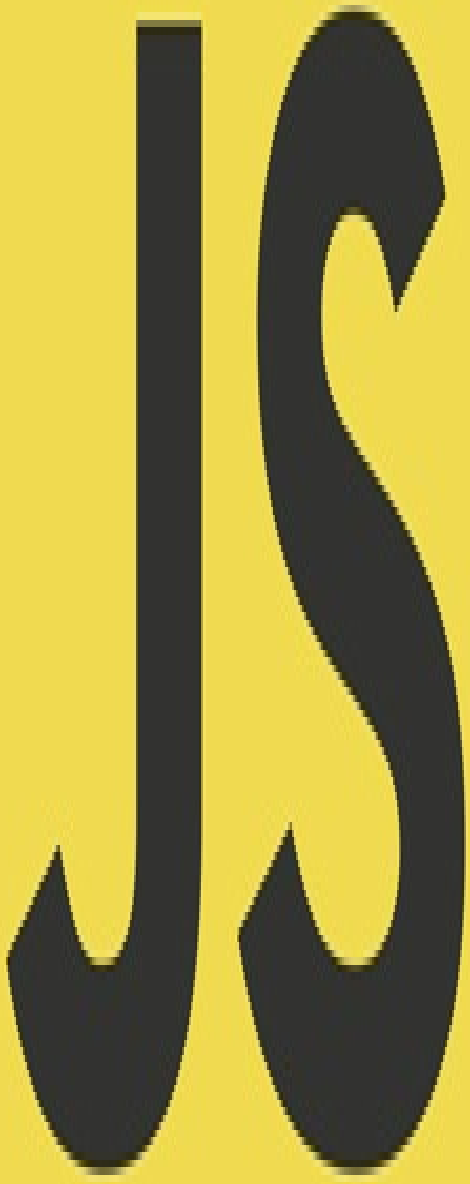


# Table of Contents

1. Introduction
2. let + const
3. arrow functions
4. default + rest + spread
5. destructuring
6. strings
7. iterators
8. generators
9. classes and inheritance
10. modules
11. promises
12. set, map, weak

# ES6-guide

**ECMAScript 6 (ES6) guide**


**CHECK SUMMARY TO SEE TABLE OF CONTENT**

I want to share with you some thoughts, snippets of code and tell you a little about the upcoming **ES6**. It's my own road to know it before it will be a standard.

You might have noticed about ES6 a lot lately. This is because the standard is targeting ratification in June 2015.

See draft - ECMAScript 2015

ECMAScript 2015 is a significant update to the language. Previous (ES5) was standardized in 2009. Frameworks like **AngularJS**, **Aurelia**, **ReactJS**, **Ionic** start using it today.

ES6 includes a lot of new features:

- arrows
- classes
- enhanced object literals
- template strings
- destructuring
- default + rest + spread
- let + const
- iterators + for..of
- generators
- unicode
- modules
- module loaders
- map + set + weakmap + weakset
- proxies
- symbols
- subclassable built-ins
- promises
- math + number + string + object APIs
- binary and octal literals
- reflect api

- tail calls

I will try to describe each of these in the next stories, so stay updated.

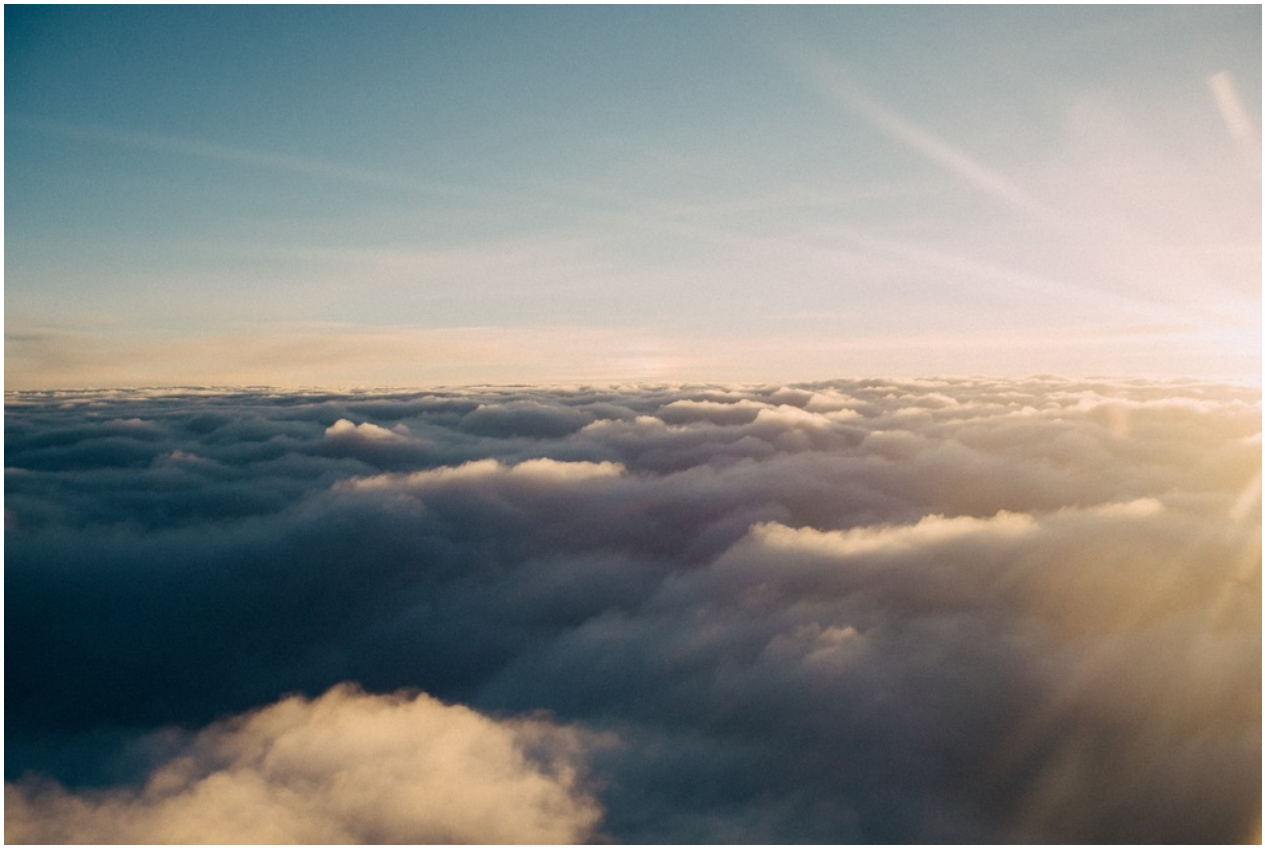
Thanks to the use of transpilers (Babel, Traceur and others) we can actually use it right now until browsers fully catch up.

