ECMAScript 6 (ES6): What's New In The Next Version Of JavaScript

You've probably heard about **ECMAScript 6** (or ES6) already. It's the next version of JavaScript, and it has some great new features. The features have varying degrees of complexity and are useful in both simple scripts and complex applications. In this article, we'll discuss a **hand-picked selection of ES6 features** that you can use in your everyday JavaScript coding.

Please note that support for these new ES6 features is well underway in modern browsers, although support varies. If you need to support old versions of browsers that lack many ES6 features, I'll touch on solutions that might help you start using ES6 today.

Most of the code samples come with an external "Run this code" link, so that you can see the code and play with it.

Variables

let

You're used to declaring variables using the `var` keyword. You can now use `let` as well. The subtle difference lies in scope. While `var` results in a variable with the surrounding function as its scope, the scope of a variable declared using `let` is only the block it is in.

```
if (true) {
  let x = 1;
}
console.log(x);  // undefined
```

This can make for cleaner code, resulting in fewer variables hanging around. Take this classic array iteration:

```
for (let i = 0, l = list.length; i < l; i++) {
  // do something with list[i]
}

console.log(i);  // undefined
```

Often one would use, for example, the `j` variable for another iteration in the same scope. But with `let`, you could safely declare `i` again, since it's defined and available only within its own block scope.

const

There is another way to declare variables. With `const`, you can declare immutable variables. This means you can't change its value once it is set. Also, you must assign a variable directly. If you try to change the variable or if you don't a set a value immediately, then you'll get an error:

```
const MY_CONSTANT = 1;
MY_CONSTANT = 2  // Error
const SOME_CONST;  // Error
```

Arrow Functions

Arrow functions are a great addition to the JavaScript language. They make for short and concise code. We are introducing arrow functions early in this article so that we can take advantage of them in other examples later on. The next code snippet shows an arrow function, with the same function written in the familiar ES5 style:

```
let books = [{title: 'X', price: 10}, {title: 'Y', price: 15...

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

// ES5 equivalent:
var titles = books.map(function(item) {
  return item.title;
...
```

If we look at the syntax of arrow functions, there is no `function` keyword. What remains is zero or more arguments, the "fat arrow" (`=>`) and the function expression. The `return` statement is implicitly added.

With zero or more than one argument, you must provide parentheses:

```
// No arguments
books.map( () => 1 ); // [1, 1]

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

Put the function expression in a block (`{ ... }`) if you need more logic or more white space:

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

Not only do arrow functions mean fewer characters to type, but they also behave differently from regular functions. An arrow function expression inherits `this` and `arguments` from the surrounding context. This means you can get rid of ugly statements like `var that = this`, and you won't need to bind functions to the correct context. Here's an example (note `this.title` versus `that.title` in the ES5 version):

```
let book = {
  title: 'X',
  sellers: ['A', 'B'],
  printSellers() {
    this.sellers.forEach(seller => console.log(seller + ' sells ' + this.title...
  }
}

// ES5 equivalent:
var book = {
  title: 'X',
  sellers: ['A', 'B'],
  printSellers: function() {
    var that = this;
    this.sellers.forEach(function(seller) {
      console.log(seller + ' sells ' + that.title)
    })
  }
}
```

Strings

Methods

A couple of convenience methods have been added to the `String` prototype. Most of them basically eliminate some workarounds with the `indexOf()` method to achieve the same:

```
'my string'.startsWith('my');  //true
'my string'.endsWith('my');  // false
'my string'.includes('str');  // true
```

Simple but effective. Another convenience method has been added to create a repeating string:

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

Template Literal

Template literals provide a clean way to create strings and perform string interpolation. You might already be familiar with the syntax; it's based on the dollar sign and curly braces `${..}`. Here's a quick demonstration:

```
let name = 'John',
  apples = 5,
  pears = 7,
  bananas = function() { return 3; }

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

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

// ES5 equivalent:
console.log('He carries ' + apples + ' apples, ' + pears + ' pears, and ' + bananas() +' bananas.');
```

In the form above, compared to ES5, they're merely a convenience for string concatenation. However, template literals can also be used for multi-line strings. Keep in mind that white space is part of the string:

```
let x = `1...
2...
3 lines long!`;  // Yay

// ES5 equivalents:
var x = "1...\n" +
"2...\n" +
"3 lines long!";

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

Arrays

The `Array` object now has some new static class methods, as well as new methods on the `Array` prototype.

