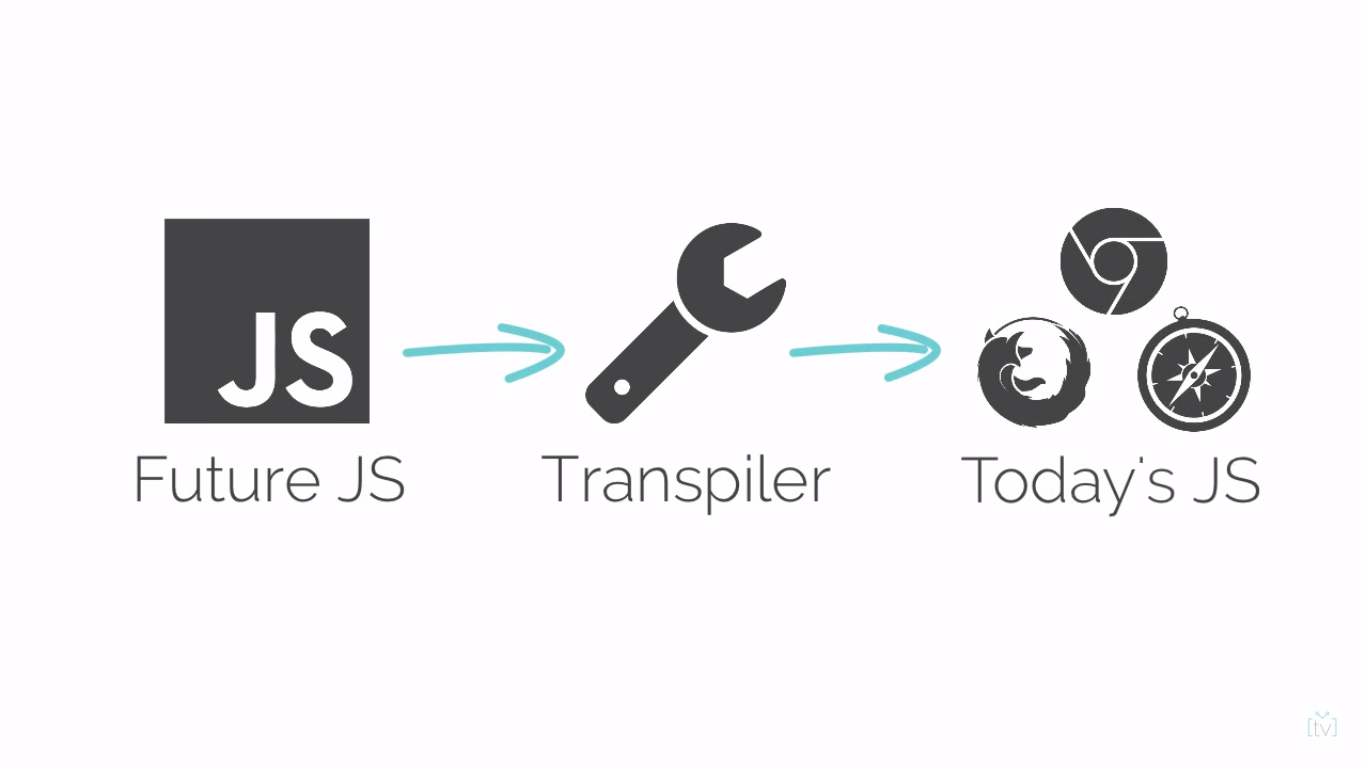
IIFE is a anonymous function which invokes itself. Creating modules using IIFE is the oldest way of creating JavaScript modules. Here is example code on how to create modules using IIFE

You can place the below code in an separate file to make it an distributable module or place it directly in your main program for your own use.

*//Module Starts*  
  
(function(window){  
  
    var sum = function(x, y){  
        return x + y;  
    }  
  
    var sub = function(x, y){  
        return x - y;  
    }  
  
    var math = {  
        findSum: function(a, b){  
            return sum(a,b);  
        },  
  
        findSub: function(a, b){  
            return sub(a, b);  
        }  
    }  
  
    window.math = math;  
  
})(window);  
  
*//Module Ends*  
  
  
console.log(math.findSum(1, 2)); *//3*  
console.log(math.findSub(1, 2)); *//-1*

Here sum and sub function objects remain in memory but there is no way the program which uses the module can access them therefore we prevented any chances of overriding global variables. The only thing thats available global is math object which exposes all the functionality of the module by hiding their implementation.



  Install Babel transpiler tool

<https://babeljs.io/docs/setup/>

<https://nodejs.org/en/>

run es6 online

<https://codepen.io/bradleyboy/pen/vEeENy>

<https://jsbin.com/?html,css,js,output>

## 2Installation

**Copy**

npm install --save-dev babel-core

## 3Usage

**Try**

**Copy**

require("babel-core").transform("code", options);

For full documentation on the Babel API see the [usage docs](https://babeljs.io/docs/usage/api/).

## 4Create .babelrc configuration file

Great! You've configured Babel but you haven't made it actually do anything. Create a [.babelrc](https://babeljs.io/docs/usage/babelrc) config in your project root and enable some [plugins](https://babeljs.io/docs/plugins).

To start, you can use the [env preset](https://babeljs.io/docs/plugins/preset-env/), which enables transforms for ES2015+

**Copy**

npm install babel-preset-env --save-dev

In order to enable the preset you have to define it in your .babelrc file, like this:

**Copy**

{

"presets": ["env"]

}

**Note**: Running a Babel 6.x project using npm 2.x can cause performance problems because of the way npm 2.x installs dependencies. This problem can be eliminated by either switching to npm 3.x or running npm 2.x with the [dedupe](https://docs.npmjs.com/cli/dedupe) flag. To check what version of npm you have run

**Copy**

npm --version

**HOW TO INSTALL ECMA6**

## Trying out ECMAScript 6

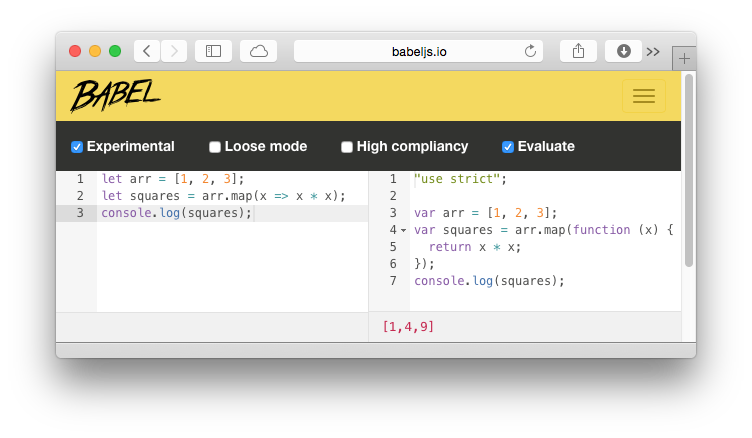
There are three simple ways to play with ES6:

1. **Web browser:** use [the online Babel REPL](http://babeljs.io/repl/), an interactive playground that compiles ES6 to ES5. There is nothing to install with this option.
2. **Command line:** use babel-node, a version of the Node.js executable that understands ES6 (and internally compiles it to ES5). It can be installed via npm.
3. **Various JavaScript engines:** check [the ES6 compatibility table by kangax](https://kangax.github.io/compat-table/es6/) to find out which ES6 features are supported natively where.

