JavaScript from ES5 to ESNext

FLAVIO COPES

Table of Contents

Preface

Welcome!

I wrote this book to help you move from pre-ES6 knowledge of JavaScript and get you quickly up to speed with the most recent advancements of the language.

JavaScript today is in the privileged position to be the only language that can run natively in the browser, and is highly integrated and optimized for that.

The future of JavaScript is going to be brilliant. Keeping up with the changes shouldn't be harder than it already is, and my goal here is to give you a quick yet comprehensive overview of the new stuff available to us.

Thank you for getting this ebook. I hope its content will help you achieve what you want.

Flavio

You can reach me via email at flavio@flaviocopes.com, on Twitter @flaviocopes.

My website is flaviocopes.com.

Introduction to ECMAScript

Whenever you read about JavaScript you'll inevitably see one of these terms:

- ES3
- ES5
- ES6
- ES7
- ES8
- ES2015
- ES2016
- ES2017
- ECMAScript 2017
- ECMAScript 2016
- ECMAScript 2015

What do they mean?

They are all referring to a **standard**, called ECMAScript.

ECMAScript is **the standard upon which JavaScript is based**, and it's often abbreviated to **ES**.

Beside JavaScript, other languages implement(ed) ECMAScript, including:

- *ActionScript* (the Flash scripting language), which is losing popularity since Flash will be officially discontinued in 2020
- *JScript* (the Microsoft scripting dialect), since at the time JavaScript was supported only by Netscape and the browser wars were at their peak, Microsoft had to build its own version for Internet Explorer

but of course JavaScript is the **most popular** and widely used implementation of ES.

Why this weird name? `Ecma International` is a Swiss standards association who is in charge of defining international standards.

When JavaScript was created, it was presented by Netscape and Sun Microsystems to Ecma and they gave it the name ECMA-262 alias **ECMAScript**.

This press release by Netscape and Sun Microsystems (the maker of Java) might help figure out the name choice, which might include legal and branding issues by Microsoft which was in the committee, according to Wikipedia.

After IE9, Microsoft stopped branding its ES support in browsers as JScript and started calling it JavaScript (at least, I could not find references to it any more)

So as of 201x, the only popular language supporting the ECMAScript spec is JavaScript.

Current ECMAScript version

The current ECMAScript version is **ES2018**.

It was released in June 2018.

What is TC39

TC39 is the committee that evolves JavaScript.

The members of TC39 are companies involved in JavaScript and browser vendors, including Mozilla, Google, Facebook, Apple, Microsoft, Intel, PayPal, SalesForce and others.

Every standard version proposal must go through various stages, which are explained here.

ES Versions

I found it puzzling why sometimes an ES version is referenced by edition number and sometimes by year, and I am confused by the year by chance being -1 on the number, which adds to the general confusion around JS/ES

Before ES2015, ECMAScript specifications were commonly called by their edition. So ES5 is the official name for the ECMAScript specification update published in 2009.

Why does this happen? During the process that led to ES2015, the name was changed from ES6 to ES2015, but since this was done late, people still referenced it as ES6, and the community has not left the edition naming behind - *the world is still calling ES releases by edition number*.

This table should clear things a bit:

Edition	Official name	Date published
ES9	ES2018	June 2018
ES8	ES2017	June 2017
ES7	ES2016	June 2016
ES6	ES2015	June 2015
ES5.1	ES5.1	June 2011
ES5	ES5	December 2009
ES4	ES4	Abandoned
ES3	ES3	December 1999
ES2	ES2	June 1998
ES1	ES1	June 1997

Let's dive into the specific features added to JavaScript since ES5.

ES2015

let and const

Until ES2015, `var` was the only construct available for defining variables.

```
var a = 0
```

If you forget to add `var` you will be assigning a value to an undeclared variable, and the results might vary.

In modern environments, with strict mode enabled, you will get an error. In older environments (or with strict mode disabled) this will initialize the variable and assign it to the global object.

If you don't initialize the variable when you declare it, it will have the `undefined` value until you assign a value to it.

```
var a //typeof a === 'undefined'
```

You can redeclare the variable many times, overriding it:

```
var a = 1
var a = 2
```

You can also declare multiple variables at once in the same statement:

```
var a = 1, b = 2
```

The **scope** is the portion of code where the variable is visible.

A variable initialized with `var` outside of any function is assigned to the global object, has a global scope and is visible everywhere. A variable initialized with `var` inside a function is assigned to that function, it's local and is visible only inside it, just like a function parameter.

Any variable defined in a function with the same name as a global variable takes precedence over the global variable, shadowing it.

It's important to understand that a block (identified by a pair of curly braces) does not define a new scope. A new scope is only created when a function is created, because `var` does not have block scope, but function scope.

Inside a function, any variable defined in it is visible throughout all the function code, even if the variable is declared at the end of the function it can still be referenced in the beginning, because JavaScript before executing the code actually *moves all variables on top* (something

that is called **hoisting**). To avoid confusion, always declare variables at the beginning of a function.

Using `let`

`let` is a new feature introduced in ES2015 and it's essentially a block scoped version of `var`. Its scope is limited to the block, statement or expression where it's defined, and all the contained inner blocks.

Modern JavaScript developers might choose to only use `let` and completely discard the use of `var`.

> If `let` seems an obscure term, just read `let color = 'red'` as *let the color be red* and it all makes much more sense

Defining `let` outside of any function - contrary to `var` - does not create a global variable.

Using `const`

Variables declared with `var` or `let` can be changed later on in the program, and reassigned. Once a `const` is initialized, its value can never be changed again, and it can't be reassigned to a different value.

```
const a = 'test'
```

We can't assign a different literal to the `a` const. We can however mutate `a` if it's an object that provides methods that mutate its contents.

`const` does not provide immutability, just makes sure that the reference can't be changed.

`const` has block scope, same as `let`.

Modern JavaScript developers might choose to always use `const` for variables that don't need to be reassigned later in the program, because we should always use the simplest construct available to avoid making errors down the road.

Arrow Functions

Arrow functions, since their introduction, changed forever how JavaScript code looks (and works).

In my opinion this change was so welcoming that you now rarely see the usage of the `function` keyword in modern codebases. Although that has still its usage.

Visually, it's a simple and welcome change, which allows you to write functions with a shorter syntax, from:

```
const myFunction = function() {
  //...
}
```

to

```
const myFunction = () => {
  //...
}
```

If the function body contains just a single statement, you can omit the brackets and write all on a single line:

```
const myFunction = () => doSomething()
```

Parameters are passed in the parentheses:

```
const myFunction = (param1, param2) => doSomething(param1, param2)
```

If you have one (and just one) parameter, you could omit the parentheses completely:

```
const myFunction = param => doSomething(param)
```

Thanks to this short syntax, arrow functions **encourage the use of small functions**.

Implicit return

Arrow functions allow you to have an implicit return: values are returned without having to use the `return` keyword.

It works when there is a one-line statement in the function body:

```
const myFunction = () => 'test'

myFunction() //'test'
```

Another example, when returning an object, remember to wrap the curly brackets in parentheses to avoid it being considered the wrapping function body brackets:

```
const myFunction = () => ({ value: 'test' })

myFunction() //{value: 'test'}
```

How `this` works in arrow functions

`this` is a concept that can be complicated to grasp, as it varies a lot depending on the context and also varies depending on the mode of JavaScript (*strict mode* or not).

