Marc-André Leprohon

ECMAScript 6 and the evolution of JavaScript

A deeper look into the language's new features

Helsinki Metropolia University of Applied Sciences

Bachelor Degree

Information Technology

Thesis

18 April 2017

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Marc-André Leprohon ES6 and the evolution of JavaScript 35 18 April 2017.
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Kimmo Sauren (project supervisor)

The thesis covers the evolution of the JavaScript programming language from its inception up until the ECMAScript 6 iteration of the standard. The goal of the thesis was to analyse the language's new features and to explain how they affect its structure, purpose and nature.

The project was carried out mostly through exhaustive literature review and as a technical analysis of the new features. The latter was often executed by comparing the ES6 features with their old ES5 equivalents. Code examples were added anywhere relevant in order to explain and support specific claims.

As a result of the technical overview of ECMAScript 6, the thesis intends to prove how the new iteration of the standard is developing the language further by introducing new innovative features thus allowing programmers to build powerful applications more simply and efficiently. Moreover, the structural changes of the languages were taken into consideration and their impact analysed.

In conclusion, the ECMAScript 6 standard certainly tries to redefine the language itself, mostly with the inclusion of classes. While JavaScript was considered a functional programming language, it is safe to conclude that it has started to move towards object-oriented principles in the recent years. With its increasing popularity, it is reasonable to expect an acceleration in the redefinition of the language.

Keywords	JavaScript, ES6, ECMAScript 2015

Contents

1 Introduction

The web is becoming more and more complex and the applications running behind the scene are growing in terms of functionality and complexity. In 2017, JavaScript is at the heart of it all. All browsers can interpret JavaScript code and most of the dynamic nature of web pages is made possible because of JavaScript. In the recent years, the language has spread even further, expanding to the server side with technologies such as Node.js. This thesis covers the birth of JavaScript, putting it in context with its predecessors, and its evolution to this day. Then are introduced the core concepts of ECMAScript6 and the new features it brings to programmers. Finally, an attempt is made to explain how this iteration of the ECMA standard will develop the web further for years to come.

2 Brief history of JavaScript

2.1 From hypertext to graphical browsers

JavaScript itself was created in 1995, but it is useful to examine the time before this to understand the context in which it was born. While the communication over a network go as far as the 1960s, the Internet as we know it really started to develop in the 1980s and 1990s. The JavaScript discussion really becomes relevant when browsers start to emerge on the Internet as a way to access web pages over the HTML protocol. In its infancy, the Internet was a mean for researchers and scholars to exchange information. It was the National Center for Supercomputing Application (NCSA) who developed the first graphical web browser [1]. Its predecessors, such as ViolaWWW and Lynx, were purely textual browsers and while they are an integral part of the development of the Internet, they never gained mainstream traction. The creator of Mosaic was Marc Andreessen who would later join Netscape and work on the Navigator product. The release of Netscape Navigator is pivotal for the history of JavaScript and the web in general.

2.2 Netscape Navigator and Brendan Eich

In 1995 Netscape started supporting Java but soon faced the problem that web designers and developers were generally not Java developers or even acquainted with the object-oriented principles. A smaller-scale, purpose-built language was needed. In 1995, the company hired Brendan Eich, who was tasked to develop a language that would allow both web administrators to connect and interact with the database and web

developers to interact with the web pages in the browser and make them more dynamic [2, 46]. Already in its inception the language had both a client and server purpose. A loosely-typed scripting language was born under the wing of Eich and was called Livescript. In the same year the language was renamed JavaScript, a marketing move that was supposed to exploit Java's momentum, but would eventually lead to a lot of confusion substantial confusion among the general public and new programmers. [3] [4]

2.3 Fighting a bad reputation

The evolution of JavaScript from its humble beginning to today is quite remarkable. In its early inception, it was considered by many a hobbyist scripting language, much different from fully fledged and tooled languages such as Java. However, its purpose-built nature allowed it to grow and be adopted overtime by other browsers. According to Shannon Horn, the language went from a phase of enthusiasm to a phase of distrust (partly because of security issues) followed by a renaissance in the more recent years. [5, 36]

2.4 Microsoft

Microsoft recognized the potential for JavaScript and created its own implementation of the language called JScript. The language, which was first released in Internet Explorer (IE) 3, was very similar to JavaScript but with a few IE-specific features. Over the next few years, both Netscape and Microsoft would continue developing their own version of the language. The languages remained very similar, but the discrepancies between led to the two led to a will to standardize. [6, 27]

2.5 Standardisation

In 1996, as JavaScript grew in popularity, it was taken to ECMA for standardisation. ECMA is an organization founded in 1961 whose focus is to provide and publish standardisation in the IT and ICT field [7]. In a press release, Netscape said that "to enable interested parties to review the JavaScript proposal, Netscape is posting the JavaScript specification on its Internet site [8]."

The first edition of the standard was ready in June 1997 and published under the codename Standard ECMA-262. The report, entitled *ECMAScript: A general purpose, cross-platform programming language*, officially defines for the first time the web scrip-

ting language called ECMAScript. While ECMASCript is a language and a standard of its own, JavaScript, JScript and ActionScript are implementations of the standard [9].

2.6 The long road to ECMAScript 6

The most important update to the standard came in 1999 when ECMAScript 3 was released. It introduced a large numbers of new concepts such as *do-while*, regular expressions, exception handling and several string methods. The next version (4), called Harmory, was eventually abandoned. According to Brendan Eich, the creator of JavaScript, “Some ES4 proposals have been deemed unsound for the Web, and are off the table for good: packages, namespaces and early binding.” [10] We had to wait until 2008 for ECMAScript 5 to be released. It includes features like *strict mode*, getters and setters and JSON support. The most recent implementation is now ECMAScript 6 which was accepted in June 2015 and is slowly being supported by browsers. Most of the code written by web developers still uses ECMASCript 5 syntax and concepts but more and more ECMAScript 6 features are being used by developer and integrated into JavaScript frameworks. The next section is the core of the thesis and will discuss the new concepts introduced by the new standards. It will do so by explaining their structure and purpose. On a more global level, this thesis attempts to explain how these new features will promote the language and move the web forward [11, 45].