First, `Array.from` creates `Array` instances from array-like and iterable objects. Examples of array-like objects include:

- the `arguments` within a function;
- a `nodeList` returned by `document.getElementsByTagName()` ;
- the new `Map` and `Set` data structures.

```
let itemElements = document.querySelectorAll('.items');
let items = Array.from(itemElements);
items.forEach(function(element) {
   console.log(element.nodeType)
...

// A workaround often used in ES5:
let items = Array.prototype.slice.call(itemElements);
```

In the example above, you can see that the `items` array has the `forEach` method, which isn't available in the `itemElements` collection.

An interesting feature of `Array.from` is the second optional `mapFunction` argument. This allows you to create a new mapped array in a single invocation:

```
let navElements = document.querySelectorAll('nav li');
let navTitles = Array.from(navElements, el => el.textContent);
```

Then, we have `Array.of`, which behaves much like the `Array` constructor. It fixes the special case when passing it a single number argument. This results in `Array.of` being preferable to `new Array()`. However, in most cases, you'll want to use array literals.

```
let x = new Array(3);  // [undefined, undefined, undefined]
let y = Array.of(8);  // [8]
let z = [1, 2, 3];  // Array literal
```

Last but not least, a couple of methods have been added to the `Array` prototype. I think the `find` methods will be very welcome to most JavaScript developers.

- `find` returns the first element for which the callback returns `true`.
- `findIndex` returns the index of the first element for which the callback returns `true`.
- `fill` "overwrites" the elements of an array with the given argument.

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

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

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

Math

A couple of new methods have been added to the `Math` object.

- `Math.sign` returns the sign of a number as `1`, `-1` or `0`.
- `Math.trunc` returns the passed number without fractional digits.
- `Math.cbrt` returns the cube root of a number.

```
Math.sign(5);  // 1
Math.sign(-9);  // -1

Math.trunc(5.9);  // 5
Math.trunc(5.123);  // 5

Math.cbrt(64);  // 4
```

If you want to learn more about the new number and math features in ES6[8], Dr. Axel Rauschmayer has you covered.

Spread Operator

The spread operator (`...`) is a very convenient syntax to expand elements of an array in specific places, such as arguments in function calls. Showing you some examples is probably the best way to demonstrate just how useful they are.

First, let's see how to expand elements of an array within another array:

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

// ES5 equivalent:
let values = [1, 2, 4];
// Iterate, push, sweat, repeat...
// Iterate, push, sweat, repeat...
```

The spread syntax is also powerful when calling functions with arguments:

```
let values = [1, 2, 4];

doSomething(...values);

function doSomething(x, y, z) {
  // x = 1, y = 2, z = 4
}

// ES5 equivalent:
doSomething.apply(null, values);
```

As you can see, this saves us from the often-used `fn.apply()` workaround. The syntax is very flexible, because the spread operator can be used anywhere in the argument list. This means that the following invocation produces the same result:

```
let values = [2, 4];
doSomething(1, ...values);
```

We've been applying the spread operator to arrays and arguments. In fact, it can be applied to all iterable objects, such as a `NodeList`:

```
let form = document.querySelector('#my-form'),
  inputs = form.querySelectorAll('input'),
  selects = form.querySelectorAll('select');

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

Now, `allTheThings` is a flat array containing the `<form>` node and its `<input>` and `<select>` child nodes.

Destructuring

Destructuring provides a convenient way to extract data from objects or arrays. For starters, a good example can be given using an array:

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

// ES5 equivalent:
var arr = [1, 2];
var x = arr[0];
var y = arr[1];
```

With this syntax, multiple variables can be assigned a value in one go. A nice side effect is that you can easily swap variable values:

```
let x = 1,
  y = 2;

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

Destructuring also works with objects. Make sure to have matching keys:

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

You could also use this mechanism to change variable names:

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

Another interesting pattern is to simulate multiple return values:

```
function doSomething() {
  return [1, 2]
}

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

Destructuring can be used to assign default values to argument objects. With an object literal, you can actually simulate named parameters.

