

Van Chau Dao

## **THE NATURE AND EVOLUTION OF JAVASCRIPT**

# THE NATURE AND EVOLUTION OF JAVASCRIPT

Van Chau Dao  
Thesis  
Autumn 2020  
Information Technology  
Oulu University of Applied Sciences

## ABSTRACT

Oulu University of Applied Sciences  
Degree Programme in Information Technology

---

Author: Van Chau Dao

Title of Bachelor's thesis: The Nature and Evolution of JavaScript

Supervisor: Kari Jyrkkä

Term and year of completion: 2020

Number of pages: 57

---

This thesis documented the essential natures of JavaScript as a programming language and reviewed the language's evolution, including the purpose of creation and historical events impacted on the language's characteristics. This thesis aimed to be a research paper by extracting the findings from multiple text-based sources and present them in two parts: the foundational definitions of JavaScript and its deposition in computer sciences and a timeline of changes in its usage, related progress, and setback from its conception point of the mid-1990s until the present day.

The materials that contribute to the thesis are internet-based articles providing accurate details about historical events and factual, technical information. The author then edits the materials to convey the thesis and the basis for the author's arguments of JavaScript's importance as a programming language. Many of the technical information came from the author's knowledge through education and working experiences.

The profile of JavaScript's technical characteristics is analyzed in a decrement at the abstract level. The history of JavaScript is featured according to its timeline. The result is a researched document with coherent evidence to support JavaScript's progress as a programming language. The conclusion is that JavaScript is a rich language and has the sustainability to surpass any limitations it faced to reach new technical horizons.

---

Keywords:

Browsers  
ECMAScript  
JavaScript  
JavaScript Engine  
Web Development

# TABLE OF CONTENTS

ABSTRACT .....	3
TABLE OF CONTENTS .....	4
VOCABULARY .....	7
<b>1 INTRODUCTION .....</b>	<b>8</b>
<b>2 WHAT IS JAVASCRIPT .....</b>	<b>9</b>
2.1 A high-level and interpreted language.....	9
2.2 Prototype-based and multi-paradigm .....	11
2.2.1 What is prototype-based? .....	11
2.2.1.1 Objects in JavaScript .....	12
2.2.1.2 Prototypal Inheritance .....	12
2.2.2 Different paradigms in JavaScript.....	14
2.2.2.1 Object-Oriented Programming .....	14
2.2.2.2 Functional programming .....	15
2.2.2.3 Event-driven programming.....	16
2.3 Single-threaded and non-blocking.....	16
<b>3 THE COMMENCEMENT OF JAVASCRIPT .....</b>	<b>18</b>
3.1 First launch.....	20
3.2 ECMAScript and the effort to standardize the language.....	21
3.2.1 ES1 & ES2 .....	21
3.2.2 ES3 .....	22
3.2.3 ES4 abandonment and the great halt.....	23
<b>4 JAVASCRIPT RENAISSANCE.....</b>	<b>25</b>
4.1 jQuery.....	25
4.1.1 Cross-browser compatibility.....	25
4.1.2 Separation from HTML.....	26
4.1.3 Short and clear syntax.....	27

4.1.4	Extensive extensions.....	28
4.2	Second generation browsers.....	29
4.2.1	Google Chrome's V8 engine .....	29
4.2.1.1	Abstract Syntax Tree (AST) .....	29
4.2.1.2	Just-In-Time (JIT) .....	30
4.2.2	The second generations of browsers .....	32
4.3	Node.js.....	32
4.3.1	The event loop.....	33
4.3.2	npm.....	35
<b>5</b>	<b>ECMASCRIPT COMEBACK &amp; UI LIBRARIES EXPLOSION .....</b>	<b>38</b>
5.1	ES5.....	38
5.1.1	Strict mode .....	38
5.1.2	Array and object methods.....	39
5.1.3	JSON support.....	40
5.2	Single Page Application.....	40
5.3	SPA UI libraries .....	42
5.3.1	2010: The beginning .....	42
5.3.2	React and many others.....	43
5.3.2.1	React.....	43
5.3.2.2	The other UI libraries .....	44
5.3.2.3	UI libraries' libraries .....	44
<b>6</b>	<b>JAVASCRIPT EVERY YEAR AND JAVASCRIPT EVERYWHERE.....</b>	<b>46</b>
6.1	ECMAScript updates .....	46
6.1.1	ES 2015 .....	46
6.1.1.1	New variable declare keywords: let and const .....	46
6.1.1.2	A new way of coding.....	48
6.1.1.3	Promise.....	51
6.1.2	Annual updates .....	51
6.2	Usage in non-web applications .....	52
6.2.1	Mobile development.....	52
6.2.2	Desktop.....	53

7	CONCLUSION.....	54
	REFERENCES .....	55

## VOCABULARY

CPU Central Processing Unit

CSS Cascading Style Sheets

DOM Document Object Model

ES ECMAScript

HTML Hypertext Markup Language

IE Internet Explorer

JS JavaScript

OOP Object-Oriented Programming

SPA Single-page Application

# 1 INTRODUCTION

When it comes to developing a website, JavaScript is an essential programming language that powered most of the modern web's view and interaction. While there are many choices to construct a server, most client-side interaction is implemented in this language, with 96.8% of the client-side programming done in JavaScript.[1] For this reason, along with many others, JavaScript has been ranked the most popular technology for eight years consecutively by Stack Overflow - an online coding exchange platform.[2]

JavaScript is a forerunner of front-end development and it became a robust, flexible tool that one might claim every type of developer can use. At the time of writing, it is entirely manageable to create any application purely with JavaScript. With just one language, one can create a server with a database and choose the front-end on multiple platforms such as web, mobile, or even desktop applications. It is even possible to create a machine learning application using JavaScript.

From those statistics, it is understandable that JavaScript has a tremendous influence on technology, and the language itself remained at the heart of the field's innovation throughout the years. Despite its humble beginning, twenty-five years ago, as a prototype scripting language, it has grown to be a giant that powers many other new technology fields of the Internet. It has also transformed into a practical tool employed outside its original purpose, branching out to different programming aspects.

For these reasons, it is worth reviewing the creation timeline of JavaScript and its characteristics to explain its phenomenon of success in the technology industry. JavaScript is undoubtedly the face of the Internet's development. It is possible that by understanding its origin story as well as its advancements after each period of time, one can gain a deeper comprehension of the landscape of the Internet's development from its debut to ultimately make a forecast of its future.



## 2 WHAT IS JAVASCRIPT

A web page, at its core, consists of Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript components. No matter how complicated and complex and architect the applied technologies are, everything will all be translated into HTML, CSS, and JavaScript blocks after the compilation phase. HTML is the blueprint of the web page giving it the structure and content, CSS to create the look of the components. And most importantly, JavaScript is a tool to manipulate all of the webpage components.[3]

The most frequently found definition of JavaScript as a programming language is that JavaScript is a high-level, multi-paradigm, non-blocking, asynchronous language. Those particular words and others, such as garbage-collected, interpreted, single-threaded, or concurrent, definitely summed up JavaScript's characteristics. However, they are often too abstract, and for the average reader, or someone new to programming, it can be hard to have a full perspective of the language.

Therefore, this chapter aims to analyze the main characteristics of JavaScript represented by the vocabulary mentioned.

### 2.1 A high-level and interpreted language

Looking at JavaScript from the utmost abstraction, it is usually regarded as a high-level programming language because of its minimal interaction with the operating system or the hardware. High-level programming languages such as JavaScript are the least complicated programming languages as they use abstraction, e.g. a garbage-collector or dynamic typing to simplify the programmer's programming concept. In contrast to this, the machine code languages use the binary expression that can be executed directly by the computer's central processing unit (CPU). When building an application in this language, a programmer does not need to manage memory or processor information and does not have to be concerned about concepts, such as pointers.[4]

The fact had posed some limitations to JavaScript. For example, it cannot read/write and copy arbitrary files on the machine's hard disk or execute any program. There is also no direct interaction with the computer's hardware, i.e. microphone or camera.[5] These limitations exist due to the usage of JavaScript on the client-side and its dependent nature on its environment it operates in. The browser plays the middleman's role between JavaScript and the machine, redirecting the user's request for permission to make use of the machine's equipment.[5] Due to JavaScript's original purpose, these limitations were imposed, creating a dynamic interaction with the user for very static web pages in the 90s. Nevertheless, as JavaScript progressed beyond the boundary of a browser, with the development of a runtime environment that executed JavaScript code in the server and overcame the obstacles stated above.

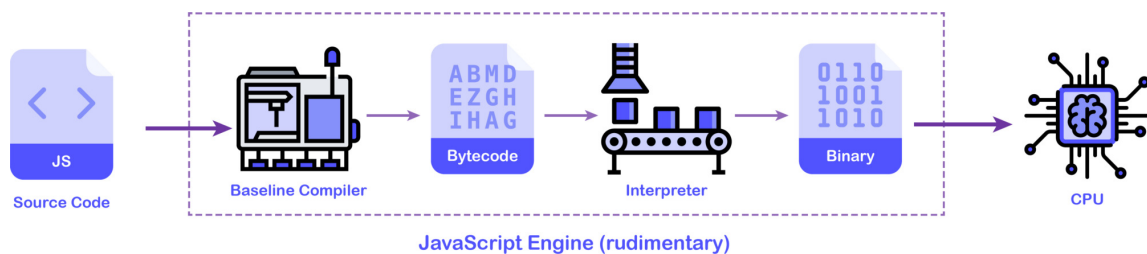


Figure 1. The Netscape/Spider Monkey JavaScript engine.[6]

JavaScript is also an interpreted language. The source code is interpreted into bytecode and executed at runtime, unlike in compiled languages, compiled during the build time. As seen in Figure 1., the Netscape browser converts JS with Spider Monkey JavaScript engine in the early days. In this fundamental process, the JavaScript engine has a baseline compiler to compile the source code into bytecode – an intermediate representation. From there, the interpreter would churn this bytecode into binary machine code that can be executed by the CPU.[6]

The problem with the rudimentary JavaScript is that it would parse all the code and hence, slow down the performance significantly when there are many codes to convert. As JavaScript and its application grew in complexity, so does the JavaScript engine that processes the source code. In chapter 4, the more JavaScript engine will be examined to highlight how the language has scaled up to accommodate its heavy usage in the future. Since each browser has a different JavaScript engine, one can deduce the JavaScript industry standard and the current practice trend. One can also cast a future prediction by having a peer review of these engines.

## 2.2 Prototype-based and multi-paradigm

When debating about the practical aspects of JavaScript, many refer to the language as a multi-paradigm. JavaScript does not restrict how a programmer should approach a problem when writing a JS program.

It is because of JS's unique prototype-based feature that gives the language the freedom to be non-restricting on the coding paradigm. This particular quality has allowed any developer from any programming background to migrate to systems using JavaScript with their current coding knowledge. JavaScript is so accommodating that Jeff Atwood, the founder of Stack Overflow, made a comment about it, now known as the Atwood's law:

Any application that can be written in JavaScript will eventually be written in JavaScript.

### 2.2.1 What is prototype-based?

Being prototype-based is the unique characteristic of JavaScript about the object inheritance behavior. The object in JavaScript is the very foundation of any JavaScript application.



```
JavaScript.js UNREGISTERED
JavaScript.js x
1
2 // Example of an object
3
4 const object = {
5   key: "value"
6
7   method() {
8     console.log("This is an object method.")
9   }
10 }
```

Line 4, Column 17 Tab Size: 4 JavaScript

Figure 2. An example of an object in JavaScript.

### 2.2.1.1 Objects in JavaScript

There are eight data types in JavaScript. There are numbers, string and large integer or BigInt, logical Boolean, null, undefined, object, and symbol used to create unique identifiers.[7] Out of the listed group, the object is an exceptional stand out as it is the only data type in JavaScript that has more than one single value, making it *non-primitive*.

In Figure 2., a structure of an object in JavaScript consists of properties, each property being a pair of a type string key and a value of any data type. An object can also have a method, which is a function that can be executed elsewhere during the program. Everything that does not belong to the primitive data types group above is an object, including function, array, JavaScript Object Notation (JSON) data.

### 2.2.1.2 Prototypal Inheritance

A screenshot of a code editor window titled 'JavaScript.js' with a 'UNREGISTERED' label in the top right corner. The code is as follows:

```
14
15 // Animal
16 function animal(name, food) {
17     this.name = name,
18     this.food = food,
19
20     this.eats = function eats() {
21         console.log(name + ' eats ' + food);
22     }
23 }
24
25 // Monkey
26 const Monkey = new animal('Monkey', 'Banana');
27
28 Monkey.eats(); // Monkey eats Banana
29
```

The editor shows line numbers on the left, and the status bar at the bottom indicates 'Line 20, Column 34', 'Tab Size: 4', and 'JavaScript'.

Figure 3. Creating the Monkey object by animal function constructor.