3 ECMAScript 6

3.1 Transition period with transpilers

Transpilers, in the most generic sense, translates code into a different standard. In the context of ES6, transpilers are very useful for programmers to write code that is not yet fully supported by all browsers. It allows the web developer to write next generation JavaScript already today and have it translated into code that browsers fully comprehend. The two most popular ES5 to ES6 transpilers today are Google Traceur and Babel (formerly *5 to 6*). As more and more features are supported by browsers, the use of transpilers will become irrelevant. The next section of the thesis will cover the different new features of the language.

3.2 New features

3.2.1 const and let

ES6 is introducing the *const* and *let* keywords. *let* allows the programmer to declare a variable which only exists within a block. It is similar to the *var* keyword which scope

extended to either the function or globally, depending where it is declared. The *let* keyword allows for more control over the scope of the variable.

```
function testingLet() {
    let x = 100;
    if(x < 150) {
        let x = 200;
        console.log(x) // 200 (inside the block)
    }
    console.log(x); // 100 (initial value because we
existed the block)
}
```

Listing 1. Code example for *let*

In the example above, the result would be different if we used *var* instead of *let*. This very simple example shows the potential for using the *let* keyword instead of *var* if we want to manipulate the value within a block only without changing the value of the variable within the function.

Constants are being introduced through the *const* keyword. The purpose of a constant is to allow for maintenance and immutability. They are used to set the value of a variable that should essentially never change. In the exceptional cases where you would change the value of a constant, it remains easy to do. Usually declared on top of the file, the constant value can be changed in a single location and let the application use the new value throughout its code [12].

3.2.2 Scopes

As we have seen with the *let* keyword, ES6 provides more flexibility when it comes to scoping. In ES5, variables could only be globally available or valid only inside a function. ES6 introduces the concept of block scoping to the language.

Block scope

A block can have different shapes, for example: an *if block* and a *for loop* are both considered as blocks. The following example shows how a constant can be defined within a *for loop* and changed on every iteration. The value is still immutable but since the

scope is limited to the block, it is reset on every iteration. Trying to redefine a constant in the same block will create an error.

```
function constScope() {
  for(var i=0; i<arguments.length; i++) {
    const OUTPUT = arguments[i];
    console.log(OUTPUT);
  }
}
constScope('first', 'second');
```

Listing 2. Code example for *block scope*

In ES5, the language has a function-level scope. Variables cannot be block-scope variables per-se. It is, however, possible to emulate block-scoping by writing the following:

```
function foo() {
    var x = 'hello';
    if (x) {
        (function () {
            var x = 'world';
         }());
    }
// x '.
}
```

Listing 3. ES5 block score emulation

Listing 3 is a good example of how ES5 emulation drove the language forward. Many of the new features introduced by ES6 have been executed in ES5 through a series of hack and third party code.

Hoisting and temporal dead zone

The variables declared with *let* are considered to be in a 'temporal dead zone' from the start of the block until the actual variable declaration. This means that if the variable is called before it is declared, a *TypeReference* error will be thrown. This is important to keep in mind since it differs from how *var* variables are treated. A *var* variable will be hoisted to the top of the block so it will be made available before it is actually declared. This is possible because the JavaScript interpreter does two passes on the code. In the first pass, it hoists the variable to the top of the block. Hence, they are available in the

second pass when other operations are processed. *Let* variable do not get such treatment.

3.2.3 Arrow Functions

To understand the new arrow functions, it is best to look at a comparison example between the ES5 and the ES6 nomenclatures:

```
function(x, y) { return x + y; } // ES5
(x, y) =>  x + y // ES6
```

Listing 4. Comparison between a regular ES5 function and a ES6 arrow function

Essentially, arrow functions allow programmers to avoid typing the *function* and *return* keywords, the curly brackets and the semi-colon. This makes the code much easier to read and maintain. Let us consider the following arrow function:

```
values => values[0];
```

Listing 5. Simple arrow function

As Listing 5 shows, the return statement of a single-statement arrow function is implicit. Moreover, since we have a single argument, we can drop the parentheses. In this case, the function returns the first value of the values array. In other programming languages, these single-statement functions are usually called Lambdas.

Arrow functions and *this*

The *this* keyword has always been slightly problematic in JavaScript since it behaves differently than other languages. In PHP for example, this refers to the instance of a class: the current object. In JavaScript however it refers to the function it sits in. Every function creates its own context.

Contrary to traditional functions, the arrow function does not create its own *this* instance. It simply inherits it from the scope surrounding it. In regular functions, the fact that every single function comes with its own scope can create problems. Let us consider the following code:

```
function testThis (service) {
  var that = this;
```

```
    that.foo = 'Hello';
    service.execute(function (response) {
      that.foo = response;
    });
  }
```

Listing 6 Storing *this* value in ES5

Listing 6 illustrates a fairly regular 'hack' in JavaScript. Upon entering the inner function whose role is to execute a function of the service, we encounter a scoping problem where *this* has a new value. For that reason, we store the value of the outer function into the variable *that* and use it in the inner function. With arrows functions, this problem is solved. Note that the problem above could also be solved with bind(), another relatively verbose alternative. In ES6, the arrow functions allows us to do the following:

```
function testThis (service) {
  this.foo = 'Hello';
  service.execute((response) => this.foo = response);
}
```

Listing 7. Arrow function and *this*

The inner function can use *this* directly without having to first store its value in a temporary variable. This is extremely useful when dealing with inner functions as it simplifies the syntax greatly which in turn leads to a less error-prone code. This is most likely the most useful feature of the arrow function in ES6 [13] [14].

3.2.4 New parameters and operators in ES6

Several new types of parameters and operators are introduced to the language with ES6. The following section will introduce the rest and default parameters as well as the spread operator.