```
function doSomething({y = 1, z = 0}) {
  console.log(y, z);
}
doSomething({y: 2...
```

Parameters

Default Values

In ES6, defining default values for function parameters is possible. The syntax is as follows:

```
function doSomething(x, y = 2) {
  return x * y;
}

doSomething(5);  // 10
doSomething(5, undefined);  // 10
doSomething(5, 3);  // 15
```

Looks pretty clean, right? I'm sure you've needed to fill up some arguments in ES5 before:

```
function doSomething(x, y) {
```

```
  y = y === undefined ? 2 : y;
  return x * y;
}
```

Either `undefined` or no argument triggers the default value for that argument.

Rest Parameters

We've been looking into the spread operator. Rest parameters are very similar. It also uses the `...` syntax and allows you to store trailing arguments in an array:

```
function doSomething(x, ...remaining) {
  return x * remaining.length;
}

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

Modules

Modules are certainly a welcome addition to the JavaScript language. I think this major feature alone is worth digging into ES6.

Any serious JavaScript project today uses some sort of module system – maybe something like the "revealing module pattern" or the more extensive formats AMD or CommonJS. However, browsers don't feature any kind of module system. You always need a build step or a loader for your AMD or CommonJS modules. Tools to handle this include RequireJS, Browserify and Webpack.

The ES6 specification includes both a new syntax and a loader mechanism for modules. If you want to use modules and write for the future, this is the syntax you should use. Modern build tools support this format, perhaps via a plugin, so you should be good to go. (No worries – we'll discuss this further in the "Transpilation" section later on.)

Now, on to the ES6 module syntax. Modules are designed around the `export` and `import` keywords. Let's examine an example with two modules right away:

```
// lib/math.js

export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;
```

```
// app.js

import { sum, pi } from "lib/math";
console.log('2π = ' + sum(pi, pi...
```

As you can see, there can be multiple `export` statements. Each must explicitly state the type of the exported value (`function` and `var` , in this example).

The `import` statement in this example uses a syntax (similar to destructuring) to explicitly define what is being imported. To import the module

as a whole, the `*` wildcard can be used, combined with the `as` keyword to give the module a local name:

```
// app.js

import * as math from "lib/math";
console.log('2π = ' + math.sum(math.pi, math.var...
```

The module system features a `default` export. This can also be a function. To import this default value in a module, you'll just need to provide the local name (i.e., no destructuring):

```
// lib/my-fn.js

export default function() {
  console.log('echo echo');
}

// app.js

import doSomething from 'lib/my-fn';
doSomething();
```

Please note that the `import` statements are synchronous, but the module code doesn't execute until all dependencies have loaded.

Classes

Classes are a well-debated feature of ES6. Some believe that they go against the prototypal nature of JavaScript, while others think they lower the barrier to entry for beginners and people coming from other languages and that they help people writing large-scale applications. In any case, they are part of ES6. Here's a very quick introduction.

Classes are built around the `class` and `constructor` keywords. Here's a short example:

```
class Vehicle {
  constructor(name) {
    this.name = name;
    this.kind = 'vehicle';
  }
  getName() {
    return this.name;
  }
}

// Create an instance
let myVehicle = new Vehicle('rocky');
```

Note that the class definition is not a regular object; hence, there are no commas between class members.

To create an instance of a class, you must use the `new` keyword. To inherit from a base class, use `extends`:

```
class Car extends Vehicle {
```

```
  constructor(name) {
    super(name);
    this.kind = 'car'
  }
}

let myCar = new Car('bumpy');

myCar.getName();  // 'bumpy'
myCar instanceof Car;  // true
myCar instanceof Vehicle;  //true
```

From the derived class, you can use `super` from any constructor or method to access its base class:

- To call the parent constructor, use `super()`.
- To call another member, use, for example, `super.getName()`.

There's more to using classes. If you want to dig deeper into the subject, I recommend "Classes in ECMAScript 6[20]" by Dr. Axel Rauschmayer.