**The Babel REPL**

The Babel REPL has four major sections:

* The top left pane contains the ES6 source code.
* The bottom left pane shows syntax errors discovered in the ES6 code.
* The top right pane contains the ES5 code that the ES6 code is compiled to.
* The bottom right pane shows output produced via console.log().



### babel-node

The babel-node executable can be installed via npm:

$ npm install --global babel

You can use it in the same way as you would the Node.js executable node. Like node, an interactive REPL is started like this:

$ babel-node

Once you are in that REPL, you can execute ES6 code:

> let arr = [1, 2, 3];

> arr.map(x => x \* x)

[ 1, 4, 9 ]

Note that [babel-node does not currently support multi-line input](https://github.com/babel/babel/issues/1741).

The Babel website has [more information the Babel CLI tools](http://babeljs.io/docs/usage/cli/).

The remaining sections of this post describe ES6 features that are easy to adopt.

## From var to let/const

ES6 has two new ways to declare variables:

* let is (roughly) a block-scoped version of var.
* const is like let, but creates constants, variables whose values can’t be changed.

You can generally replace each var with a let or a const. But you shouldn’t do so blindly, because the different kind of scoping can change how code behaves. As an example, look at the following ES5 code:

**var** x = 3;

**function** **func**(randomize) {

**if** (randomize) {

**var** x = **Math**.random(); // (A) scope: whole function

**return** x;

}

**return** x; // accesses the x from line A

}

func(false); // undefined

That func() returns undefined may be surprising. You can see why if you rewrite the code so that it more closely reflects what is actually going on:

**var** x = 3;

**function** **func**(randomize) {

**var** x;

**if** (randomize) {

x = **Math**.random();

**return** x;

}

**return** x;

}

func(false); // undefined

If you replace var with let in the initial version, you get different behavior:

**let** x = 3;

**function** **func**(randomize) {

**if** (randomize) {

**let** x = **Math**.random();

**return** x;

}

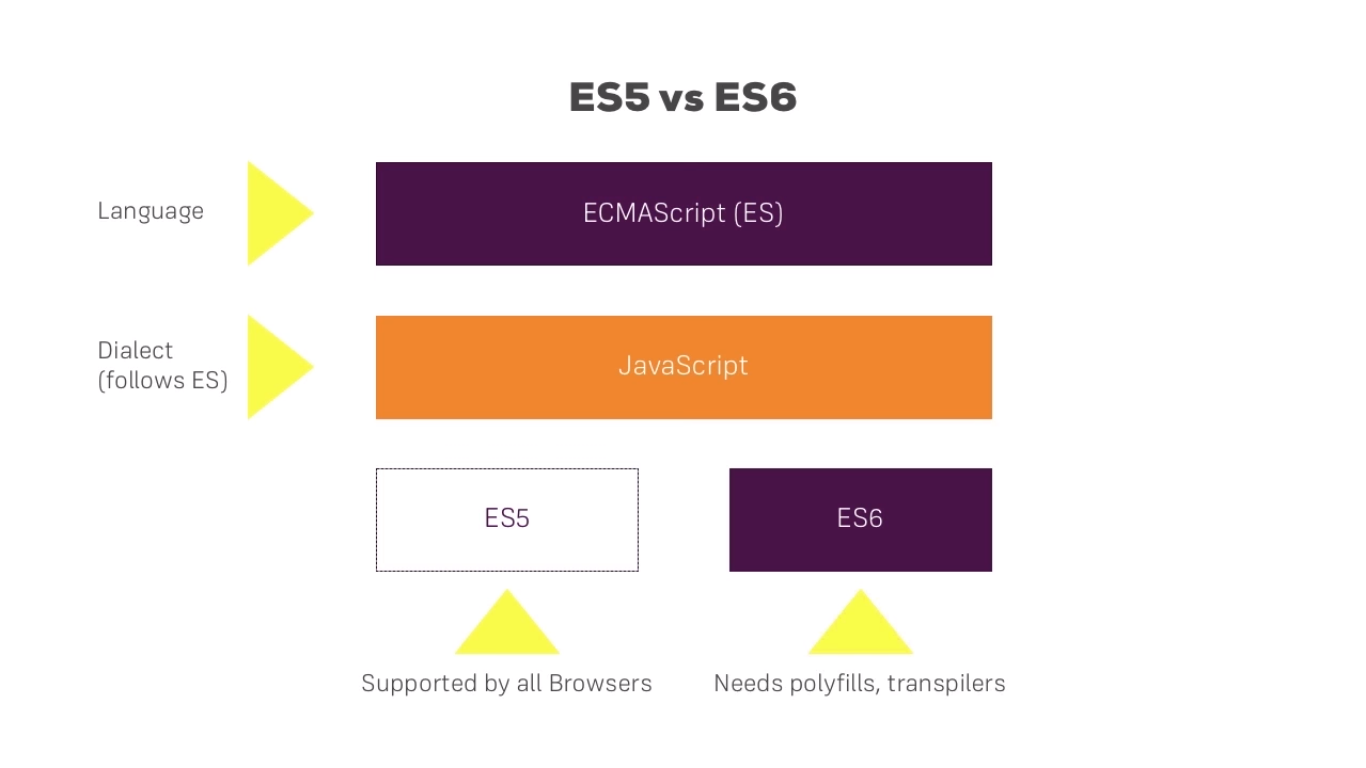
**return** x;

}

func(false); // 3

Thus, blindly replacing var with let or const is risky. My advice is:

* Only use let/const in new code.
* Leave old code as is or refactor it carefully.



<https://jsbin.com/porayoduri/edit?css,js,console,output>

## ES6 FEATURES

There are a lot of new features, but for now we'll cover a subset of the new features below.

* [const and let](https://learn.co/lessons/introduction-to-es6#block-scoping)
* [Classes](https://learn.co/lessons/introduction-to-es6#classes-use-strict)
* [Arrow Functions](https://learn.co/lessons/introduction-to-es6#arrow-functions)
* [Promises](https://learn.co/lessons/introduction-to-es6#promises)
* [Object Literal Extensions](https://learn.co/lessons/introduction-to-es6#object-literal-extensions)
* [Spread Operator](https://learn.co/lessons/introduction-to-es6#spread-operator)
* [Template Strings](https://learn.co/lessons/introduction-to-es6#template-strings)
* [Destructuring](https://learn.co/lessons/introduction-to-es6#destructuring)

### BLOCK SCOPING

#### LET ("USE STRICT")

The keyword [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let) is a new way of declaring local variables. How does it differ from good ol' var? Variables declared with let have block-level scope:

*// https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#Scoping\_rules*

**function** letTest() {

**let** x **=** 31;

**if** (**true**) {

**let** x **=** 71; *// different variable*

console.log(x); *// 71*

}

console.log(x); *// 31*

}

Notice how x declared outside of the if block differs from the x declared inside the block. Block-level scope means that the variable is available only in the block (if, for, while, etc.) in which it is defined; it differs from JavaScript's normal function-level scope, which restricts the variable to the function in which it is defined (or global/window if it's a global variable).