Rest parameter

Rest parameters are denoted with the ellipsis (...). In ES5, when wanting a function containing an undefined amount of arguments, programmers can use the arguments keyword which is an array containing everything that was passed to the function. Let us compare ES5 and ES6 codes.

```
// ES5
function hobbyComparer(myHobbies) {
```

```
  for (var i = 1; i < arguments.length; i++) {
    var hobby = arguments[i];
    if (myHobbies.indexOf(hobby) === -1) {
      console.log(false)
      return false;
    }
  }
  console.log(true);
  return true;
}
var myHobbies = ['volleyball', 'painting', 'fishing'];
hobbyComparer(myHobbies, "volleyball", "painting");


// ES6
function hobbyComparer(myHobbies, ...yourHobbies) {
  for (var yourHobby of yourHobbies) {
    if (myHobbies.indexOf(yourHobby) === -1) {
      console.log(false);
      return false;
    }
  }
  console.log(true);
  return true;
}
var myHobbies = ['volleyball', 'painting', 'fishing'];
hobbyComparer(myHobbies, "volleyball", "painting");
```

Listing 8. Use of the rest operator in ES6

The sample function in Listing 8 checks if both parties share all of the same hobbies. In the ES6 code, it is made clearer that the function is expecting both my hobbies (commonly referred as the haystack) and your hobbies (referred as needles). We do not have to rely on the arguments keyword since everything that comes after it is considered as being part of the same argument and stored into an array. In ES5, the arguments keyword contains all of the arguments passed to a function while the rest parameter only contains the last one. One more difference is that the rest parameter is an actual array, which allows you access to all native array methods in order to manipulate it.

Default parameter

Default parameters are a feature already available in several traditional programming languages such as Java or PHP. It is finally being included in the language as part of ES6. It can alleviate issues related to undefined variables within a function. The following code snippet clearly shows the usefulness of default parameters.

```
// ES5
function greet(name) {
    var name = typeof name !== 'undefined' ?  name :
'visitor';
    console.log('Hello '+ name);
}
greet();

// ES6
function greet(name='visitor') {
    console.log('Hello '+ name);
}
greet();
```

Listing 9. Default parameters in ES6

With the new syntax, there is no more need to check if the argument passed is undefined. It is done on the argument level directly by assigning it a default value in the case of an undefined value. Even in the case of an argument specifically set to ‘undefined’, the default value will be used.

Spread operator

As with the rest parameter, the spread operator uses the ellipsis to represent a single entity that could be separate entities. It is most often used as an example replacement for the Function.prototype.apply() approach utilised in ES5.

```
// ES5
array = [1, 2 ,3];
function.apply(null, array);

// ES6
```

```
array = [1, 2 ,3];
function(...array));

//ES6
var numbers = [3, 4];
var numbers2 = [1,2, ...numbers 5,6]; // [1,2,3,4,5,6]

//ES6
numbers2.push(...numbers); //[1,2,3,4,5,6,3,4]
```

Listing 10. Array manipulation with the spread operator

As Listing 10 shows, it is possible to insert an array within another array with the help of the spread operator. It also allows for array easy array pushing. Spread operator makes array manipulation easier and cleaner.

3.2.5 String manipulation: template strings, interpolation, tags and raw strings

Template strings are string literals allowing for simpler multi-line strings. By using the backtick character (`) around strings, it is possible to easily output multiline strings. A perhaps more useful feature is the possibility to interpolate, or substitute a variable directly within a string without having to rely on the concatenation operator.

```
//ES6
let x = 5;
let y= 6;
console.log(`You are ${x * y} years old`); //Will out-
put: You are 30 years old
```

Listing 11. Interpolation in ES6

The tagging functionality of template strings allows the programmer to further customize the output of a string by allowing for manipulation within a function.

```
let x = 30;
function templateFunction(strings, ...values) {
  console.log(`${strings[0]}${values[0]}
${strings[1]}`);
  return 'Processed!';
}
```

```
console.log(templateFunction`You are ${x} years old`);
//You are 30 years old
// Processed!
```

Listing 12. String manipulation with tags

In the example code of Listing 12, we use a function to manipulate the passed string by using interpolation. The function itself also returns a string that we output. This is a useful feature of the language when it comes to developing a user interface where, for example, the current username needs to be output within a predefined and hardcoded string.

Raw strings are an additional string feature offered by ES6. It is a very straightforward and allows for easy access to the string without escaping any characters or interpreting characters such as backslash.

```
//ES6
let 'rawString = String.raw`First line \n Second
line`;
console.log(rawString); //First line \n Second line';
```

Listing 13. Example of raw string usage in ES6

We can use this functionality shown in Listing 13 within a tag function as well with string.raw[x]. There is no such equivalent in ES5.

3.2.6 Modules

Prior to ES6, modularisation was not possible with pure JavaScript. To circumvent the problem, standards such as CommonJS Modules and Asynchronous Module Definition (AMD) were created. They allowed programmers to load JS files (modules) within other JavaScript files. The main advantage to working with modules is code organization and dependency management.

With the advent of ES6, modules are supported at a language level and are denoted with the *module* keyword within the script tag. Both regular scripts and modules use the same .js file extension, but while regular scripts are loaded synchronously by the browser, modules are loaded asynchronously.

Exporting

The first step in creating modular JavaScript is to export code that can be used by other modules.

```
// ES6
// myExporterFile.js
export function myExportedMethod() {
    console.log('Hello from the exported method!');
}
// Export simple variables
export let myExportedVar = {};
// Export whole classes
export class myClass {}
```

Listing 14. Example code for export functionality in ES6

As seen in Listing 14, it is possible to export just about anything, ranging from simple data so full fledged classes. This is the first step in making JavaScript truly modular.

Importing

The counterpart of exporting is the *import* keyword. After exporting a block of code, it becomes possible to import it and use the contained methods directly. The following code example shows different ways to import code.

```
Import { myExportedMethod as myRenamedMethod } from
'./myExporterFile';
myRenamedMethod(); // Hello from the exported method!

//Import everything
Import * as imported from './lib.js';
//Import whatever was defined as default
import imported from ./greet.js
//Importing both default and non-default
import myDefault, { someNonDefault } from ./myfile.js
```

Listing 15. Different ways to use imports in ES6

It is also possible to import a whole module by using the star (*) symbol as depicted in Listing 15. This imports the whole file as a single object and allows for easy access, using the dot notation, to all of the file's methods and variables as regular object properties. This type of import is called namespace import [15].

When it comes to importing default values, the syntax is very simple. Contrarily to the regular, non-default, exports, it does not require any curly braces. The import is named import will import whatever was set as default export within greet.js. It is even possible to export/import both default and non-default bindings. When importing, one simply needs to separate them by a comma. They are easy to differentiate since the default import has no curly braces surrounding it.

Restrictions regarding exports and imports

Both modular expressions are meant to be fairly static. This means that it is not possible to export a variable inside a conditional expression. Similarly, one cannot import anything with a function or statement. In that sense, the modules feature of ES6 is not dynamic [16].