It's important to clarify this concept because arrow functions behave very differently compared to regular functions.

When defined as a method of an object, in a regular function `this` refers to the object, so you can do:

```
const car = {
  model: 'Fiesta',
  manufacturer: 'Ford',
  fullName: function() {
    return `${this.manufacturer} ${this.model}`
  }
}
```

calling `car.fullName()` will return `"Ford Fiesta"`.

The `this` scope with arrow functions is **inherited** from the execution context. An arrow function does not bind `this` at all, so its value will be looked up in the call stack, so in this code `car.fullName()` will not work, and will return the string `"undefined undefined"`:

```
const car = {
  model: 'Fiesta',
  manufacturer: 'Ford',
  fullName: () => {
    return `${this.manufacturer} ${this.model}`
  }
}
```

Due to this, arrow functions are not suited as object methods.

Arrow functions cannot be used as constructors either, when instantiating an object will raise a `TypeError`.

This is where regular functions should be used instead, **when dynamic context is not needed**.

This is also a problem when handling events. DOM Event listeners set `this` to be the target element, and if you rely on `this` in an event handler, a regular function is necessary:

```
const link = document.querySelector('#link')
link.addEventListener('click', () => {
  // this === window
})
```

```
const link = document.querySelector('#link')
link.addEventListener('click', function() {
  // this === link
})
```

Classes

JavaScript has a quite uncommon way to implement inheritance: prototypical inheritance. Prototypal inheritance, while in my opinion great, is unlike most other popular programming language's implementation of inheritance, which is class-based.

People coming from Java or Python or other languages had a hard time understanding the intricacies of prototypal inheritance, so the ECMAScript committee decided to sprinkle syntactic sugar on top of prototypical inheritance so that it resembles how class-based inheritance works in other popular implementations.

This is important: JavaScript under the hood is still the same, and you can access an object prototype in the usual way.

A class definition

This is how a class looks.

```
class Person {
  constructor(name) {
    this.name = name
  }

  hello() {
    return 'Hello, I am ' + this.name + '.'
  }
}
```

A class has an identifier, which we can use to create new objects using `new ClassIdentifier()`.

When the object is initialized, the `constructor` method is called, with any parameters passed.

A class also has as many methods as it needs. In this case `hello` is a method and can be called on all objects derived from this class:

```
const flavio = new Person('Flavio')
flavio.hello()
```

Class inheritance

A class can extend another class, and objects initialized using that class inherit all the methods of both classes.

If the inherited class has a method with the same name as one of the classes higher in the hierarchy, the closest method takes precedence:

```
class Programmer extends Person {
  hello() {
    return super.hello() + ' I am a programmer.'
  }
}

const flavio = new Programmer('Flavio')
flavio.hello()
```

(the above program prints "*Hello, I am Flavio. I am a programmer.*")

Classes do not have explicit class variable declarations, but you must initialize any variable in the constructor.

Inside a class, you can reference the parent class calling `super()`.

Static methods

Normally methods are defined on the instance, not on the class.

Static methods are executed on the class instead:

```
class Person {
  static genericHello() {
    return 'Hello'
  }
}

Person.genericHello() //Hello
```

Private methods

JavaScript does not have a built-in way to define private or protected methods.

There are workarounds, but I won't describe them here.

Getters and setters

You can add methods prefixed with `get` or `set` to create a getter and setter, which are two different pieces of code that are executed based on what you are doing: accessing the variable, or modifying its value.

```
class Person {
  constructor(name) {
    this.name = name
  }

  set name(value) {
    this.name = value
  }

  get name() {
    return this.name
  }
}
```

If you only have a getter, the property cannot be set, and any attempt at doing so will be ignored:

```
class Person {
  constructor(name) {
    this.name = name
  }

  get name() {
    return this.name
  }
}
```

If you only have a setter, you can change the value but not access it from the outside:

```
class Person {
  constructor(name) {
    this.name = name
  }

  set name(value) {
    this.name = value
  }
}
```

Default parameters

Default parameter values have been introduced in ES2015, and are widely implemented in modern browsers.

This is a `doSomething` function which accepts `param1`.

```
const doSomething = (param1) => {

}
```

We can add a default value for `param1` if the function is invoked without specifying a parameter:

```
const doSomething = (param1 = 'test') => {

}
```

This works for more parameters as well, of course:

```
const doSomething = (param1 = 'test', param2 = 'test2') => {

}
```

What if you have an unique object with parameters values in it?

Once upon a time, if we had to pass an object of options to a function, in order to have default values of those options if one of them was not defined, you had to add a little bit of code inside the function:

```
const colorize = (options) => {
  if (!options) {
    options = {}
  }

  const color = ('color' in options) ? options.color : 'yellow'
  ...
}
```

With destructuring you can provide default values, which simplifies the code a lot:

```
const colorize = ({ color = 'yellow' }) => {
  ...
}
```

If no object is passed when calling our `colorize` function, similarly we can assign an empty object by default:

```
const spin = ({ color = 'yellow' } = {}) => {
  ...
}
```

Template Literals

Template Literals allow you to work with strings in a novel way compared to ES5 and below.

The syntax at a first glance is very simple, just use backticks instead of single or double quotes:

```
const a_string = `something`
```

They are unique because they provide a lot of features that normal strings built with quotes do not, in particular:

- they offer a great syntax to define multiline strings
- they provide an easy way to interpolate variables and expressions in strings
- they allow you to create DSLs with template tags (DSL means domain specific language, and it's for example used in React by Styled Components, to define CSS for a component)

Let's dive into each of these in detail.

Multiline strings

Pre-ES6, to create a string spanning over two lines you had to use the `\` character at the end of a line:

```
const string =
  'first part \
second part'
```

This allows to create a string on 2 lines, but it's rendered on just one line:

```
first part second part
```

To render the string on multiple lines as well, you explicitly need to add `\n` at the end of each line, like this:

```
const string =
  'first line\n \
second line'
```

or

```
const string = 'first line\n' + 'second line'
```

Template literals make multiline strings much simpler.

Once a template literal is opened with the backtick, you just press enter to create a new line, with no special characters, and it's rendered as-is:

```
const string = `Hey
this

string
is awesome!`
```

Keep in mind that space is meaningful, so doing this:

```
const string = `First
               Second`
```

is going to create a string like this:

```
First
               Second
```

an easy way to fix this problem is by having an empty first line, and appending the trim() method right after the closing backtick, which will eliminate any space before the first character:

```
const string = `
First
Second`.trim()
```

Interpolation

Template literals provide an easy way to interpolate variables and expressions into strings.

You do so by using the `${...}` syntax:

```
const var = 'test'
const string = `something ${var}` //something test
```

inside the `${}` you can add anything, even expressions:

```
const string = `something ${1 + 2 + 3}`
const string2 = `something ${foo() ? 'x' : 'y'}`
```

Template tags

Tagged templates is one feature that might sound less useful at first for you, but it's actually used by lots of popular libraries around, like Styled Components or Apollo, the GraphQL client/server lib, so it's essential to understand how it works.