Browser support matrix

ES6 Repl in Chrome Devl Tools - Scratch JS

```
Future is bright.
```

People from the community denounced these words. I have only one to add: we can handle it easily!



# let + const

First topic about ECMAScript 2015 is **let + const**. If you are familiar with JavaScript, you have probably known the term: **scope**. If you are not that lucky, don't worry about it. I'll explain that in a few words below.

Why I mentioned something about JavaScript **scope**? This is because **let** and **const** have a very strong connection with that word. Firstly, imagine and old way (and still valid) to declare a new variable in your JS code using ES5:

```
// ES5

var a = 1;

if (1 === a) {
  var b = 2;
}

for (var c = 0; c < 3; c++) {
  // …
}

function letsDeclareAnotherOne() {
  var d = 4;
}

console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
console.log(d); // ReferenceError: d is not defined

// window
console.log(window.a); // 1
console.log(window.b); // 2
console.log(window.c); // 3
console.log(window.d); // undefined
```

1. We can see that variable **a** is declared as global. Nothing surprising.
2. Variable **b** is inside an **if block**, but in JavaScript it doesn't create a new scope. If you are familiar with other languages, you can be disappointed, but this is JavaScript and it works as you see.
3. The next statement is a **for loop**. **C** variable is declared in this for loop, but also in the global scope.
4. Until variable **d** is declared in his own scope. It's inside a function and only **function creates new scopes**.

**Variables in JavaScript are hoisted to the top!**

**Hoisting** is JavaScript's default behavior of moving all declarations to the top of the current scope (**to the top of the current script** or **the current function**).

In JavaScript, a variable can be declared after it has been used. In other words - a variable can be used before it has been declared!

One more rule, more aware: JavaScript only hoists declarations, not initialization.

```
// scope and variable hoisting

var n = 1;

(function () {
  console.log(n);
  var n = 2;
```

```
  console.log(n);
})();

console.log(n);
```

Let's look for the new keywords in JavaScript ECMAScript 2015: **let** and **const**.

# let

We can imagine that **let** is a new var statement. What is the difference? let is block scoped. Let's see an example:

```
// ES6 – let

let a = 1;

if (1 === a) {
  let b = 2;
}

for (let c = 0; c < 3; c++) {
  // …
}

function letsDeclareAnotherOne() {
  let d = 4;
}

console.log(a); // 1
console.log(b); // ReferenceError: b is not defined
console.log(c); // ReferenceError: c is not defined
console.log(d); // ReferenceError: d is not defined

// window
console.log(window.a); // 1
console.log(window.b); // undefined
console.log(window.c); // undefined
console.log(window.d); // undefined
```

As we can see, this time only variable **a** is declared as a global. **let** gives us a way to declare block scoped variables, which is undefined outside it.

**I use Chrome (stable version) with #enable-javascript-harmony flag enabled. Visit chrome://flags/#enable-javascript-harmony, enable this flag, restart Chrome and you will get many new features.**

You can also use **BabelJS repl** or **Traceur repl** and compare results.

# const

**const** is single-assignment and like a **let**, block-scoped declaration.

```
// ES6 const

{
  const PI = 3.141593;
  PI = 3.14; // throws “PI” is read-only
}

console.log(PI); // throws ReferenceError: PI is not defined
```

const cannot be reinitialized. It will throw an Error when we try to assign another value.

Let's look for the equivalent in ES5:

```
// ES5 const

var PI = (function () {
  var PI = 3.141593;
  return function () { return PI; };
})();
```

# arrow functions

A new syntactic sugar which ES6 brings us soon, called **arrow functions** (also known as a fat arrow function). It's a shorter syntax compared to function expressions and lexically binds **this** value.

REMEMBER - **Arrow functions are always anonymous.**

# Syntactic sugar

How does it look? It's a signature:

```
([param] [, param]) => {
 statements
}

           param => expression
(param1, param2) => { block }
```

..and it could be translated to:

```
    () => { … }       // no argument
     x => { … }       // one argument
(x, y) => { … }       // several arguments

x => { return x * x } // block
x => x * x            // expression, same as above
```

Lambda expressions in JavaScript! Cool!

Instead of writing:

```
[3, 4, 5].map(function (n) {
  return n * n;
});
```

..you can write something like this:

```
[3, 4, 5].map(n => n * n);
```

Awesome. Isn't it? There is more!

# Fixed "this" = lexical "this"

**The value of this inside of the function is determined by where the arrow function is defined not where it is used.**

No more **bind**, **call** and **apply**! No more:

```
var self = this;
```

It solves a major pain point (from my point of view) and has the added bonus of improving performance through JavaScript engine optimizations.

```
// ES5
function FancyObject() {
 var self = this;

 self.name = 'FancyObject';
 setTimeout(function () {
  self.name = 'Hello World!';
 }, 1000);
}

// ES6
function FancyObject() {
  this.name = 'FancyObject';
  setTimeout(() => {
    this.name = 'Hello World!'; // properly refers to FancyObject
  }, 1000);
}
```

## The same function

- **typeof** returns **function**
- **instanceof** returns **Function**

## Limitations

- It's cannot be used as a constructor and will throw an error when used with **new**.
- Fixed **this** means that you cannot change the **value of this** inside of the function. It remains the same value throughout the entire lifecycle of the function.
- Regular functions can be **named**.
- Functions declarations are **hoisted** (can be used before they are declared).

# default + rest + spread

ECMAScript 2015 functions made a significant progress, taking into account years of complaints. The result is a number of improvements that make programming in JavaScript **less error-prone** and **more powerful**.

Let's see three new features which give us **extended parameter handling**.

