ES6

GETTING STARTED WITH EcmaScript 6

made with ❤ and a lot of ☕ by js-craft.io

Table of contents

Chapter 1: Introduction

You have probably heard of ECMAScript 6, also known as the most recent version of the standard that defines JavaScript. Maybe you haven't, in which case you shall be informed that we're basically talking about the newest version of JavaScript here. This specification includes a whole bunch of exciting changes intending to make the language more flexible and powerful.

To get a better and more profound understanding of those technologies we'll cover the most important essentials in this ebook. We'll start by taking a look into the ECMAScript 6 features in chapter 2. You'll get an overview of key concept like classes, promises or arrow functions.

So let's start and explore some of the most important features of ECMAScript 6:

- Scoping
- Classes
- Modules
- Promises
- Arrow Functions
- Template Literals

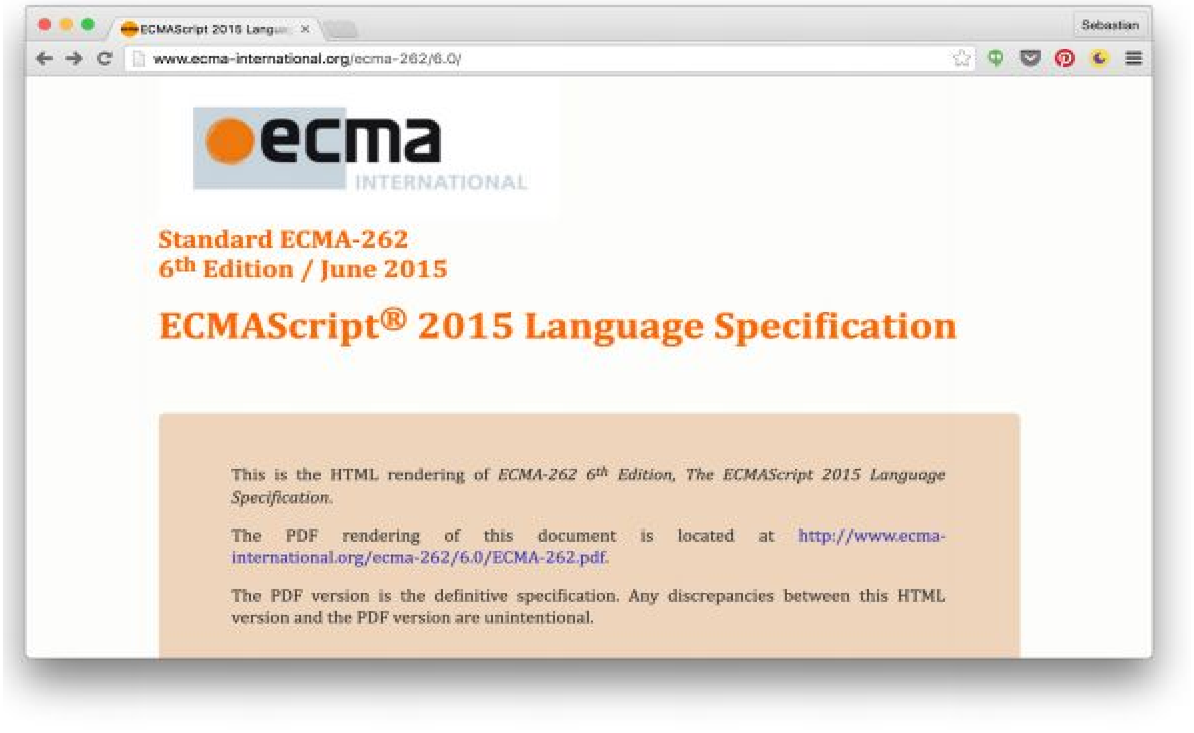
Chapter 2: ECMAScript 6

When talking about ECMAScript 6 (ECMAScript 2015) it's most important to understand that ECMAScript is just a specification. JavaScript is the corresponding implementation which is done by browser vendors.