In Styled Components template tags are used to define CSS strings:

```
const Button = styled.button`
  font-size: 1.5em;
  background-color: black;
  color: white;
`
```

In Apollo template tags are used to define a GraphQL query schema:

```
const query = gql`
  query {
    ...
  }
`
```

The `styled.button` and `gql` template tags highlighted in those examples are just **functions**:

```
function gql(literals, ...expressions) {}
```

this function returns a string, which can be the result of *any* kind of computation.

`literals` is an array containing the template literal content tokenized by the expressions interpolations.

`expressions` contains all the interpolations.

If we take an example above:

```
const string = `something ${1 + 2 + 3}`
```

`literals` is an array with two items. The first is `something `, the string until the first interpolation, and the second is an empty string, the space between the end of the first interpolation (we only have one) and the end of the string.

`expressions` in this case is an array with a single item, `6`.

A more complex example is:

```
const string = `something
another ${'x'}
new line ${1 + 2 + 3}
test`
```

in this case `literals` is an array where the first item is:

```
;`something
another `
```

the second is:

```
;`
new line `
```

and the third is:

```
;`
test`
```

`expressions` in this case is an array with two items, `x` and `6`.

The function that is passed those values can do anything with them, and this is the power of this kind feature.

The most simple example is replicating what the string interpolation does, by joining `literals` and `expressions`:

```
const interpolated = interpolate`I paid ${10}€`
```

and this is how `interpolate` works:

```
function interpolate(literals, ...expressions) {
  let string = ``
  for (const [i, val] of expressions) {
    string += literals[i] + val
  }
  string += literals[literals.length - 1]
  return string
}
```

Destructuring assignments

Given an object, you can extract just some values and put them into named variables:

```
const person = {
  firstName: 'Tom',
  lastName: 'Cruise',
  actor: true,
  age: 54, //made up
}

const {firstName: name, age} = person
```

`name` and `age` contain the desired values.

The syntax also works on arrays:

```
const a = [1,2,3,4,5]
const [first, second] = a
```

This statement creates 3 new variables by getting the items with index 0, 1, 4 from the array `a`:

```
const [first, second, , , fifth] = a
```

Enhanced Object Literals

In ES2015 Object Literals gained superpowers.

Simpler syntax to include variables

Instead of doing

```
const something = 'y'
const x = {
  something: something
}
```

you can do

```
const something = 'y'
const x = {
  something
}
```

Prototype

A prototype can be specified with

```
const anObject = { y: 'y' }
const x = {
  __proto__: anObject
}
```

super()

```
const anObject = { y: 'y', test: () => 'zoo' }
const x = {
  __proto__: anObject,
  test() {
    return super.test() + 'x'
  }
}
x.test() //zoox
```

Dynamic properties

```
const x = {
```

```
  ['a' + '_' + 'b']: 'z'
}
x.a_b //z
```

For-of loop

ES5 back in 2009 introduced `forEach()` loops. While nice, they offered no way to break, like `for` loops always did.

ES2015 introduced the **`for-of` loop**, which combines the conciseness of `forEach` with the ability to break:

```
//iterate over the value
for (const v of ['a', 'b', 'c']) {
  console.log(v);
}

//get the index as well, using `entries()`
for (const [i, v] of ['a', 'b', 'c'].entries()) {
  console.log(index) //index
  console.log(value) //value
}
```

Notice the use of `const`. This loop creates a new scope in every iteration, so we can safely use that instead of `let`.

The difference with `for...in` is:

- `for...of` **iterates over the property values**
- `for...in` **iterates the property names**

Promises

A promise is commonly defined as **a proxy for a value that will eventually become available**.

Promises are one way to deal with asynchronous code, without writing too many callbacks in your code.

Async functions use the promises API as their building block, so understanding them is fundamental even if in newer code you'll likely use async functions instead of promises.

How promises work, in brief

Once a promise has been called, it will start in **pending state**. This means that the caller function continues the execution, while it waits for the promise to do its own processing, and give the caller function some feedback.

At this point, the caller function waits for it to either return the promise in a **resolved state**, or in a **rejected state**, but as you know JavaScript is asynchronous, so *the function continues its execution while the promise does it work*.

Which JS API use promises?

In addition to your own code and library code, promises are used by standard modern Web APIs such as:

- the Battery API
- the Fetch API
- Service Workers

It's unlikely that in modern JavaScript you'll find yourself *not* using promises, so let's start diving right into them.

Creating a promise

The Promise API exposes a Promise constructor, which you initialize using `new Promise()`:

```
let done = true

const isItDoneYet = new Promise((resolve, reject) => {
  if (done) {
```

```
    const workDone = 'Here is the thing I built'
    resolve(workDone)
  } else {
    const why = 'Still working on something else'
    reject(why)
  }
})
```

As you can see the promise checks the `done` global constant, and if that's true, we return a resolved promise, otherwise a rejected promise.

Using `resolve` and `reject` we can communicate back a value, in the above case we just return a string, but it could be an object as well.

Consuming a promise

In the last section, we introduced how a promise is created.

Now let's see how the promise can be *consumed* or used.

```
const isItDoneYet = new Promise()
//...

const checkIfItsDone = () => {
  isItDoneYet
    .then(ok => {
      console.log(ok)
    })
    .catch(err => {
      console.error(err)
    })
}
```

Running `checkIfItsDone()` will execute the `isItDoneYet()` promise and will wait for it to resolve, using the `then` callback, and if there is an error, it will handle it in the `catch` callback.

Chaining promises

A promise can be returned to another promise, creating a chain of promises.

A great example of chaining promises is given by the Fetch API, a layer on top of the XMLHttpRequest API, which we can use to get a resource and queue a chain of promises to execute when the resource is fetched.

The Fetch API is a promise-based mechanism, and calling `fetch()` is equivalent to defining our own promise using `new Promise()`.

Example of chaining promises

```
const status = response => {
  if (response.status >= 200 && response.status < 300) {
    return Promise.resolve(response)
  }
  return Promise.reject(new Error(response.statusText))
}

const json = response => response.json()

fetch('/todos.json')
  .then(status)
  .then(json)
  .then(data => {
    console.log('Request succeeded with JSON response', data)
  })
  .catch(error => {
    console.log('Request failed', error)
  })
```

In this example, we call `fetch()` to get a list of TODO items from the `todos.json` file found in the domain root, and we create a chain of promises.

Running `fetch()` returns a response, which has many properties, and within those we reference:

- `status`, a numeric value representing the HTTP status code
- `statusText`, a status message, which is `OK` if the request succeeded

`response` also has a `json()` method, which returns a promise that will resolve with the content of the body processed and transformed into JSON.

So given those premises, this is what happens: the first promise in the chain is a function that we defined, called `status()`, that checks the response status and if it's not a success response (between 200 and 299), it rejects the promise.

This operation will cause the promise chain to skip all the chained promises listed and will skip directly to the `catch()` statement at the bottom, logging the `Request failed` text along with the error message.

If that succeeds instead, it calls the json() function we defined. Since the previous promise, when successful, returned the `response` object, we get it as an input to the second promise.

In this case, we return the data JSON processed, so the third promise receives the JSON directly:

```
.then((data) => {
  console.log('Request succeeded with JSON response', data)
})
```

and we log it to the console.

Handling errors

In the above example, in the previous section, we had a `catch` that was appended to the chain of promises.