## default

It's a simple, little addition that makes it much easier to handle function parameters. Functions in JavaScript allow any number of parameters to be passed regardless of the number of declared parameters in the function definition. You probably know **commonly seen pattern** in current JavaScript code:

```
function inc(number, increment) {
  // set default to 1 if increment not passed
  // (or passed as undefined)
  increment = increment || 1;
  return number + increment;
}

console.log(inc(2, 2)); // 4
console.log(inc(2));    // 3
```

The logical **OR operator** (||) always returns the second operand when the first is falsy.

**ES6** gives us a way to set default function parameters. Any parameters with a default value are considered to be optional.

ES6 version of **inc** function looks like this:

```
function inc(number, increment = 1) {
  return number + increment;
}

console.log(inc(2, 2)); // 4
console.log(inc(2));    // 3
```

You can also set default values to parameters that appear **before arguments** without default values:

```
function sum(a, b = 2, c) {
  return a + b + c;
}

console.log(sum(1, 5, 10));         // 16 -> b === 5
console.log(sum(1, undefined, 10)); // 13 -> b as default
```

You can even execute a function to set default parameter. It's not restricted to **primitive values**.

```
function getDefaultIncrement() {
  return 1;
}

function inc(number, increment = getDefaultIncrement()) {
  return number + increment;
}
```

```
console.log(inc(2, 2)); // 4
console.log(inc(2));    // 3
```

# rest

Let's rewrite **sum** function to handle all arguments passed to it (without validation - just to be clear). If we want to use **ES5**, we probably also want to use **arguments** object.

```
function sum() {
   var numbers = Array.prototype.slice.call(arguments),
       result = 0;
   numbers.forEach(function (number) {
       result += number;
   });
   return result;
}

console.log(sum(1));             // 1
console.log(sum(1, 2, 3, 4, 5)); // 15
```

But it's not obvious that the function is capable of handling any parameters. W have to scan body of the function and find **arguments** object.

**ECMAScript 6** introduces **rest** parameters to help us with this and other pitfalls.

**arguments - contains all parameters including named parameters**

Rest parameters are indicated by three dots **…** preceding a parameter. Named parameter becomes an **array** which contain the rest of the parameters.

**sum** function can be rewritten using ES6 syntax:

```
function sum(…numbers) {
  var result = 0;
  numbers.forEach(function (number) {
    result += number;
  });
  return result;
}

console.log(sum(1)); // 1
console.log(sum(1, 2, 3, 4, 5)); // 15
```

Restriction: **no other named arguments can follow in the function declaration.**

```
function sum(…numbers, last) { // causes a syntax error
  var result = 0;
  numbers.forEach(function (number) {
    result += number;
  });
  return result;
}
```

# spread

The **spread** is closely related to rest parameters, because of **…** (three dots) notation. It allows to split an array to single

arguments which are passed to the function as separate arguments.

Let's define our **sum** function an pass **spread** to it:

```
function sum(a, b, c) {
  return a + b + c;
}

var args = [1, 2, 3];
console.log(sum(…args)); // 6
```

ES5 equivalent is:

```
function sum(a, b, c) {
  return a + b + c;
}

var args = [1, 2, 3];
console.log(sum.apply(undefined, args)); // 6
```

Instead using an **apply** function, we can just type **…args** and pass all array argument separately.

We can also mix standard arguments with spread operator:

```
function sum(a, b, c) {
  return a + b + c;
}

var args = [1, 2];
console.log(sum(…args, 3)); // 6
```

The result is the same. First two arguments are from args array, and the last passed argument is 3.

# destructuring

**Destructuring** is one more little addition to the upcoming JavaScript standard, which helps us write code more flexibly and effectively.

It allows binding using pattern matching. We can use it for matching arrays and objects. It's similar to standard object look up and returns **undefined** when value is not found.

## arrays

Today it's common to see the code such as this.

```
// ES5
var point = [1, 2];
var xVal = point[0],
    yVal = point[1];

console.log(xVal); // 1
console.log(yVal); // 2
```

ES6 gives us destructuring of arrays into individual variables during assignment which is intuitive and flexible.

```
// ES6
let point = [1, 2];
let [xVal, yVal] = point;

console.log(xVal); // 1
console.log(yVal); // 2
// .. and reverse!
[xVal, yVal] = [yVal, xVal];

console.log(xVal); // 2
console.log(yVal); // 1
```

We can even omit some values..

```
let threeD = [1, 2, 3];
let [a, , c] = threeD;

console.log(a); // 1
console.log(c); // 3
```

..and have nested array destructuring.

```
let nested = [1, [2, 3], 4];
let [a, [b], d] = nested;

console.log(a); // 1
console.log(b); // 2
console.log(d); // 4
```

# objects

As well as the array syntax, ES6 also has the ability to destructure objects. It uses an object literal on the left side of an assignment operation. Object pattern is very similar to array pattern seen above. Let's see:

```
let point = {
  x: 1,
  y: 2
};
let { x: a, y: b } = point;

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

It supports nested object as well as array pattern.

```
let point = {
  x: 1,
  y: 2,
  z: {
    one: 3,
    two: 4
  }
};
let { x: a, y: b, z: { one: c, two: d } } = point;

console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
console.log(d); // 4
```

# mixed

We can also mix objects and arrays together and use theirs literals.

```
let mixed = {
  one: 1,
  two: 2,
  values: [3, 4, 5]
};
let { one: a, two: b, values: [c, , e] } = mixed;

console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
console.log(e); // 5
```

But I think the most interesting is that we are able to use **functions** which return destructuring assignment.

```
function mixed () {
  return {
    one: 1,
    two: 2,
    values: [3, 4, 5]
  };
}
let { one: a, two: b, values: [c, , e] } = mixed();

console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
console.log(e); // 5
```

The same result! It gives us a lot of possibilities to use it in our code.

# attention!

**If the value of a destructuring assignment isn't match, it evaluates to undefined.**

```
let point = {
  x: 1
};
let { x: a, y: b } = point;

console.log(a); // 1
console.log(b); // undefined
```

**If we try to omit var, let or const, it will throw an error, because block code can't be destructuring assignment.**

```
let point = {
  x: 1
};

{ x: a } = point; // throws error
```

We have to wrap it in parentheses. Just that ☺

```
let point = {
  x: 1
};
({ x: a } = point);

console.log(a); // 1
```

# strings

I gonna show you a couple of changes to **strings** in JavaScript, which will be available when ES6 comes. A syntactic sugar, which could be helpful in daily work.

## template strings

First, a **string interpolation**. Yep, template strings (finally) support string interpolation. ES6 brings us also support for **multi-line** syntax and **raw literals**.