Default exports

It is also possible to define default exports anonymously. There can only be one default export per file. There are several ways to export something by default. One example is to use the renaming scheme using the JavaScript *default* keyword within it:

```
function greet (name) {
    return 'Hello ' + name;
}
export { greet as default };
```

Listing 16. Default exports in ES6

Loading modules in the browser

The traditional and most common way to load JavaScript in a webpage is to include a <script> tag, to provide the location of the script in the *src* attribute and to set the type as *text/javascript*. Loading a module in the browser is very similar except for the type attribute that changes to *module*. This has the effect of excluding all of its content from the global scope. It is encapsulated within the module unless it is specifically exported and imported.

Differed loading

Perhaps the most important aspect of the new modular nature of ES6 is that there is no need to load all of the code at once. This was already possible with technology such as CommonJS and AMD which purpose was exactly to provide JS with a modular functionality. With ES6, however, this becomes part of the language itself.

By default, when including a module in a web page, the browser will load the code sequentially as it appears. In practice, if the first module that the browser encounters contains imports, it will recursively load all the dependencies. The recursive nature of the loading means that the browser will go as deep as needed before coming back to the top-level and load the second module directly included on the web page.

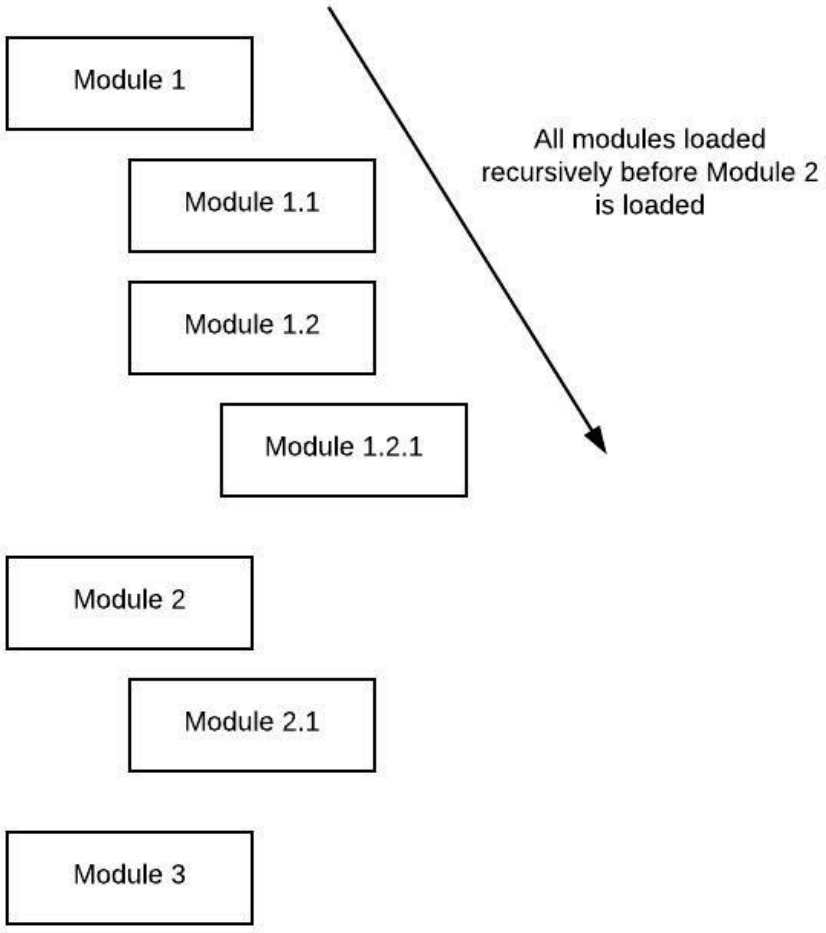

Figure 1. Module loading in the browser

Figure 1 depicts the way the modules are loaded within the browser. The execution of the code follows a similar but slightly different model. Instead of loading the module 1 first, it needs to loads all the dependencies within it. This makes sense because the higher level modules depend on the lower level ones; hence, they are importing them.

It is also possible to load modules using web workers. Web workers provide an API that allows the loading of script to run as background tasks. It can help with page responsiveness since it does not have to wait and does not prevent other scripts or modules from loading. This thesis will not go into many details about web workers, but it is good to note that it is perfectly possible for web workers to handle modules, just like they do with regular scripts [17].

Overall, modules are a way to encapsulate logic. It is also a way to protect your variables and functions from being accessed if necessary. To make the code available to other parts of the application, the programmer must make use of exports and imports.

3.2.7 Classes

It took much iteration of the standards for classes to arrive in the JavaScript world. Object-oriented languages like Java have had access to classes for a long time but it never got included by default within the language. This never stopped third party library developers from creating solutions that emulate classes. With classes come many important concepts such as inheritance, extension, polymorphism, etc.

Prototypal OOP and the constructor pattern

The traditional way to emulate classes in JavaScript is to use a prototypal approach. First, you declare a function. Let us consider an Animal function:

```
// We declare the function
function Animal (color, size) {
    this.color = color;
    this.size = size;
}
// We add prototypal functions
Animal.prototype.describe = function() {
```

```
        console.log('The animal is of size ' + size  + '
and of colour ' + color;
}
//We create a new instance of the "class" and use it
var animal = new Animal('grey', 'medium');
animal.describe();
```

Listing 17. Prototypal approach

Listing 17 shows the ES5 approach to class emulation. The function Animal acts as a class with properties colour and size. Following the declaration, we added new methods via the prototype functionality of the language. Finally, we can create an instance of the class and access its methods.