#### CONST

The keyword [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const) does not require strict mode. Like let, it declares a variable with block-level scope; additionally, it prevents its variable identifier from being reassigned.

That means that the following will throw an error:

**const** myVariable **=** 1;

myVariable **=** 2; *// syntax error*

However, this does not mean that the variable's value is immutable — the value can still change.

**const** myObject **=** {};

*// this works*

myObject.myProperty **=** 1;

*// 1*

console.log(myObject.myProperty)

### CLASSES ("USE STRICT")

"Wait," you say. "JavaScript has prototypal, not class-based, inheritance."

You're still right. But [classes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes) in JavaScript are awesome, and you'll be seeing them increasingly as ES6 adoption increases.

Consider the following simple example, based largely on the examples from MDN. We want to create a Polygon prototype and inherit from it. We'll start with ES5:

**function** Polygon(height, width) {

**this**.height **=** height;

**this**.width **=** width;

}

Cool, so we got to the ES6 class constructor, which works just like a constructor in ES5. But now how do we implement the area() getter? Well, not very nicely — let's back up and rewrite what we just wrote:

**function** Polygon(height, width) {

**this**.height **=** height;

**this**.width **=** width;

*// :(*

**this**.area **=** **this**.calcArea();

}

Polygon.prototype.calcArea **=** **function**() {

**return** **this**.height **\*** **this**.width;

};

**const** rectangle **=** **new** Polygon(10, 5);

console.log(rectangle.area);

Okay, so that worked, but look at how difficult it is to reason about. We have to plan in advance for the properties that we want to set, and area is not calculated dynamically — it's set when the Polygon is instantiated and then forgotten about, so if somehow a Polygon's height and width changed, its area would need to be updated separately. Gross.

Moreover, extending this ES5 Polygon is a bit onerous:

**function** Square(sideLength) {

Polygon.call(**this**, sideLength, sideLength);

}

Square.prototype **=** **new** Polygon();

**const** square **=** **new** Square(5);

*// Polygon { height: 5, width: 5, area: 25 }*

console.log(square);

Well, that seems like it just about works. But what if we check the variable square's constructor?

*// [Function: Polygon]*

square.constructor;

That's not right. It should be [Function: Square]. We can fix it, though:

Square.prototype.constructor **=** Square;

**const** square2 **=** **new** Square(6);

*// { [Function: Square] constructor: [Circular] }*

square2.constructor

Eh, close enough?

(Note: The point here isn't to land on a perfect approach to object inheritance in JavaScript, it's to show that such a goal isn't feasible and won't be achieved in a nice way.)

Now let's try with ES6:

**class** Polygon {

constructor(height, width) {

**this**.height **=** height;

**this**.width **=** width;

}

*// whaaaaat -- getters!*

get area() {

**return** **this**.calcArea();

}

calcArea() {

**return** **this**.height **\*** **this**.width;

}

}

**const** rectangle **=** **new** Polygon(10, 5);

console.log(rectangle.area);

Let's extend it:

**class** Square **extends** Polygon {

constructor(sideLength) {

**super**(sideLength, sideLength)

}

}

**const** mySquare **=** **new** Square(5);

*// Square { height: 5, width: 5 }*

mySquare;

*// [Function: Square]*

mySquare.constructor;

*// 25*

mySquare.area;

Whoa. That was easy.

### ARROW FUNCTIONS

[Arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) provide not only a terser way to define a function but also lexically bind the current *this* value. This ain't your grandpa's JS.

**const** greet **=** (greeting, person) **=>** {

**return** greeting **+** ', ' **+** person **+** '!';

};

*// 'Hello, Marv'*

greet('Hello', 'Marv');

**var** a **=** [

'Hydrogen',

'Helium',

'Lithium',

'Beryl­lium'

];

*// compare this implementation...*

**var** a2 **=** a.map(**function**(s){ **return** s.length });

*// ... to this implementation with the fat arrow*

**var** a3 **=** a.map(s **=>** s.length);

Fat arrows also have implicit returns — the following are equivalent:

**var** a3 **=** a.map(s **=>** s.length);

**var** a4 **=** a.map(s **=>** {

**return** s.length;

});

If the function only accepts one argument, parentheses are optional:

*// this...*

**var** a3 **=** a.map(s **=>** s.length);

*// ... is the same as this*

**var** a3 **=** a.map((s) **=>** s.length);

If there are zero or two or more arguments, though, you must use parens:

**var** evens **=** [1, 2, 3, 4].reduce((memo, i) **=>** {

**if** (i **%** 2 **===** 0) {

memo.push(i)

}

**return** memo;

}, []);

### PROMISES

[Promises](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) offer a new way of handling asynchronicity.

We'll cover them in greater detail in a later lesson, but for now know that Promise is available globally in Node.js.

Also know that it's awesome.

**const** promise **=** **new** Promise((resolve, reject) **=>** {

**return** someIntenseTask().then(result **=>** {

**if** (result.success) {

**return** resolve(result)

}

**return** reject(result.error)

})

})

promise.then(result **=>** {

**return** doSomething(result);

}).**catch**(error **=>** handleError(error))

### OBJECT LITERAL EXTENSIONS

ES6 gives us a number of handy [new ways to deal with objects](https://github.com/lukehoban/es6features#enhanced-object-literals). They're features that you either wish JavaScript had, or ones you didn't know you needed.

**const** prop **=** **function**() {

**return** "I'm a prop!";

}

**const** myObj **=** {

*// computed (dynamic) property names*

*// methods*

shout() {

**return** 'AH!'

},

*// short for `prop: prop`*

prop

}

*// 'something'*

myObj.foobar

*// "I'm a prop!"*

myObj.prop()

*// 'AH!'*

myObj.shout()

### SPREAD OPERATOR

The [spread operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator) — ... — is unassuming but incredibly powerful.

We can use it for arrays:

**const** a **=** [1, 2, 3]

**const** b **=** [0, ...a, 4, 5]

*// [0, 1, 2, 3, 4, 5]*

b

functions:

**function** printArgs() {

*// recall that every function gets an `arguments`*

*// object*

console.log(arguments);

}

*// using `a` from above*

*// [1, 2, 3]*

printArgs(...a);

### TEMPLATE STRINGS

[Template strings](https://nodejs.org/en/docs/es6/) in ES6 are most commonly used for string interpolation. Instead of writing:

**var** foo **=** 'bar';

**var** sentence **=** 'I went to the ' **+** foo **+** ' after working in ES5 for too long.';

we can now write:

**var** foo **=** 'bar';

**var** sentence **=** `I went to the ${foo} after working in ES5 for too long.`;

and we'll get the same result.

You can also use tagged template literals to perform more advanced manipulation:

A tag is simply a function whose first argument is an array of strings and whose subsequent arguments are the values of the substitution expressions (the things in ${}).