It is a fact that code recycling is an essential part of programming, especially in a large scale project. A programmer would often have an original block of code outlining its purpose, methods, and features, e.g. a class in object-oriented programming (OOP). They then copy it in its origin or apply extension logic to modify it according to their needs.

In JavaScript, all the objects have a hidden `[[Prototype]]` property pointing to another object referencing from or a `null` value.[8] Figure 3. described the `animal` function as a constructor taking in the name and the food and `eats` that would output to the browser console. It should be noted that this is one of many methods to instantiate an object.

In this example, by creating a new `Monkey` object inline 26, it inherits the method `eats` and, when executed, produced a text 'Monkey eats Banana '. This relationship is prototypal inheritance. Hence, the `animal` function is the prototype object of the `Monkey` object, evidenced by Figure 4.

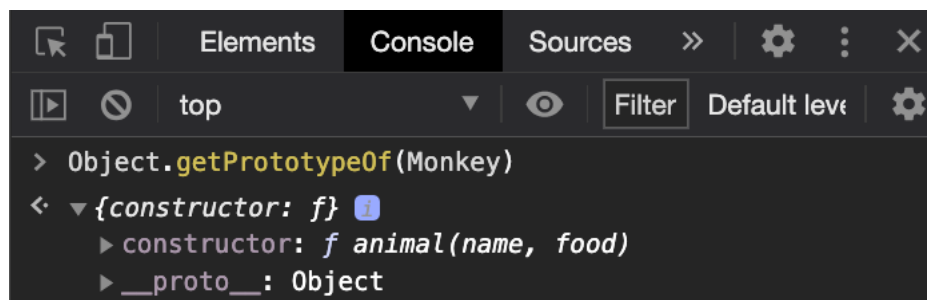


Figure 4. Prototype relationship demonstration.

By instantiating a new object on an existing object, as demonstrated in the example in Figure 3., an inheritance relationship is set automatically. However, an object can also be assigned of prototype object by `__proto__` getter/setter, or modern `Object.getPrototypeOf/Object.setPrototypeOf` functions.

This prototype inheritance can be chaining; object A can have a prototype object B, object B, in turn, has a prototype object C. When a property of an object is in demand and does not exist, JS will try to shift through the prototype objects to locate the property and only returns `undefined` when the search finds nothing.[8]

## 2.2.2 Different paradigms in JavaScript

As the title has characterized, multiple programming paradigms can be used in JavaScript. These are JavaScript's most fashionable programming schemes at the time of writing. The following sections specify each paradigm approach, uses, and relates to JavaScript.

### 2.2.2.1 Object-Oriented Programming

Thanks to the prototypal inheritance, programmers from the OOP background can effortlessly adopt JavaScript for their projects. However, despite the similarity of 'inheritance', JS's prototypal inheritance still has a few critical differences from the class inheritance of a proper OOP language.

The first difference is in the concept of the original code block itself. A class in OOP is not a direct object; it represents a blueprint of an object. It provides the base properties and the methods that an object, when instantiating from it, would have. A class is an abstraction of an object.

Therefore, when a class inherits from another class, the class itself becomes an abstraction of an object.[9] Class inheritance is truly an immutable inheritance. An object cannot modify any methods at runtime. The many levels of inheritance in OOP are many layers of abstractions of the object. This tightly coupled subclass relationship is costly in the development, particularly in the debugging process.

In contrast, every complex structure element in JavaScript, despite its declaration notation being a function, is merely an object. An object can have multiple prototype objects, and those objects can be chaining, as explained in the previous section. Those prototype reference layers will just be purely objects. Compared to this, prototypal inheritance is more flexible for its nature as a prototype object. Without the heavy abstraction concept of a class, the programmer can customize their implementation of the 'class' regarding mutability and the object's complex structure.

### 2.2.2.2 Functional programming

Functional programming is a paradigm that belongs to the declarative category. It has not been as popular as OOP in the early days but it has since caught on with JavaScript's prominent usage. Especially, since the debut of JS User Interface (UI) libraries such as React or Angular. It is a design based on mathematical functions, applying these so-called **pure functions** solely based on the input and producing the same output whenever given the same input.

Functional programming promotes the concept of modularity, generating small blocks of pure functions to comprise the more prominent application.[10] A hierarchy of a functional programming system is exhibited in Figure 5. This hierarchy can be found in many UI libraries where they construct the application using mainly functions to build the HTML elements.

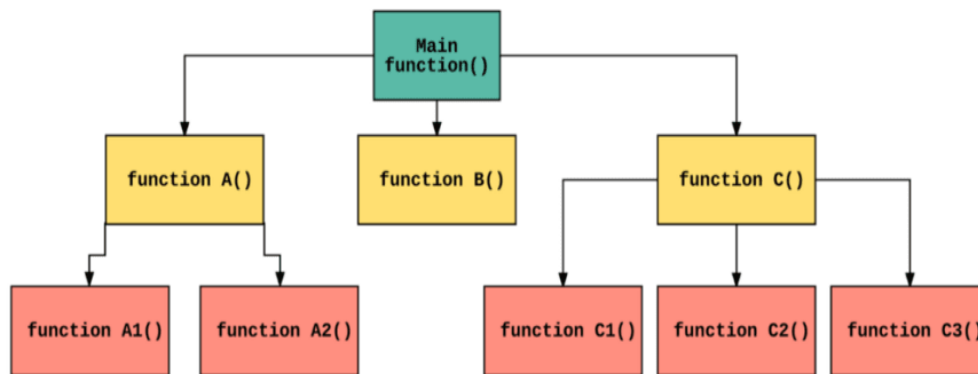


Figure 5. The functional Flowchart.[10]

Compared with OOP, which has a more rigid structure, creating a tightly coupled system through abstraction inheritance, functional programming is quite the opposite with a more directive approach. Its purpose is to answer to the question what the data is and transformed it independently. A functional programming system is less coupled and produces fewer side effects by not having a shared state between each component. It also accepts a function as an input; the concept is referred to as a **higher-order function**, whereas only data or objects can be passed through in OOP.

### 2.2.2.3 Event-driven programming

The flow of event-driven programming is essentially based on user interaction. It is different from the two previous programming paradigms. It always listens to the user's events, namely `onClick` or `onChange`, as a trigger to the performed intended action. The jQuery library popularizes it.

The HTML element will be tagged with JavaScript events with a task function stand by. Once the user commits the event, the callback function is executed. Event-driven programming is a user-centered approach and often uses for web pages with existing HTML code based.

### 2.3 Single-threaded and non-blocking

JavaScript is a single-threaded runtime language. It can only process one task at a time. This main execution thread supervises all the events when the website is opened on the browser.

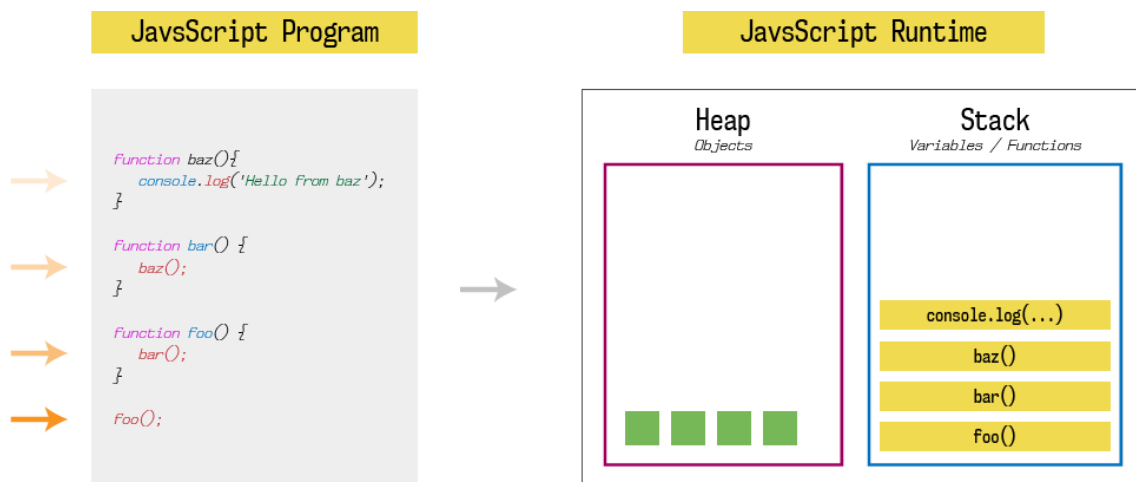


Figure 6. The JavaScript Runtime Environment.[6]

The runtime process is exemplified in Figure 6.; the functions are queueing up in the JavaScript Runtime Call Stack. It is a last-in-first-out data storage.[6] When the `foo` function is being called, it appears in the stack and then continues to run `bar`. `bar` would then trigger `baz` that finally has a console logging task.



Until a function returns a value, it will not be considered *completed* and it remains in the stack by the code's order. In each stack frame, the stack will conduct one single action, taking an entry of the signal's function in the codebase or processing the return.[6] When the console finishes logging out the text, it is marked as completed as the returned value, i.e. it has been consumed. The `console.log(...)` task would disappear from the stack, consequently ending the `baz`, `bar`, and `foo` functions.

When the JavaScript Runtime encountered a never-ending task, notably a `while (true) {}` loop, or a time consuming one such as fetching the server's data, it would block all the other remaining tasks – a clear example of blocking code. As modern JavaScript is often referred to as non-blocking or asynchronous, there is a way to combat these recursive functions call blocking the programming and using up all the system's resources.

To simplify, for the function that requires more time, the later version of JS would create another thread for that function to wait out until it is completed with a returned value. Afterward, the thread will be closed, and it will collide into the main thread. Any other function required to run when the task is completed will be pushed to the main thread. This concurrent process will be further clarified in chapter 4 with the introduction of the event loop.

### 3 THE COMMENCEMENT OF JAVASCRIPT

The history of JavaScript began shortly after the establishment of the Internet. Despite the invention of the first browser – the WorldWideWeb – in 1990 by Tim Berners-Lee, the Internet remained an unfamiliar concept to the mainstream.[11]

However, in 1993, when the commercial Mosaic browser was released, the Internet began to flourish. Marc Andreessen and Eric Bina created the Mosaic browser while working at the National Center for Supercomputing Applications. It was first released in January 1993 for the UNIX system and later in September 1993 for the Macintosh and Windows system. It featured a graphical interface that allowed the image to be displayed 'inline' with the document text. It also introduced the concept of the Document Object Model (DOM) structure in the browser.[11]

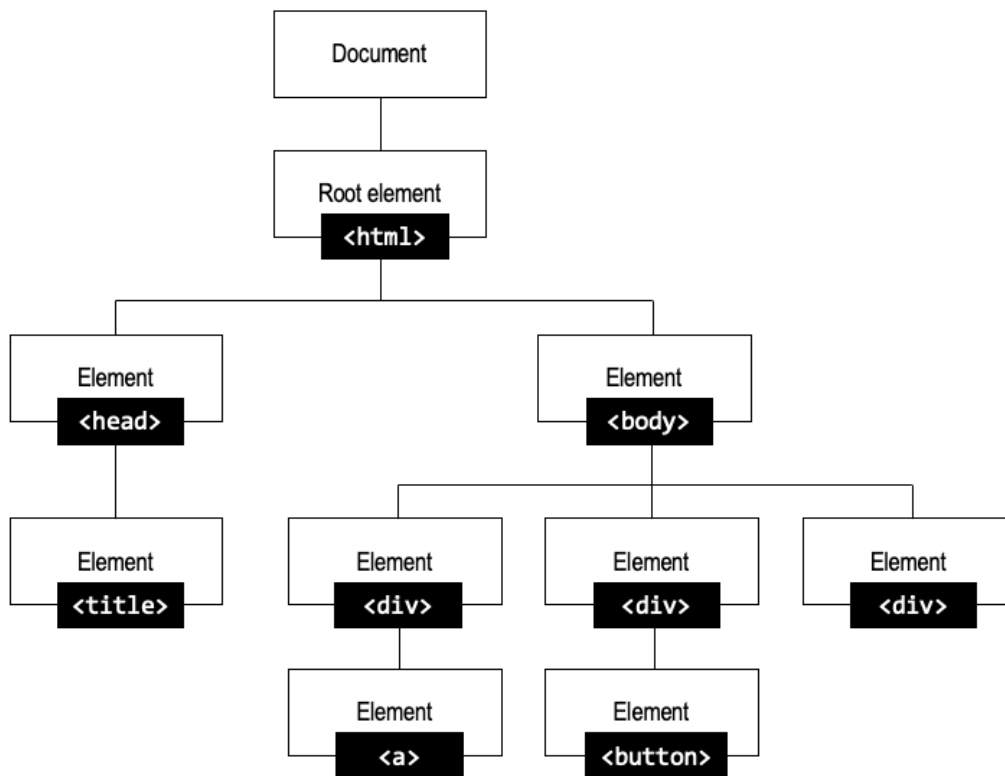


Figure 7. A representation of the HTML Document Object Model hierarchy.

