ESSENTIALS

{ ECMAScript 6 and TypeScript }

CodingTheSmartWay.com

Sebastian Eschweiler

Essentials - ECMAScript 6 and TypeScript

Sebastian Eschweiler

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.

Maybe you've also heard about TypeScript. TypeScript is a new Microsoft offering that seeks to change the way we write JavaScript. As the name implies, TypeScript associates a strongly typed layer in conjunction with JavaScript. Both, ECMAScript 6 and TypeScript, are great extensions on top of classic JavaScript. With the features provided, JavaScript programming becomes more mature, easier to read and understand and more powerful, especially when used in bigger projects.

To get a better and more profound understanding of those technologies we'll cover the most important essentials in this comprehensive 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.

In chapter 3 we're going to explore key concepts of TypeScript. By using simple and easy to understand examples you'll learn how to apply typing in your code.

Many modern web frameworks are already making use of ECMAScript 6 and TypeScript. For Angular 2 and Ionic 2 (which is based on Angular 2) ECMAScript 6 and TypeScript are fundamental. Many of the concepts are an integral part of the frameworks. Investing your time to learn ECMAScript 6 and TypeScript fundamentals is an investment which surely will pay off in many ways.

So let's start and explore the most important features of ECMAScript 6 and TypeScript.

Make Sure To Check Out Our Resources

Website

http://CodingTheSmartWay.com

Twitter

@codingsmartway

YouTube

https://www.youtube.com/Codingthesmartway

Ebooks

Book Bundle

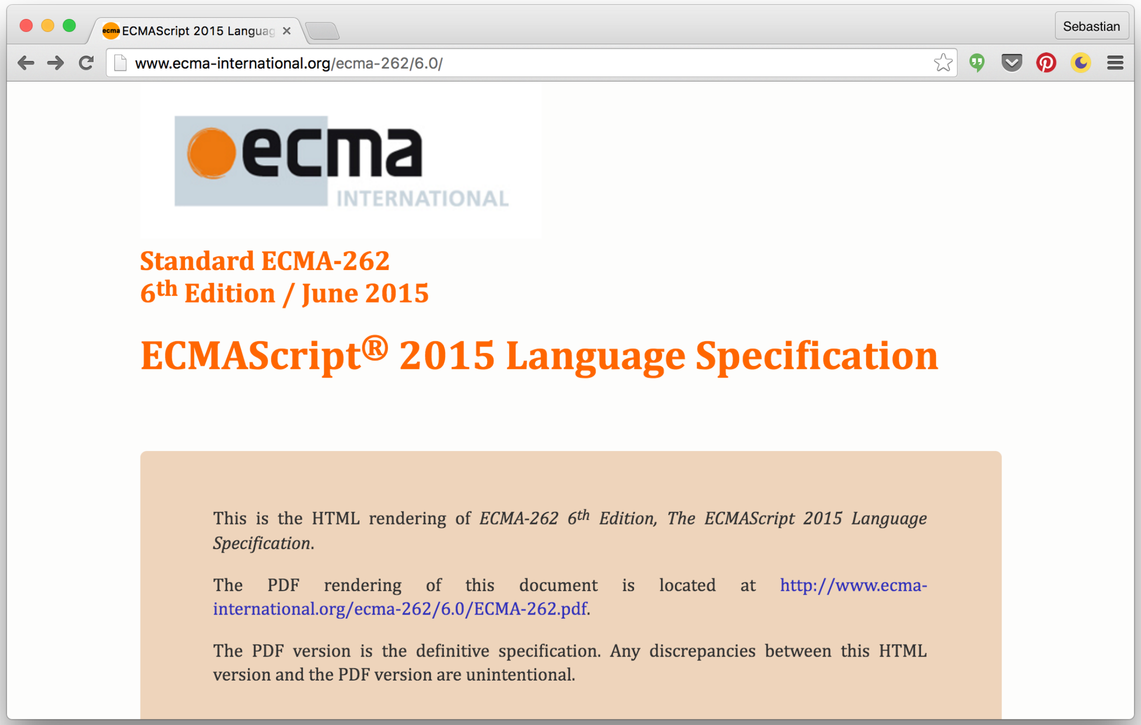
Chapter 2: ECMAScript 6

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.ecma-international.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-orientied progamming 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 Angular 2 and Ionic 2 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 Angular 2 components:

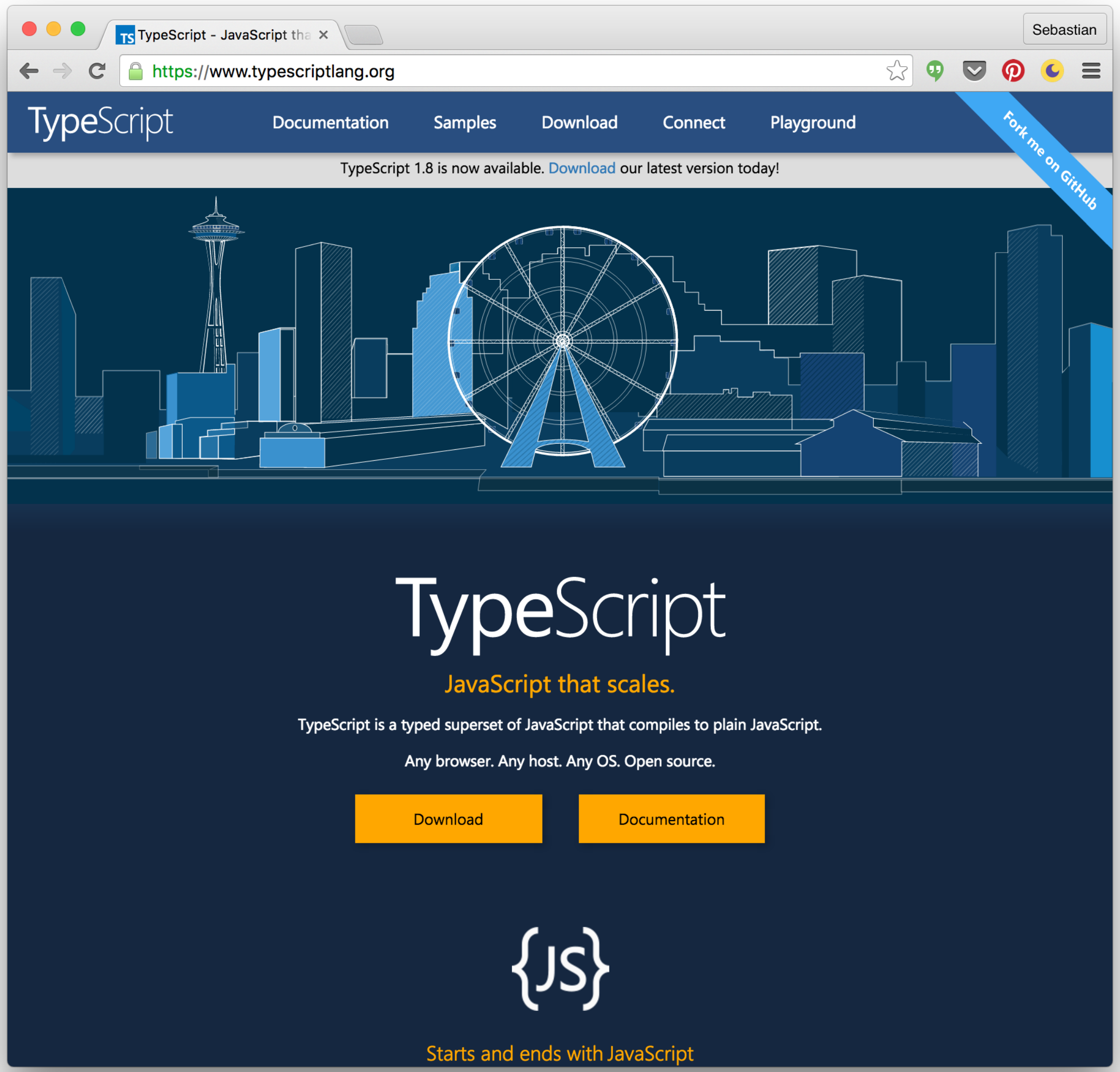
```
let template = `<div>
<h1>Hello World</h1>
</div>`;
```

Chapter 3: TypeScript

Another important concept which is used heavily in Angular 2 and Ionic 2 is TypeScript. TypeScript is a typed superset of JavaScript. It compiles to plain JavaScript, so that the resulting code can be interpreted by any browser.

Using TypeScript allows you to program with JavaScript in a way you might already know from more stricter, object-orientied languages like C++, Java or C#. As we've already learned that with ECMAScript 2015 we're getting the feature of classes TypeScript is the perfect extension to that. Beside being able to work with class types TypeScript offers us the ability to also work with static typing in JavaScript. The difference here is that this typing can be evaluated already at compile time and you do not have to wait until runtime.

The TypeScript website can be found at https://www.typescriptlang.org/.



TypeScript 1.8 is now available. Download our latest version today!

TypeScript

JavaScript that scales.

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.

Any browser. Any host. Any OS. Open source.