Here's the example from [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals#Tagged_template_literals):

**var** a **=** 5;

**var** b **=** 10;

**function** tag(strings, ...values) {

console.log(strings[0]); *// "Hello "*

console.log(strings[1]); *// " world "*

console.log(values[0]); *// 15*

console.log(values[1]); *// 50*

**return** "Bazinga!";

}

tag`Hello ${ a **+** b } world ${ a **\*** b }`;

*// "Bazinga!"*

### DESTRUCTURING

Destructuring makes it easier than ever to pull values out of objects and arrays and store them in variables. We destructure an array by putting our new variable names at the corresponding index and an object by giving our variable the same name as the key we are interested in.

**const** [a, b] **=** [1, 2];

*// a === 1 && b === 2*

**const** { a, b } **=** { a: 1, b: 2 }

*// a === 1 && b === 2*

To see what other amazing things we can to with destructuring, check out the [docs](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment).

## CONCLUSION

**More information:** chapter “[Variables and scoping](http://exploringjs.com/es6/ch_variables.html)” in “Exploring ES6”.

## From IIFEs to blocks

In ES5, you had to use an IIFE if you wanted to keep a variable local:

(**function** () { // open IIFE

**var** tmp = ···;

···

}()); // close IIFE

**console**.log(tmp); // ReferenceError

In ECMAScript 6, you can simply use a block and a let declaration:

{ // open block

**let** tmp = ···;

···

} // close block

**console**.log(tmp); // ReferenceError

**More information:** section “[Avoid IIFEs in ES6](http://exploringjs.com/es6/ch_callables.html#sec_iifes-in-es6)” in “Exploring ES6”.

## From concatenating strings to template literals

With ES6, JavaScript finally gets literals for string interpolation and multi-line strings.

### String interpolation

In ES5, you put values into strings by concatenating those values and string fragments:

**function** **printCoord**(x, y) {

**console**.log('('+x+', '+y+')');

}

In ES6 you can use string interpolation via template literals:

**function** **printCoord**(x, y) {

**console**.log(`(${x}, ${y})`);

}

### Multi-line strings

Template literals also help with representing multi-line strings.

For example, this is what you have to do to represent one in ES5:

**var** HTML5\_SKELETON =

'<!doctype html>\n' +

'<html>\n' +

'<head>\n' +

' <meta charset="UTF-8">\n' +

' <title></title>\n' +

'</head>\n' +

'<body>\n' +

'</body>\n' +

'</html>\n';

If you escape the newlines via backslashes, things look a bit nicer (but you still have to explicitly add newlines):

**var** HTML5\_SKELETON = '\

<!doctype html>\n\

<html>\n\

<head>\n\

<meta charset="UTF-8">\n\

<title></title>\n\

</head>\n\

<body>\n\

</body>\n\

</html>';

ES6 template literals can span multiple lines:

**const** HTML5\_SKELETON = `

<!doctype html>

<html>

<head>

<meta charset="UTF-8">

<title></title>

</head>

<body>

</body>

</html>`;

(The examples differ in how much whitespace is included, but that doesn’t matter in this case.)

**More information:** chapter “[Template literals and tagged templates](http://exploringjs.com/es6/ch_template-literals.html)” in “Exploring ES6”.

## From function expressions to arrow functions

In current ES5 code, you have to be careful with this whenever you are using function expressions. In the following example, I create the helper variable \_this (line A) so that the this of UiComponent can be accessed in line B.

**function** **UiComponent** {

**var** \_this = **this**; // (A)

**var** button = **document**.getElementById('myButton');

button.addEventListener('click', **function** () {

**console**.log('CLICK');

\_this.handleClick(); // (B)

});

}

UiComponent.prototype.handleClick = **function** () {

···

};

In ES6, you can use arrow functions, which don’t shadow this (line A, lexical *this*):

**class** **UiComponent** {

**constructor**() {

**let** button = **document**.getElementById('myButton');

button.addEventListener('click', () => {

**console**.log('CLICK');

**this**.handleClick(); // (A)

});

}

handleClick() {

···

}

}

Arrow functions are especially handy for short callbacks that only return results of expressions.

In ES5, such callbacks are relatively verbose:

**var** arr = [1, 2, 3];

**var** squares = arr.map(**function** (x) { **return** x \* x });

In ES6, arrow functions are much more concise:

**let** arr = [1, 2, 3];

**let** squares = arr.map(x => x \* x);

When defining parameters, you can even omit parentheses if the parameters are just a single identifier. Thus: (x) => x \* x and x => x \* x are both allowed.

**More information:** chapter “[Arrow functions](http://exploringjs.com/es6/ch_arrow-functions.html)” in “Exploring ES6”.

## Handling multiple return values

Some functions or methods return multiple values via arrays or objects. In ES5, you always need to create intermediate variables if you want to access those values. In ES6, you can avoid intermediate variables via destructuring.

### Multiple return values via arrays

exec() returns captured groups via an Array-like object. In ES5, you need an intermediate variable (matchObj in the example below), even if you are only interested in the groups:

**var** matchObj =

/^(\d\d\d\d)-(\d\d)-(\d\d)$/

.exec('2999-12-31');

**var** year = matchObj[1];

**var** month = matchObj[2];

**var** day = matchObj[3];

In ES6, destructuring makes this code simpler:

**let** [, year, month, day] =

/^(\d\d\d\d)-(\d\d)-(\d\d)$/

.exec('2999-12-31');

The empty slot at the beginning of the Array pattern skips the Array element at index zero.

### Multiple return values via objects

The method Object.getOwnPropertyDescriptor() return a property descriptors, an object that holds multiple values in its properties.

In ES5, even if you are only interested in the properties of an object, you still need an intermediate variable (propDesc in the example below):

**var** obj = { foo: 123 };

**var** propDesc = **Object**.getOwnPropertyDescriptor(obj, 'foo');

**var** writable = propDesc.writable;

**var** configurable = propDesc.configurable;

**console**.log(writable, configurable); // true true

In ES6, you can use destructuring:

**let** obj = { foo: 123 };

**let** {writable, configurable} =

**Object**.getOwnPropertyDescriptor(obj, 'foo');

**console**.log(writable, configurable); // true true

{writable, configurable} is an abbreviation for:

{ writable: writable, configurable: configurable }

**More information:** chapter “[Destructuring](http://exploringjs.com/es6/ch_destructuring.html)” in “Exploring ES6”.

## From for to forEach() to for-of

Prior to ES5, you iterated over Arrays as follows:

**var** arr = ['a', 'b', 'c'];

**for** (**var** i=0; i<arr.length; i++) {

**var** elem = arr[i];

**console**.log(elem);

}

In ES5, you have the option of using the Array method forEach():

arr.forEach(**function** (elem) {

**console**.log(elem);

});

A for loop has the advantage that you can break from it, forEach() has the advantage of conciseness.

In ES6, the for-of loop combines both advantages:

**let** arr = ['a', 'b', 'c'];

**for** (**let** elem **of** arr) {

**console**.log(elem);

}

If you want both index and value of each array element, for-of has got you covered, too, via the new Array method entries() and destructuring:

**for** (**let** [index, elem] **of** arr.entries()) {

**console**.log(index+'. '+elem);

}

**More information:** section “[The for-of loop](http://exploringjs.com/es6/ch_iteration.html#sec_for-of-loop)” in “Exploring ES6”.