In essence, JavaScript uses prototypal inheritance to encapsulate all the characteristics of a class, namely its property and methods. There are other approaches than the prototypal one, namely the constructor pattern. This pattern tries to emulate even more closely the classical OOP approach by instantiating an object and calling a constructor. The following example illustrates this approach:

```
function Programmer(name) {
          Person.call(this, name);
}
Programmer.prototype                =                Ob-
ject.create(Person.prototype);
Programmer.prototype.constructor = Programmer;
```

Listing 18. Constructor approach

Listing 18 is a good example of a classic JavaScript malpractice because it throws shade on the real language's own implementation of OOP. In the words of Douglas Crockford, one of the main contributors to the development of the language: "JavaScript's constructor pattern did not appeal to the classical crowd. It also obscured JavaScript's true prototypal nature. As a result, there are very few programmers who know how to use the language effectively." [18]

ES6 approach to classes

ES6 brings solutions to some of those problems and complexity by introducing classes to the language. Let us re-create the simple code above in a more classical approach, using the new ES6 classes.

```
class Animal {
    constructor(color, size) {
        this.color = color;
        this. size = size;
     }

     describe() {
          console.log('The animal if of ' + size  + '
          size and of ' + color + ' color.';
     }
}
let animal = new Animal('blue', 'small');
animal.describe();
```

Listing 19. ES6 classes

In Listing 19 we have declared a class with self-contained properties and methods and utilised it. One of the main differences compared to the ES5 approach to classes is the disappearance of the function keyword within the class, prototype and method declaration. The new syntax is much more concise and in line with other object-oriented programming. This is, however, a superficial difference since JavaScript actually considers classes a function; classes do behave differently than regular functions.

Primary characteristics of classes

Classes are not hoisted, meaning that they behave like a declarative *let*. In practice, this means that functions that make use of a class cannot be called before the class itself is declared and evaluated. Moreover, everything in a class acts in strict mode. And commas between member declarations are forbidden [19].

Class declaration and expression

While the previous example class was named Animal, it is possible to declare classes anonymously:

```
let myClass = class {
```

```
    …
}
let instanceOfMyClass = new myClass("foo");
```

Listing 20. Alternative class declaration

Listing 20 shows that there is nothing required after the class keyword. The main difference with the first declaration resides in the fact that the anonymous class' name property is always empty. It is, however, possible to create non-anonymous class expressions by simply adding an expression after the class keyword.

Getting and setting values

The way to set the value of a property or retrieving its value is very simple. Let us consider a simple example with class foo:

```
class Foo {
    constructor(name) {
        this.name = name;
    }

    get name() {
        return this.name;
    }

    set name(myName) {
        this.name = myName;
    }
}
```

Listing 21. Getter and setter example

In Listing 21 we declared a class with a constructor, a getter and a setter. Calling the getter after instantiating the class will allow the user to get the current name set for the object (done within the constructor). Alternatively, calling the setter will allow us to redefine the name property.

Concept of first-class citizen in JavaScript

Anything that can be passed as a value is considered to be a "first-class" citizen. Classes are no exception to the rule in JavaScript. They can indeed be used as an argument to a function.

Static method

Static methods do not depend on the objects but on the class itself. They can be called directly on the class without the need to instantiate the class first. Let us create the Car class with a setMake static method:

```
class Car {
    ...
    static setMake(name) {
        return new Car(name);
    }
}

let car = Car.setMake('Ford');
```

Listing 22. Static method example in ES6

If we try to call the staticMethod on an instantiated object in Listing 22, an error will be thrown. To avoid the error, the static method should be called directly on the class. The static methods can be used in subclasses as well. Static classes are mostly useful for their utility purpose. For example, if we have a calculator class with methods offering different mathematical operations, it does not make sense to create a new calculator every time. We can simply use the class directly to execute the mathematical operation needed. For small applications, the difference in execution speed is minimal. The following code example consists of a small test that shows the difference between the two:

```
var t0 = performance.now();
class Car2 {
   doSquareRoot(number) {
      return Math.sqrt(number);
   }
}

let car2 = new Car2();
let result2 = car2.doSquareRoot("123456789");
console.log(result2);
var t1 = performance.now();
```

```
console.log("Call to doSquareRoot took " + (t1 - t0) +
" milliseconds.")

var t0 = performance.now();
class Car {
    static doSquareRoot(number) {
        return Math.sqrt(number);
    }
}

let result = Car.doSquareRoot("123456789");
console.log(result);
var t1 = performance.now();
console.log("Call to doSquareRoot took " + (t1 - t0) +
" milliseconds.")
```

Listing 23. Performance test for static methods

The first test in Listing 23 will yield a result of 1.535 milliseconds, while the second one returns 0.585 milliseconds. The difference between the two is only one millisecond. For larger applications, however, where the instantiated class needs to also instantiate a bunch of objects, using static methods will result in less overhead and consequently to more performing applications [20].

Generator methods

A generator allows you to iterate, exit and resume the execution of a function or a method. In the ES6 world, just like with regular functions, the symbol used to denote generators is the star symbol (*). An iterator method must contain the yield keyword. Used alongside promises, generators allow programmers to write asynchronous code by being able to pause the execution within an iteration [21]. The generators themselves are covered more in depth within another section of the thesis.

Subclassing

With the arrival of classes also comes the possibility to extend said classes and create subclasses. The advantage of a subclass is that it possesses all the properties and methods of the superclass, but can add its own methods and properties, without influencing the superclass. A subclass can also be called a derived class while the super-

class is usually referred to as the base class. Let us look at an example where we have a Car base class and a ColouredCar derived class.

```
class Car {
    constructor(make, model) {
        this.make = make;
        this.model = model;
    }
    stringify() {
        return          this.make.toString()          +
this.model.toString()
    }
}

class DatedCar extends Car {
    constructor(make, model, year) {
        super(make, model);
        this.year = year;
    }
    stringify() {
        return super.stringify() + ' was built in ' +
this.year;
    }
}

datedcar = new DatedCar("ford", "taurus", 1998);
datedcar = datedcar.toString();
console.log(datedcar);
```

Listing 24. Subclassing in ES6

The derived class depicted in Listing 24 does not need to assign the values in the constructor since the base class already does it. It can simply use the keyword *super* which in this case refers to the constructor of the base class. In its own implementation of the *stringify* method, it also calls the *stringify* method of the base class.

It is important to note that calling the superconstructor is mandatory before being able to use the keyword *this* or even before instantiating an instance of the derived class

with an empty constructor. In both cases, failing to do so with result in a ReferenceError [16].

Protecting class data

In certain programming languages, protecting data by setting them as private is done via a keyword of the same name. All properties that can only be changed by the current class will be formally denoted as such. To be able to access those properties from instantiated objects, the keyword *public* is used.