```
let x = 1;
let y = 2;
let sumTpl = `${x} + ${y} = ${x + y}`;

console.log(sumTpl); // 1 + 2 = 3
```

As you can see, we can inject values to string by using **${value}** syntax. Another thing to consider is **grave accent** - a char under the tilde (~) on a keyboard. A template literal string must be wrapped by it, to work properly.

The example above is an equivalent (in ES5) to simply (Babel version):

```
var x = 1;
var y = 2;
var sumTpl = "" + x + " + " + y + " = " + (x + y);

console.log(sumTpl); // 1 + 2 = 3
```

This feature is very useful and almost removes the need for a template system.

Template strings provide also **multi-line syntax**, which is not legal in ES5 and earlier.

```
let types = `Number
String
Array
Object`;
console.log(types); // Number
                    // String
                    // Array
                    // Object
```

ES5 equivalent:

```
var types = "Number\nString\nArray\nObject";
console.log(types); // Number
                    // String
                    // Array
                    // Object
```

The last thing is access the **raw template string** content where backslashes are not interpreted. We don't have equivalent in ES5 here.

```
let interpreted = 'raw\nstring';
```

```
let esaped = 'raw\\nstring';
let raw = String.raw`raw\nstring`;

console.log(interpreted);    // raw
                             // string
console.log(raw === esaped); // true
```

# extended support for Unicode

ES6 gives us full support for **Unicode** within strings and regular expressions. It's non-breaking addition allows to building global apps.

Let's see an example:

```
let str = '' ;

console.log(str.length);             // 2
console.log(str === '\uD842\uDFB7'); // true
```

You can see that character ***** represented by two 16-bit code units. It's a surrogate pair in which we have a single code point represented by two code units. The length of that string is also 2.

**Surrogate pairs are used in UTF-16 to represent code points above U+FFFF.**

```
console.log(str.charCodeAt(0)); // 55362
console.log(str.charCodeAt(1)); // 57271
```

The **charCodeAt()** method returns the 16-bit number for each code unit.

ES6 allows encoding of strings in UTF-16. JavaScript can now support work with surrogate pairs. It gives us also a new method **codePointAt()** that returns Unicode code point instead of Unicode code unit.

```
console.log(str.codePointAt(0)); // 134071
console.log(str.codePointAt(1)); // 57271
console.log(str.codePointAt(0) === 0x20BB7); // true
```

It works the same as **charCodeAt()** except for non-BMP characters.

**BMP - Basic Multilingual Plane - the first 2^16 code points.**

**codePointAt()** returns full code point at the 0 **position**. codePointAt() and charCodeAt() return the same value for **position 1**.

We can also do a reverse operation with another new method added to ES6: **fromCodePoint()**.

```
console.log(String.fromCodePoint(134071));  // ""
console.log(String.fromCodePoint(0x20BB7)); // ""
```

**Unicode code unit escape sequences consist of six characters, namely \u plus four hexadecimal digits, and contribute one code unit.**

**Unicode code point escape sequences consist of five to ten characters, namely \u{ 1–6 hexadecimal digits }, and**

**contribute one or two code units.**

Dealing with that two definitions, above example could be represented by one code point in **ES6**:

```
// ES6
console.log('\u{20BB7}'); //
console.log('\u{20BB7}' === '\uD842\uDFB7'); // true

// ES5
console.log('\u20BB7); // 7!
console.log('\u20BB7' === '\uD842\uDFB7'); // false
```

In ES5 we get an unexpected result when we try to match one single character using regular expression.

```
console.log(/^.$/.test(str)); // false - length is 2
```

ES6 allows us to use new RegExp **u mode** to handle code points. It is simply a new **u** flag (u == Unicode).

```
console.log(/^.$/u.test(str)); // true
```

Adding u flag allows to correctly match the string by characters instead of code units.

# strings are iterable

Strings are iterable by using the **for-of** loop which I will cover in more detail in iterators article later. I write about it now because it enumerate Unicode code points and each may comprise one or two characters.

```
let str = 'abc\uD842\uDFB7';
console.log(str.length); // 5
for (let c of str) {
  console.log(c); // a
                  // b
                  // c
                  //
}
```

We can also use **spread** operator to transform string into an array with full Unicode support.

```
let str = 'abc\uD842\uDFB7';
let chars = […str];
console.log(chars); // ['a', 'b', 'c', '']
```

# new string methods

**repeat(n)** - string repeats by **n** times

```
console.log('abc|'.repeat(3)); // 'abc|abc|abc|'
```

**startsWith(str, starts = 0) : boolean** - check if string starts with **str**, starting from **starts**

```
console.log('ecmascript'.startsWith('ecma'));      // true
console.log('ecmascript'.startsWith('script', 4)); // true
```

**endsWith(str, ends = str.length) : boolean** - check if string ends with **str**, **ends** - where the string to be checked ends

```
console.log('ecmascript'.endsWith('script'));  // true
console.log('ecmascript'.endsWith('ecma', 4)); // true
```

**includes(str, starts = 0) : boolean** - check if string contain **str**, starting from **starts**

```
console.log('ecmascript'.includes('ecma'));      // true
console.log('ecmascript'.includes('script', 4)); // true
```

# iterators

## iterator & iterable

**An iterator is an object with a next method that returns { done, value } tuples.**

ES6 gives us a pattern for creating custom iterators and it has a similar implementation to Java **Iterable** or .NET **IEnumerable**. It has also built-in **iterables**: **String**, **Array**, **TypedArray**, **Map** and **Set**. An iterator object can be any object with a next() method.

**Iterable is an object which has Symbol.iterator method inside.**

**Symbol is in turn an unique and immutable data type which can be used as an identifier for object properties—no equivalent in ES5.**

```
// Symbol
let s1 = Symbol('abc');
let s2 = Symbol('abc');

console.log(s1 !== s2); // true
console.log(typeof s1); // 'symbol'

let obj = {};
obj[s1] = 'abc';
console.log(obj); // Object { Symbol(abc): 'abc' }
```

Let's see a simple iterator written from scratch, which allows us to iterate through random numbers which are dynamically generated by **next()** method. A function returning iterable object take one argument (**items**) which is used to determine if the iterator should stop and returns **done = true**.