## Handling parameter default values

In ES5, you specify default values for parameters like this:

**function** **foo**(x, y) {

x = x || 0;

y = y || 0;

···

}

ES6 has nicer syntax:

**function** **foo**(x=0, y=0) {

···

}

An added benefit is that in ES6, a parameter default value is only triggered by undefined, while it is triggered by any falsy value in the previous ES5 code.

**More information:** section “[Parameter default values](http://exploringjs.com/es6/ch_parameter-handling.html#sec_parameter-default-values)” in “Exploring ES6”.

## Handling named parameters

A common way of naming parameters in JavaScript is via object literals (the so-called options object pattern):

selectEntries({ start: 0, end: -1 });

Two advantages of this approach are: Code becomes more self-descriptive and it is easier to omit arbitrary parameters.

In ES5, you can implement selectEntries() as follows:

**function** **selectEntries**(options) {

**var** start = options.start || 0;

**var** end = options.end || -1;

**var** step = options.step || 1;

···

}

In ES6, you can use destructuring in parameter definitions and the code becomes simpler:

**function** **selectEntries**({ start=0, end=-1, step=1 }) {

···

}

### Making the parameter optional

To make the parameter options optional in ES5, you’d add line A to the code:

**function** **selectEntries**(options) {

options = options || {}; // (A)

**var** start = options.start || 0;

**var** end = options.end || -1;

**var** step = options.step || 1;

···

}

In ES6 you can specify {} as a parameter default value:

**function** **selectEntries**({ start=0, end=-1, step=1 } = {}) {

···

}

**More information:** section “[Simulating named parameters](http://exploringjs.com/es6/ch_parameter-handling.html#sec_named-parameters)” in “Exploring ES6”.

## From arguments to rest parameters

In ES5, if you want a function (or method) to accept an arbitrary number of arguments, you must use the special variable arguments:

**function** **logAllArguments**() {

**for** (**var** i=0; i < **arguments**.length; i++) {

**console**.log(**arguments**[i]);

}

}

In ES6, you can declare a rest parameter (args in the example below) via the ... operator:

**function** **logAllArguments**(...args) {

**for** (**let** arg **of** args) {

**console**.log(arg);

}

}

Rest parameters are even nicer if you are only interested in trailing parameters:

**function** **format**(pattern, ...args) {

···

}

Handling this case in ES5 is clumsy:

**function** **format**() {

**var** pattern = **arguments**[0];

**var** args = [].slice.call(**arguments**, 1);

···

}

Rest parameters make code easier to read: You can tell that a function has a variable number of parameters just by looking at its parameter definitions.

**More information:** section “[Rest parameters](http://exploringjs.com/es6/ch_parameter-handling.html#sec_rest-parameters)” in “Exploring ES6”.

## From apply() to the spread operator (...)

In ES5, you turn arrays into parameters via apply(). ES6 has the spread operator for this purpose.

### Math.max()

ES5 – apply():

> Math.max.apply(null, [-1, 5, 11, 3])

11

ES6 – spread operator:

> Math.max(...[-1, 5, 11, 3])

11

### Array.prototype.push()

ES5 – apply():

**var** arr1 = ['a', 'b'];

**var** arr2 = ['c', 'd'];

arr1.push.apply(arr1, arr2);

// arr1 is now ['a', 'b', 'c', 'd']

ES6 – spread operator:

**let** arr1 = ['a', 'b'];

**let** arr2 = ['c', 'd'];

arr1.push(...arr2);

// arr1 is now ['a', 'b', 'c', 'd']

**More information:** section “[The spread operator (...)](http://exploringjs.com/es6/ch_parameter-handling.html#sec_spread-operator)” in “Exploring ES6”.

## From concat() to the spread operator (...)

The spread operator can also turn the contents of its operand into array elements. That means that it becomes an alternative to the Array method concat().

ES5 – concat():

**var** arr1 = ['a', 'b'];

**var** arr2 = ['c'];

**var** arr3 = ['d', 'e'];

**console**.log(arr1.concat(arr2, arr3));

// [ 'a', 'b', 'c', 'd', 'e' ]

ES6 – spread operator:

**let** arr1 = ['a', 'b'];

**let** arr2 = ['c'];

**let** arr3 = ['d', 'e'];

**console**.log([...arr1, ...arr2, ...arr3]);

// [ 'a', 'b', 'c', 'd', 'e' ]

**More information:** section “[The spread operator (...)](http://exploringjs.com/es6/ch_parameter-handling.html#sec_spread-operator)” in “Exploring ES6”.

## From constructors to classes

ES6 classes are mostly just more convenient syntax for constructor functions.

### Base classes

In ES5, you implement constructor functions directly:

**function** **Person**(name) {

**this**.name = name;

}

Person.prototype.describe = **function** () {

**return** 'Person called '+**this**.name;

};

In ES6, classes provide slightly more convenient syntax for constructor functions:

**class** **Person** {

**constructor**(name) {

**this**.name = name;

}

describe() {

**return** 'Person called '+**this**.name;

}

}

### Derived classes

Subclassing is complicated in ES5, especially referring to super-constructors and super-properties. This is the canonical way of creating a sub-constructor of Person, Employee:

**function** **Employee**(name, title) {

Person.call(**this**, name); // super(name)

**this**.title = title;

}

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

Employee.prototype.constructor = Employee;

Employee.prototype.describe = **function** () {

**return** Person.prototype.describe.call(**this**) // super.describe()

+ ' (' + **this**.title + ')';

};

ES6 has built-in support for subclassing, via the extends clause:

**class** **Employee** **extends** **Person** {

**constructor**(name, title) {

**super**(name);

**this**.title = title;

}

describe() {

**return** **super**.describe() + ' (' + **this**.title + ')';

}

}

**More information:** chapter “[Classes](http://exploringjs.com/es6/ch_classes.html)” in “Exploring ES6”.

## From custom error constructors to subclasses of Error

In ES5, it is impossible to subclass the built-in constructor for exceptions, Error(the chapter “[Subclassing Built-ins](http://speakingjs.com/es5/ch28.html)” in “Speaking JavaScript” explains why). The following code shows a work-around that gives the constructor MyErrorimportant features such as a stack trace:

**function** **MyError**() {

// Use Error as a function

**var** superInstance = **Error**.apply(null, **arguments**);

copyOwnPropertiesFrom(**this**, superInstance);

}

MyError.prototype = **Object**.create(**Error**.prototype);

MyError.prototype.constructor = MyError;

In ES6, all built-in constructors can be subclassed, which is why the following code achieves what the ES5 code can only simulate:

**class** **MyError** **extends** **Error** {

}

**More information:** section “[Subclassing built-in constructors](http://exploringjs.com/es6/ch_classes.html" \l "subclassing-builtin-constructors)” in “Exploring ES6”.

## From function expressions in object literals to method definitions

In JavaScript, methods are properties whose values are functions.

In ES5 object literals, methods are created like other properties. The property values are provided via function expressions.