Symbols

Symbols are a new primitive data type, like `Number` and `String`. You can use symbols to create unique identifiers for object properties or to create unique constants.

```
const MY_CONSTANT = Symbol();

let obj = ...
obj[MY_CONSTANT] = 1;
```

Note that key-value pairs set with symbols are not returned by `Object.getOwnPropertyNames()`, and they are not visible in `for...in` iterations, `Object.keys()` or `JSON.stringify()`. This is in contrast to regular string-based keys. You can get an array of symbols for an object with `Object.getOwnPropertySymbols()`.

Symbols work naturally with `const` because of their immutable character:

```
const CHINESE = Symbol();
const ENGLISH = Symbol();
const SPANISH = Symbol();

switch(language) {
  case CHINESE:
    //
    break;
  case ENGLISH:
    //
    break;
  case SPANISH:
    //
    break;
  default:
```

```
        //
        break;
    }
```

You can give a symbol a description. You can't use it to access the symbol itself, but it's useful for debugging.

```
const CONST_1 = Symbol('my symbol');
const CONST_2 = Symbol('my symbol');

typeof CONST_1 === 'symbol';  // true

CONST_1 === CONST_2;  // false
```

Want to learn more about symbols? Mozilla Developer Network has a good page about the new symbol primitive[21].

Transpilation

We can write our code in ES6 today. As mentioned in the introduction, browser support for ES6 features is not extensive yet and varies a lot. It's very likely that not all of the ES6 code you write will be understood by the browsers of your users. This is why we need to convert it to the previous version of JavaScript (ES5), which runs fine in any modern browser. This conversion is often referred to as "transpilation." We'll need to do this with our applications until the browsers we want to support understand ES6.

Getting Started

Transpiling code is not hard. You can transpile code directly from the command line, or you can include it as a plugin for a task runner, such as Grunt or Gulp. Plenty of transpilation solutions are out there, including Babel, Traceur and TypeScript. See, for instance, the many ways to start using ES6[22] with Babel (formerly "6to5"). Most features of ES6 are at your disposal!

Now that you're hopefully enthusiastic about using ES6, why not start using it? Depending on the features you want to use and the browsers or environments you need to support (such as Node.js), you'll probably want to incorporate a transpiler in your workflow. And if you're up for it, there are also file watchers and live browser reloaders to make your coding experience seamless.

If you're starting from scratch, you might just want to transpile your code from the command line (see, for example, the Babel CLI documentation[23]). If you are already using a task runner, such as Grunt or Gulp, you can add a plugin such as gulp-babel[24] or the babel-loader[25] for Webpack. For Grunt, there is grunt-babel[26] and many other ES6-related plugins[27]. Folks using Browserify might want to check out babelify[28].

Many features can be converted to ES5-compatible code without significant overhead. Others do require extra stopgaps (which can be provided by the transpiler) and/or come with a performance penalty. Some are simply impossible. To play around with ES6 code and see what the transpiled code looks like, you can use various interactive environments (also known as REPLs):

- Traceur: website[29], REPL[30]
- Babel: website[31], REPL[32]
- TypeScript: website[33], REPL[34]
- ScratchJS[35] (Chrome extension)

Note that TypeScript is not exactly a transpiler. It's a typed superset of JavaScript that compiles to JavaScript. Among other features, it supports many ES6 features, much like the other transpilers.

So, What Exactly Can I Use?

In general, some of the features of ES6 can be used almost for "free," such as modules, arrow functions, rest parameters and classes. These features can be transpiled to ES5 without much overhead. Additions to `Array`, `String` and `Math` objects and prototypes (such as `Array.from()` and `"it".startsWith("you")`) require so-called "polyfills." Polyfills are stopgaps for functionality that a browser doesn't natively support yet. You can load a polyfill first, and your code will run as if the browser has that functionality. Both Babel and Traceur do provide such polyfills.

See Kangax's ES6 compatibility table[36] for a full overview of ES6 features that are supported by both transpilers and browsers. It is motivating to see that, at the time of writing, the latest browsers already support 55% to over 70% of all ES6 features. Microsoft Edge, Google Chrome and Mozilla's Firefox are really competing with each other here, which is great for the web at large.