```
let random1_10 = function (items = 1) {
  return {
    [Symbol.iterator]() {
      let cur = 0;
      return {
        next() {
          let done = cur === items,
              random = Math.floor(Math.random() * 10) + 1;
          ++cur;
          return {
            done: done,
            value: random
          }
        }
      }
    }
  };
};

for (let n of random1_10(5)) {
  console.log(n); // prints 5 random numbers
}
```

Every time **for-of** loop call **next()** method, an iterator generate a random number and returns it to the loop.

**If the iterator returns done = true, you can omit value, so the result will be { done: true }**

```
let random1_10 = function (items = 1) {
```

```
  return {
    [Symbol.iterator]() {
      let cur = 0;
      return {
        next() {
          if (cur === items) {
            return {
              done: true
            }
          }
          ++cur;
          return {
            done: false,
            value: Math.floor(Math.random() * 10) + 1
          }
        }
      }
    }
  };
};

for (let n of random1_10(5)) {
 console.log(n); // prints 5 random numbers
}
```

# for-of loop

ES6 has a new loop—**for-of**. It works with iterables. Let's look at his signature:

```
for (LET of ITERABLE) {
  CODE BLOCK
}
```

It's similar to **for-in** loop, which can be used to iterate through object properties (plain old Objects).

**Arrays** in ES6 are iterable by default, so we finally can use **for-of** for looping over the elements.

```
const arr = [1, 2, 3, 4, 5];

for (let item of arr) {
  console.log(item); // 1
                     // 2
                     // 3
                     // 4
                     // 5
}
```

Inside the for-of loop, we can even use a **break**, **continue** and **return**.

```
const arr = [1, 2, 3, 4, 5];

for (let item of arr) {
  if (item > 4) {
    break;
  }
  if (0 !== item % 2) {
    continue;
  }
  console.log(item); // 2
                     // 4
}
```

# generators

**Generators** are simply subtypes of **Iterators** which I wrote about previously. They are a special kind of function that can be suspended and resumed, which is making a difference to iterators. Generators use **function*** and **yield** operators to work their magic.

**The yield operator returns a value from the function and when the generator is resumed, execution continues after the yield.**

**We also have to use function* (with star character) instead of a function to return a generator instance.**

**!!! Generators have been borrowed from Python language.**

**The most magical feature in ES6!**

Why? Take a look:

```
function* generator () {
  yield 1;
  // pause
  yield 2;
  // pause
  yield 3;
  // pause
  yield 'done?';
  // done
}
let gen = generator(); // [object Generator]

console.log(gen.next()); // Object {value: 1, done: false}
console.log(gen.next()); // Object {value: 2, done: false}
console.log(gen.next()); // Object {value: 3, done: false}
console.log(gen.next()); // Object {value: 'done?', done: false}
console.log(gen.next()); // Object {value: undefined, done: true}
console.log(gen.next()); // Object {value: undefined, done: true}

for (let val of generator()) {
  console.log(val); // 1
                    // 2
                    // 3
                    // 'done?'
}
```

As you can see, the generator has four **yield** statements. Each returns a value, pauses execution and moves to the next yield when **next()** method is called. Calling a function produces an object for controlling generator execution, a so-called **generator object**.

Use is similar to iterators. We have **next()** method as I mentioned above and we can even use it with **for-of** loop.

Below is an example of a generator called **random1_10**, which returns random numbers from 1 to 10.

```
function* random1_10 () {
  while (true) {
    yield Math.floor(Math.random() * 10) + 1;
  }
}

let rand = random1_10();
console.log(rand.next());
console.log(rand.next());
// …
```

Generator has never ending **while** loop. It produces random numbers every time when you call **next()** method.

ES5 implementation:

```
function random1_10 () {
  return {
    next: function() {
      return {
        value: Math.floor(Math.random() * 10) + 1,
        done: false
      };
    }
  };
}

let rand = random1_10();
console.log(rand.next());
console.log(rand.next());
// …
```

We can also mix generators together:

```
function* random (max) {
  yield Math.floor(Math.random() * max) + 1;
}

function* random1_20 () {
  while (true) {
    yield* random(20);
  }
}

let rand = random1_20();
console.log(rand.next());
console.log(rand.next());
// …
```

**random1_20** generator returns random values from 1 to 20. It uses **random** generator inside to create random number each time when yield statement is reached.

# classes and inheritance

**OO** keywords is probably the most awaited features in ES6. **Classes** are something like another syntactic sugar over the prototype-based OO pattern. We now have one, concise way to make class patterns easier to use.

**Over the prototype-based OO pattern to ensure backwards compatibility.**

# Overview—an example of ES6 class syntax and ES5 equivalent

```
class Vehicle {

  constructor (name, type) {
    this.name = name;
    this.type = type;
  }

  getName () {
    return this.name;
  }

  getType () {
    return this.type;
  }

}

let car = new Vehicle('Tesla', 'car');
console.log(car.getName()); // Tesla
console.log(car.getType()); // car
```

It's naive example, but we can see a new keywords as class and constructor.

ES5 equivalent could be something like this:

```
function Vehicle (name, type) {
  this.name = name;
  this.type = type;
};

Vehicle.prototype.getName = function getName () {
  return this.name;
};

Vehicle.prototype.getType = function getType () {
  return this.type;
};

var car = new Vehicle('Tesla', 'car');
console.log(car.getName()); // Tesla
console.log(car.getType()); // car
```

**Classes support prototype-based inheritance, super calls, instance and static methods and constructors.**

It's simple. We instantiate our classes the same way, but let's add some..

# inheritance

..to it and start from ES5 example:

```
function Vehicle (name, type) {
  this.name = name;
  this.type = type;
};

Vehicle.prototype.getName = function getName () {
  return this.name;
};