**var** obj = {

foo: **function** () {

···

},

bar: **function** () {

**this**.foo();

}, // trailing comma is legal in ES5

}

ES6 has method definitions, special syntax for creating methods:

**let** obj = {

foo() {

···

},

bar() {

**this**.foo();

},

}

**More information:** section “[Method definitions](http://exploringjs.com/es6/ch_oop-besides-classes.html#object-literal-method-definitions)” in “Exploring ES6”.

## From objects to Maps

Using the language construct object as a map from strings to arbitrary values (a data structure) has always been a makeshift solution in JavaScript. The safest way to do so is by creating an object whose prototype is null. Then you still have to ensure that no key is ever the string '\_\_proto\_\_', because that property key triggers special functionality in many JavaScript engines.

The following ES5 code contains the function countWords that uses the object dict as a map:

**var** dict = **Object**.create(null);

**function** **countWords**(word) {

**var** escapedWord = escapeKey(word);

**if** (escapedWord **in** dict) {

dict[escapedWord]++;

} **else** {

dict[escapedWord] = 1;

}

}

**function** **escapeKey**(key) {

**if** (key.indexOf('\_\_proto\_\_') === 0) {

**return** key+'%';

} **else** {

**return** key;

}

}

In ES6, you can use the built-in data structure Map and don’t have to escape keys. As a downside, incrementing values inside Maps is less convenient.

**let** map = **new** **Map**();

**function** **countWords**(word) {

**let** count = map.get(word) || 0;

map.set(word, count + 1);

}

Another benefit of Maps is that you can use arbitrary values as keys, not just strings.

**More information:**

* Section “[The dict Pattern: Objects Without Prototypes Are Better Maps](http://speakingjs.com/es5/ch17.html#dict_pattern)” in “Speaking JavaScript”
* Chapter “[Maps and Sets](http://exploringjs.com/es6/ch_maps-sets.html)” in “Exploring ES6”

## From CommonJS modules to ES6 modules

Even in ES5, module systems based on either AMD syntax or CommonJS syntax have mostly replaced hand-written solutions such as [the revealing module pattern](http://christianheilmann.com/2007/08/22/again-with-the-module-pattern-reveal-something-to-the-world/).

ES6 has built-in support for modules. Alas, no JavaScript engine supports them natively, yet. But tools such as browserify, webpack or jspm let you use ES6 syntax to create modules, making the code you write future-proof.

### Multiple exports

In CommonJS, you export multiple entities as follows:

//------ lib.js ------

**var** sqrt = **Math**.sqrt;

**function** **square**(x) {

**return** x \* x;

}

**function** **diag**(x, y) {

**return** sqrt(square(x) + square(y));

}

**module**.exports = {

sqrt: sqrt,

square: square,

diag: diag,

};

//------ main1.js ------

**var** square = **require**('lib').square;

**var** diag = **require**('lib').diag;

**console**.log(square(11)); // 121

**console**.log(diag(4, 3)); // 5

Alternatively, you can import the whole module as an object and access squareand diag via it:

//------ main2.js ------

**var** lib = **require**('lib');

**console**.log(lib.square(11)); // 121

**console**.log(lib.diag(4, 3)); // 5

In ES6, multiple exports are called named exports and handled like this:

//------ lib.js ------

**export** **const** sqrt = **Math**.sqrt;

**export** **function** **square**(x) {

**return** x \* x;

}

**export** **function** **diag**(x, y) {

**return** sqrt(square(x) + square(y));

}

//------ main1.js ------

**import** { square, diag } **from** 'lib';

**console**.log(square(11)); // 121

**console**.log(diag(4, 3)); // 5

The syntax for importing modules as objects looks as follows (line A):

//------ main2.js ------

**import** \* **as** lib **from** 'lib'; // (A)

**console**.log(lib.square(11)); // 121

**console**.log(lib.diag(4, 3)); // 5

### Single exports

Node.js extends CommonJS and lets you export single values from modules, via module.exports:

//------ myFunc.js ------

**module**.exports = **function** () { ··· };

//------ main1.js ------

**var** myFunc = **require**('myFunc');

myFunc();

In ES6, the same thing is done via export default:

//------ myFunc.js ------

**export** **default** **function** () { ··· } // no semicolon!

//------ main1.js ------

**import** myFunc **from** 'myFunc';

myFunc();

**More information:** chapter “[Modules](http://exploringjs.com/es6/ch_modules.html)” in “Exploring ES6”.

## What to do next

Now that you got a first taste of ES6, what should you do next? I have two suggestions:

* In “[Exploring ES6](http://exploringjs.com/es6/)”, each major feature of ES6 has its own chapter, which starts with an overview. Browsing the chapters is therefore a good way of getting a more complete picture of ES6.
* The chapter “[Deploying ECMAScript 6](http://exploringjs.com/es6/ch_deploying-es6.html)” describes the options you have for deploying ES6 includes the following new features:
* [arrows](https://github.com/lukehoban/es6features/blob/master/README.md#arrows)
* [classes](https://github.com/lukehoban/es6features/blob/master/README.md#classes)
* [enhanced object literals](https://github.com/lukehoban/es6features/blob/master/README.md#enhanced-object-literals)
* [template strings](https://github.com/lukehoban/es6features/blob/master/README.md#template-strings)
* [destructuring](https://github.com/lukehoban/es6features/blob/master/README.md#destructuring)
* [default + rest + spread](https://github.com/lukehoban/es6features/blob/master/README.md#default--rest--spread)
* [let + const](https://github.com/lukehoban/es6features/blob/master/README.md#let--const)
* [iterators + for..of](https://github.com/lukehoban/es6features/blob/master/README.md#iterators--forof)
* [generators](https://github.com/lukehoban/es6features/blob/master/README.md#generators)
* [unicode](https://github.com/lukehoban/es6features/blob/master/README.md#unicode)
* [modules](https://github.com/lukehoban/es6features/blob/master/README.md#modules)
* [module loaders](https://github.com/lukehoban/es6features/blob/master/README.md#module-loaders)
* [map + set + weakmap + weakset](https://github.com/lukehoban/es6features/blob/master/README.md#map--set--weakmap--weakset)
* [proxies](https://github.com/lukehoban/es6features/blob/master/README.md#proxies)
* [symbols](https://github.com/lukehoban/es6features/blob/master/README.md#symbols)
* [subclassable built-ins](https://github.com/lukehoban/es6features/blob/master/README.md#subclassable-built-ins)
* [promises](https://github.com/lukehoban/es6features/blob/master/README.md#promises)
* [math + number + string + array + object APIs](https://github.com/lukehoban/es6features/blob/master/README.md#math--number--string--array--object-apis)
* [binary and octal literals](https://github.com/lukehoban/es6features/blob/master/README.md#binary-and-octal-literals)
* [reflect api](https://github.com/lukehoban/es6features/blob/master/README.md#reflect-api)
* [tail calls](https://github.com/lukehoban/es6features/blob/master/README.md#tail-calls)

## ECMAScript 6 Features

### Arrows

Arrows are a function shorthand using the => syntax. They are syntactically similar to the related feature in C#, Java 8 and CoffeeScript. They support both statement block bodies as well as expression bodies which return the value of the expression. Unlike functions, arrows share the same lexical this as their surrounding code.