In ES5 this was accomplished by appending an underscore character to the variable that should be considered *private*. However, while widely used, this offered no real protection as it is merely a naming convention and a guideline not to alter the data at hand.

```
//ES5
Class MakeItPrivateES5 {
    constructor(x, y) {
          this._x = x;
          this_y = y;
     }
}

//ES6 with WeakMaps
let _x = new WeakMap();
let _y = new WeakMap();

class MakePrivateES6 {
    constructor(x, y) {
        _x.set(this, x);
        _y.set(this, y);
}
...
}
//ES6 with Symbols
const _x = new Symbol('x');
cont _y = new Symbol('y');

class MakePrivateES6 {
```

```
        constructor(x, y) {
            this[_x] = x;
            this[_y] = y;
    }
    ...
    }
```

Listing 25. Private properties in ES5 and ES6

The example in Listing 25 shows how to make properties private within ES5 and ES6. For the latter, Weakmaps and symbols are utilised to accomplish the desired result. The constructor sets its passed data to the WeakMap's' own private data, effectively encapsulating it within itself. In a WeakMap, the keys are weak meaning that the references are cleared by the garbage collector [23].

The idea is the same with Symbols. This type of encapsulation however remains very 'hacky', but there are proposals being outlined to make the private keyword part of the language. Yehud Katz from the ECMAScript's TC39 standards committee, and co-founder of ember.js, is currently preparing such a proposal [24].

3.2.8 Promises

JavaScript has always been an interactive language because of its application on the web. As users interact with a page, the language needs to be able to react accordingly and effectively. As a result, functionality such as events and callbacks were created and perfected throughout the language evolution.

The asynchronous nature of JavaScript

To understand why being asynchronous is important, it is relevant to have a look at the concurrency model and the event loop. At runtime, JavaScript operates using the concepts of stack, heap and queue. The queue consists of a series of messages that are lined-up in the memory, ready to be processed within the stack. Once the stack is empty, the first message in the queue gets taken and a function stack is created. A stack is created when going through the code. The global scope is first analysed and each call to functions is pushed to the top of the stack as a separate frame. Once the

whole stack is executed, the next message in the queue is processed. The heap in this case is a separate part of the memory used to store data.

One of the aspects that differentiate JavaScript from other languages is that it needs to finish the whole messages before it starts to process a new one. If the message takes too long to execute, most browsers present the user with a dialog warning about the script's excessive execution time, even allowing the user to stop the script completely. The simplest example to illustrate this functionality is as follow:

```
function callback() {
    console.log('Hi from the callback');
}
setTimeout(callback, o);
console.log('Hi from the global space');

The output in the console is as follow:
Hi from the global space
Hi from the callback
```

Listing 26. Simple callback example

Listing 26 shows that the callback function was added to the stack first, followed by the output in the global space. Even without a timer (the value is set to 0), the callback has to wait for the rest of the stack to complete.

Always completing but never blocking

One of the key features of the language resides in its I/O processing which never blocks. It can process other inputs while waiting for events or callbacks to complete. This is the essence of the asynchronous nature of JavaScript. In practice this means that while the result of an operation is awaiting its completion in a callback or an event, the application is free to continue executing the stack. When the operation in the callback terminates, the application is notified. The simplest example is a network request; the application will not halt because a 3rd party API is not responding with the expected data. The programmer is strongly encouraged to create asynchronous and sound code but even in the event of a purposefully blocking code. For example, the browser will not accept a synchronous sleep() method [25] [26].

Events and callbacks

Events and callbacks are a major part of JavaScript and should be recognizable to all programmers familiar with the language. To illustrate the event model, let us consider one of the most popular browser events: the click.

```
let el = document.getElementById('button1');
el.onclick = function(event) {
    alert('clicked!');
}
```

Listing 27. Simple event example

In Listing 27, the *onclick* function is not added to the message queue until the user actually clicks the element in question. Events become verbose for more complex scenario and are usually reserved for user interactions such as clicks and taps.

Complex tasks are better handled via a callback function. Callbacks originate from functional programming which makes use of functions as arguments as a basic principle. Here's a basic example using the forEach method.

```
let myArray = ["hehe", "haha", "hihi"];
myArray.forEach(function (el) {
    console.log(el);
});
```

Listing 28. Example with forEach and callback

The forEach method [27], part of the Array class, expects a callback as argument in Listing 26. If successful, the callback then outputs the content of the array to the console. The callback itself, taking the form of an anonymous function, is not executed until the forEach method itself is executed. The function can be either named or anonymous. For example, with a custom piece of code it is easy to simply have callback as a parameter for the function declaration and let the implementation decide on the callback. As long as the callback is effectively executed within the function declaration, the paradigm is valid [28].

This long-winded introduction to promises starts to take form when considering nested callback. As an example, if a callback depends on the completion of another function which also depends on the execution of a callback, we are required to use nested call-

backs. The classic example of callback hell (inspired by callbackhell.com) consists of a function responsible for reading and writing to a file. It is composed of a series of callbacks to determine if the file actually exists, is valid, is readable and is writable. In essence, all these callbacks are necessary and valid, but in reality they create very complicated code which is prone to errors. The callbacks are also limitative for tracking multi-operations process state. This is where ES6 promises shine and improve where callbacks fell short [29].

Improved callbacks

After going through the evolution of asynchronous JavaScript, it is time to introduce the latest and best instalments of the asynchronous world: the JavaScript promise. As the name describes, a promise provide the expected result, at some point. The promise lifecycle comprises of a few different steps. The first state of the promise is pending and it occurs before it is either accepted or rejected. This image taken from the official Mozilla documentation describes the lifecycle perfectly [30].

During the pending state, the application executes the asynchronous actions required and subsequently enters 2 possible states: fulfilled or rejected. In either case, the code will have to be able to handle the scenario. In the case of a rejection, the most logical measure to take is to handle the error correctly, for example, by letting the user know what went wrong. Before going any further, it is important to note the use of the *then* method of the promise. This method allows the programmer to take action whether the promise has been fulfilled or rejected. The recapitulate, a promise has 3 possible states:

- Pending
- Fulfilled
- Rejected

Then method

Each promise needs to implement the *then* method in order to be successful. The method takes 2 arguments: a function to handle a successful promise and a function to handle a failed promise. Let us consider a very simple and abstract scenario where the code promises to do something.

```
let myPromise = doSomething(someParam);
```

```
myPromise.then(function(contents) {
     // fulfilled, do something with contents.
}, function(err) {
    // rejection
    console.error(err.message);
});
```

Listing 29. Example with *then*

In the rare case where a programmer would only want to handle a failed promise, it is possible to replace the then method from Listing 29 with the catch method. It is, however, not mandatory to handle the rejection of a failure [29].