When a website is loaded, the browser created the DOM tree using the HTML file code as the blueprint. In Figure 7., the webpage is the 'Document', which has the root element represented by the <html> tag. The <html></html> tags enclosed all the other elements that marked up the webpage. In this example, the webpage hierarchy showcased how the element's position made up the DOM tree.



*Figure 8. The Mosaic Browser for the Macintosh system.[11]*

At the time, Mosaic was very well-known and praised for bridging the gap between internet multimedia and non-technical users. (Figure 8.) Nevertheless, as Marc Andreessen established Netscape Communications Corporation, the Mosaic browser quickly lost its market share to the new innovative Netscape Navigator browser in 1995.[12] Netscape introduced various

monumental concepts in its product, such as rendering as the page is loaded, cookies, animation frames, proxy auto-config, and the significant JavaScript.

### 3.1 First launch

Until this point, a standard website was only a static product. A non-technical user was able to view the site with ease but was completely unable to interact with it. It was clear that the next evolutionary step in the field was to deliver dynamic content to the end-user.

To fulfill this request, Netscape naturally gravitated towards the popular programming language at the time – Java. In the beginning, the plan was that Java would be the leading 'professional' choice for a more complex embedded web system and a scripting language whose codes would be placed inside the HTML document for the casual scripters and amateurs as a 'companion' language.[13]

This approach soon proved to be fruitless for the following reasons. When considering the user experience, it required the user to have a Java Virtual Machine, handle Java's security prompt, and boot the corresponding Java program when visiting a particular website.[13] When considering the development process, if Netscape were to develop their own Java Virtual Machine, it would “never achieve perfect bug-for-bug compatibility” with Java's original virtual machine by their company Sun Microsystems.[14]

Netscape called off cooperating with Sun Microsystems and famously employed Brendan Eich to complete the latter goal: creating a glue-like scripting language to enhance the webpage experience on an extreme deadline of ten days. This time restraint led Eich to create the multi-paradigm, flexible programming language to let the developers, including the non-engineers, apply their programming pattern. Syntactically, Mocha was a curly bracket language similar to Java at the request of Netscape's executives. However, Mocha has many characteristics of a modern-day JS inspired by different programming languages. It has first-class functions like Scheme, dynamic typing like Lisp, and prototype inheritance like Self. JavaScript made a tremendous impact on the web browsing experience from the beginning. The websites became

strikingly lively with animation enhancements, and the most notable change lasting until this day: popup advertisements.

The language and its interpreter soon changed the name to Live in September 1995 upon the launch of Netscape 2.0 but changed it to JavaScript three months later.[15] Brendan Eich continued working for Netscape following the release of JavaScript, overseeing the implementation of SpiderMonkey. Eich reportedly refactored most of Mocha's codebase to settle the technical debt left behind when rushing Mocha to release a year ago, including the JavaScript engine renamed SpiderMonkey.[14] (Figure 1.)

In 1995, an increasingly popular Microsoft launched its browser program named Internet Explorer (IE). To compete with Netscape Navigator on JavaScript, Microsoft reverse-engineered JavaScript into Jscript and premiered in August 1996.[15]

### **3.2 ECMAScript and the effort to standardize the language**

The Internet started to proliferate, going from just 3,000 websites to 258,000 by 1996.[16] With two similar browser scripting languages JavaScript and JScript, there was a clear need to standardize JavaScript. In November 1996, Netscape submitted JavaScript to the European Computer Manufacturers Association (ECMA) – a neutral organization founded in 1961 for the sole purpose of standardizing the computer systems. [17, 18]

#### **3.2.1 ES1 & ES2**

By June 1997, the Standard ECMA-262 or the first edition of ECMAScript's specified document (ES1) was introduced. It established the programming language itself as ECMAScript, the standardized name of JavaScript.[18] This document provided the browser vendors and server-side applications a consistent set of specifications, guidelines on the implementation of JavaScript. Both JavaScript and Jscript were guaranteed to follow the principles set forth by the ES1 with added features.

In June 1998, the second version ES2 was published with editorial changes to align with the ISO/IEC 16262 international standard specification.

### 3.2.2 ES3

The first version of JavaScript was undoubtedly an awe-inspiring invention for its time in its core value and its applications. It was a magnificent achievement, considering the limitation in labor and time resources. However, the infamous ten-day creation process unavoidably led to errors and shortcomings in the language.

The ECMAScript version 3 (ES3), released in December 1999, addressed these problems. It featured better error handlings, including a tighter definition of errors.[18] It also introduced new control statements, such as `if` statement, `do-while`, and `try/catch` blocks, and covered the definitions of functions and data types, numeric output formatting.[19]



```
32
33 // Equal operator
34
35 90 == '90' // True
36
37
38 // Strict operator
39
40 90 === '90' // False
41
42
```

Figure 9. The difference between equality operators in JavaScript.

Moreover, especially, it presented the regular expressions, with the much-needed strict equality operator. Some of the early JavaScript testers proposed the first equal operator in the first version

of JavaScript (demonstrated by line 35 in Figure 9.) based on convenience if a number were to be directly equal to a string. In the beginning, JavaScript was destined to be accessible to a non-programmer audience, and therefore, it implemented this lenient abstract equality proposal. The new ES3 offered the strict equality operator that would compare based on value and type, as seen in line 40 in Figure 9., making the system more coherent and versatile.

### **3.2.3 ES4 abandonment and the great halt**

With the release of ES3, JavaScript was on its way to advancement, and the work for ES4 started in February 1999. This newer version, alongside fixing the occurring bugs from ES3, also aimed to better support programming-in-the-large.[18] The proposing features listed to attain this ambition grew immensely through the years, including but not limited to: classes and interface implementation, module system, optional types annotation, destructuring assignment, constant bindings, array comprehensions.[18]

ES4 would have brought significant changes to JavaScript compared to ES3. Its proposal was growing in scale to serve its ambition of being an enterprise-level programming language. The features bear a resemblance to the modern-day TypeScript, and the changes would have left a mark on JavaScript. Unfortunately, such greatness did not come to fruition because of the political differences in the field.

Despite the Netscape Navigator standing position in the browser marketplace in the mid-1990s, it lost its dominance to Microsoft by the end of the decade. During the first 'Browser Wars', IE had claimed the superior position by 2002, taking almost 90% of the overall market share. (Figure 10.)

After AOL acquired Netscape in November 1998, Netscape Navigator released a new version in Standard and Gold Edition with the latter version packed full of e-mail, newsreaders functionalities. These functionalities resulted in costing the browser its performance with constant crashing and incomplete render.[12] The Netscape browser also lost its edge by not supporting CSS – a feature developed by Microsoft Internet Explorer that soon became a new foundational pillar of a website. Microsoft had Internet Explorer built into every Windows and MacOS computer as a standard browser due to a deal worth 150 million dollars with Apple in August 1997.[12]

## Browser Wars

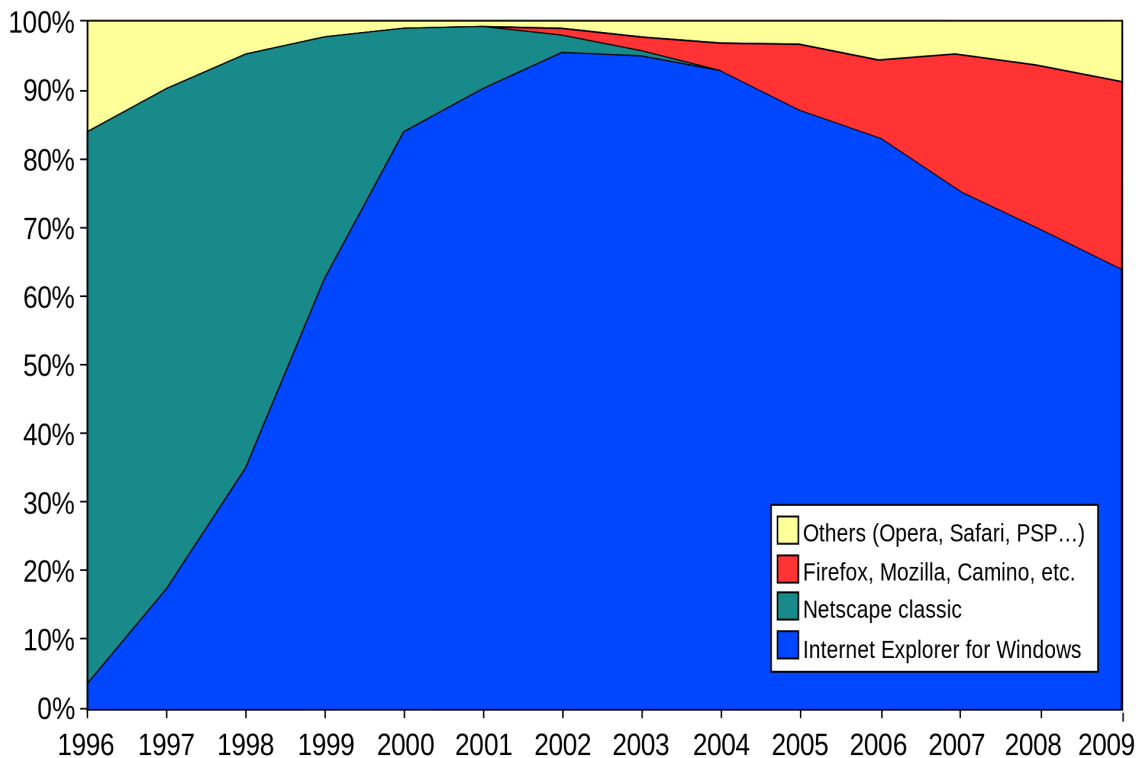


Figure 10. The market share for several browsers between 1995 and 2010.[20]

The combination of low performances and Microsoft's tremendous influence terminated Netscape Navigator by the early 2000s. (Figure 10.) With the newfound victory of the first 'Browser Wars', Microsoft pushed Jscript forward to its separate agenda by creating its unique JavaScript extensions. One notable extension was a function that allows a browser to perform an asynchronous HTTP request called XMLHttpRequest. This technique, later known as AJAX, created the opportunity to have content updated on-the-fly.[14]

Upon reviewing the proposal of ES4, Microsoft concluded that the plan was too complicated and withdrew from the development process.[18] This discontinued collaboration created multiple disputes between Brendan Eich and the creative force at IE and it was carried on until 2008, when all effort to construct ES4 ended for good. Part of the ES4 proposal, mainly the fixes and JSON support, was made into the latter ECMAScript version 5.



## 4 JAVASCRIPT RENAISSANCE

Following the previously mentioned events, JavaScript's innovation came to a halt when Netscape lost its success to Microsoft Internet Explorer and because of the dispute between the ES4 contributing parties. This dark age slowly faded as breakthroughs in the industry emerged. This chapter will review the three most historic achievements that took JavaScript to a new renaissance horizon.

The first is the jQuery library, the earliest JavaScript library that unified and empowered the web applications by enhancing their architecture complexity, interaction, and user accessibility with minimal development efforts. The second is the next generation of browsers whose JavaScript engine optimization speeded up the JavaScript's performances by a landslide. Finally, the chapter will study Node.js and how it transcended JavaScript into a language that can exist beyond the front-end frontier.

### 4.1 jQuery

jQuery is a lightweight JavaScript library which was created in 2006 by John Resig. It has been the most used library since its release. *As of February 2020, jQuery is used by 74.4% of the top 10 million websites.*[21] jQuery rose to the top of the stardom for serving the fundamental need of 'write less, do more' in JS applications. jQuery's core values are all aimed to smoothen the process of web development to deliver a more dynamic product with less time and coding effort.

#### 4.1.1 Cross-browser compatibility

After Mosaic's success, many different web browsing products entered the market though they only managed to have an insignificant portion of the market share. The second generation of web browsers started with the remanence of the first 'Browser Wars'. When Netscape lost its part to Microsoft and the funding for its commercial activities, it turned the product into an open-source version named Mozilla in 2002. Mozilla managed to produce a stable internet and a spinoff

project which eventually became the Mozilla Firefox browser, in the same year. [22] Along this time, Apple ended the deal with Microsoft and launched its web browser product Safari. These browsers and Opera – a popular browser for handheld devices, had the market share fight as IE peaked at more than 92% in 2004. The widespread use of these different browsers created the compatibility issue for the developers. Since each browser had its version of a JavaScript engine, the same script code would deliver inconsistent results across different browsers.

jQuery offered an interface that guarantees to eliminate the particular problem.[22] Using jQuery, developers do not need to research the differences and rewrite the JS code multiple times.