When anything in the chain of promises fails and raises an error or rejects the promise, the control goes to the nearest `catch()` statement down the chain.

```
new Promise((resolve, reject) => {
  throw new Error('Error')
}).catch(err => {
  console.error(err)
})

// or

new Promise((resolve, reject) => {
  reject('Error')
}).catch(err => {
  console.error(err)
})
```

Cascading errors

If inside the `catch()` you raise an error, you can append a second `catch()` to handle it, and so on.

```
new Promise((resolve, reject) => {
  throw new Error('Error')
})
  .catch(err => {
    throw new Error('Error')
  })
  .catch(err => {
    console.error(err)
  })
```

Orchestrating promises

`Promise.all()`

If you need to synchronize different promises, `Promise.all()` helps you define a list of promises, and execute something when they are all resolved.

Example:

```
const f1 = fetch('/something.json')
const f2 = fetch('/something2.json')

Promise.all([f1, f2])
  .then(res => {
    console.log('Array of results', res)
  })
  .catch(err => {
    console.error(err)
  })
```

The ES2015 destructuring assignment syntax allows you to also do

```
Promise.all([f1, f2]).then(([res1, res2]) => {
  console.log('Results', res1, res2)
})
```

You are not limited to using `fetch` of course, **any promise is good to go**.

`Promise.race()`

`Promise.race()` runs as soon as one of the promises you pass to it resolves, and it runs the attached callback just once with the result of the first promise resolved.

Example:

```
const promiseOne = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'one')
})
const promiseTwo = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'two')
})

Promise.race([promiseOne, promiseTwo]).then(result => {
  console.log(result) // 'two'
})
```

Common errors

Uncaught TypeError: undefined is not a promise

If you get the `Uncaught TypeError: undefined is not a promise` error in the console, make sure you use `new Promise()` instead of just `Promise()`

Modules

ES Modules is the ECMAScript standard for working with modules.

While Node.js has been using the CommonJS standard for years, the browser never had a module system, as every major decision such as a module system must be first standardized by ECMAScript and then implemented by the browser.

This standardization process completed with ES2015 and browsers started implementing this standard trying to keep everything well aligned, working all in the same way, and now ES Modules are supported in Chrome, Safari, Edge and Firefox (since version 60).

Modules are very cool, because they let you encapsulate all sorts of functionality, and expose this functionality to other JavaScript files, as libraries.

The ES Modules Syntax

The syntax to import a module is:

```
import package from 'module-name'
```

while CommonJS uses

```
const package = require('module-name')
```

A module is a JavaScript file that **exports** one or more values (objects, functions or variables), using the `export` keyword. For example, this module exports a function that returns a string uppercase:

> uppercase.js

```
export default str => str.toUpperCase()
```

In this example, the module defines a single, **default export**, so it can be an anonymous function. Otherwise it would need a name to distinguish it from other exports.

Now, **any other JavaScript module** can import the functionality offered by uppercase.js by importing it.

An HTML page can add a module by using a `<script>` tag with the special `type="module"` attribute:

```
<script type="module" src="index.js"></script>
```

> Note: this module import behaves like a `defer` script load. See efficiently load JavaScript with defer and async

It's important to note that any script loaded with `type="module"` is loaded in strict mode.

In this example, the `uppercase.js` module defines a **default export**, so when we import it, we can assign it a name we prefer:

```
import toUpperCase from './uppercase.js'
```

and we can use it:

```
toUpperCase('test') //'TEST'
```

You can also use an absolute path for the module import, to reference modules defined on another domain:

```
import toUpperCase from 'https://flavio-es-modules-example.glitch.me/uppercase.js'
```

This is also valid import syntax:

```
import { toUpperCase } from '/uppercase.js'
import { toUpperCase } from '../uppercase.js'
```

This is not:

```
import { toUpperCase } from 'uppercase.js'
import { toUpperCase } from 'utils/uppercase.js'
```

It's either absolute, or has a `./` or `/` before the name.

Other import/export options

We saw this example above:

```
export default str => str.toUpperCase()
```

This creates one default export. In a file however you can export more than one thing, by using this syntax:

```
const a = 1
const b = 2
const c = 3

export { a, b, c }
```

Another module can import all those exports using

```
import * from 'module'
```

You can import just a few of those exports, using the destructuring assignment:

```
import { a } from 'module'
import { a, b } from 'module'
```

You can rename any import, for convenience, using `as`:

```
import { a, b as two } from 'module'
```

You can import the default export, and any non-default export by name, like in this common React import:

```
import React, { Component } from 'react'
```

You can see an ES Modules example here: https://glitch.com/edit/#!/flavio-es-modules-example?path=index.html

CORS

Modules are fetched using CORS. This means that if you reference scripts from other domains, they must have a valid CORS header that allows cross-site loading (like `Access-Control-Allow-Origin: *`)

What about browsers that do not support modules?

Use a combination of `type="module"` and `nomodule`:

```
<script type="module" src="module.js"></script>
<script nomodule src="fallback.js"></script>
```

Conclusion

ES Modules are one of the biggest features introduced in modern browsers. They are part of ES6 but the road to implement them has been long.

We can now use them! But we must also remember that having more than a few modules is going to have a performance hit on our pages, as it's one more step that the browser must perform at runtime.

Webpack is probably going to still be a huge player even if ES Modules land in the browser, but having such a feature directly built in the language is huge for a unification of how modules work client-side and on Node.js as well.

New String methods

Any string value got some new instance methods:

- `repeat()`
- `codePointAt()`

repeat()

Repeats the strings for the specificed number of times:

```
'Ho'.repeat(3) //'HoHoHo'
```

Returns an empty string if there is no parameter, or the parameter is `0`. If the parameter is negative you'll get a RangeError.

codePointAt()

This method can be used to handle Unicode characters that cannot be represented by a single 16-bit Unicode unit, but need 2 instead.

Using `charCodeAt()` you need to retrieve the first, and the second, and combine them. Using `codePointAt()` you get the whole character in one call.

For example, this chinese character "吉" is composed by 2 UTF-16 (Unicode) parts:

```
"吉".charCodeAt(0).toString(16) //d842
"吉".charCodeAt(1).toString(16) //dfb7
```

If you create a new character by combining those unicode characters:

```
"\ud842\udfb7" //"吉"
```

You can get the same result usign `codePointAt()`:

```
"吉".codePointAt(0) //20bb7
```

If you create a new character by combining those unicode characters:

```
"\u{20bb7}" //"吉"
```

More on Unicode and working with it in my Unicode guide: https://flaviocopes.com/unicode/

New Object methods

ES6 introduced several static methods under the Object namespace:

- `Object.is()` determines if two values are the same value
- `Object.assign()` used to shallow copy an object
- `Object.setPrototypeOf` sets an object prototype

Object.is()

This methods aims to help comparing values.

Usage:

```
Object.is(a, b)
```

The result is always `false` unless:

- `a` and `b` are the same exact object
- `a` and `b` are equal strings (strings are equal when composed by the same characters)
- `a` and `b` are equal numbers (numbers are equal when their value is equal)
- `a` and `b` are both `undefined`, both `null`, both `NaN`, both `true` or both `false`

`0` and `-0` are different values in JavaScript, so pay attention in this special case (convert all to `+0` using the `+` unary operator before comparing, for example).