It is also possible to chain the promises, meaning that you call an already settled promise. This is due to the fact that *then* always returns a promise which can be re-evaluated, for example to assign a different value to different objects. In chained promises, the errors are forwarded until a catch or an error handler is encountered.

Promise global object method

There are 4 methods which can be use to interact with promises.

- Promise.all(iterable)
- Promise.race(iterable)
- Promise.reject(reason)
- Promise.resolve(value)

Resolved() returns a resolved promise object. It can either go through the *then* chain and resolve at the end of it, or else it will be fulfilled with the provided value. Let us consider the simplest scenario where the resolve method is used.

```
let myPromise = new Promise(function(resolve, reject) {
    console.log("This is a promise");
    resolve();
    //reject();
}).then(function() {
    console.log("Promise resolved!");
}).catch(function() {
     console.log("Promise unfulfilled!");
});
```

Listing 30. Example of resolve() function

When the promise of Listing 30 calls the resolve method, the promised is considered fulfilled and resolved. However, if we were to uncomment the reject call and comment resolve, the promise would become unfulfilled and the error would be caught by the catch prototype method [29].

Finally, if a program needs to address several operations happening before doing something else, then all method becomes useful. The method allows you to make sure that all operations, for example, network requests, have been executed before executing something else. This enables easy concurrency and keeps the code short and concise. Overall, the main advantage of promises is that it allows for truly asynchronous, non-blocking code to be executed. It also allows for cleaner code and better error handling than regular callbacks. One point to remember is that while callbacks are functions, promises are simply JavaScript objects [33].

4 JavaScript as an object-oriented language

In the words of Eric Ellitott, a leader figure in the JavaScript World, functional programming can be defined as follows: “the process of building software by composing pure functions, avoiding shared state, mutable data, and side-effects. Functional programming is declarative rather than imperative, and application state flows through pure functions.” [34] Functions accomplish all of the roles accomplished by classes and objects in the object-oriented paradigm. More specifically, pure functions play a central role in functional programming. They are defined by two rules: the function must consistently return the same value given the same input arguments and it should have no side effects. They are independent, self-contained and are not affected by external states. They are not affecting external states either. Traditional JavaScript code usually includes a large number of pure functions; it is, consequently, at least partly a functional programming language.

4.1 Shared states

Object oriented programming (OOP), as its name implies, revolves around objects. Objects are an instance of classes (a concept that is now part of JavaScript) that possess defined states and behaviours. In OOP, the complexity is broken down into smaller chunks of manageable contexts; the classes. In opposition to functional pro-

gramming, OOP objects do alter external states. A class' whole purpose might be to change the value of a "master object" which lives outside of its own scope. This concept is called shared state and is omnipresent in OOP code.

4.2 Race condition

While shared state allows for a considerable amount of flexibility, it also contains many caveats. One of them being the race condition problem. In an asynchronous program, very popular concept all over the web, an operation on an object may occur while several other operations have been triggered on the same object. A good example of object being altered simultaneously could be a user profile on a website. Shared states create a timing issue as well as an order of execution issue which can be mitigated by pure functions in functional programming. This is one of the main arguments for proponents of a purely-functional approach to JavaScript programming.

4.3 Advantages of a OOP approach

While functional programming has a large number of advantages, there are several reasons why the language has been making a shift towards OOP principles. A program using classes can be much easier to understand; the state and behaviour are much more clearly defined with class methods. Classes also help define "things" within our applications. If you can put a name on a concept, it is most likely definable as a class with its own set of properties and methods. In a sense, OOP is less abstract than functional programming as it tries to encapsulate a concept understandable by the human within its own container. This has the advantage of being easily extendable when a large amount of programming work on the same program, which is usually the case in an industrial context.

4.4 Best of both worlds

JavaScript has traditionally been considered a functional programming language but some of the changes brought by ES6 are starting to integrate more and more object-oriented principles within the language. In reality, the two concepts are not mutually exclusive and both co-exist within most JavaScript code. Some programmers may prefer a functional-programming-oriented approach while others will choose the OOP route, now more than ever with the inclusion of native classes within the language.

4.5 Performance improvements

According to a report entitles “Performance of ES6 features relative to the ES5 baseline operations per second”, [35] the performance difference between the new ES6 features and their ES5 equivalent varies considerably depending on which browser is interpreting the code. Among the biggest surprise is that only map-set-lookups seem to be faster on ES6 across all browsers. On the other side of the spectrum we find ‘map-set-object’ which is executing slower than its ES5 counterpart on all browsers. However, as browsers improve their internal interpretation of JavaScript, we can expect newer features of the languages to become more powerful than their ES5 counterparts, for which browsers have been optimized for years. For example, from version to version, the V8 engine (the engine powering Chromium and NodeJS) sees incredible speed improvement. The following figure is just an example on how fast JavaScript engines are being optimized to execute recent features such as Promises (ES6) and Async/Await. [36]

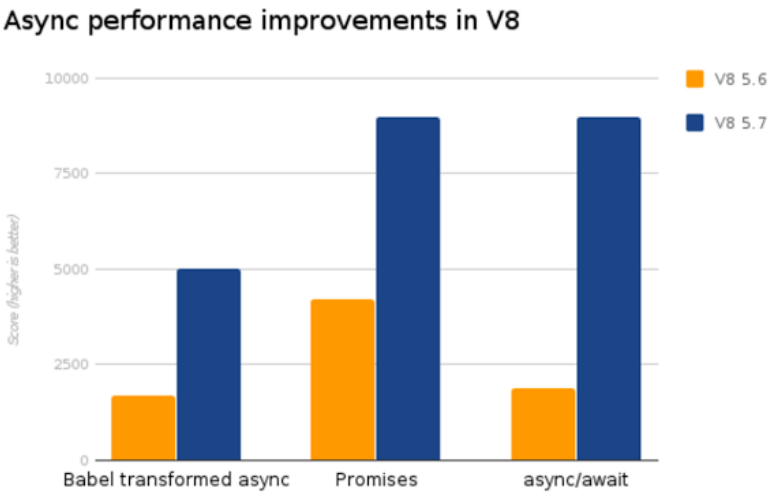

Figure 2. Async performance improvement in V8 engine

5 Conclusion

JavaScript is a language that is continuously being developed. ES7 has been published and ratified in June 2016 and ES8 is already in development and some features being made public. This thesis did not cover the ES7 but a large amount of documentation is already available and some transpilers allows the programmer to take advantage of ES7 features already today. For reference, ES7 includes new features such as exponentiation operator and Array.prototype.includes(), while ES8 will bring Await/async features to the language. [37]

In the TIOBE Index for March 2017, a yearly ranking of most popular programming language, JavaScript comfortably sits in 8th position with a 0,33% usage increase from 2016 [38]. It is safe to say that the future of JavaScript is bright. With the rise of server-side JavaScript environments such as Node.JS, the language that has often being diminishingly called a scripting language by classic programmer is more relevant than ever and will continue to drive and power the web forward.