#### 4.1.2 Separation from HTML

A screenshot of a code editor window titled 'index.html' with 'UNREGISTERED' in the top right corner. The editor shows a single document containing both HTML and JavaScript code. The code is as follows:

```
1 <!DOCTYPE html>
2 <html>
3   <script type="text/javascript">
4     function sayHi() {
5       alert('Hello!')
6     }
7   </script>
8
9   <button onclick="sayHi()">
10    Click
11  </button>
12 </html>
```

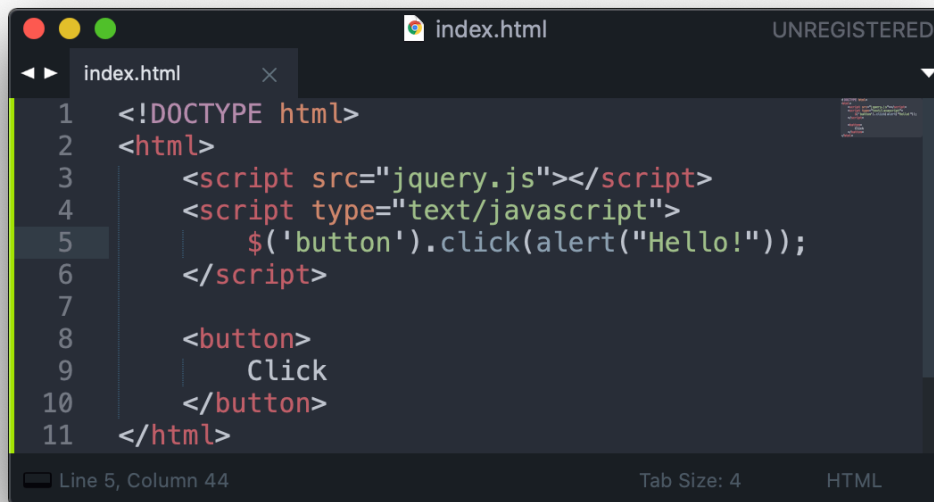
The editor interface includes a tab labeled 'index.html', a line and column indicator at the bottom left ('Line 1, Column 16'), and a tab size indicator at the bottom right ('Tab Size: 4').

Figure 11. An example of a traditional approach using JavaScript.

In Figure 11., a traditional HTML code block with a script element co-exist in the same document. For an alert to trigger when clicking the button element, the button tag must declare the `onClick` event handler and the function `sayHi` in line number 9.

jQuery promotes the separation of HTML by providing the syntax to add the event handlers to the DOM element by using JavaScript. Figure 12. presents an example of an application using jQuery.

First, the jQuery code was added in line 3. In the script, jQuery identifies the element `button` and triggers the `alert` when the element is clicked. This separation often resulted in better readability and over code hygiene.



```
1 <!DOCTYPE html>
2 <html>
3   <script src="jquery.js"></script>
4   <script type="text/javascript">
5     $('button').click(alert("Hello!"));
6   </script>
7
8   <button>
9     Click
10  </button>
11 </html>
```

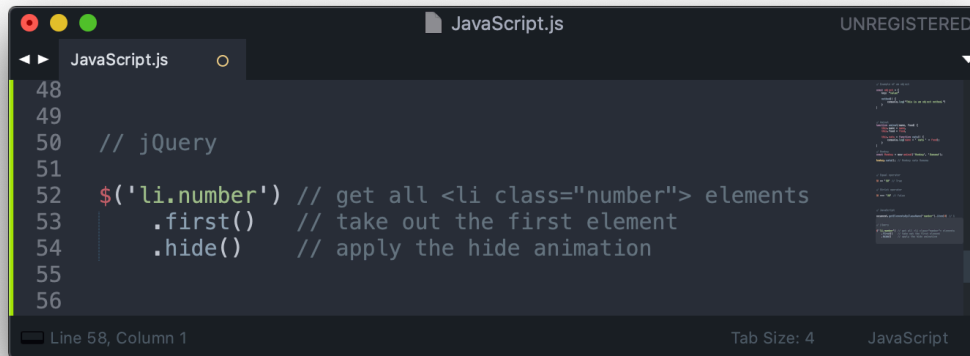
Figure 12. An example of jQuery with a click function.

### 4.1.3 Short and clear syntax

At its core, jQuery is a DOM manipulate library which provides normal or static utility functions and jQuery object methods to a selected element. jQuery's primary identifier received the CSS selector string argument with the `$` alias and reference through the DOM to find all the matching elements. For example, upon calling `$("li.number").hide()`, jQuery would locate all the list `<li>` elements with the class name of `number` and apply the hide animation effect.

In comparison to vanilla JavaScript's `document.getElementById/document.getElementsByTagName` methods, the selector alias is much shorter and more effective at spotting the HTML element.

The jQuery selector's return value is always a jQuery object able to chain different methods. Figure 13. analyzes the process of the method chain.

A screenshot of a code editor window titled 'JavaScript.js' with a 'UNREGISTERED' watermark in the top right. The editor shows a JavaScript code snippet with line numbers 48 to 56. The code is as follows:

```
48  
49  
50 // jQuery  
51  
52 $('li.number') // get all <li class="number"> elements  
53   .first()     // take out the first element  
54   .hide()     // apply the hide animation  
55  
56
```

The status bar at the bottom indicates 'Line 58, Column 1', 'Tab Size: 4', and 'JavaScript'.

Figure 13. An example of jQuery method chains.

#### 4.1.4 Extensive extensions

Over the years, jQuery has built up a thorough supported method list. It has a variety of methods to change visual aspects, such as animation effects and direct CSS manipulation. It has robust methods for completing DOM structure manipulation, such as element insertion, removal, or replacement, and traversing the DOM tree to find other elements. jQuery also has events and form handling properties and the well-implemented AJAX request handler for asynchronous programming. Some of these methods were so outstanding that many of today's browser vendors standardized them into their products.

There is also an abundance of plugins libraries to support jQuery further. An average website developer can choose the plugin – a ready-made method – that best serves their purpose instead of building the feature from scratch with JavaScript code.

On these four principles, jQuery was revolutionary to front-end development across different browsers. It was comfortable to learn and even more comfortable to use with the meticulous documentation. It has a substantial built-in set of methods that speed up the progress significantly. Furthermore, the quick adoption from the community, in turn, supplies new extensions that enrich its ecosystems.

## 4.2 Second generation browsers

The invention of jQuery has led to an increase of JavaScript usage both at number and complexity levels. While jQuery is not hard to use, it is hard to parse to the machine code level, particularly with its famous DOM elements selector. Facing this problem is the second generation of browsers in the late 2000s.

These browsers came with the improved JavaScript engine that provided a better loading and overall processing experience on the client-side. The transformation in these JavaScript engines, especially the V8 engine from Google's newly released Chrome browser, drastically accelerated JavaScript performances and ushered into a new Renaissance era.

### 4.2.1 Google Chrome's V8 engine

In 2005, Google launched Google Maps, an online global mapping service. The nature of the product required the platform to have the power to handle the heavy interaction from the users. It was undoubtedly a complicated and grand coding construction project. However, when it was brought into use, Google quickly realized that the limited browser hindered Google Maps' performances. In particular, the traditional method of the JavaScript Engine used at the time to parse the code was not efficient enough to handle such an enormous site as Google Maps.[6]

In 2008, Google presented its browser, Google Chrome, with the brand new V8 engine to solve this problem. The engine's main improvement laid at the core of its working principle: delivering the code promptly and heavily optimizing the frequently executed code.

#### 4.2.1.1 Abstract Syntax Tree (AST)

The engine first parsed the source code into an Abstract Syntax Tree (AST) – a tree branch representation. Figure 14. describes an example of AST representation. The code can be a

function call, a class representation or a code module. The branches split to the most detailed representation of the code line, such as a declaration and an assignment statement.

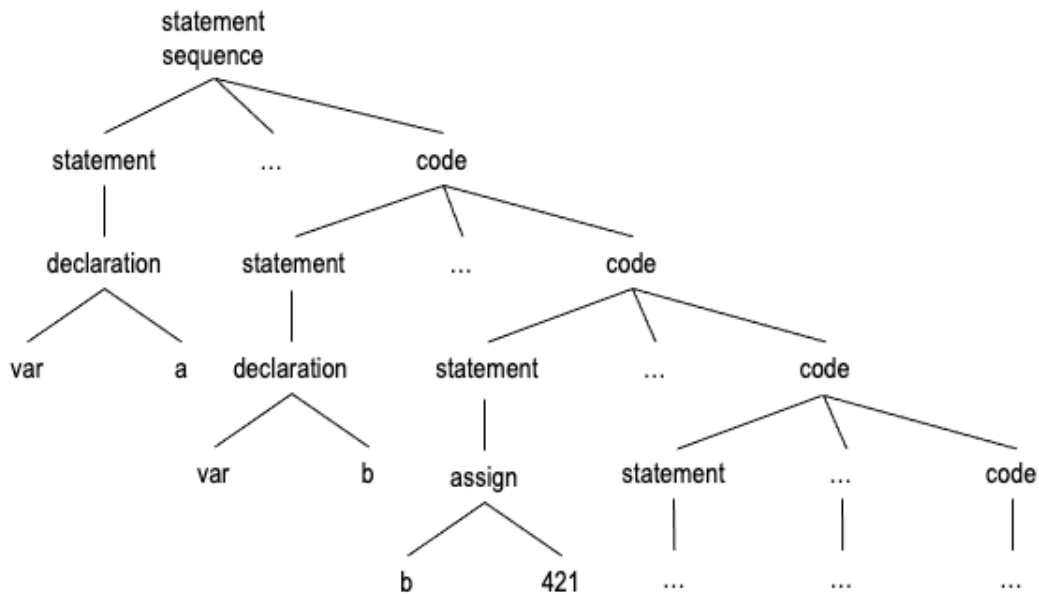


Figure 14. An abstract Syntax Tree Example.

By having the source code mapped out in the AST model, the system can identify the priority of compilation of different code blocks, contributing to an overall faster application bootstrap.[6]

#### 4.2.1.2 Just-In-Time (JIT)

Essentially, V8 used the Just-In-Time compilation process to boost up the performance. First, the base compiler would kickstart the pipeline by quickly generating the non-optimized machine code from the JavaScript source code. The V8's base compiler is well run and produces 30% less code than the last-generation browser's base compiler.[23] When the application has started, the **runtime compiler** will surveil the system to pinpoint 'hot code'. This code spent a significant amount of time running either complicated or frequently used code. These profiled hot codes will be recompiled and optimized by the optimization compiler.[23] A visualization of this JIT process can be found in Figure 15.

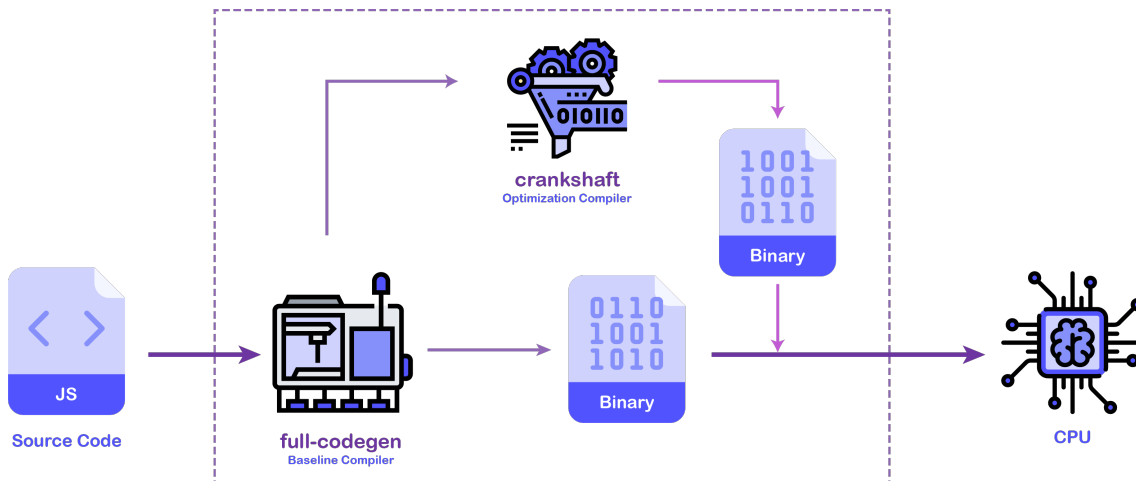


Figure 15. The V8 JavaScript Engine Pipeline.[6]

This JIT compilation approach replaced the interpreter found in the earlier version of the JavaScript engine. JIT happened during the runtime, in Figure 15., when it started simultaneously at the time of the initial baseline compiler. JIT allows the runtime compiler and optimization compiler to access the dynamic runtime information to make better optimizations, such as inline function—positioning the compiled function in the main body (Figure 16.) and casting types for variables based on declared values during compilation.