This can be compared to the HTML specification (e.g. HTML5) which is defined by the World Wide Web Consortium (W3C) and implemented by browser vendors too.

Different browsers are implementing specifications in different ways and most of current browsers are not able to support ECMAScript 6 features by default right now. That is the reason why we need to transfer JavaScript code, which is based on ECMAScript 6, back to JavaScript code which complies to ECMAScript 5. Fortunately this step is done automatically as part of the Ionic 2 build process.

The ECMAScript 6 specification is containing some important and fundamental changes that are taking JavaScript to the next level. The specification can be found at http://www.ecmainternational.org/ecma-262/6.0/.



Standard ECMA-262
6th Edition / June 2015

ECMAScript® 2015 Language Specification

This is the HTML rendering of *ECMA-262 6th Edition, The ECMAScript 2015 Language Specification*.

The PDF rendering of this document is located at http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf.

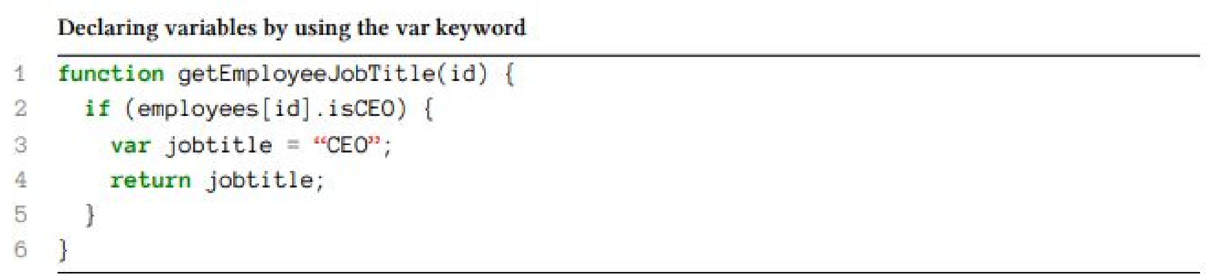
The PDF version is the definitive specification. Any discrepancies between this HTML version and the PDF version are unintentional.

ECMAScript 6 specification website

Let’s take a look at the most important ECMAScript 6 features in the following:

Scoping

If you have been using ECMAScript 5, declaring variables by using the var keyword should sound familiar to you:



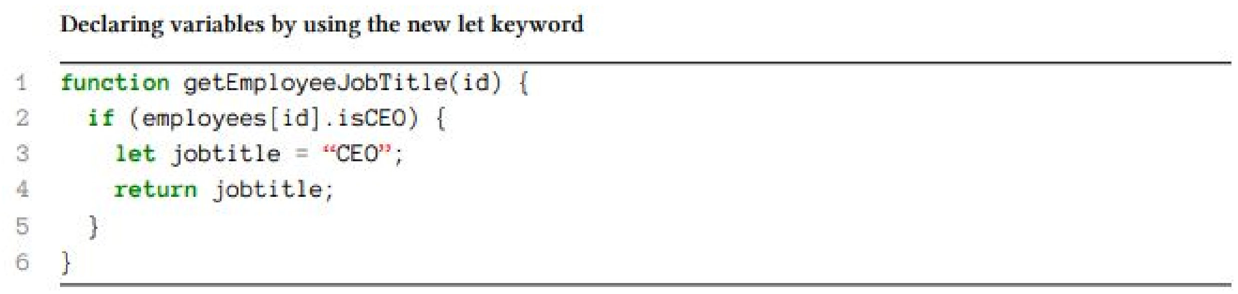
Declaring variables by using the var keyword

```
function getEmployeeJobTitle(id) {
  if (employees[id].isCEO) {
    var jobtitle = “CEO”;
    return jobtitle;
  }
}
```

In this example the variable jobtitle is declared within the if block. In pretty much any other language, this would limit the availability of jobtitle to the if block. Outside of this block the variable would not be defined and therefore not be accessible.

In JavaScript the opposite is the case. Even if jobtitle is declared in the if block the declaration is done on the top of the function and therefore jobtitle becomes available outside the if block.