// Expression bodies

var odds = evens.map(v => v + 1);

var nums = evens.map((v, i) => v + i);

var pairs = evens.map(v => ({even: v, odd: v + 1}));

// Statement bodies

nums.forEach(v => {

if (v % 5 === 0)

fives.push(v);

});

// Lexical this

var bob = {

\_name: "Bob",

\_friends: [],

printFriends() {

this.\_friends.forEach(f =>

console.log(this.\_name + " knows " + f));

}

}

More info: [MDN Arrow Functions](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

### Classes

ES6 classes are a simple sugar over the prototype-based OO pattern. Having a single convenient declarative form makes class patterns easier to use, and encourages interoperability. Classes support prototype-based inheritance, super calls, instance and static methods and constructors.

class SkinnedMesh extends THREE.Mesh {

constructor(geometry, materials) {

super(geometry, materials);

this.idMatrix = SkinnedMesh.defaultMatrix();

this.bones = [];

this.boneMatrices = [];

//...

}

update(camera) {

//...

super.update();

}

get boneCount() {

return this.bones.length;

}

set matrixType(matrixType) {

this.idMatrix = SkinnedMesh[matrixType]();

}

static defaultMatrix() {

return new THREE.Matrix4();

}

}

More info: [MDN Classes](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes)

### Enhanced Object Literals

Object literals are extended to support setting the prototype at construction, shorthand for foo: foo assignments, defining methods, making super calls, and computing property names with expressions. Together, these also bring object literals and class declarations closer together, and let object-based design benefit from some of the same conveniences.

var obj = {

// \_\_proto\_\_

\_\_proto\_\_: theProtoObj,

// Shorthand for ‘handler: handler’

handler,

// Methods

toString() {

// Super calls

return "d " + super.toString();

},

// Computed (dynamic) property names

[ 'prop\_' + (() => 42)() ]: 42

};

More info: [MDN Grammar and types: Object literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#Object_literals)

### Template Strings

Template strings provide syntactic sugar for constructing strings. This is similar to string interpolation features in Perl, Python and more. Optionally, a tag can be added to allow the string construction to be customized, avoiding injection attacks or constructing higher level data structures from string contents.

// Basic literal string creation

`In JavaScript '\n' is a line-feed.`

// Multiline strings

`In JavaScript this is

not legal.`

// String interpolation

var name = "Bob", time = "today";

`Hello ${name}, how are you ${time}?`

// Construct an HTTP request prefix is used to interpret the replacements and construction

POST`http://foo.org/bar?a=${a}&b=${b}

Content-Type: application/json

X-Credentials: ${credentials}

{ "foo": ${foo},

"bar": ${bar}}`(myOnReadyStateChangeHandler);

More info: [MDN Template Strings](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/template_strings)

### Destructuring

Destructuring allows binding using pattern matching, with support for matching arrays and objects. Destructuring is fail-soft, similar to standard object lookup foo["bar"], producing undefined values when not found.

// list matching

var [a, , b] = [1,2,3];

// object matching

var { op: a, lhs: { op: b }, rhs: c }

= getASTNode()

// object matching shorthand

// binds `op`, `lhs` and `rhs` in scope

var {op, lhs, rhs} = getASTNode()

// Can be used in parameter position

function g({name: x}) {

console.log(x);

}

g({name: 5})

// Fail-soft destructuring

var [a] = [];

a === undefined;

// Fail-soft destructuring with defaults

var [a = 1] = [];

a === 1;

More info: [MDN Destructuring assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

### Default + Rest + Spread

Callee-evaluated default parameter values. Turn an array into consecutive arguments in a function call. Bind trailing parameters to an array. Rest replaces the need for arguments and addresses common cases more directly.

function f(x, y=12) {

// y is 12 if not passed (or passed as undefined)

return x + y;

}

f(3) == 15

function f(x, ...y) {

// y is an Array

return x \* y.length;

}

f(3, "hello", true) == 6

function f(x, y, z) {

return x + y + z;

}

// Pass each elem of array as argument

f(...[1,2,3]) == 6

More MDN info: [Default parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters), [Rest parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters), [Spread Operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator)

### Let + Const

Block-scoped binding constructs. let is the new var. const is single-assignment. Static restrictions prevent use before assignment.

function f() {

{

let x;

{

// okay, block scoped name

const x = "sneaky";

// error, const

x = "foo";

}

// error, already declared in block

let x = "inner";

}

}

More MDN info: [let statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let), [const statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const)

### Iterators + For..Of

Iterator objects enable custom iteration like CLR IEnumerable or Java Iterable. Generalize for..in to custom iterator-based iteration with for..of. Don’t require realizing an array, enabling lazy design patterns like LINQ.

let fibonacci = {

[Symbol.iterator]() {

let pre = 0, cur = 1;

return {

next() {

[pre, cur] = [cur, pre + cur];

return { done: false, value: cur }

}

}

}

}

for (var n of fibonacci) {

// truncate the sequence at 1000

if (n > 1000)

break;

console.log(n);

}

Iteration is based on these duck-typed interfaces (using [TypeScript](http://typescriptlang.org/) type syntax for exposition only):

interface IteratorResult {

done: boolean;

value: any;

}

interface Iterator {

next(): IteratorResult;

}

interface Iterable {

[Symbol.iterator](): Iterator

}

More info: [MDN for...of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of)

### Generators

Generators simplify iterator-authoring using function\* and yield. A function declared as function\* returns a Generator instance. Generators are subtypes of iterators which include additional next and throw. These enable values to flow back into the generator, so yield is an expression form which returns a value (or throws).

Note: Can also be used to enable ‘await’-like async programming, see also ES7 await proposal.

var fibonacci = {

[Symbol.iterator]: function\*() {

var pre = 0, cur = 1;

for (;;) {

var temp = pre;

pre = cur;

cur += temp;

yield cur;

}

}

}

for (var n of fibonacci) {

// truncate the sequence at 1000

if (n > 1000)

break;

console.log(n);

}

The generator interface is (using [TypeScript](http://typescriptlang.org/) type syntax for exposition only):

interface Generator extends Iterator {

next(value?: any): IteratorResult;

throw(exception: any);

}

More info: [MDN Iteration protocols](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols)

### Unicode

Non-breaking additions to support full Unicode, including new Unicode literal form in strings and new RegExp u mode to handle code points, as well as new APIs to process strings at the 21bit code points level. These additions support building global apps in JavaScript.

// same as ES5.1

"𠮷".length == 2

// new RegExp behaviour, opt-in ‘u’

"𠮷".match(/./u)[0].length == 2

// new form

"\u{20BB7}"=="𠮷"=="\uD842\uDFB7"

// new String ops

"𠮷".codePointAt(0) == 0x20BB7

// for-of iterates code points

for(var c of "𠮷") {

console.log(c);

}

More info: [MDN RegExp.prototype.unicode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp/unicode)

### Modules

Language-level support for modules for component definition. Codifies patterns from popular JavaScript module loaders (AMD, CommonJS). Runtime behaviour defined by a host-defined default loader. Implicitly async model – no code executes until requested modules are available and processed.