Vehicle.prototype.getType = function getType () {
  return this.type;
};

function Car (name) {
  Vehicle.call(this, name, 'car');
}

Car.prototype = Object.create(Vehicle.prototype);
Car.prototype.constructor = Car;
Car.parent = Vehicle.prototype;
Car.prototype.getName = function () {
  return 'It is a car: '+ this.name;
};

var car = new Car('Tesla');
console.log(car.getName()); // It is a car: Tesla
console.log(car.getType()); // car
```

And now look at the ES6 version:

```
class Vehicle {

  constructor (name, type) {
    this.name = name;
    this.type = type;
  }

  getName () {
    return this.name;
  }

  getType () {
    return this.type;
  }

}

class Car extends Vehicle {

  constructor (name) {
    super(name, 'car');
  }

  getName () {
    return 'It is a car: ' + super.getName();
  }

}

let car = new Car('Tesla');
console.log(car.getName()); // It is a car: Tesla
console.log(car.getType()); // car
```

We see how easy is to implement inheritance with ES6. It's finally looking like in other OO programming languages. We use **extends** to inherit from another class and the **super** keyword to call the parent class (function). Moreover, **getName()** method was overridden in subclass **Car**.

**super—previously to achieve such functionality in Javascript required the use of call or apply**

# static

```
class Vehicle {

  constructor (name, type) {
    this.name = name;
    this.type = type;
  }

  getName () {
    return this.name;
  }

  getType () {
    return this.type;
  }

  static create (name, type) {
    return new Vehicle(name, type);
  }

}

let car = Vehicle.create('Tesla', 'car');
console.log(car.getName()); // Tesla
console.log(car.getType()); // car
```

Classes give us an opportunity to create static members. We don't have to use the **new** keyword later to instantiate a class.

static methods (properties) are also inherited and could be called by super

# get / set

Other great things in upcoming ES6 are **getters** and **setters** for object properties. They allow us to run the code on the reading or writing of a property.

```
class Car {

  constructor (name) {
    this._name = name;
  }

  set name (name) {
    this._name = name;
  }

  get name () {
    return this._name;
  }

}

let car = new Car('Tesla');
console.log(car.name); // Tesla

car.name = 'BMW';
console.log(car.name); // BMW
```

I use '_' prefix to create a (**tmp**) field to store name property.

# Enhanced Object Properties

The last thing I have to mention is **property shorthand**, **computed property names** and **method properties**.

ES6 gives us shorter syntax for common **object property** definition:

```
// ES6
let x = 1,
    y = 2,
    obj = { x, y };

console.log(obj); // Object { x: 1, y: 2 }

// ES5
var x = 1,
    y = 2,
    obj = {
      x: x,
      y: y
    };

console.log(obj); // Object { x: 1, y: 2 }
```

As you can see, this works because the property value has the same name as the property identifier.

Another thing is ES6 support for **computed names** in object property definitions:

```
// ES6
let getKey = () => 'key_123',
    obj = {
      foo: 'bar',
      ['key_' + getKey()]: 123
    };

console.log(obj); // Object { foo: 'bar', key_123: 123 }

// ES5
var getKey = function () {
      return 'key_123';
    },
    obj = {
      foo: 'bar'
    };
obj['key_' + getKey()] = 123;

console.log(obj); // Object { foo: 'bar', key_123: 123 }
```

The one last thing is **method properties** seen in classes above. We can even use it in object definitions:

```
// ES6
let obj = {
  name: 'object name',
  toString () { // 'function' keyword is omitted here
    return this.name;
  }
};


console.log(obj.toString()); // object name

// ES5
var obj = {
  name: 'object name',
  toString: function () {
    return this.name;
  }
```

```
};

console.log(obj.toString()); // object name
```

# modules

Today we have a couple of ways to create **modules**, export & import them. JavaScript doesn't have any built-in module loader yet. Upcoming ECMAScript 2015 standard gives us a reason to make people happy. Finally ;)

We have third party standards: **CommonJS** and **AMD**. The most popular, but, unfortunately, incompatible standards for module loaders.

**CommonJS is known from Node.js. It's mostly dedicated for servers and it supports synchronous loading. It also has a compact syntax focused on export and require keywords.**

**AMD and the most popular implementation - RequireJS are dedicated for browsers. AMD supports asynchronous loading, but has more complicated syntax than CommonJS.**

The goal for ES6 is (was) to mix these two standards and make both user groups happy.

ES6 gives us an easy syntax and support for asynchronous and configurable module loading.

**Async model**—no code executes until requested modules are available and processed.

## Named export

Modules can export multiple objects, which could be simple variables or functions.

```
export function multiply (x, y) {
  return x * y;
};
```

We can also export a function stored in a variable, but we have to wrap the variable in a set of curly braces.

```
var multiply = function (x, y) {
  return x * y;
};

export { multiply };
```

We can even export many objects and like in the above example—we have to wrap exported statements in a set of curly braces if we use one export keyword.

```
export hello = 'Hello World';
export function multiply (x, y) {
  return x * y;
};

// === OR ===

var hello = 'Hello World',
    multiply = function (x, y) {
      return x * y;
    };

export { hello, multiply };
```

Let's just imagine that we have **modules.js** file with all exported statements. To import them in another file (**in the same**

**directory**) we use … **import { .. } from** .. syntax:

```
import { hello } from 'modules';
```

**We can omit .js extension just like in CommonJS and AMD.**

We can even import many statements:

```
import { hello, multiply } from 'modules';
```

Imports may also be **aliased**:

```
import { multiply as pow2 } from 'modules';
```

..and use **wildcard** (*) to import all exported statemets:

```
import * from 'modules';
```

# Default export

In our module, we can have many named exports, but we can also have a **default export**. It's because our module could be a large library and with default export we can import then an entire module. It could be also useful when our module has single value or model (class / constructor).

**One default export per module.**

```
export default function (x, y) {
  return x * y;
};
```

This time we don't have to use curly braces for importing and we have a chance to name imported statement as we wish.