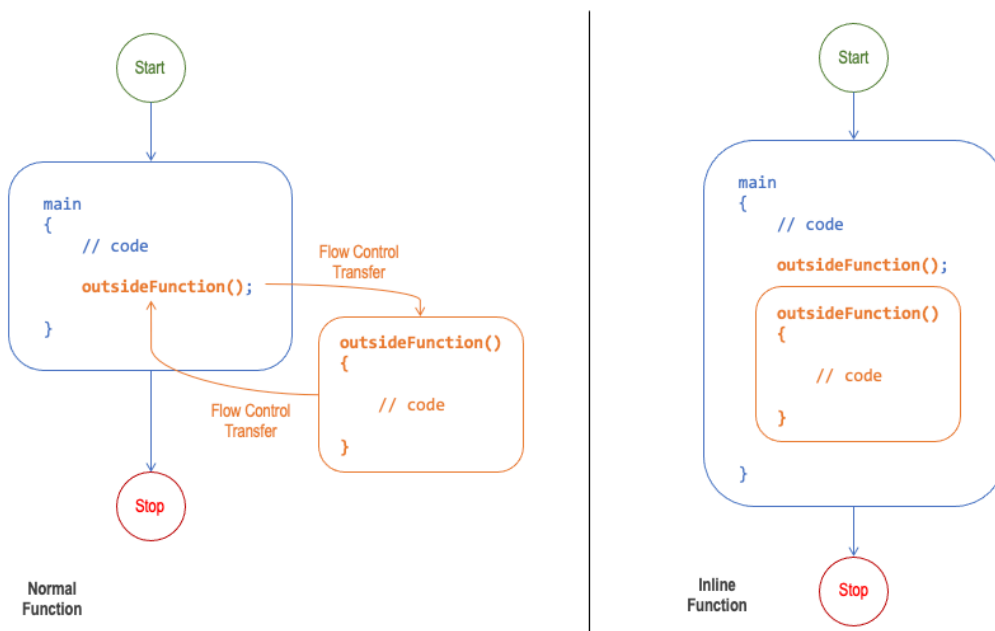


Figure 16. A comparison between the normal function and the inline function.

The main disadvantage of this process is its operational cost of CPU and memory resources because the optimization and surveillance occur at runtime. Furthermore, the Crankshaft would often result in slower initial rendering, and often the Full-code generator (full-codegen) would not meet up the compiling expectation in larger applications. Moreover, despite its impressive initial architect, Crankshaft was not sustainable since it was only created to optimize a subset of the JS.[24]

#### 4.2.2 The second generations of browsers

V8's cutting-edge approaches powered the Google Chrome browser to rise among its competitors. However, the impact it had made spread beyond its product. It had inspired the advancement of using the Just-In-Time compilation, the Abstract Syntax Tree, and a different compilation to machine code models by other browser vendors.

Netscape's protégé Mozilla also made advancements with the traditional SpiderMonkey JS engine incorporating the JIT compiler into its pipeline to compile the source code to the native code. The Chakra engine of Internet Explorer, Safari's JavaScriptCore engine, and many others followed the initiative to adopt the same optimization model of V8, producing the strong second generation of browsers with a noticeable productivity improvement. The performance progress across the browsers led to Microsoft losing its dominance in the early 2010s, dropping below the 50% mark for the first time in October 2010 with 49.87% of the market share. [20]

The V8 engine is also the inspiration and the foundation of Node.js – the invention that brought JavaScript to a new frontier.

#### 4.3 Node.js

When Netscape was launched in the mid-1990s, it also formulated a server-side JavaScript service called Netscape's LiveWire Pro Web. Ryan Dahl took the lead of the project's development and maintenance team.[25] Even though LiveWire did not meet any success, Ryan Dahl continued pursuing server-side scripting and created the runtime environment **Node.js**. Most popular website servers at the time were implemented in sequential programming. They



lacked the scalability to handle a large volume of concurrent connections limited by blocking codes that ran in sequence.

Dahl invented the Node.js in 2009 as an open-source, single-thread, non-blocking, and cross-platform back-end solution to address this problem.[25] Despite his experience with Netscape, Dahl did not initially aim to write the program in JavaScript. Nevertheless, with the advanced V8 engine's release, it was an epiphany for him to combine the project with JavaScript.[26]

Node.js became a sensation in web-programming upon its debut for being the first representation of JavaScript outside the browser's scope, pioneering in asynchronous programming with the breakthrough **event loop** concept as its core. It also featured a package manager **npm** that productively control the project's module dependencies.

#### 4.3.1 The event loop



```
JavaScript.js UNREGISTERED
JavaScript.js
58 // Event loop
59
60 function bar(){
61     console.log('bar');
62 }
63
64 function baz(){
65     console.log('baz');
66 }
67
68 function foo(){
69     console.log('foo');
70     setTimeout(bar, 0);
71     baz();
72 }
73
74 foo();
75
```

Line 81, Column 1 Tab Size: 4 JavaScript

Figure 17. Code demonstration for the event loop.

Whenever a Node.js application executes a time-consuming Input/Output (I/O) operation, such as fetching data from the servers, it will be set aside along with its callback function to avoid blocking the main thread. Once the task finishes, the process will resume, and the callback function will execute. The Node.js has a built-in set of asynchronous I/O primitives to prevent any blocking task that would interfere with the application and waste the CPU cycles.[25] The event loop is the queuing mechanism that manages the *call stack* as well as the *message queue*. It ensures that the Node.js application can be asynchronous and have a non-blocking I/O.[27]

A code example to demonstrate this concept can be found in Figure 17. There are three functions: `bar`, `baz`, and `foo` that output the name of their function when activated. The `foo` function also activates the `bar` function as a callback function in a `setTimeout` function and then executes `baz`. The execution order of the main call stack is visualized in Figure 18.



Figure 18. The execution order for the functions in Figure 17. code demonstration.[27]

In the first iteration, the `foo` function enters the stack and executes the output control in the second iteration (Figure 17., line 69). Next, the `setTimeout` function enters the stack with the `bar` function and timeout period of zero seconds (Figure 17., line 70). Since the `setTimeout` requires the stack to wait for a declared amount of time to pass before executing the callback function, the main call stack immediately moves the `setTimeout` function aside to await its completion. The `baz` function follows with the output to the console. The `foo` function at this point reaches the end, exiting the call stack.

Once the `setTimeout` function waits for the specified time, the call back function `bar` is moved to the message queue. In this example, the waiting time is set to zero seconds, which should be done instantly. The event loop only refers to the message queue once the main call stack is empty as it always prioritizes every task in the main call stack first.[27] Therefore, in the last three frames of Figure 18, after the `foo` function disappears entirely, it demonstrates the `bar` function's entry and its execution.

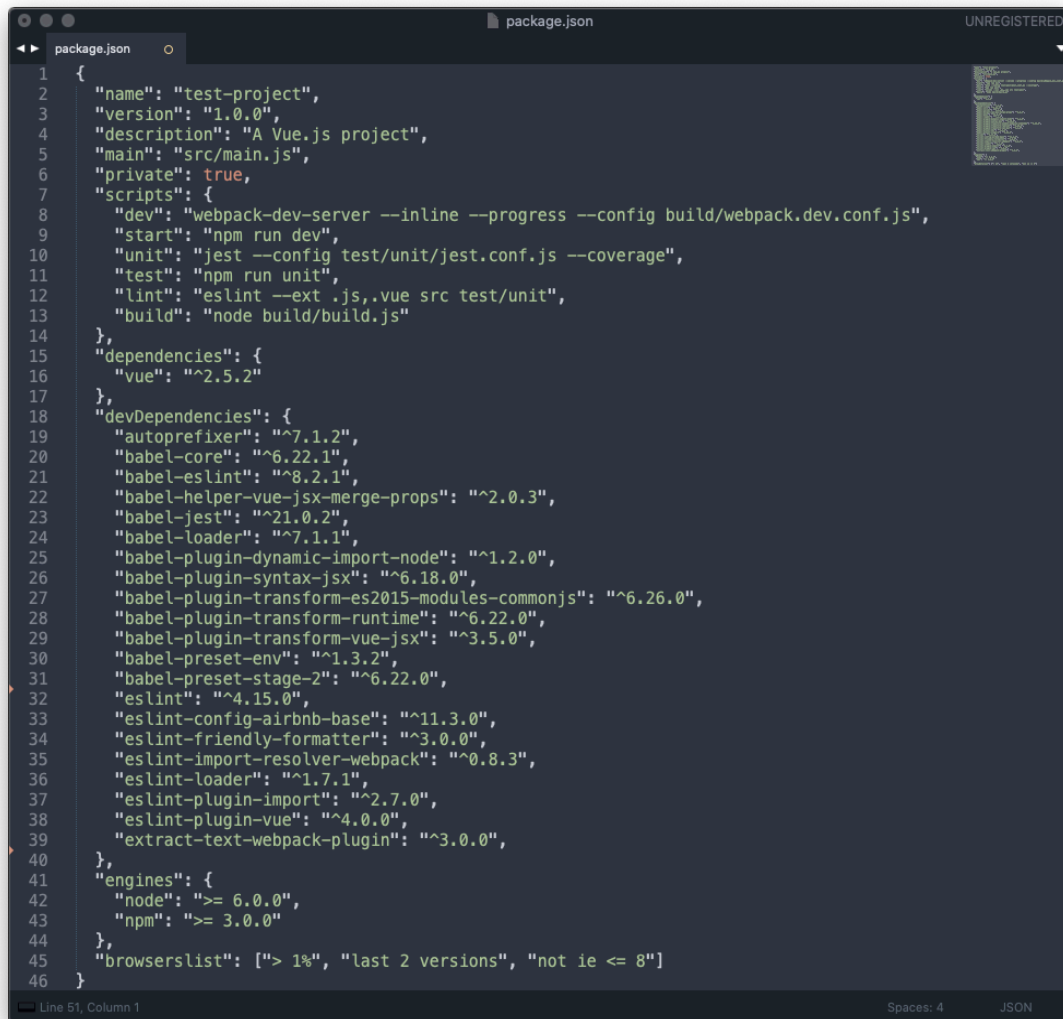
This unique event loop concept recognizes all the time-consuming functions and removes them from the main call stack to eliminate any possible blockage. The message queue is also known as the callback queue for receiving the callback functions when the main task completes. The event loop would first go through the main call stack and clear them before introducing the callback functions or microtasks from the callback/message queue and clearing them on the main stack.

The concept of event loop gave rise to Node.js. Its outstanding success led to adoption within the browsers where they embrace the message queue and repetitive event loop to manage the front-end actions.

### 4.3.2 npm

Another concept introduced that became extremely popular with the Node.js debut is the Node package manager, frequently known as **npm**. Isaac Z. Schlueter, the npm's creator, disapproving how the dependencies were managed and packaged at the time, wrote npm in JavaScript as a proper package manager for Node.js.[28]

npm features a command-line client that allows the user to manage the module dependencies in their project. It connects to a remote module registry allowing easy access to download and update dependency. The details of the project are stored in the `package.json` file, including the record of the dependencies.



```
1 {
2   "name": "test-project",
3   "version": "1.0.0",
4   "description": "A Vue.js project",
5   "main": "src/main.js",
6   "private": true,
7   "scripts": {
8     "dev": "webpack-dev-server --inline --progress --config build/webpack.dev.conf.js",
9     "start": "npm run dev",
10    "unit": "jest --config test/unit/jest.conf.js --coverage",
11    "test": "npm run unit",
12    "lint": "eslint --ext .js,.vue src test/unit",
13    "build": "node build/build.js"
14  },
15  "dependencies": {
16    "vue": "^2.5.2"
17  },
18  "devDependencies": {
19    "autoprefixer": "^7.1.2",
20    "babel-core": "^6.22.1",
21    "babel-eslint": "^8.2.1",
22    "babel-helper-vue-jsx-merge-props": "^2.0.3",
23    "babel-jest": "^21.0.2",
24    "babel-loader": "^7.1.1",
25    "babel-plugin-dynamic-import-node": "^1.2.0",
26    "babel-plugin-syntax-jsx": "^6.18.0",
27    "babel-plugin-transform-es2015-modules-commonjs": "^6.26.0",
28    "babel-plugin-transform-runtime": "^6.22.0",
29    "babel-plugin-transform-vue-jsx": "^3.5.0",
30    "babel-preset-env": "^1.3.2",
31    "babel-preset-stage-2": "^6.22.0",
32    "eslint": "^4.15.0",
33    "eslint-config-airbnb-base": "^11.3.0",
34    "eslint-friendly-formatter": "^3.0.0",
35    "eslint-import-resolver-webpack": "^0.8.3",
36    "eslint-loader": "^1.7.1",
37    "eslint-plugin-import": "^2.7.0",
38    "eslint-plugin-vue": "^4.0.0",
39    "extract-text-webpack-plugin": "^3.0.0",
40  },
41  "engines": {
42    "node": ">= 6.0.0",
43    "npm": ">= 3.0.0"
44  },
45  "browserslist": ["> 1%", "last 2 versions", "not ie <= 8"]
46 }
```

Figure 19. An example of a `package.json` file.