With ECMAScript 6 a new keyword is introduced which can be used: let. Declaring variables with let is achieving a behavior which is more like you would expect:



Declaring variables by using the new let keyword

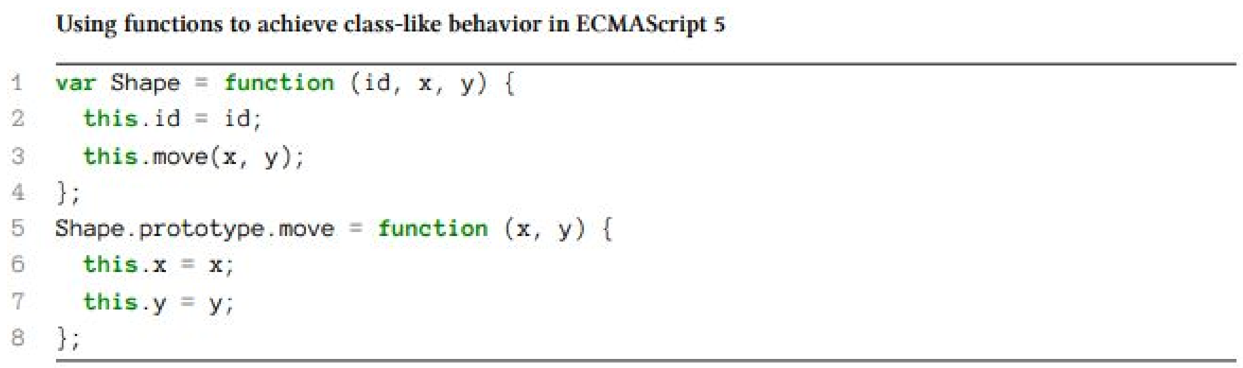
```
function getEmployeeJobTitle(id) {
  if (employees[id].isCEO) {
    let jobtitle = “CEO”;
    return jobtitle;
  }
}
```

In this example jobtitle is only available within the if block. Accessing the variable outside the if block is not possible.

It is planned to replace var completely with let with the upcoming versions of the ECMAScript specification. So, it would be best to stop using var right now and only declare variable by using let.

Classes

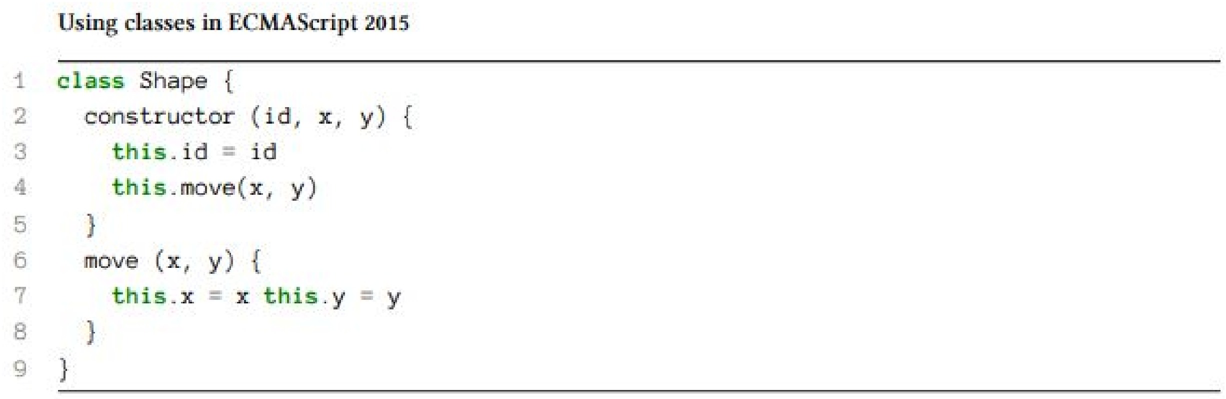
With ECMAScript 2015 the JavaScript language becomes fully object-oriented. Classes can be defined and inheritance can be applied. These are features you might already know from working with object-oriented programming languages like C++ and Java. In the previous version of the ECMAScript specification a common approach was to use functions to rebuild the missing class features which resulted in code like the following:



Using functions to achieve class-like behavior in ECMAScript 5

```
var Shape = function (id, x, y) {
  this.id = id;
  this.move(x, y);
};
Shape.prototype.move = function (x, y) {
  this.x = x;
  this.y = y;
};
```

Now, in ECMAScript 2015 the class keyword is available to define classes. The example from above can be rewritten by using the following code:



Using classes in ECMAScript 2015

```
class Shape {
  constructor (id, x, y) {
    this.id = id
    this.move(x, y)
  }
  move (x, y) {
    this.x = x this.y = y
  }
}
```

The class Shape is defined with two elements:

- A class constructor is defined by implementing a constructor function. This is a special function which is called once when a new object instance is created
- The class method move is defined

A new object instance of Shape can be created by using the new keyword:

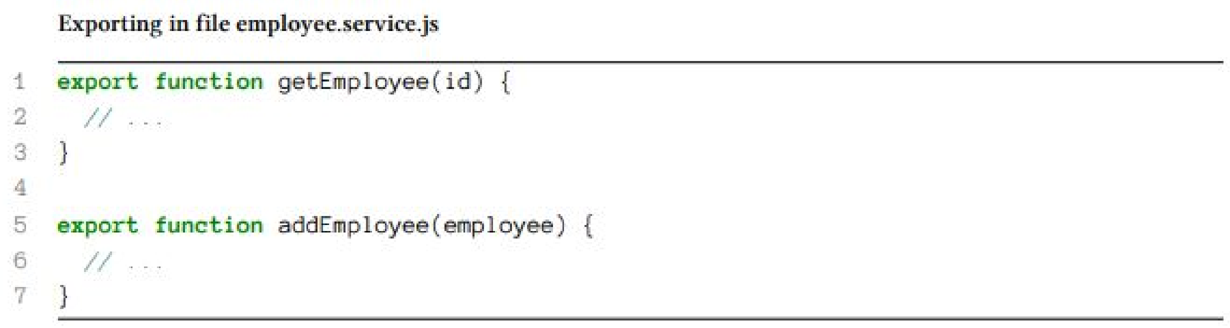
let shape = new Shape(1, 15, 20);

The object instance can then be used to execute class methods:
shape.move(20,25);

Modules

Before ECMAScript 6 there has been no standard way of organizing functions in namespaces and dynamically load code. Third party solutions like CommonJS and AMD (Asynchronous Module Definition) have been available. As non of these extensions have been defined as standard, there have been a lot of discussions of what approach is the best and should be used.

With the ECMAScript 6 specification now a standard way for handling modules is defined and available for use. This new syntax makes exporting and importing code very easy. Let’s take a look at a simple example. In the following listing you can see the implementation of two functions in file employee.services.js. Both functions are exported by using the export keyword:


Exporting in file employee.service.js

```
export function getEmployee(id) {
  // ...
}

export function addEmployee(employee) {
  // ...
}
```

Let’s say we want to use both service functions in one of our application components, e.g. in app.component.js. First, we need to add a corresponding import statement on top of the file:

import { getEmployee, addEmployee } from './employee.service';

Now both functions are available in app.component.ts and can be used.

It’s also possible to assign alias names. In the following example the function getEmployee is imported with the alias myOwnMethodName:

import { getEmployee as myOwnMethodName } from './employee.service';

In this case you must use myOwnMethodName to access the getEmployee function. Another way to import from a module is to use the asterisk sign as a wildcard:

*import * as EmployeeService from './employee.service';*

Here we’re importing everything from employee.service.ts. At the same time all exports are made available via

EmployeeService: EmployeeService.getEmployee(12);

The module concept is very important in React as well. All standard components are made available as modules. In order to use things the frameworks are providing you need to import first.