```
import multiply from 'modules';

// === OR ===

import pow2 from 'modules';

// === OR ===

...
```

**Module can have both named exports and a default export.**

```
// modules.js
export hello = 'Hello World';
export default function (x, y) {
  return x * y;
};

// app.js
```

```
import pow2, { hello } from 'modules';
```

**The default export is just a named export with the special name default.**

```
// modules.js
export default function (x, y) {
  return x * y;
};

// app.js
import { default } from 'modules';
```

# API

In addition, there is also a programmatic API and it allows to:

- Programmatically work with modules and scripts
- Configure module loading

**SystemJS—universal dynamic module loader—loads ES6 modules, AMD, CommonJS and global scripts in the browser and NodeJS. Works with both Traceur and Babel.**

Module loader should support:

- Dynamic loading
- Global namespace isolation
- Nested virtualization
- Compilation hooks

# promises

Promises aren't a new and shiny idea. I use it every day in my AngularJS code. It's based on kriskowal / q library:

**A tool for creating and composing asynchronous promises in JavaScript.**

It's a library for asynchronous programming, to make our life easier. But, before I describe promises, I have to write something about callbacks.

## Callbacks and callback hell

Until I remember, JavaScript coders use callbacks for all browser-based asynchronous functions (setTimeout, XMLHttpRequest, etc.).

Look at naive example:

```
console.log('start!');
setTimeout(function () {
  console.log('ping');
  setTimeout(function () {
    console.log('pong');
    setTimeout(function () {
      console.log('end!');
    }, 1000);
  }, 1000);
}, 1000);

// start!
// after 1 sec: ping
// .. 1 sec later: pong
// .. and: end!
```

We have simple code which prints some statements to the console. I used a **setTimeout** function here, to show callback functions passed to invoke later (1 sec here). It looks terrible and we have only 3 steps here. Let's imagine more steps. It will look like you build a pyramid, not nice, readable code. Awful, right? It's called **callback hell** and we have it everywhere.

Callback Hell

## Promises

Support for promises is a very nice addition to the language. It's finally native in the ES6.

**Promises are a first class representation of a value that may be made available in the future.**

A promise can be:

- **fulfilled** - promise succeeded
- **rejected** - promise failed
- **pending** - not fulfilled or not rejected yet
- **settled** - fulfilled or rejected

Every returned **promise object** also has a **then** method to execute code when a promise is **settled**.

Yep, promise **object**, because..

**callbacks are functions, promises are objects.**

Callbacks are blocks of code to execute in response to.. something (event). Promises are objects which store an information about the state.

How does it look like? Let's see:

```
new Promise((resolve, reject) => {
  // when success, resolve
  let value = 'success';
  resolve(value);

  // when an error occurred, reject
  reject(new Error('Something happened!'));
});
```

Promise calls its **resolve function** when it's fulfilled (success) and **reject function** otherwise (failure).

Promises are objects, so it's not passed as arguments like callbacks, it's **returned**. The return statement is an object which is a placeholder for the result, which will be available in the future.

Promises have just one responsibility-**they represent only one** event. Callbacks can handle multiple events, many times.

We can assign returned value (object) to the **let statement**:

```
let promise = new Promise((resolve, reject) => {
  // when success, resolve
  let value = 'success';
  resolve(value);

  // when an error occurred, reject
  reject(new Error('Something happened!'));
});
```

As I mentioned above-promise object also has a **then** method to execute code when the promise is settled.

```
promise.then(onResolve, onReject)
```

We can use this function to handle **onResolve** and **onReject** values returned by a promise. We can handle **success**, **failure** or **both**.

```
let promise = new Promise((resolve, reject) => {
  // when success, resolve
  let value = 'success';
  resolve(value);

  // when an error occurred, reject
  reject(new Error('Something happened!'));
});

promise.then(response => {
  console.log(response);
}, error => {
  console.log(error);
});
```

```
// success
```

Our code above never executes **reject** function, so we can omit it for simplicity:

```
let promise = new Promise(resolve => {
  let value = 'success';
  resolve(value);
});

promise.then(response => {
  console.log(response); // success
});
```

Handlers passed to **promise.then** don't just handle the result of the previous promise-they return is turned into a **new promise**.

```
let promise = new Promise(resolve => {
  let value = 'success';
  resolve(value);
});

promise.then(response => {
  console.log(response); // success
  return 'another success';
}).then(response => {
  console.log(response); // another success
});
```

You can see, that the code based on promises is always *flat*. No more **callback hell**.

If you are only interested in rejections, you can omit the first parameter.

```
let promise = new Promise((resolve, reject) => {
  let reason = 'failure';
  reject(reason);
});

promise.then(
  null,
  error => {
    console.log(error); // failure
  }
);
```

But is a more compact way of doing the same thing-**catch()** method.

```
let promise = new Promise((resolve, reject) => {
  let reason = 'failure';
  reject(reason);
});

promise.catch(err => {
  console.log(err); // failure
});
```

If we have more than one **then()** call, the error is passed on until there is an error handler.

```
let promise = new Promise(resolve => {
  resolve();
});
```

```
promise
  .then(response => {
    return 1;
  })
  .then(response => {
    throw new Error('failure');
  })
  .catch(error => {
    console.log(error.message); // failure
  });
```

We can even combine **one or more promises** into new promises without having to take care of ordering of the underlying asynchronous operations yourself.

```
let doSmth = new Promise(resolve => {
    resolve('doSmth');
  }),
  doSmthElse = new Promise(resolve => {
    resolve('doSmthElse');
  }),
  oneMore = new Promise(resolve => {
    resolve('oneMore');
  });

Promise.all([
    doSmth,
    doSmthElse,
    oneMore
  ])
  .then(response => {
    let [one, two, three] = response;
    console.log(one, two, three); // doSmth doSmthElse oneMore
  });
```

**Promise.all()** takes an array of promises and when all of them are fulfilled, it put their values into the array.

There are two more functions which are useful:

- **Promise.resolve(value)** - it returns a promise which resolves to a **value** or returns **value** if **value** is already a promise
- **Promise.reject(value)** - returns rejected promise with **value** as **value**

# Pitfall

Promises have its pitfall as well. Let's image that when any exception is thrown within a **then** or the function passed to **new Promise**, will be silently disposed of **unless manually handled**.

# set, map, weak

Sets and maps will be (are) finally available in ES6! No more spartan way to manipulate data structures. This chapter explains how we can deal with **Map**, **Set**, **WeakMap** and **WeakSet**.

## Map

Maps are a store for **key** / **value** pairs. Key and value could be a primitives or object references.

Let's create a map:

```
let map = new Map(),
    val2 = 'val2',
    val3 = {
      key: 'value'
    };

map.set(0, 'val1');
map.set('1', val2);
map.set({ key: 2 }, val3);

console.log(map); // Map {0 => 'val1', '1' => 'val2', Object {key: 2} => Object {key: 'value'}}
```

We can also use a constructor to create the sam map, based on array param passed to the constructor:

```
let map,
    val2 = 'val2',
    val3 = {
      key: 'value'
    };

map = new Map([[0, 'val1'], ['1', val2], [{ key: 2 }, val3]]);

console.log(map); // Map {0 => 'val1', '1' => 'val2', Object {key: 2} => Object {key: 'value'}}
```

To get a value by using a key, we have to use a **get()** method to do it (surprising):

```
let map = new Map(),
    val2 = 'val2',
    val3 = {
      key: 'value'
    };

map.set(0, 'val1');
map.set('1', val2);
map.set({ key: 2 }, val3);

console.log(map.get('1')); // val2
```

To iterate over the map collection, we can use built-in **forEach** method or use new **for..of** structure:

```
// forEach
let map = new Map(),
    val2 = 'val2',
    val3 = {
      key: 'value'
    };
```

```
map.set(0, 'val1');
map.set('1', val2);
map.set({ key: 2 }, val3);

map.forEach(function (value, key) {
  console.log(`Key: ${key} has value: ${value}`);
  // Key: 0 has value: val1
  // Key: 1 has value: val2
  // Key: [object Object] has value: [object Object]
});
```

```
// for..of
let map = new Map(),
    val2 = 'val2',
    val3 = {
      key: 'value'
    };

map.set(0, 'val1');
map.set('1', val2);
map.set({ key: 2 }, val3);

for (let entry of map) {
  console.log(`Key: ${entry[0]} has value: ${entry[1]}`);
  // Key: 0 has value: val1
  // Key: 1 has value: val2
  // Key: [object Object] has value: [object Object]
};
```

We can also use a couple of methods to iterate:

- **entries()**—get all entries
- **keys()**—get only all keys
- **values()**—get only all values

To check if value is stored in our map, we can use **has()** method:

```
let map = new Map(),
    val2 = 'val2',
    val3 = {
      key: 'value'
    };

map.set(0, 'val1');
map.set('1', val2);
map.set({ key: 2 }, val3);

console.log(map.has(0));     // true
console.log(map.has('key')); // false
```

To delete entry, we have **delete()** method:

```
let map = new Map(),
    val2 = 'val2',
    val3 = {
      key: 'value'
    };

map.set(0, 'val1');
map.set('1', val2);
map.set({ key: 2 }, val3);

console.log(map.size); // 3

map.delete('1');
```

```
console.log(map.size); // 2
```

..and to clear all collection, we use clear() method:

```
let map = new Map(),
    val2 = 'val2',
    val3 = {
      key: 'value'
    };

map.set(0, 'val1');
map.set('1', val2);
map.set({ key: 2 }, val3);

console.log(map.size); // 3

map.clear();

console.log(map.size); // 0
```

# Set

It's a collection for *unique* values. The values could be also a primitives or object references.

```
let set = new Set();

set.add(1);
set.add('1');
set.add({ key: 'value' });

console.log(set); // Set {1, '1', Object {key: 'value'}}
```

Like a map, set allows to create collection by passing an array to its constructor:

```
let set = new Set([1, '1', { key: 'value' }]);

console.log(set); // Set {1, '1', Object {key: 'value'}}
```

To iterate over sets we have the same two options—built-in **forEach** function or **for..of** structure:

```
// forEach
let set = new Set([1, '1', { key: 'value' }]);

set.forEach(function (value) {
  console.log(value);
  // 1
  // '1'
  // Object {key: 'value'}
});
```

```
// for..of
let set = new Set([1, '1', { key: 'value' }]);

for (let value of set) {
  console.log(value);
  // 1
  // '1'
  // Object {key: 'value'}
};
```

**Set doesn't allow to add duplicates.**

```
let set = new Set([1, 1, 1, 2, 5, 5, 6, 9]);
console.log(set.size); // 5!
```

We can also use **has()**, **delete()**, **clear()** methods, which are similar to the Map versions.

# WeakMap

WeakMaps provides leak-free object keyed side tables. It's a Map that doesn't prevent its keys from being **garbage-collected**. We don't have to worry about memory leaks.

If the object is destroyed, the garbage collector removes an entry from the WeakMap and frees memory.

**Keys must be objects.**

It has almost the same API like a Map, but we **can't iterate** over the WeakMap collection. We can't even determine the length of the collection because we don't have **size** attribute here.

The API looks like this:

```
new WeakMap([iterable])

WeakMap.prototype.get(key)        : any
WeakMap.prototype.set(key, value) : this
WeakMap.prototype.has(key)        : boolean
WeakMap.prototype.delete(key)     : boolean
```

```
let wm = new WeakMap(),
    obj = {
      key1: {
        k: 'v1'
      },
      key2: {
        k: 'v2'
      }
   };

wm.set(obj.key1, 'val1');
wm.set(obj.key2, 'val2');

console.log(wm); // WeakMap {Object {k: 'v1'} => 'val1', Object {k: 'v2'} => 'val2'}
console.log(wm.has(obj.key1)); // true

delete obj.key1;

console.log(wm.has(obj.key1)); // false
```

# WeakSet

Like a WeakMap, WeakSet is a Seat that doesn't prevent its values from being garbage-collected. It has simpler API than WeakMap, because has only three methods:

```
new WeakSet([iterable])
```

```
WeakSet.prototype.add(value)    : any
WeakSet.prototype.has(value)    : boolean
WeakSet.prototype.delete(value) : boolean
```

**WeakSets are collections of unique objects only.**

WeakSet collection can't be iterated and we cannot determine its size.

```
let ws = new WeakSet(),
    obj = {
      key1: {
        k: 'v1'
      },
      key2: {
        k: 'v2'
      }
   };

ws.add(obj.key1);
ws.add(obj.key2);

console.log(ws); // WeakSet {Object {k: 'v1'}, Object {k: 'v2'}}
console.log(ws.has(obj.key1)); // true

delete obj.key1;

console.log(ws.has(obj.key1)); // false
```