Personally, I'm finding that being able to easily use new ES6 features such as modules, arrow functions and rest parameters is a relief and a significant improvement to my own coding. Now that I'm comfortable with writing in ES6 and transpiling my code to ES5, more ES6 goodness will follow naturally over time.

What's Next?

Once you've installed a transpiler, you might want to start using "small" features, such as `let` and arrow functions. Keep in mind that code that is already written as ES5 will be left untouched by the transpiler. As you enhance your scripts with ES6 and enjoy using it, you can gradually sprinkle more and more ES6 features onto your code. Perhaps convert some of the code to the new modules or class syntax. I promise it'll be good!

There is much more to ES6 than we were able to cover in this article. Uncovered features include `Map`, `Set`, tagged template strings, generators, `Proxy` and `Promise`. Let me know if you want these features to be covered in a follow-up article. In any case, a book that covers all of ES6 is *Exploring ES6*[37] by Dr. Axel Rauschmayer, which I can happily recommend for a deep dive.

Closing Thought

By using a transpiler, all of your code is effectively "locked" to ES5, while browsers keep adding new features. So, even if a browser fully supports a particular ES6 feature, the ES5-compatible version will be used, possibly performing worse[38]. You can count on the fact that any ES6 feature, and eventually all of them, will be supported at some point (in the browsers and environments you need to support at that time). Until then, we need to manage this and selectively disable ES6 features from getting transpiled to ES5 and prevent unnecessary overhead. With this in mind, decide for yourself whether it is time to start using (parts of) ES6. Some companies think it is[39].

Excerpt Image: Ruiwen Chua[40]

(al, ml)

Footnotes

Click the note number to return to the link spot in the text.

1 Run the example code using babeljs.io/repl/

2 Run the example code using babeljs.io/repl/

3 Run the example code using babeljs.io/repl/

4 Run the example code using babeljs.io/repl/

5 Run the example code using babeljs.io/repl/

6 Run the example code using babeljs.io/repl/

7 Run the example code using babeljs.io/repl/

8 http://www.2ality.com/2015/04/numbers-math-es6.html

9 Run the example code using babeljs.io/repl/

10 Run the example code using babeljs.io/repl/

11 Run the example code using babeljs.io/repl/

12 http://jsbin.com/vibaxerino/edit?html,js,console

13 Run the example code using babeljs.io/repl/

14 Run the example code using babeljs.io/repl/

15 Run the example code using babeljs.io/repl/

16 Run the example code using babeljs.io/repl/

17 Run the example code using babeljs.io/repl/

18 Run the example code using babeljs.io/repl/

19 Run the example code using babeljs.io/repl/

20 http://www.2ality.com/2015/02/es6-classes-final.html

21 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol

22 http://babeljs.io/docs/setup/

23 https://babeljs.io/docs/usage/cli/

24 https://www.npmjs.com/package/gulp-babel

25 https://github.com/babel/babel-loader

26 https://github.com/babel/grunt-babel

27 http://gruntjs.com/plugins?q=es6

28 https://github.com/babel/babelify

29 https://github.com/google/traceur-compiler

30 https://google.github.io/traceur-compiler/demo/repl.html

31 https://babeljs.io/

32 https://babeljs.io/repl/

33 http://www.typescriptlang.org/

34 http://www.typescriptlang.org/Playground

35 https://github.com/richgilbank/Scratch-JS

36 https://kangax.github.io/compat-table/es6/

37 http://exploringjs.com/

38 http://kpdecker.github.io/six-speed/

39 http://babeljs.io/users/

40 https://www.flickr.com/photos/ruiwen/3260095534/in/

JavaScriptWeb Development

↑ Back to top

Lars Kappert

Lars Kappert is a Dutch front-end solution architect & lead developer. He specializes in architecture, solutions, tooling, performance, and development of web sites and applications. Core web technologies include HTML5, JavaScript, Node.js, and CSS. Lars contributes to OSS at GitHub, publishes articles at Medium, and you can follow @webprolific on Twitter.

↑ Back to top