Object.assign()

Introduced in `ES2015`, this method copies all the **enumerable own properties** of one or more objects into another.

Its primary use case is to create a shallow copy of an object.

```
const copied = Object.assign({}, original)
```

Being a shallow copy, values are cloned, and objects references are copied (not the objects themselves), so if you edit an object property in the original object, that's modified also in the copied object, since the referenced inner object is the same:

```
const original = {
  name: 'Fiesta',
```

```
  car: {
    color: 'blue'
  }
}
const copied = Object.assign({}, original)

original.name = 'Focus'
original.car.color = 'yellow'

copied.name //Fiesta
copied.car.color //yellow
```

I mentioned "one or more":

```
const wisePerson = {
  isWise: true
}
const foolishPerson = {
  isFoolish: true
}
const wiseAndFoolishPerson = Object.assign({}, wisePerson, foolishPerson)

console.log(wiseAndFoolishPerson) //{ isWise: true, isFoolish: true }
```

Object.setPrototypeOf()

Set the prototype of an object. Accepts two arguments: the object and the prototype.

Usage:

```
Object.setPrototypeOf(object, prototype)
```

Example:

```
const animal = {
  isAnimal: true
}
const mammal = {
  isMammal: true
}

mammal.__proto__ = animal
mammal.isAnimal //true

const dog = Object.create(animal)

dog.isAnimal  //true
console.log(dog.isMammal)  //undefined
```

```
Object.setPrototypeOf(dog, mammal)

dog.isAnimal //true
dog.isMammal //true
```

The spread operator

You can expand an array, an object or a string using the spread operator `...`.

Let's start with an array example. Given

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

you can create a new array using

```
const b = [...a, 4, 5, 6]
```

You can also create a copy of an array using

```
const c = [...a]
```

This works for objects as well. Clone an object with:

```
const newObj = { ...oldObj }
```

Using strings, the spread operator creates an array with each char in the string:

```
const hey = 'hey'
const arrayized = [...hey] // ['h', 'e', 'y']
```

This operator has some pretty useful applications. The most important one is the ability to use an array as function argument in a very simple way:

```
const f = (foo, bar) => {}
const a = [1, 2]
f(...a)
```

(in the past you could do this using `f.apply(null, a)` but that's not as nice and readable)

The **rest element** is useful when working with **array destructuring**:

```
const numbers = [1, 2, 3, 4, 5]
[first, second, ...others] = numbers
```

and **spread elements**:

```
const numbers = [1, 2, 3, 4, 5]
const sum = (a, b, c, d, e) => a + b + c + d + e
const sum = sum(...numbers)
```

ES2018 introduces rest properties, which are the same but for objects.

Rest properties:

```
const { first, second, ...others } = {
  first: 1,
  second: 2,
  third: 3,
  fourth: 4,
  fifth: 5
}

first // 1
second // 2
others // { third: 3, fourth: 4, fifth: 5 }
```

Spread properties allow to create a new object by combining the properties of the object passed after the spread operator:

```
const items = { first, second, ...others }
items //{ first: 1, second: 2, third: 3, fourth: 4, fifth: 5 }
```

Set

A Set data structure allows to add data to a container.

A Set is a collection of objects or primitive types (strings, numbers or booleans), and you can think of it as a Map where values are used as map keys, with the map value always being a boolean true.

Initialize a Set

A Set is initialized by calling:

```
const s = new Set()
```

Add items to a Set

You can add items to the Set by using the `add` method:

```
s.add('one')
s.add('two')
```

A set only stores unique elements, so calling `s.add('one')` multiple times won't add new items.

You can't add multiple elements to a set at the same time. You need to call `add()` multiple times.

Check if an item is in the set

Once an element is in the set, we can check if the set contains it:

```
s.has('one') //true
s.has('three') //false
```

Delete an item from a Set by key

Use the `delete()` method:

```
s.delete('one')
```

Determine the number of items in a Set

Use the `size` property:

```
s.size
```

Delete all items from a Set

Use the `clear()` method:

```
s.clear()
```

Iterate the items in a Set

Use the `keys()` or `values()` methods - they are equivalent:

```
for (const k of s.keys()) {
  console.log(k)
}

for (const k of s.values()) {
  console.log(k)
}
```

The `entries()` method returns an iterator, which you can use like this:

```
const i = s.entries()
console.log(i.next())
```

calling `i.next()` will return each element as a `{ value, done = false }` object until the iterator ends, at which point `done` is `true`.

You can also use the forEach() method on the set:

```
s.forEach(v => console.log(v))
```

or you can just use the set in a for..of loop:

```
for (const k of s) {
  console.log(k)
}
```

Initialize a Set with values

You can initialize a Set with a set of values:

```
const s = new Set([1, 2, 3, 4])
```

Convert to array

Convert the Set keys into an array

```
const a = [...s.keys()]

// or

const a = [...s.values()]
```

A WeakSet

A WeakSet is a special kind of Set.

In a Set, items are never garbage collected. A WeakSet instead lets all its items be freely garbage collected. Every key of a WeakSet is an object. When the reference to this object is lost, the value can be garbage collected.

Here are the main differences:

1. you cannot iterate over the WeakSet
2. you cannot clear all items from a WeakSet
3. you cannot check its size

A WeakSet is generally used by framework-level code, and only exposes these methods:

- add()
- has()
- delete()

Map

A Map data structure allows to associate data to a key.

Before ES6

Before its introduction, people generally used objects as maps, by associating some object or value to a specific key value:

```
const car = {}
car['color'] = 'red'
car.owner = 'Flavio'
console.log(car['color']) //red
console.log(car.color) //red
console.log(car.owner) //Flavio
console.log(car['owner']) //Flavio
```

Enter Map

ES6 introduced the Map data structure, providing us a proper tool to handle this kind of data organization.

A Map is initialized by calling:

```
const m = new Map()
```

Add items to a Map

You can add items to the map by using the `set` method:

```
m.set('color', 'red')
m.set('age', 2)
```

Get an item from a map by key

And you can get items out of a map by using `get`:

```
const color = m.get('color')
const age = m.get('age')
```

Delete an item from a map by key

Use the `delete()` method:

```
m.delete('color')
```

Delete all items from a map

Use the `clear()` method:

```
m.clear()
```

Check if a map contains an item by key

Use the `has()` method:

```
const hasColor = m.has('color')
```

Find the number of items in a map

Use the `size` property:

```
const size = m.size
```

Initialize a map with values

You can initialize a map with a set of values:

```
const m = new Map([['color', 'red'], ['owner', 'Flavio'], ['age', 2]])
```

Map keys

Just like any value (object, array, string, number) can be used as the value of the key-value entry of a map item, **any value can be used as the key**, even objects.

If you try to get a non-existing key using `get()` out of a map, it will return `undefined`.

Weird situations you'll almost never find in real life

```
const m = new Map()
m.set(NaN, 'test')
m.get(NaN) //test
```

```
const m = new Map()
m.set(+0, 'test')
m.get(-0) //test
```

Iterating over a map

Iterate over map keys

Map offers the `keys()` method we can use to iterate on all the keys:

```
for (const k of m.keys()) {
  console.log(k)
}
```

Iterate over map values

The Map object offers the `values()` method we can use to iterate on all the values:

```
for (const v of m.values()) {
  console.log(v)
}
```

Iterate over map key, value pairs

The Map object offers the `entries()` method we can use to iterate on all the values:

```
for (const [k, v] of m.entries()) {
  console.log(k, v)
}
```

which can be simplified to

```
for (const [k, v] of m) {
  console.log(k, v)
```

```
}
```

Convert to array

Convert the map keys into an array

```
const a = [...m.keys()]
```

Convert the map values into an array

```
const a = [...m.values()]
```

WeakMap

A WeakMap is a special kind of map.