In Figure 19., an anatomy of a `package.json` file is exhibited. The file acts as the blueprint of the project. It details the descriptive information with data fields such as `name`, `version`, `description`, `engines`. More importantly, it lists all the modules used in the project's dependencies and `devDependencies` (for the development environment only) fields with

the name and the minimum or specific version that the application required. Every time a module is added or removed, the `package.json` will update itself to keep track of the changes.

npm revolutionized the delivery of applications in both development and production environment. The `package.json` file can be deposited in the repository along with the rest of the application's source code. The developers can simply run `npm install` in the project folder's command prompt to install all the modules with the up-to-date version with the blueprint file. The command would also add, update, and remove the dependency modules exactly outlined by the details in the `package.json`. This method ensures consistency between the development environment, contributing to the coherence and overall productivity, especially in larger-scale programming. The deployment process also benefits from the npm's dependency list, installing the dependencies sufficiently.

npm was widely famous for its launch with the advantages cited above. It provides a better flow for the development process, especially with the remote registry collection of modules. This popularity of the module library and its easy implementation has led to malicious code laced into the modules. Thus, the developers ought to have a careful curation of the dependencies.

## 5 ECMASCRIPT COMEBACK & UI LIBRARIES EXPLOSION

The innovations in the past decade powered the rise of JavaScript. Its popularity had brought the community together to collaborate towards ECMAScript's comeback and they continued pushing the web-constructing boundaries with JavaScript.

This chapter concentrates on adding the next version of ECMAScript and the critical feature list that gave JavaScript a shift in flexibility as the web development process moved towards a single-page application with a new generation of JavaScript-based UI libraries.

### 5.1 ES5

The ES4 was a bold step up but it was never public with a lack of support. The committee's corporate members released an ECMAScript 3.1 later as an update of ES3 focusing on security, library updates, and compatibility.[18] Brendan Eich announced that the Ecma TC39 committee's unanimous effort would be spent on the further development of ECMAScript 3.1, which was later named the ECMAScript 5<sup>th</sup> Edition.

The ES5 was published in December 2009, exactly ten years from the last version, the ES3.[18] This version certainly did not carry all of the ES4 draft's potential. However, it still presented many salient changes to the language and became the cornerstone of the next innovation wave of JavaScript.

#### 5.1.1 Strict mode

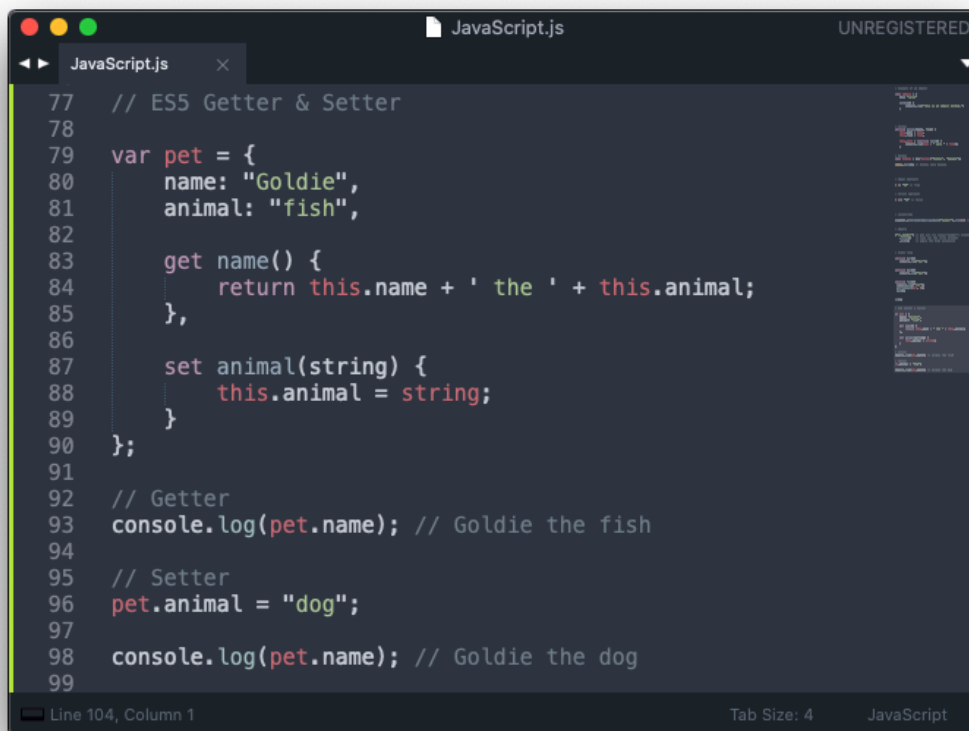
The ES5 introduced a strict mode declaration "`use strict`" for code to be used in strict mode.[18] It is a string expression which assists the developers in writing cleaner, more secure code. JS has the reputation of being a 'too-easy' programming language that lacks coding discipline since the code would still run even with less than appropriate syntax. The strict mode

would turn the sloppy syntax or coding choices into actual error signals, such as the usage of undeclared variables, mistype variables, and deleting variables.

### 5.1.2 Array and object methods

There are many excellent methods put forth in the 5<sup>th</sup> version of ECMAScript for array and object manipulation. As JavaScript usage increased in volume and complexity, these methods shortened the developing time and were excellent tools for developers.

For the array, there are an `Array.isArray()` method for type verification, `Array.indexOf()` and `Array.lastIndexOf()` for an element's index identification. Especially, for the array iteration purpose, there are a wide range of methods to choose from such as `Array.map()`, `Array.forEach()`, `Array.reduce()`. Another useful method is `Array.filter()` which incorporates iteration into filtration.



```
77 // ES5 Getter & Setter
78
79 var pet = {
80     name: "Goldie",
81     animal: "fish",
82
83     get name() {
84         return this.name + ' the ' + this.animal;
85     },
86
87     set animal(string) {
88         this.animal = string;
89     }
90 };
91
92 // Getter
93 console.log(pet.name); // Goldie the fish
94
95 // Setter
96 pet.animal = "dog";
97
98 console.log(pet.name); // Goldie the dog
99
```

Figure 20. Property Getter and Setter in ES5.

For the object, the ES5 provided the syntax to have a getter and setter function. Figure 20 is an example of a `pet` object with a getter method `name` that returns the name with the animal breed and a setter method `animal` that changes the breed.

### 5.1.3 JSON support

After the ES5 had been added, the JSON started to gain popularity as a data communication format between the server and the front-end. The server would send the data as a string to the client-side, with the JSON method `JSON.parse()`, and the string would turn a JS object. There is also the `JSON.stringify()` to turn a JS object into a plain string for sending data back to the server.

## 5.2 Single Page Application

At the end of the 2000s, there is an incredible number of web-applications. What used to be a portal to display description text and images to give a virtual representation, has been engineered to become highly interactive applications like Google Maps. Behind this web-applications phenomenon was the concept of single-page application – an invention that marked the rebirth of JavaScript and solidified its leading position for the next decade to come.

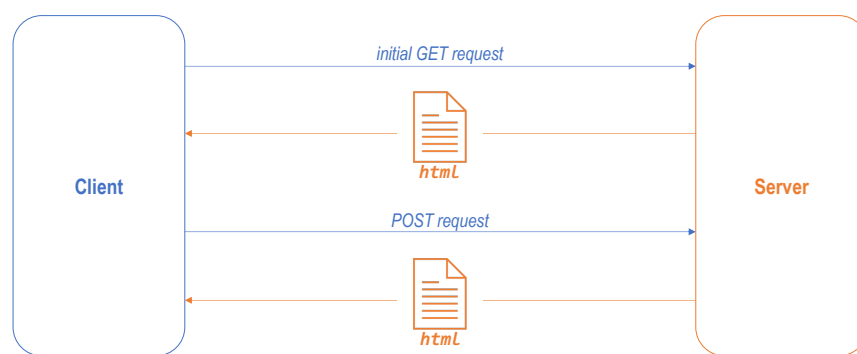


Figure 21. Traditional Multi-Page Lifecycle.

Traditionally, each website view is a *single* HTML file. When the user inputs the URL, the browser will process URL requests with the server, and in response, the server will send back the HTML



file for the browser to render. When the client submits a POST request to send out an update form, the server will return another HTML file for render. (Figure 21.)

This traditional method would require the website to reload multiple times whenever it receives a new HTML file to render the name multi-page application. This approach is a lamentable option for more ambitious web projects due to its absolute sending/receiving mechanism and front-end/back-end decoupling by responding with an entire HTML file. A single HTML file will have to carry many static asset files, such as CSS, images and JS codes, making it bulky to transfer and hinder the overall performance.

A Single-page Application (SPA) is the agile solution embraced by the JS developers to overcome these limitations. A SPA is an application that only has a single loaded page responsible for all of the interaction.

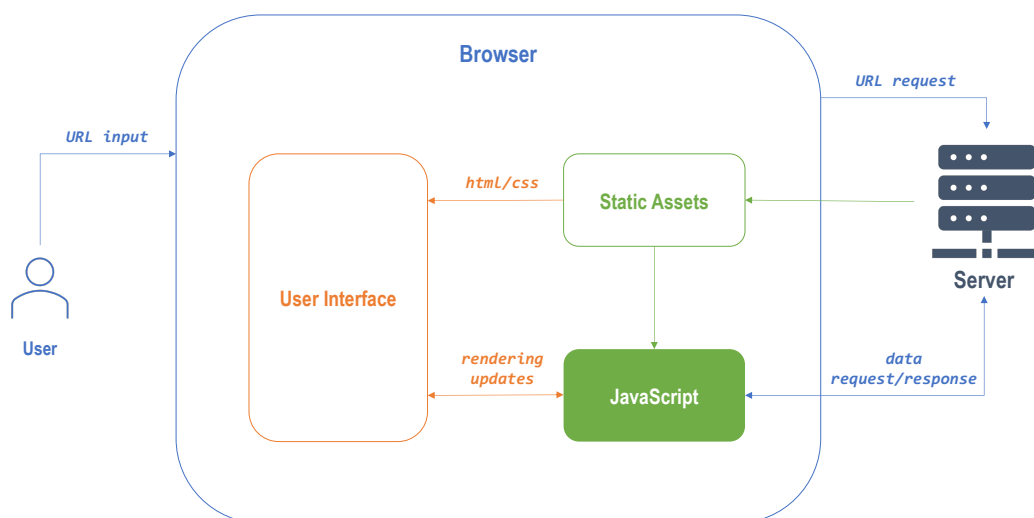


Figure 22. The lifecycle of Single-Page Application.

In Figure 22., the demonstration shows that when the browser sends the first *initial* URL request to the server, it responds by sending the webpage's 'Static Assets'—building blocks of the UI. However, the JavaScript will handle the interaction and simultaneously request the server's data using AJAX requests. This initial data is stored temporarily (this is also known as caching). If there are changes triggered by the server or recognized by the JavaScript, the JavaScript will

activate the rendering updates. The UI will stay-up-date, flexible, responsive, and highly interactive.

When compared to the traditional approach, the SPA presents many superior advantages. The first and most important is the product quality: re-rendering only happens when necessary, the UI update is noticeably faster because of the lightweight JSON-formatted string response from the server, the UX is greatly improved with less interruption of reloading. The second is the development quality with a better separation between front-end and back-end, and easier debugging process.

### **5.3 SPA UI libraries**

The adoption of SPA in web development overgrew regardless of its flaws (length initial loading time, CPU process requirement) exclusively with JavaScript.

#### **5.3.1 2010: The beginning**

AngularJS and Backbone.js were the first significant SPA JavaScript front-end solutions. Both debuted in October of 2010 and made a dramatic impact upon their release.

Backbone.js is a lighter front-end library using a RESTful JSON interface and it practices the SPA in a Model-View-Controller dynamic. It was a popular choice for smaller projects with less data traffic.

AngularJS is a framework which extends HTML with new attributes for a two-way data binding, which essentially means to create a connection between the HTML element and the data from the server. It was geared towards enterprise-level programming projects with supported unit testing and a two-way data-binding activity for faster updates.

### 5.3.2 React and many others

Despite SPA's frameworks/libraries using JavaScript were the first-generation, they were extremely successful. They paved the way for an explosion of the JavaScript-based UI libraries in the following five years.

#### 5.3.2.1 React

React is a one-way data flow UI library created in 2003 by Jordan Walke – an engineer working for Facebook.[29] React was considered a game-changer for many groundbreaking concepts, including the virtual DOM. The virtual DOM is an in-memory UI representation of the *actual* DOM tree which acts as a reference for React to calculate and compare the exact change and selectively trigger rerender in the real DOM through the ReactDOM library.[29] A React application comprises pure JavaScript components which will be rendered to a DOM element through the ReactDOM library. These components can have values that are passed down to them, known as *props*. They make up the node in the virtual DOM tree. If there is a change, for example, a change of *prop* value text line in the <p> element, only the <p> element in the real DOM would rerender with the changed value. This use of virtual DOM resulted in the ultra-fast performance over React's UI library peer that only uses regular DOM updates.

Another reason for its acclaimed reputation is the promotion of functional programming with pure JavaScript components. The front-end development becomes less tangled in a complicated DOM tree by isolating the components to their single functionality or purpose. The code leans on a single purpose. This approach means less code, easier maintenance, less repetition with a high capability of code recycling.