// lib/math.js

export function sum(x, y) {

return x + y;

}

export var pi = 3.141593;

// app.js

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

alert("2π = " + math.sum(math.pi, math.pi));

// otherApp.js

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

alert("2π = " + sum(pi, pi));

Some additional features include export default and export \*:

// lib/mathplusplus.js

export \* from "lib/math";

export var e = 2.71828182846;

export default function(x) {

return Math.log(x);

}

// app.js

import ln, {pi, e} from "lib/mathplusplus";

alert("2π = " + ln(e)\*pi\*2);

More MDN info: [import statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import), [export statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export)

### Module Loaders

Module loaders support:

* Dynamic loading
* State isolation
* Global namespace isolation
* Compilation hooks
* Nested virtualization

The default module loader can be configured, and new loaders can be constructed to evaluate and load code in isolated or constrained contexts.

// Dynamic loading – ‘System’ is default loader

System.import('lib/math').then(function(m) {

alert("2π = " + m.sum(m.pi, m.pi));

});

// Create execution sandboxes – new Loaders

var loader = new Loader({

global: fixup(window) // replace ‘console.log’

});

loader.eval("console.log('hello world!');");

// Directly manipulate module cache

System.get('jquery');

System.set('jquery', Module({$: $})); // WARNING: not yet finalized

### Map + Set + WeakMap + WeakSet

Efficient data structures for common algorithms. WeakMaps provides leak-free object-key’d side tables.

// Sets

var s = new Set();

s.add("hello").add("goodbye").add("hello");

s.size === 2;

s.has("hello") === true;

// Maps

var m = new Map();

m.set("hello", 42);

m.set(s, 34);

m.get(s) == 34;

// Weak Maps

var wm = new WeakMap();

wm.set(s, { extra: 42 });

wm.size === undefined

// Weak Sets

var ws = new WeakSet();

ws.add({ data: 42 });

// Because the added object has no other references, it will not be held in the set

More MDN info: [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map), [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set), [WeakMap](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap), [WeakSet](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakSet)

### Proxies

Proxies enable creation of objects with the full range of behaviors available to host objects. Can be used for interception, object virtualization, logging/profiling, etc.

// Proxying a normal object

var target = {};

var handler = {

get: function (receiver, name) {

return `Hello, ${name}!`;

}

};

var p = new Proxy(target, handler);

p.world === 'Hello, world!';

// Proxying a function object

var target = function () { return 'I am the target'; };

var handler = {

apply: function (receiver, ...args) {

return 'I am the proxy';

}

};

var p = new Proxy(target, handler);

p() === 'I am the proxy';

There are traps available for all of the runtime-level meta-operations:

var handler =

{

get:...,

set:...,

has:...,

deleteProperty:...,

apply:...,

construct:...,

getOwnPropertyDescriptor:...,

defineProperty:...,

getPrototypeOf:...,

setPrototypeOf:...,

enumerate:...,

ownKeys:...,

preventExtensions:...,

isExtensible:...

}

More info: [MDN Proxy](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy)

### Symbols

Symbols enable access control for object state. Symbols allow properties to be keyed by either string (as in ES5) or symbol. Symbols are a new primitive type. Optional description parameter used in debugging - but is not part of identity. Symbols are unique (like gensym), but not private since they are exposed via reflection features like Object.getOwnPropertySymbols.

var MyClass = (function() {

// module scoped symbol

var key = Symbol("key");

function MyClass(privateData) {

this[key] = privateData;

}

MyClass.prototype = {

doStuff: function() {

... this[key] ...

}

};

return MyClass;

})();

var c = new MyClass("hello")

c["key"] === undefined

More info: [MDN Symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol)

### Subclassable Built-ins

In ES6, built-ins like Array, Date and DOM Elements can be subclassed.

Object construction for a function named Ctor now uses two-phases (both virtually dispatched):

* Call Ctor[@@create] to allocate the object, installing any special behavior
* Invoke constructor on new instance to initialize

The known @@create symbol is available via Symbol.create. Built-ins now expose their @@create explicitly.

// Pseudo-code of Array

class Array {

constructor(...args) { /\* ... \*/ }

static [Symbol.create]() {

// Install special [[DefineOwnProperty]]

// to magically update 'length'

}

}

// User code of Array subclass

class MyArray extends Array {

constructor(...args) { super(...args); }

}

// Two-phase 'new':

// 1) Call @@create to allocate object

// 2) Invoke constructor on new instance

var arr = new MyArray();

arr[1] = 12;

arr.length == 2

### Math + Number + String + Array + Object APIs

Many new library additions, including core Math libraries, Array conversion helpers, String helpers, and Object.assign for copying.

Number.EPSILON

Number.isInteger(Infinity) // false

Number.isNaN("NaN") // false

Math.acosh(3) // 1.762747174039086

Math.hypot(3, 4) // 5

Math.imul(Math.pow(2, 32) - 1, Math.pow(2, 32) - 2) // 2

"abcde".includes("cd") // true

"abc".repeat(3) // "abcabcabc"

Array.from(document.querySelectorAll('\*')) // Returns a real Array

Array.of(1, 2, 3) // Similar to new Array(...), but without special one-arg behavior

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

[1, 2, 3].find(x => x == 3) // 3

[1, 2, 3].findIndex(x => x == 2) // 1

[1, 2, 3, 4, 5].copyWithin(3, 0) // [1, 2, 3, 1, 2]

["a", "b", "c"].entries() // iterator [0, "a"], [1,"b"], [2,"c"]

["a", "b", "c"].keys() // iterator 0, 1, 2

["a", "b", "c"].values() // iterator "a", "b", "c"

Object.assign(Point, { origin: new Point(0,0) })

More MDN info: [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number), [Math](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math), [Array.from](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from), [Array.of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/of), [Array.prototype.copyWithin](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/copyWithin), [Object.assign](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign)

### Binary and Octal Literals

Two new numeric literal forms are added for binary (b) and octal (o).

0b111110111 === 503 // true

0o767 === 503 // true

### Promises

Promises are a library for asynchronous programming. Promises are a first class representation of a value that may be made available in the future. Promises are used in many existing JavaScript libraries.

function timeout(duration = 0) {

return new Promise((resolve, reject) => {

setTimeout(resolve, duration);

})

}

var p = timeout(1000).then(() => {

return timeout(2000);

}).then(() => {

throw new Error("hmm");

}).catch(err => {

return Promise.all([timeout(100), timeout(200)]);

})

More info: [MDN Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)

### Reflect API

Full reflection API exposing the runtime-level meta-operations on objects. This is effectively the inverse of the Proxy API, and allows making calls corresponding to the same meta-operations as the proxy traps. Especially useful for implementing proxies.

// No sample yet

More info: [MDN Reflect](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect)

### Tail Calls

Calls in tail-position are guaranteed to not grow the stack unboundedly. Makes recursive algorithms safe in the face of unbounded inputs.

function factorial(n, acc = 1) {

'use strict';

if (n <= 1) return acc;

return factorial(n - 1, n \* acc);

}

// Stack overflow in most implementations today,

// but safe on arbitrary inputs in ES6

factorial(100000)