References

1. National Science Foundation [online]. URL: https://www.nsf.gov/about/history/nsf0050/internet/mosaic.htm. Accessed 16 April 2017.

2. Goodman, Danny Morrison, Michael Novitski, Paul Rayl, Tia Gustaff Rayl, Tia Gustaff, JavaScript Bible. Indianapolis, IN: John Wiley & Sons, Incorporated; 2010.

3. Champeon S. JavaScript: How Did We Get Here? [online]. June 2001. URL: http://archive.oreilly.com/pub/a/javascript/2001/04/06/js_history.html. Accessed 16 April 2017.

4. Short History of JavaScript [online]. June 2012. URL: https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript. Accessed 16 April 2017.

5. White, Alexei. JavaScript Programmer's Reference. Indianapolis, IN: Wiley Publishing Inc; 2009.

6. Stefanov, Stoyan. Object-Oriented JavaScript. Burmingham, UK: Packt Publishing; 2008.

7. Ecma International, What is Ecma International [online]. URL: http://www.ecma-international.org/memento/index.html. Accessed 17 April 2017.

8. Industry leaders to advance standardization of Netscape's JavaScript at standards body meeting. Netscape to post JavaScript specification and licensing information on the Internet site [online]. November 1996. URL: http://web.archive.org/web/19981203070212/http://cgi.netscape.com/newsref/pr/newsrelease289.html. Accessed 17 April 2017.

9. ECMA. ECMAScript: A general purpose, cross-platform programming language [online]. June 1997. URL: http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf. Accessed 17 April 2017.

10. Eich B. ECMAScript Harmony [online]. August 2008. URL: https://mail.mozilla.org/pipermail/es-discuss/2008-August/006837.html. Accessed 17 April 2017.

11 Rauschmayer , Axel. Speaking JavaScript: An In-Depth Guide for Programmers. Sebastopol, CA: O'Reilley Media; 2014.

12. const [online]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const. Accessed 18 April 2017.

13. Motto T. ES6 arrow functions, syntax and lexical scoping [online]. URL: https://toddmotto.com/es6-arrow-functions-syntaxes-and-lexical-scoping. Accessed 18 April 2017.

14. Orendorff J. ES6 In Depth: Arrow functions [online]. URL: https://hacks.mozilla.org/2015/06/es6-in-depth-arrow-functions/. Accessed 18 April 2017.

15. Rauschmayer A. Exploring ES6: 16. Modules [online]. URL: http://exploringjs.com/es6/ch_modules.html. Accessed 18 April 2017.

16. Zakas N. Understanding ECMAScript 6: Encapsulating Code With Modules [online]. URL: https://leanpub.com/understandinges6/read#leanpub-auto-encapsulating-code-with-modules. Accessed 18 April 2017.

17. W3C, Web workers [online]. September 2015. URL: https://www.w3.org/TR/workers. Accessed 18 April 2017.

18. Maujood M. Prototypal Object-Oriented Programming using JavaScript [online]. April 2016. URL: https://alistapart.com/article/prototypal-object-oriented-programming-using-javascript. Accessed 18 April 2017.

19. Rauschmayer A. Exploring ES6: 15. Classes [online]. URL: http://exploringjs.com/es6/ch_classes.html. Accessed 18 April 2017.

20. static [online]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static. Accessed 18 April 2017.

21. Lindesay F. Generators [online]. URL: https://www.promisejs.org/generators/. Accessed 18 April 2017.

23. Rauschmayer A. Managing the private data of ES6 classes [online]. January 2016. URL: http://www.2ality.com/2016/01/private-data-classes.html. Accessed 18 April 2017.

24. Private State for ECMAScript Objects [online]. URL: https://github.com/wycats/javascript-private-state. Accessed 18 April 2017.

25. Concurrency model and Event Loop [online]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop. Accessed 18 April 2017.

26. Rauschmayer A. Exploring ES6: 24. Asynchronous programming (background) [online]. URL: http://exploringjs.com/es6/ch_async.html#ch_async. Accessed 18 April 2017.

27. Array.prototype.forEach() [online]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach. Accessed 18 April 2017.

28. Understand JavaScript Callback Functions and Use Them [online]. URL: http://javascriptissexy.com/understand-javascript-callback-functions-and-use-them/. Accessed 18 April 2017.

29. Zakas N. Understanding ECMAScript 6: Promises and Asynchronous Programming [online]. URL: https://leanpub.com/understandinges6/read#leanpub-auto-promises-and-asynchronous-programming. Accessed 18 April 2017.

30. Promise [online]. URL: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise. Accessed 18 April 2017.

33. Nelson J. Introduction to ES6 Promises – The Four Functions You Need To Avoid Callback Hell [online]. May 2015. URL: http://jamesknelson.com/grokking-es6-promises-the-four-functions-you-need-to-avoid-callback-hell/. Accessed 17 April 2017.

34. Elliott E. Master the JavaScript Interview: What is Functional Programming? [online]. January 2017. URL: https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0#.nh6w6psg7. Accessed 17 April 2017.

35. Decker K. Performance of ES6 features relative to the ES5 baseline operations per second [online]. URL: http://kpdecker.github.io/six-speed. Accessed 17 April 2017.

36. Hablich M. V8 JavaScript Engine: Retiring Octane [online]. April 2017. URL: https://v8project.blogspot.fi/. Accessed 18 April 2017.

37. ECMAScript [online]. URL: https://en.wikipedia.org/wiki/ECMAScript. Accessed 17 April 2017.

38.Tiobe. TIOBE Index for April 2017 [online]. URL: https://www.tiobe.com/tiobe-index/. Accessed 18 April 2017.