React won the favor of the developers with its premiere which led to substantial support from Facebook. It snowballed in a short period from the number of developers using React to contribute to React by building specialized library extensions for it.

### 5.3.2.2 The other UI libraries

Single-page applications UI libraries soon exploded with different frameworks that cater to different programmers' coding styles and requirements. Under the hood, they are practicing SPA philosophy. However, they are distinct from one another based on their paradigm approaches, their built-in functionalities' extensiveness, and their aim of the production scale.

AngularJS favors a declarative logical/functional programming style, while Backbone.js prefers an imperative structured/procedural programming paradigm. Many libraries adopt the virtual DOM following React's footsteps, e.g. Vue and Ember, and many still use the regular DOM. The abundance of choices gave the developer the freedom to choose the best tool that best meets the project and their preferences.

### 5.3.2.3 UI libraries' libraries

The success of the SPA UI libraries is measured in the usage and great amount of the UI components libraries and in the variety of their corresponding supporting libraries.

If considering the Vue framework as an example: there are standard libraries from the same development company recommended for different assistance purposes. Vuex for the state manager, a centralized *store* that keeps track of all the data changes that a component can connect to, retrieves and stays current with the relevant data. For UI elements, such as a tooltip, there are many libraries for Vue alone that can be easily installed through the module manager npm. These open-source libraries are an incredible time-saver as the mundane UI elements or more functional libraries such as map rendering, are much harder to create from scratch.

The wide acceptance from the SPA front-end libraries system's programming community has brought JavaScript to another milestone. It accelerated the development process and fueled a new era of large-scale interactive web applications.

The early 2010s witnessed a new beginning of JavaScript. It had been at the heart of internet innovations for more than a decade and became an essential tool that powers modern-day webpages. The language had surpassed its existence creation purpose of being an *optional*

temporary scripting language between the DOM elements. It had become indispensable for web development and would continue to thrive thanks to the massive developer audience that had adopted its extension application.

## 6 JAVASCRIPT EVERY YEAR AND JAVASCRIPT EVERYWHERE

In web development, the standing position of JavaScript after the appearance of Angular and jQuery is deep-rooted. The technology is continuously evolving, and yet JavaScript has remained in the center position of internet building from its starting days. JavaScript has advanced tremendously in the past decade, with more frequent ECMAScript updates, a growing number of JavaScript-based libraries/frameworks and helpers, and an ever-increasing number of developers learning, using, and expanding the language.

JavaScript has transformed into an industry of its own. This chapter will browse through all the annual updates of the ECMAScript and the critical features with the industry's adoption response of what each version provides. Finally, it will examine the concept of 'JavaScript Everywhere'.

### 6.1 ECMAScript updates

#### 6.1.1 ES 2015

The 6<sup>th</sup> version of ECMAScript was published in June of 2015. The six-year gap from the last ECMAScript and the rise of the SPA UI libraries made this a lengthy revision in functionality and coding styles.

There are new methods for the array, such as the `find` and `findIndex` function. For the Number object, and the global scope: `isFinite` and `isNaN`. However, the most significant changes are the new keywords for the variable, an introduction of a new syntax set for a new wave of coding and an excitingly new concept of Promise.[30]

##### 6.1.1.1 New variable declare keywords: `let` and `const`

JavaScript had only had one keyword to declare a variable so far: `var`. It is a function and global scoped variable. When declared without a default variable, it has an undefined value. In the example of Figure 23., the variable `year` is first declared in line 102 with a 2020 value. It is then

redeclared in line 106 with a 2021 value. When the variable is logged in, and outside of the code block (from line 106 to 110) in line 113, the logging value is 2021, the last to be declared.

```
101 // Var vs Let
102 var year = 2020;
103 var month = 'December';
104
105 {
106   var year = 2021;
107   let month = 'January';
108
109   console.log(year); // 2021
110   console.log(month); // 'January'
111 }
112
113 console.log(year); // 2021
114 console.log(month); // 'December'
115
```

Figure 23. A comparison of var and let.

However, with the month variable, the let keyword, acting as a local declaration, assigns the month variable to another value, but only in the block scope, resulting in the logging value unchanged outside of the block scope on line 114. The main difference between var and let is in their changes in the perspective scopes. var can be redeclared multiple times, as demonstrated in Figure 23., while let can only be declared once in each scope, the variable exists in.

Another new keyword is const, which stands for constant and is similar in the scope scale, like let. It is meant for elements that are never changing in value, it can only be instantiated once with a value, and most importantly, it cannot reassign.

These new keywords would give out an error notice when the usage is outside of the outlined specifications. With this new addition, the ES6 aimed to promote a more structured coding pattern throughout the development process.

#### 6.1.1.2 A new way of coding

The ES6's most defining modifications are the cleaner solution to construct more complex coding projects.



```
145
146 // Spread operator
147
148 function toArray(...elems) {
149   return Array.isArray(elems);
150 }
151
152 let numbers = toArray(1,2,3,4,5,6);
153 let pets = toArray('dog', 'cat', 'fish');
154
155 console.log(numbers); // true
156 console.log(pets); //true
157
```

Figure 24. Spread operator example.

One of these expressions is the spread operator (...) or the rest parameter that gathers the elements and combine them into an array. Figure 24 shows that the function toArray uses the spread operator in the input parameter next to the elems parameter. This spread operator would merge all the inputs into an array.

It is also possible to do the opposite to split the array and object parameter. This expression helps the code block employ the specific variable immediately without accessing the array or object and its index. In the example of Figure 25., the function takes in the array and splits it into two first individual elements of a and b when the rest of the elements in the array are now in a rest



array. `a` and `b` are now can be called immediately exhibited in line 162 (Figure 25.) instead of the traditional method of accessing through the array index.



```
159 // Destructuring
160
161 function splitArray([a, b, ...rest]) {
162     return a + b;
163 }
164
165 let sum = splitArray([1,2,3,4,5]);
166
167 console.log(sum); // 3
168
```

The screenshot shows a code editor window titled 'JavaScript.js' with a dark theme. The code is as follows: Line 159: // Destructuring; Line 160: (blank); Line 161: function splitArray([a, b, ...rest]) {; Line 162: return a + b;; Line 163: }; Line 164: (blank); Line 165: let sum = splitArray([1,2,3,4,5]);; Line 166: (blank); Line 167: console.log(sum); // 3; Line 168: (blank). The status bar at the bottom indicates 'Line 182, Column 1', 'Tab Size: 4', and 'JavaScript'.

Figure 25. An example of array destructuring.

Another unique expression is the arrow function. (Figure 26.) It is a short-handed syntax for the function. The arrow function leaves out the function and the return keywords as well as the curly brackets.[30] It is a more minimal method to write functions that would consume lesser code lines, especially in layering callback functions inside the running function. The arrow function works in conjunction with the new variable keyword `const` in promoting a superior coding pattern. In line 167, the `const` declaration can apply to the function since the function's mechanism once declared is mostly unchanged afterward.

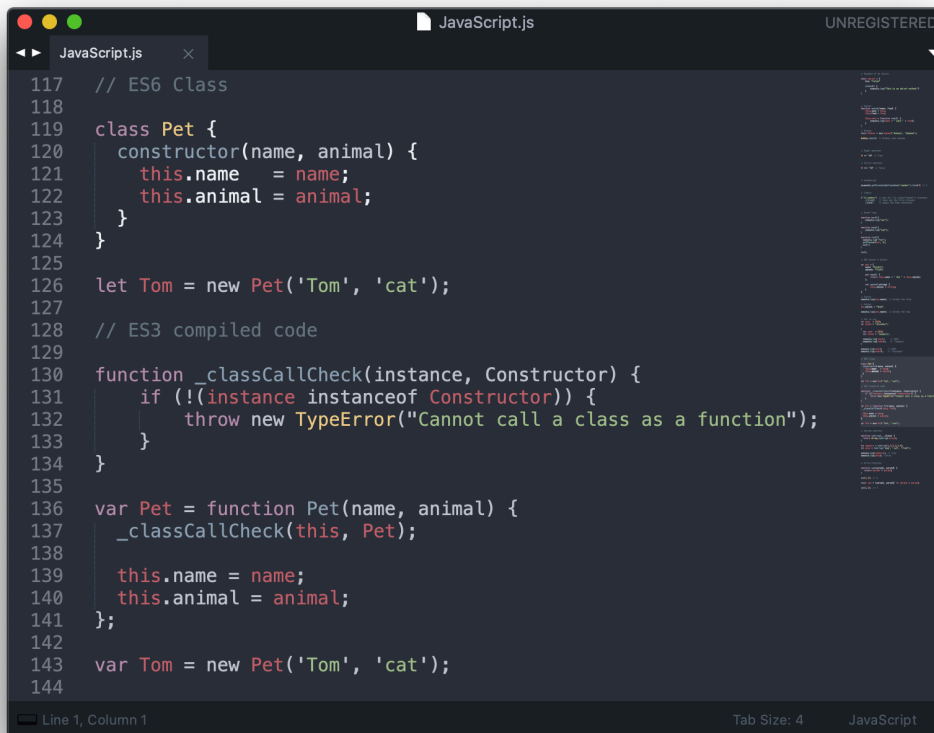
Finally, the ES2015 offers the syntax for classes which have a background or reference for OOP. Figure 27. describes a simple class of `Pet` with a constructor that receives a name and an animal. The syntax is simple and bears a close resemblance to the OOP style in other OOP languages.

The expression update of the 6<sup>th</sup> ECMAScript is a turning point for JavaScript. The syntax and variable keywords accommodate different coding styles and thus inspired a new range of developers from different corners of the programming world to take up the language. Sadly, these advancements are too cutting-edge for the browsers, and some browsers are just not as adaptive as others. Therefore, this browser limitation has sparked the growth of the transpiler library, such as Babel. This translation tool can take in any JavaScript version to turn the code into any version

of JS. Figure 27 is an example of a class declaration in the ES6 and a translated version of the same code in the ES3 – the most supported JS version among the browsers.



```
JavaScript.js UNREGISTERED
JavaScript.js x
159 // Arrow function
160
161 function sum(param1, param2) {
162     return param1 + param2;
163 }
164
165 sum(1,2); // 3
166
167 const sum = (param1, param2) => param1 + param2;
168
169 sum(1,2); // 3
170
Line 1, Column 1 Tab Size: 4 JavaScript
```



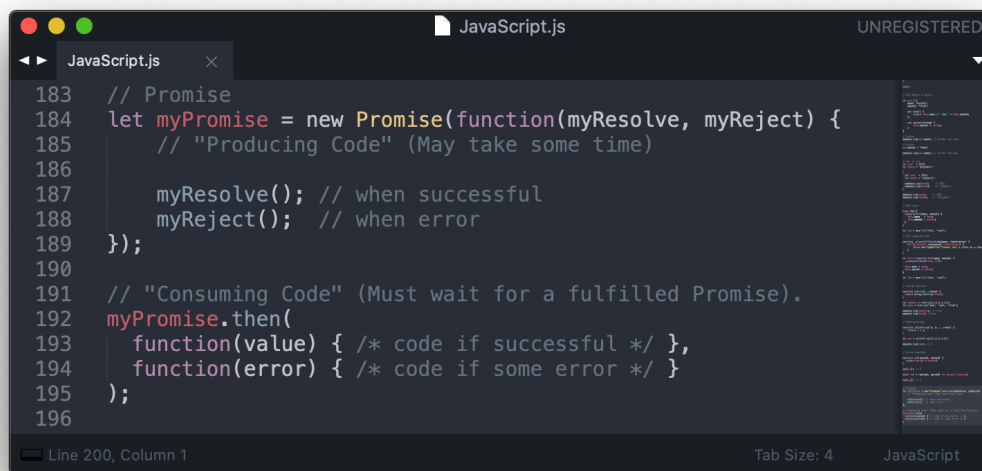
```
JavaScript.js UNREGISTERED
JavaScript.js x
117 // ES6 Class
118
119 class Pet {
120     constructor(name, animal) {
121         this.name = name;
122         this.animal = animal;
123     }
124 }
125
126 let Tom = new Pet('Tom', 'cat');
127
128 // ES3 compiled code
129
130 function _classCallCheck(instance, Constructor) {
131     if (!(instance instanceof Constructor)) {
132         throw new TypeError("Cannot call a class as a function");
133     }
134 }
135
136 var Pet = function Pet(name, animal) {
137     _classCallCheck(this, Pet);
138
139     this.name = name;
140     this.animal = animal;
141 };
142
143 var Tom = new Pet('Tom', 'cat');
144
Line 1, Column 1 Tab Size: 4 JavaScript
```

Figure 26. The arrow function demonstration.

Figure 27. The ES6 class declaration and compiled code.