In a map object, items are never garbage collected. A WeakMap instead lets all its items be freely garbage collected. Every key of a WeakMap is an object. When the reference to this object is lost, the value can be garbage collected.

Here are the main differences:

1. you cannot iterate over the keys or values (or key-values) of a WeakMap
2. you cannot clear all items from a WeakMap
3. you cannot check its size

A WeakMap exposes those methods, which are equivalent to the Map ones:

- `get(k)`
- `set(k, v)`
- `has(k)`
- `delete(k)`

The use cases of a WeakMap are less evident than the ones of a Map, and you might never find the need for them, but essentially it can be used to build a memory-sensitive cache that is not going to interfere with garbage collection, or for careful encapsualtion and information hiding.

Generators

Generators are a special kind of function with the ability to pause itself, and resume later, allowing other code to run in the meantime.

See the full JavaScript Generators Guide for a detailed explanation of the topic.

The code decides that it has to wait, so it lets other code "in the queue" to run, and keeps the right to resume its operations "when the thing it's waiting for" is done.

All this is done with a single, simple keyword: `yield`. When a generator contains that keyword, the execution is halted.

A generator can contain many `yield` keywords, thus halting itself multiple times, and it's identified by the `*function` keyword, which is not to be confused with the pointer dereference operator used in lower level programming languages such as C, C++ or Go.

Generators enable whole new paradigms of programming in JavaScript, allowing:

- 2-way communication while a generator is running
- long-lived while loops which do not freeze your program

Here is an example of a generator which explains how it all works.

```
function *calculator(input) {
    var doubleThat = 2 * (yield (input / 2))
    var another = yield (doubleThat)
    return (input * doubleThat * another)
}
```

We initialize it with

```
const calc = calculator(10)
```

Then we start the iterator on our generator:

```
calc.next()
```

This first iteration starts the iterator. The code returns this object:

```
{
  done: false
  value: 5
}
```

What happens is: the code runs the function, with `input = 10` as it was passed in the generator constructor. It runs until it reaches the `yield`, and returns the content of `yield`: `input / 2 = 5`. So we got a value of 5, and the indication that the iteration is not done (the function is just paused).

In the second iteration we pass the value `7`:

```
calc.next(7)
```

and what we got back is:

```
{
  done: false
  value: 14
}
```

`7` was placed as the value of `doubleThat`. Important: you might read like `input / 2` was the argument, but that's just the return value of the first iteration. We now skip that, and use the new input value, `7`, and multiply it by 2.

We then reach the second yield, and that returns `doubleThat`, so the returned value is `14`.

In the next, and last, iteration, we pass in 100

```
calc.next(100)
```

and in return we got

```
{
  done: true
  value: 14000
}
```

As the iteration is done (no more yield keywords found) and we just return `(input * doubleThat * another)` which amounts to `10 * 14 * 100`.

ES2016

Array.prototype.includes()

This feature introduces a more readable syntax for checking if an array contains an element.

With ES6 and lower, to check if an array contained an element you had to use `indexOf`, which checks the index in the array, and returns `-1` if the element is not there.

Since `-1` is evaluated as a true value, you could **not** do for example

```
if (![1,2].indexOf(3)) {
  console.log('Not found')
}
```

With this feature introduced in ES7 we can do

```
if (![1,2].includes(3)) {
  console.log('Not found')
}
```

Exponentiation Operator

The exponentiation operator `**` is the equivalent of `Math.pow()`, but brought into the language instead of being a library function.

```
Math.pow(4, 2) == 4 ** 2
```

This feature is a nice addition for math intensive JS applications.

The `**` operator is standardized across many languages including Python, Ruby, MATLAB, Lua, Perl and many others.

ES2017

String padding

The purpose of string padding is to **add characters to a string**, so it **reaches a specific length**.

ES2017 introduces two `String` methods: `padStart()` and `padEnd()`.

```
padStart(targetLength [, padString])
padEnd(targetLength [, padString])
```

Sample usage:

padStart()	
'test'.padStart(4)	`'test'`
'test'.padStart(5)	`' test'`
'test'.padStart(8)	`' test'`
'test'.padStart(8, 'abcd')	`'abcdtest'`

padEnd()	
'test'.padEnd(4)	`'test'`
'test'.padEnd(5)	`'test '`
'test'.padEnd(8)	`'test '`
'test'.padEnd(8, 'abcd')	`'testabcd'`

Object.values()

This method returns an array containing all the object own property values.

Usage:

```
const person = { name: 'Fred', age: 87 }
Object.values(person) // ['Fred', 87]
```

`Object.values()` also works with arrays:

```
const people = ['Fred', 'Tony']
Object.values(people) // ['Fred', 'Tony']
```

Object.entries()

This method returns an array containing all the object own properties, as an array of `[key, value]` pairs.

Usage:

```
const person = { name: 'Fred', age: 87 }
Object.entries(person) // [['name', 'Fred'], ['age', 87]]
```

`Object.entries()` also works with arrays:

```
const people = ['Fred', 'Tony']
Object.entries(people) // [['0', 'Fred'], ['1', 'Tony']]
```

Object.getOwnPropertyDescriptors()

This method returns all own (non-inherited) properties descriptors of an object.

Any object in JavaScript has a set of properties, and each of these properties has a descriptor.

A descriptor is a set of attributes of a property, and it's composed by a subset of the following:

- **value**: the value of the property
- **writable**: true the property can be changed
- **get**: a getter function for the property, called when the property is read
- **set**: a setter function for the property, called when the property is set to a value
- **configurable**: if false, the property cannot be removed nor any attribute can be changed, except its value
- **enumerable**: true if the property is enumerable

`Object.getOwnPropertyDescriptors(obj)` accepts an object, and returns an object with the set of descriptors.

In what way is this useful?

ES6 gave us `Object.assign()`, which copies all enumerable own properties from one or more objects, and return a new object.

However there is a problem with that, because it does not correctly copies properties with non-default attributes.

If an object for example has just a setter, it's not correctly copied to a new object, using `Object.assign()`.

For example with

```
const person1 = {
    set name(newName) {
        console.log(newName)
    }
}
```

This won't work:

```
const person2 = {}
Object.assign(person2, person1)
```

But this will work:

```
const person3 = {}
Object.defineProperties(person3,
  Object.getOwnPropertyDescriptors(person1))
```

As you can see with a simple console test:

```
person1.name = 'x'
"x"

person2.name = 'x'

person3.name = 'x'
"x"
```

`person2` misses the setter, it was not copied over.

The same limitation goes for shallow cloning objects with **Object.create()**.

Trailing commas

This feature allows to have trailing commas in function declarations, and in functions calls:

```
const doSomething = (var1, var2,) => {
  //...
}

doSomething('test2', 'test2',)
```

This change will encourage developers to stop the ugly "comma at the start of the line" habit.