TypeScript website

Transpiling

Transpiling is the term which is used to describe the process of transferring TypeScript code to plain JavaScript code. If you would like to use TypeScript standalone you first have to install the corresponding NPM package via:

```
$ npm install -g typescript
```

To start transpiling you can execute the TypeScript compiler by using the *tsc* command:

```
$ tsc myfile.ts
```

The JavaScript output is written to *myfile.js*.

Types & Return Types

Adding type information to a variable is very easy by using the following syntax:

```
let varname: [type];
```

The `[type]` placeholder needs to be replaces by one of the basic types:

- `boolean`: this is the most basics datatype with just two values - *true* and *false*
- `number`: a floating point value
- `string`: textual values (use double quotes, single quotes or backticks to surround string data)

E.g. if you want varname to be of type string you simple need to write:

```
let varname: string;
```

If you would like to assign a value at the same time you can simple extend the statement to:

```
let varname: string = "just some text";
```

By using backticks to assign a multiline string or embed expressions:

`let varname: string = Hello` ${ fullName };

Furthermore there are two special basic datatypes:

- any
- void

By using *any* as datatype we're able to describe the type of a variable which we do not know when writing an application. The values are assigned dynamically and can be of different types:

```
let notSureAboutType: any;
```

The *void* type is indicating the absence of any type at all. A common use case of *void* is the return type of a function which is not returning any values:

```
function notReturningAnything(): void { ... }
```

Declaring a variable of type *void* does not make much sense because the only values which could be assigned are *undefined* or *null*.

The general syntax for defining a function with return type is:

```
function myFunc(): [type] { ... }
```

Enums

TypeScript also offers enums. You can define an enum by using the enum keyword and by defining possible values. The values are put into curly braces and are separated by comma as you can see in the following example:

```
enum TodoStatus {Open, Started, Done}
let myTodo: Todo = new Todo();
myTodo.status = TodoStatus.Open;
```

Internally all values of an enum are represented by a numeric value. The values are starting at `0`. If you want, you can set your own value:

```
enum TodoStatus {Open = 1, Started, Done}
```

Arrays

Defining typed arrays is also possible by using so called *Generics*. Here is a simple example:

```
let myArray: Array<string>;
myArray.push(“some string”);
```

Beside native datatypes you can also use your own (class) types in TypeScript:

```
let myTodo: Todo;
```

In this case *Todo* is a class you have defined in your project. Using that class type together with Generics is also possible:

```
let todos: Array<Todo>;
```

If you now try to insert anything into the array which is different from an object of type *Todo* the TypeScript compiler is returning an error.

Interfaces

TypeScript is supporting interfaces. Interfaces are just descriptions of the actions that an object of a certain type can perform. By applying interfaces to classes we’re able to enforce the class to provide certain methods. For example, let’s say we have a *Car* and a *Truck* class. Both classes should have a *startEngine* action. The exact implementation - how the engine is started - should be left to each particular class. The implementation of the *startEngine* method of class *Car* differs from the implementation you can find in the *Truck* class.

Let’s define an interface named *Vehicle* which contains the method which all concrete class implementations should share:

```
interface Vehicle {
  startEngine();
}
```

Note, the interface is only including the method signature without the method body. Of course you can also define properties in the interface, e.g.:

```
interface Vehicle {
  engineStarted: boolean;
  startEngine();
}
```

In this case the property *engineStarted* is defined with the *datatype* boolean.

Implementing the interface is very easy by using the following syntax:

Implementing interfaces with TypeScript

```
class Car implements Vehicle {
  engineStarted: boolean;
  constructor() {
    this.engineStarted = false;
  }
  engineStart() {
    // start the engine of a car
    this.engineStarted = true;
  }
}
```

By using the *implements* keyword we're defining that the class *Car* should implement the previously defined *Vehicle* interface. Herewith we make sure, that the TypeScript compiler checks that class *Car* is containing an implementation of the *engineStart* and is containing the property engineStarted. Furthermore a constructor is added to set the *engineStart* property initially to *false*. The second implementation of the *Vehicle* interface is done for the *Truck* class:

Implementing interfaces with TypeScript

```
class Truck implements Vehicle {
  engineStarted: boolean;
  constructor() {
    this.engineStarted = false;
  }
  engineStart() {
    // start the engine of a truck
    this.engineStarted = true;
  }
}
```

Here you have the option to implement a different logic of *engineStart*.

Having defined an interface type like *Vehicle* enables you to use that type where you would expect to get a *Car* or *Truck* instance. E.g. if you want to define a typed array which should be capable of containing *Car* and *Truck* objects at the same time you only need to use the following code:

```
let vehicles: Array<Vehicle>;
```

Now you can push *Car* and *Truck* elements to the vehicles array without getting error messages from the TypeScript compiler.