### 6.1.1.3 Promise

The `Promise` is another meaningful innovation from the ES2015. It is the concept in response to the asynchronous efforts from the popular usage of jQuery's AJAX calls and Node.js. The `Promise` consists of two parts: the asynchronous action and the callback function after the first asynchronous action is completed.[30]



```
183 // Promise
184 let myPromise = new Promise(function(myResolve, myReject) {
185     // "Producing Code" (May take some time)
186
187     myResolve(); // when successful
188     myReject(); // when error
189 });
190
191 // "Consuming Code" (Must wait for a fulfilled Promise).
192 myPromise.then(
193     function(value) { /* code if successful */ },
194     function(error) { /* code if some error */ }
195 );
196
```

Figure 28. A demonstration of the `Promise` function.[30]

In the an example of Figure 28., the function `myPromise` creates a `Promise` executor and would process the "Producing Code" (line 185). `myPromise` would also take in `myResolve` and `myReject` functions. These callback functions are to be executed based on the result of the "Producing Code" which usually are time-consuming tasks, such as fetching data.

### 6.1.2 Annual updates

After the publication of the ECMAScript 6<sup>th</sup> version, JavaScript has grown beyond the influence of the industry's forces. Most of JavaScript's most prominent library frameworks became open-source-maintained and are developed mainly by the community on the Internet. This peaceful

development of JavaScript and its sub-branches products are not confined to any political opinions. ECMAScript has since arrived annually, without much interruption.

The priority is developing more extending methods based on the community of JavaScript usage. For instance, in the more complicated application with a more layered data scheme, there is a need to have the array 'flatten'. The flatten method must be coded by the developer from scratch or it must be taken out from a utility library, e.g. underscore.js. Despite their lightweight size, these utility libraries require to be fully installed even if there is only a need for a couple of methods. For this reason, the ECMAScript developers usually take note of the most used methods to include in the official JavaScript for better performances, such as the `Array.flat()` method debut in the ECMAScript 2021.[31]

Another improvement that ECMAScript frequently features in the annual updates is in the syntax. With JavaScript being the most malleable programming language that caters to a whole range of coding styles, the coding convention upgrade is as important as the functional ones. The `async/await` syntax introduced in the 7<sup>th</sup> Edition of ECMAScript 2016 improves the Promise function's readability and writability. [18]

## **6.2 Usage in non-web applications**

JavaScript exists everywhere in the technology ecosystem. It has undoubtedly outgrown its designated HTML markup script tag. While in newer ventures such as machine learning, JavaScript is still in its early phase, with TensorFlow.js launched in March 2018.[32] In native mobile and desktop development, however, JavaScript is relatively well-established, making the 'JavaScript everywhere' ideology going beyond web development.

### **6.2.1 Mobile development**

The first framework to use JavaScript for native development is React Native – an open-source application framework started by Facebook. The first release of React Native was around the same time as Facebook's sibling product React in 2015.[33]

React Native has the same working principles as React, with the exception of DOM manipulation through the virtual DOM. Instead of the index HTML file, there is a main.js file as the root place to ignite the application from there; it followed the same nested JavaScript architect as React on the web client-side.

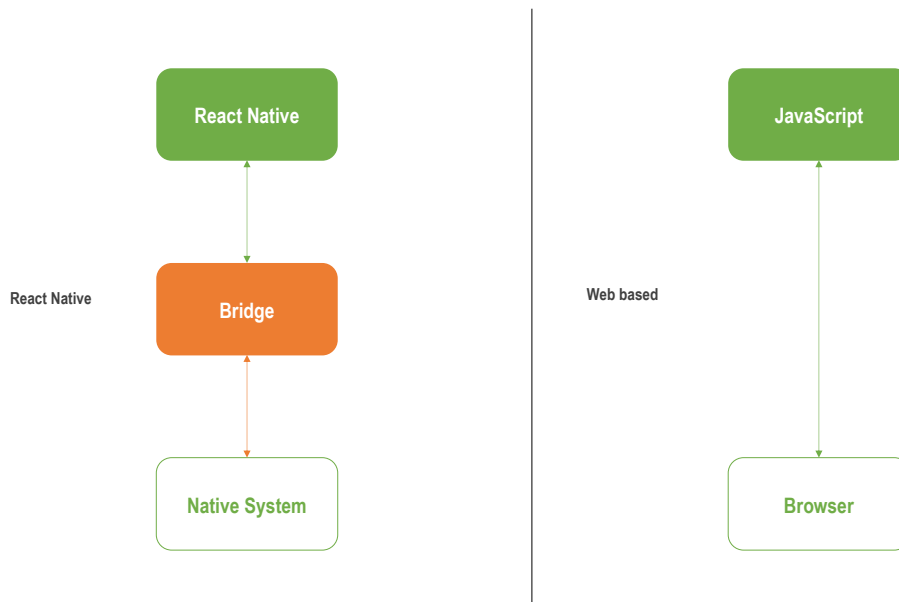


Figure 29. A comparison between the React Native and web-based application process.

The main difference between React Native and React/normal JavaScript is the middleman Bridge. The JavaScript React Native is interpreted and communicates to the native device system through the middleman with serialized native modules, asynchronous data in JSON format, and batch processing—a queue of jobs to be run in the background with the users' permission.

### 6.2.2 Desktop

The desktop application is made possible with the Node.js server runtime and Electron – a JavaScript framework that renders the Graphical User Interface (GUI) with Google Chrome's Chromium rendering engine.[34] Essentially, it used the minimalist version of Chrome, the Chromium, to act as a mini browser in the desktop application to render the GUI and to handle the client interaction while communicating to the server with Node.js.

## 7 CONCLUSION

Back from its creation date in 1995, the potential of JavaScript's success was dimmed. It was a language that was crafted in ten days facing many limits and challenges that were not only technical.

While reviewing the language's characteristics, JavaScript is a rich and flexible language with a soft learning curve but powerful performance. JavaScript gave the developers the luxury of building any application in any platform with their most comfortable coding styles. Its continued flourishing expands the language to be more capable, more accommodating towards many more generations of developers, and to conquer different programming fields.

Moreover, by going through the history, a comprehensive perspective of JavaScript's evolution is revealed. The progress of JavaScript appears both fortuitous and expected at the same time. JavaScript started as a project backed by a large corporation, but it carried on even when the supporting forces died out. Despite its questionable features in the early days, it is still the foundation of many groundbreaking technology innovations. The language has single-handedly transformed the landscape of webpage technology in the past decade. It gave rise to many libraries and frameworks that, in turn, became the building blocks to many impressive software projects. In the past 25 years of JavaScript, it has gone from a secondary choice to web-programming to an absolute pillar of it..

JavaScript's widespread adoption and the contribution effort from the worldwide community of developers has accelerated JavaScript's growth as a programming language. The JavaScript language is a rare programming language which continually redefines its role, purposes, and capability. It did the unthinkable jump to the server-side in the late 2000s and outlived its creative purpose of a glue language in the front-end. JavaScript has also pushed to the native device application in recent years, bringing a new dimension to the concept 'JavaScript Everywhere'.

## REFERENCES

1. Web Technology Surveys. Usage statistics of JavaScript as a client-side programming language on websites. Date of retrieval 07.10.2020.  
<https://w3techs.com/technologies/details/cp-javascript>
2. Stack Overflow Insights. 2020 Developer Survey. Date of retrieval 08.10.2020.  
<https://insights.stackoverflow.com/survey/2020>.
3. MDN Web Docs. What is JavaScript. Date of retrieval 10.10.2020.  
[https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First\\_steps/What\\_is\\_JavaScript](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript).
4. Tom McFarlin. What is JavaScript. Date of retrieval 10.10.2020.  
<https://code.tutsplus.com/tutorials/what-is-javascript--cms-26177#:~:text=JavaScript%20is%20a%20high%2Dlevel,the%20server%2Dside%2C%20too>.
5. JavaScript.Info. An introduction to JavaScript. Date of retrieval 10.10.2020.  
<https://javascript.info/intro>.
6. Uday Hiwarale. How does JavaScript and JavaScript engine work in the browser and node? Date of retrieval 11.10.2020. <https://medium.com/jspoint/how-javascript-works-in-browser-and-node-ab7d0d09ac2f>.
7. JavaScript.Info. Data types. Date of retrieval 11.10.2020. <https://javascript.info/types>.
8. JavaScript.Info. Prototype Inheritance. Date of retrieval 12.10.2020.  
<https://javascript.info/prototype-inheritance>.
9. Cris Hanks. Classical vs. Prototypal Inheritance. Date of retrieval 13.10.2020.  
<https://dev.to/crishanks/classical-vs-prototypal-inheritance-2o5a>.
10. Navneet Sahota. Why Functional Programming Matters. Date of retrieval 13.10.2020.  
<https://dev.to/navi/why-functional-programming-matters-2o95>.
11. History-Computer.com. Mosaic Browser. Date of retrieval 14.10.2020. <https://history-computer.com/Internet/Conquering/Mosaic.html>
12. Wikipedia. Netscape Navigator. Date of retrieval 15.10.2020.  
[https://en.wikipedia.org/wiki/Netscape\\_Navigator](https://en.wikipedia.org/wiki/Netscape_Navigator).

13. Gabriel Lebec. JavaScript: A History for Beginners. Date of retrieval 16.10.2020.  
[coursereport.com/blog/history-of-javascript](https://coursereport.com/blog/history-of-javascript)
14. Sebastian Peyrott. A Brief History of JavaScript. Date of retrieval 16.10.2020.  
<https://auth0.com/blog/a-brief-history-of-javascript/>
15. Wikipedia. JavaScript. Date of retrieval 20.10.2020.  
<https://en.wikipedia.org/wiki/JavaScript>.
16. Martin Armstrong. How Many Websites Are There? Date of retrieval 20.10.2020.  
<https://www.statista.com/chart/19058/how-many-websites-are-there/>.
17. Wikipedia. Ecma International. Date of retrieval 21.10.2020.  
[https://en.wikipedia.org/wiki/Ecma\\_International](https://en.wikipedia.org/wiki/Ecma_International).
18. Wikipedia. ECMAScript. Date of retrieval 21.10.2020.  
<https://en.wikipedia.org/wiki/ECMAScript>.
19. ECMA Standardizing Information and Communication Systems. ECMAScript Language Specification 3<sup>rd</sup> Edition. December 1999.
20. Wikipedia. Browser Wars. Date of retrieval 25.10.2020.  
[https://en.wikipedia.org/wiki/Browser\\_wars](https://en.wikipedia.org/wiki/Browser_wars).
21. Wikipedia. jQuery. Date of retrieval 26.10.2020. <https://en.wikipedia.org/wiki/JQuery>.
22. Wikipedia. History of the Web Browser. Date of retrieval 27.10.2020.  
[https://en.wikipedia.org/wiki/History\\_of\\_the\\_web\\_browser](https://en.wikipedia.org/wiki/History_of_the_web_browser).
23. Kevin Millikin, Florian Schneider. A New Crankshaft for V8. Date of retrieval 30.10.2020.  
<https://blog.chromium.org/2010/12/new-crankshaft-for-v8.html>.
24. V8 Team. Launching Ignition and TurboFan. Date of retrieval 30.10.2020.  
<https://v8.dev/blog/launching-ignition-and-turbofan#:~:text=Crankshaft%20can%20only%20optimize%20a%20subset%20of%20the%20JavaScript%20language.&text=TurboFan%20was%20designed%20from%20the,planned%20for%20ES2015%20and%20beyond>.
25. Wikipedia. Node.js. Date of retrieval 2.11.2020. <https://en.wikipedia.org/wiki/Node.js>.
26. Node.js Contributors. Introduction to Node.js. Date of retrieval 2.11.2020.  
<https://Node.js.dev/learn/introduction-to-Node.js>.
27. Node.js Contributors. The Node.js Event Loop. Date of retrieval 2.11.2020.  
<https://Node.js.dev/learn/the-Node.js-event-loop>.



28. Wikipedia. npm. Date of retrieval 12.11.2020.  
[https://en.wikipedia.org/wiki/Npm\\_\(software\)](https://en.wikipedia.org/wiki/Npm_(software)).
29. Wikipedia. React. Date of retrieval 26.11.2020.  
[https://en.wikipedia.org/wiki/React\\_\(web\\_framework\)](https://en.wikipedia.org/wiki/React_(web_framework)).
30. W3Schools.com. ECMAScript 2015 – ES6. Date of retrieval 26.11.2020.  
[https://www.w3schools.com/js/js\\_es6.asp#mark\\_class](https://www.w3schools.com/js/js_es6.asp#mark_class).
31. ECMA Standardizing Information and Communication Systems. ECMAScript 2021 Language Specification. December 2020.
32. Wikipedia. TensorFlow. Date of retrieval 6.12.2020.  
<https://en.wikipedia.org/wiki/TensorFlow>.
33. Wikipedia. React Native. Date of retrieval 6.12.2020.  
[https://en.wikipedia.org/wiki/React\\_Native](https://en.wikipedia.org/wiki/React_Native).
34. Wikipedia. Electron. Date of retrieval 6.12.2020.  
[https://en.wikipedia.org/wiki/Electron\\_\(software\\_framework\)](https://en.wikipedia.org/wiki/Electron_(software_framework))