Async functions

JavaScript evolved in a very short time from callbacks to promises (ES2015), and since ES2017 asynchronous JavaScript is even simpler with the async/await syntax.

Async functions are a combination of promises and generators, and basically, they are a higher level abstraction over promises. Let me repeat: **async/await is built on promises**.

Why were async/await introduced?

They reduce the boilerplate around promises, and the "don't break the chain" limitation of chaining promises.

When Promises were introduced in ES2015, they were meant to solve a problem with asynchronous code, and they did, but over the 2 years that separated ES2015 and ES2017, it was clear that *promises could not be the final solution*.

Promises were introduced to solve the famous *callback hell* problem, but they introduced complexity on their own, and syntax complexity.

They were good primitives around which a better syntax could be exposed to developers, so when the time was right we got **async functions**.

They make the code look like it's synchronous, but it's asynchronous and non-blocking behind the scenes.

How it works

An async function returns a promise, like in this example:

```
const doSomethingAsync = () => {
  return new Promise(resolve => {
    setTimeout(() => resolve('I did something'), 3000)
  })
}
```

When you want to **call** this function you prepend `await`, and **the calling code will stop until the promise is resolved or rejected**. One caveat: the client function must be defined as `async`. Here's an example:

```
const doSomething = async () => {
  console.log(await doSomethingAsync())
```

```
}
```

A quick example

This is a simple example of async/await used to run a function asynchronously:

```
const doSomethingAsync = () => {
  return new Promise(resolve => {
    setTimeout(() => resolve('I did something'), 3000)
  })
}

const doSomething = async () => {
  console.log(await doSomethingAsync())
}

console.log('Before')
doSomething()
console.log('After')
```

The above code will print the following to the browser console:

```
Before
After
I did something //after 3s
```

Promise all the things

Prepending the `async` keyword to any function means that the function will return a promise.

Even if it's not doing so explicitly, it will internally make it return a promise.

This is why this code is valid:

```
const aFunction = async () => {
  return 'test'
}

aFunction().then(alert) // This will alert 'test'
```

and it's the same as:

```
const aFunction = async () => {
  return Promise.resolve('test')
}
```

```
aFunction().then(alert) // This will alert 'test'
```

The code is much simpler to read

As you can see in the example above, our code looks very simple. Compare it to code using plain promises, with chaining and callback functions.

And this is a very simple example, the major benefits will arise when the code is much more complex.

For example here's how you would get a JSON resource, and parse it, using promises:

```
const getFirstUserData = () => {
  return fetch('/users.json') // get users list
    .then(response => response.json()) // parse JSON
    .then(users => users[0]) // pick first user
    .then(user => fetch(`/users/${user.name}`)) // get user data
    .then(userResponse => response.json()) // parse JSON
}

getFirstUserData()
```

And here is the same functionality provided using await/async:

```
const getFirstUserData = async () => {
  const response = await fetch('/users.json') // get users list
  const users = await response.json() // parse JSON
  const user = users[0] // pick first user
  const userResponse = await fetch(`/users/${user.name}`) // get user data
  const userData = await user.json() // parse JSON
  return userData
}

getFirstUserData()
```

Multiple async functions in series

Async functions can be chained very easily, and the syntax is much more readable than with plain promises:

```
const promiseToDoSomething = () => {
  return new Promise(resolve => {
    setTimeout(() => resolve('I did something'), 10000)
  })
}
```

```
const watchOverSomeoneDoingSomething = async () => {
  const something = await promiseToDoSomething()
  return something + ' and I watched'
}

const watchOverSomeoneWatchingSomeoneDoingSomething = async () => {
  const something = await watchOverSomeoneDoingSomething()
  return something + ' and I watched as well'
}

watchOverSomeoneWatchingSomeoneDoingSomething().then(res => {
  console.log(res)
})
```

Will print:

```
I did something and I watched and I watched as well
```

Easier debugging

Debugging promises is hard because the debugger will not step over asynchronous code.

Async/await makes this very easy because to the compiler it's just like synchronous code.

Shared Memory and Atomics

WebWorkers are used to create multithreaded programs in the browser.

They offer a messaging protocol via events. Since ES2017, you can create a shared memory array between web workers and their creator, using a `SharedArrayBuffer`.

Since it's unknown how much time writing to a shared memory portion takes to propagate, **Atomics** are a way to enforce that when reading a value, any kind of writing operation is completed.

Any more detail on this can be found in the spec proposal, which has since been implemented.

ES2018

Rest/Spread Properties

ES2015 introduced the concept of a **rest element** when working with **array destructuring**:

```
const numbers = [1, 2, 3, 4, 5]
[first, second, ...others] = numbers
```

and **spread elements**:

```
const numbers = [1, 2, 3, 4, 5]
const sum = (a, b, c, d, e) => a + b + c + d + e
const sum = sum(...numbers)
```

ES2018 introduces the same but for objects.

Rest properties:

```
const { first, second, ...others } = { first: 1, second: 2, third: 3, fourth: 4, fifth: 5
}

first // 1
second // 2
others // { third: 3, fourth: 4, fifth: 5 }
```

Spread properties allow to create a new object by combining the properties of the object passed after the spread operator:

```
const items = { first, second, ...others }
items //{ first: 1, second: 2, third: 3, fourth: 4, fifth: 5 }
```

Asynchronous iteration

The new construct `for-await-of` allows you to use an async iterable object as the loop iteration:

```
for await (const line of readLines(filePath)) {
  console.log(line)
}
```

Since this uses `await`, you can use it only inside `async` functions, like a normal `await`.

Promise.prototype.finally()

When a promise is fulfilled, successfully it calls the `then()` methods, one after another.

If something fails during this, the `then()` methods are jumped and the `catch()` method is executed.

`finally()` allow you to run some code regardless of the successful or not successful execution of the promise:

```
fetch('file.json')
  .then(data => data.json())
  .catch(error => console.error(error))
  .finally(() => console.log('finished'))
```

Regular Expression improvements

ES2018 introduced a number of improvements regarding Regular Expressions. I recommend my tutorial on them, available at https://flaviocopes.com/javascript-regular-expressions/.

Here are the ES2018 specific additions.

RegExp lookbehind assertions: match a string depending on what precedes it

This is a lookahead: you use `?=` to match a string that's followed by a specific substring:

```
/Roger(?=Waters)/

/Roger(?= Waters)/.test('Roger is my dog') //false
/Roger(?= Waters)/.test('Roger is my dog and Roger Waters is a famous musician') //true
```

`?!` performs the inverse operation, matching if a string is **not** followed by a specific substring:

```
/Roger(?!Waters)/

/Roger(?! Waters)/.test('Roger is my dog') //true
/Roger(?! Waters)/.test('Roger Waters is a famous musician') //false
```

Lookaheads use the `?=` symbol. They were already available.

Lookbehinds, a new feature, uses `?<=` .

```
/(?<=Roger) Waters/

/(?<=Roger) Waters/.test('Pink Waters is my dog') //false
/(?<=Roger) Waters/.test('Roger is my dog and Roger Waters is a famous musician') //true
```

A lookbehind is negated using `?<!` :

```
/(?<!Roger) Waters/

/(?<!Roger) Waters/.test('Pink Waters is my dog') //true
/(?<!Roger) Waters/.test('Roger is my dog and Roger Waters is a famous musician') //false
```

Unicode property escapes `\p{…}` and `\P{…}`

In a regular expression pattern you can use `\d` to match any digit, `\s` to match any character that's not a white space, `\w` to match any alphanumeric character, and so on.