Optional and Default Parameter

We've already learnt how TypeScript enables us to define datatypes of function parameters and return types. By default every parameter is assumed to be required. For a function call the compiler will check if all the defined parameters are supplied with corresponding values. Furthermore the compiler will check if the number of supplied values is matching with the number of defined parameters. Passing in more values than parameter is not possible by default.

This is a different behavior than in JavaScript. In JavaScript parameters are optional and you can decide to leave them off when calling a function. In this case the value is automatically set to *undefined*.

Fortunately there is a way to handle optional parameters in TypeScript to. You simply need to add *?* to the end of the parameter definition, like you can see in the following example:

```
function stringBuilder(string1: string, string2?: string) {
  if (string2)
    return string1 + " " + string2;
  else
    return string1;
}
```

In this case the function takes up to two string parameters: *string1* and *string2*. At the end of *string2* the *?* is added so this parameter is optional. Now you have the option to pass one or two string parameters when calling the function. You can use it with just one parameter:

```
let result = stringBuilder("Hello");
```

Or you can use it with two parameters:

```
let result = stringBuilder("Hello", "World");
```

But calling the function with more than two parameters will lead to an error returned by the TypeScript compiler.

Please note: optional parameters must follow required parameters. Changing the order in the previous example to

```
function stringBuilder(string2?: string, string1: string) { … }
```

is not possible because now the first parameter is optional and the second parameter is mandatory.

Another option TypeScript offers is to get function parameters initialized by default. We're able to rewrite the function *stringBuilder* to contain a default-initialized parameter:

```
function stringBuilder(string1: string, string2 = "World") { … }
```

Here you can see that *string2* is getting the value "World" by default. This means that *string2* is still optional, but, if you do not supply a value for that parameter, it will be automatically set to the default value.

Decorators

Decorators are an easy way to extend class definition (methods, accessors, properties and parameters) with additional metadata. If you have been working with languages like Java or C# before, you might already be familiar with the concept of annotations which is similar to decorators.

In Angular 2 and Ionic 2 decorators are used in many ways. In this case we're using standard decorators the frameworks are providing. The standard framework decorators enable us to add metadata to various things. By using these decorators we can say that a class should be a component (@Component), define input (@Input) and output (@Output) properties, declare a pipe (@Pipe) or define that a class has dependencies that should be injected into the constructur (@Injectable).

Using these built-in decorators is very easy. Decorators which are applied on class-level are added just before a class declaration as you can see in the following example:

Implementation of an Angular 2 component using the @Component class decorator

```
import {Component} from '@angular/core'

@Component({
  selector: 'my-app',
  template: `
    <h4>Todos List</h4>
    <h5>Number of Todos: <span class="badge">{{todos.length}}</span></h5>
    <ul class="list-group">
      <li *ngFor="let todo of todos" class="list-group-item">
       {{todo}}
      </li>
    </ul>
    <div class="form-inline">
      <input class="form-control" #todotext>
      <button class="btn btn-default" (click)="addTodo(todotext.value)">Add Todo\
</button>
    </div>
  `
})
export class AppComponent {
  todos: Array<string>;
  constructor() {
    this.todos = ["Todo 1", "Todo 2", "Todo 3"];
  }
  addTodo(todo: string) {
    this.todos.push(todo);
  }
}
```

In this Angular 2 example we're using the *@Component({...})* class decorator to declare that the class *AppComponent* should be an Angular 2 component. The decorator *@Component({...})* is added before the class declaration. The component decorator gets one argument. This argument is an object containing properties. By using these properties we're able to further specify the Angular 2 component.

Chapter 4: Conclusion

Now you have a basic understanding of both *ECMAScript 6* and *TypeScript*. As you saw in the examples both 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 and TypeScript is optional. You can still use modern frameworks like Angular 2 and Ionic 2 without the need of writing code with ECMAScript 6 and TypeScript 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.

Make Sure To Check Out Our Resources

Website

http://CodingTheSmartWay.com

Twitter

@codingsmartway

YouTube

https://www.youtube.com/Codingthesmartway

Ebooks

Angular 2 - A Practical Introduction To The New Web Development Platform

https://leanpub.com/angular2-book/

Ionic 2 - A Practical Introduction To Hybrid Mobile Apps Development

https://leanpub.com/ionic2-book/

Book Bundle

Angular 2 and Ionic 2 Book Bundle

https://leanpub.com/b/angular2-ionic2-books