Promises

Promises are used to simplify asynchronous programming and are an integral part of ECMAScript 6. Before we had to use callback functions to handle asynchronous code executing. That would have looked like the code you can see in the following:

Handling asynchronous code by using callback functions

```
getEmployee(id, function(employee) {
  getSupervisor(employee, function(supervisor) {
    // further processing
  });
});
```

The getEmployee and the getSupervisor method are executed asynchronously. To receive the result of each method a callback function is passed as a second parameter to the function call. The anonymous callback method is called when the result is available. As getEmployee needs to be executed first and getSupervisor second, after the result of the first function call is available, we're calling getSupervisor inside the callback function of getEmployee. As you can see, callback inside of callback makes the code heard to read and understand. A much cleaner way is to use promises instead:

Handling asynchronous code by using promises

```
getEmployee(id)
  .then(function(employee) {
    return getSupervisor(employee);
  })
  .then(function(supervisor) {
    // further processing
  })
```

As you can see this code is much better to read because the pieces are executed as you read it. A promise is basically handled by attaching the then method. The then method can take up to two parameters:

- a success callback function
- a reject callback function

In the example we're using just a success callback function, so only one parameter is passed to the then method. Promises are always stateful. There are three different states:

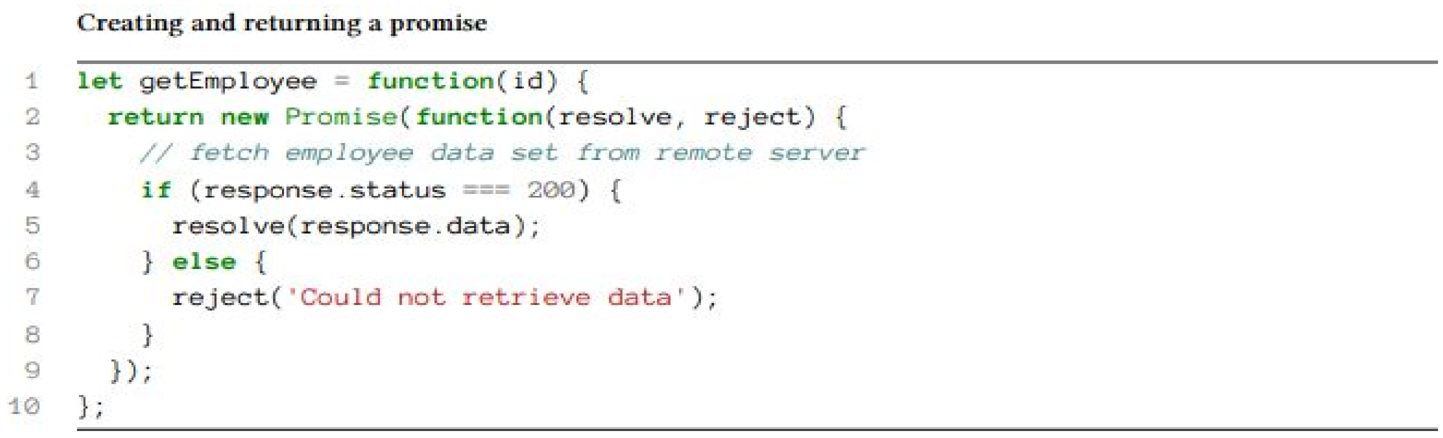
- Pending
- Fulfilled
- Rejected

If a promise is in state pending the asynchronous work is not finished yet (e.g. a remote server call). The then method is not yet executed.

When the promise is fulfilled then the asynchronous work has been finished with success. Then then method is executed and the success callback function passed in as the first parameter is called.

A promise can end in rejected state. That's the case when the asynchronous work has not been finished successfully (e.g. a remote server call has resulted in an error). In this case the second method callback is used to handle the rejected state. The argument of that function would by a rejected value or an error.