This new feature extends this concept to all Unicode characters introducing `\p{}` and is negation `\P{}`.

Any unicode character has a set of properties. For example `Script` determines the language family, `ASCII` is a boolean that's true for ASCII characters, and so on. You can put this property in the graph parentheses, and the regex will check for that to be true:

```
/^\p{ASCII}+$/u.test('abc')   //
/^\p{ASCII}+$/u.test('ABC@')  //
/^\p{ASCII}+$/u.test('ABC  ') //
```

`ASCII_Hex_Digit` is another boolean property, that checks if the string only contains valid hexadecimal digits:

```
/^\p{ASCII_Hex_Digit}+$/u.test('0123456789ABCDEF') //
/^\p{ASCII_Hex_Digit}+$/u.test('h')                //
```

There are many other boolean properties, which you just check by adding their name in the graph parentheses, including `Uppercase`, `Lowercase`, `White_Space`, `Alphabetic`, `Emoji` and more:

```
/^\p{Lowercase}$/u.test('h') //
/^\p{Uppercase}$/u.test('H') //

/^\p{Emoji}+$/u.test('H')   //
/^\p{Emoji}+$/u.test('    ') //
```

In addition to those binary properties, you can check any of the unicode character properties to match a specific value. In this example, I check if the string is written in the greek or latin alphabet:

```
/^\p{Script=Greek}+$/u.test('ελληνικά') //
/^\p{Script=Latin}+$/u.test('hey') //
```

Read more about all the properties you can use directly on the proposal.

Named capturing groups

In ES2018 a capturing group can be assigned to a name, rather than just being assigned a slot in the result array:

```
const re = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/
const result = re.exec('2015-01-02')

// result.groups.year === '2015';
// result.groups.month === '01';
// result.groups.day === '02';
```

The `s` flag for regular expressions

The `s` flag, short for *single line*, causes the `.` to match new line characters as well. Without it, the dot matches regular characters but not the new line:

```
/hi.welcome/.test('hi\nwelcome') // false
/hi.welcome/s.test('hi\nwelcome') // true
```

ESNext

What's next? ESNext.

ESNext is a name that always indicates the next version of JavaScript.

The current ECMAScript version is **ES2018**. It was released in June 2018.

Historically JavaScript editions have been standardized during the summer, so we can expect **ECMAScript 2019** to be released in summer 2019.

So at the time of writing, ES2018 has been released, and **ESNext is ES2019**

Proposals to the ECMAScript standard are organized in stages. Stages 1-3 are an incubator of new features, and features reaching Stage 4 are finalized as part of the new standard.

At the time of writing we have a number of features at **Stage 4**. I will introduce them in this section. The latest versions of the major browsers should already implement most of those.

Some of those changes are mostly for internal use, but it's also good to know what is going on.

There are other features at Stage 3, which might be promoted to Stage 4 in the next few months, and you can check them out on this GitHub repository: https://github.com/tc39/proposals.

Array.prototype.{flat,flatMap}

`flat()` is a new array instance method that can create a one-dimensional array from a multidimensional array.

Example:

```
['Dog', ['Sheep', 'Wolf']].flat()
//[ 'Dog', 'Sheep', 'Wolf' ]
```

By default it only "flats" up to one level, but you can add a parameter to set the number of levels you want to flat the array to. Set it to `Infinity` to have unlimited levels:

```
['Dog', ['Sheep', ['Wolf']]].flat()
//[ 'Dog', 'Sheep', [ 'Wolf' ] ]

['Dog', ['Sheep', ['Wolf']]].flat(2)
//[ 'Dog', 'Sheep', 'Wolf' ]

['Dog', ['Sheep', ['Wolf']]].flat(Infinity)
//[ 'Dog', 'Sheep', 'Wolf' ]
```

If you are familiar with the JavaScript `map()` method of an array, you know that using it you can execute a function on every element of an array.

`flatMap()` is a new Array instance method that combines `flat()` with `map()`. It's useful when calling a function that returns an array in the map() callback, but you want your resulted array to be flat:

```
['My dog', 'is awesome'].map(words => words.split(' '))
//[ [ 'My', 'dog' ], [ 'is', 'awesome' ] ]

['My dog', 'is awesome'].flatMap(words => words.split(' '))
//[ 'My', 'dog', 'is', 'awesome' ]
```

Optional catch binding

Sometimes we dont need to have a parameter binded to the catch block of a try/catch.

We previously had to do:

```
try {
  //...
} catch (e) {
  //handle error
}
```

Even if we never had to use `e` to analyze the error. We can now simply omit it:

```
try {
  //...
} catch {
  //handle error
}
```

Object.fromEntries()

Objects have an `entries()` method, since ES2017.

It returns an array containing all the object own properties, as an array of `[key, value]` pairs:

```
const person = { name: 'Fred', age: 87 }
Object.entries(person) // [['name', 'Fred'], ['age', 87]]
```

ES2019 introduces a new `Object.fromEntries()` method, which can create a new object from such array of properties:

```
const person = { name: 'Fred', age: 87 }
const entries = Object.entries(person)
const newPerson = Object.fromEntries(entries)

person !== newPerson //true
```

String.prototype.{trimStart,trimEnd}

This feature has been part of v8/Chrome for almost a year now, and it's going to be standardized in ES2019.

`trimStart()`

Return a new string with removed white space from the start of the original string

```
'Testing'.trimStart() //'Testing'
' Testing'.trimStart() //'Testing'
' Testing '.trimStart() //'Testing '
'Testing'.trimStart() //'Testing'
```

`trimEnd()`

Return a new string with removed white space from the end of the original string

```
'Testing'.trimEnd() //'Testing'
' Testing'.trimEnd() //' Testing'
' Testing '.trimEnd() //' Testing'
'Testing '.trimEnd() //'Testing'
```

Symbol.prototype.description

You can now retrieve the description of a symbol by accessing its `description` property instead of having to use the `toString()` method:

```
const testSymbol = Symbol('Test')
testSymbol.description // 'Test'
```

JSON improvements

Before this change, the line separator (\u2028) and paragraph separator (\u2029) symbols were not allowed in strings parsed as JSON.

Using JSON.parse(), those characters resulted in a `SyntaxError` but now they parse correctly, as defined by the JSON standard.

Well-formed JSON.stringify()

Fixes the `JSON.stringify()` output when it processes surrogate UTF-8 code points (U+D800 to U+DFFF).

Before this change calling `JSON.stringify()` would return a malformed Unicode character (a "  ").

Now those surrogate code points can be safely represented as strings using `JSON.stringify()`, and transformed back into their original representation using `JSON.parse()`.

Function.prototype.toString()

Functions have always had an instance method called `toString()` which return a string containing the function code.

ES2019 introduced a change to the return value to avoid stripping comments and other characters like whitespace, exactly representing the function as it was defined.

If previously we had

```
function /* this is bar */ bar () {}
```

The behavior was this:

```
bar.toString() //'function bar() {}
```

now the new behavior is:

```
bar.toString(); // 'function /* this is bar */ bar () {}'
```