Ok, now we've learned how to handle promises. But, how to create a promise so that we can return a promise object from functions which are running asynchronous code? In fact, that's very easy because there is a new class called Promise. The class constructor of Promise is taking a function as an argument which is expecting two parameters: resolve and reject. With that information available we're able to implement the getEmployee function from above to return a Promise object:


Creating and returning a promise

```
let getEmployee = function(id) {
  return new Promise(function(resolve, reject) {
    // fetch employee data set from remote server
    if (response.status === 200) {
      resolve(response.data);
    } else {
      reject('Could not retrieve data');
    }
  });
};
```

Arrow Functions

With the new arrow function syntax, ECMAScript makes writing anonymous functions (like callbacks) a lot easier. To apply the new syntax you simply need to define methods by using the fat arrow operator (⇒). The previous example Handling asynchronous code by using promises can be rewritten.

Before:

Handling asynchronous code by using promises

```
getEmployee(id)
  .then(function(employee) {
    return getSupervisor(employee);
  })
  .then(function(supervisor) {
    // further processing
  })
```

After:

Handling asynchronous code by using promises and arrow functions

```
getEmployee(id)
  .then(employee => getSupervisor(employee))
  .then(... )
```

Much shorter than before. With the new syntax a lot is happening implicitly. E.g. the return value is assigned automatically to the variable which is written before the operator. No need to use the return keyword explicitly. On the right side of the fat arrow operator you can find the function body which consists of only one method call (getSupervisor).

If your function body needs to comprise multiple statements you can use a block on the right side:

Handling asynchronous code by using promises and arrow functions with block

```
getEmployee(id)
  .then(employee => {
    console.log(employee);
    return getSupervisor(employee);
  })
  .then(... )
```

Note that we now need to use the return keyword because the arrow function block contains multiple statements.

For arrow functions there is another thing which is different from normal function: this is lexically bound. So what does this exactly mean? It's quite simple: There is no new this context which is only valid inside of the function. The this scope from outside the function is also valid within the arrow function. You can access all this properties which are valid outside from inside of the arrow function without needing to pass those values as a parameter into the function. Ok let's take a look at a simple example, first by using ECMAScript 5:

Workaround to access context this in ECMAScript 5

```
var self = this;
this.nums.forEach(function (v) {
  if (v % 5 === 0)
  self.fives.push(v);
});
```

Here you can see that the outside this is not accessible in the function which is passed as an argument to the forEach call. To access this inside the function a workaround needs to be applied. A variable named self is declared outside the function block. self is initialized with this. As self is available inside the function block as well, we can use this variable to access the this context from outside the function.

In ECMAScript 5.1 a little extension helped us to avoid using a separate variable to make this accessible inside a nested function block:

Using the bind function in ECMAScript 5.1

```
this.nums.forEach(function (v) {
  if (v % 5 === 0)
    this.fives.push(v);
}.bind(this));
```

By using bind(this) we're able to lexical bind this to the function and to make it accessible.

Using arrow functions in ECMAScript 6 makes this step obsolete.

Lexical this in ECMAScript 6

```
this.nums.forEach((v) => {
  if (v % 5 === 0)
    this.fives.push(v)
})
```

Template Literals

Composing strings in former versions of JavaScript has always been done by using something similar to:

```
fullname = 'Mr ' + firstname + ' ' + lastname;
```

With ECMAScript 6 composing of string is getting much easier by using template literals:

```
let fullname = `Mr ${firstname} ${lastname}`;
```

Using template literals require us to use backticks (‘) instead of simple quotes. Another advantage of using backticks is that multiline strings are supported. This is a great feature for writing embedded HTML template code, e.g. for React or Angular 2 components:

```
let template = `<div>
        <h1>Hello World</h1>
</div>`;
```

Chapter 3: Conclusion

Now you have a basic understanding of ECMAScript 6. As you saw in the examples the concepts are a superset of JavaScript based on ECMAScript 5.x. Many new concepts have been introduced to make JavaScript programming more mature and easier to use for bigger projects.

Using ECMAScript 6 is still optional. You can still use modern frameworks like React or Angular 2 without the need of writing code with ECMAScript 6 features included. Nevertheless, my advice would be to try it out. Many things get easier to implement, less code is needed and the code you're writing becomes much easier to read.