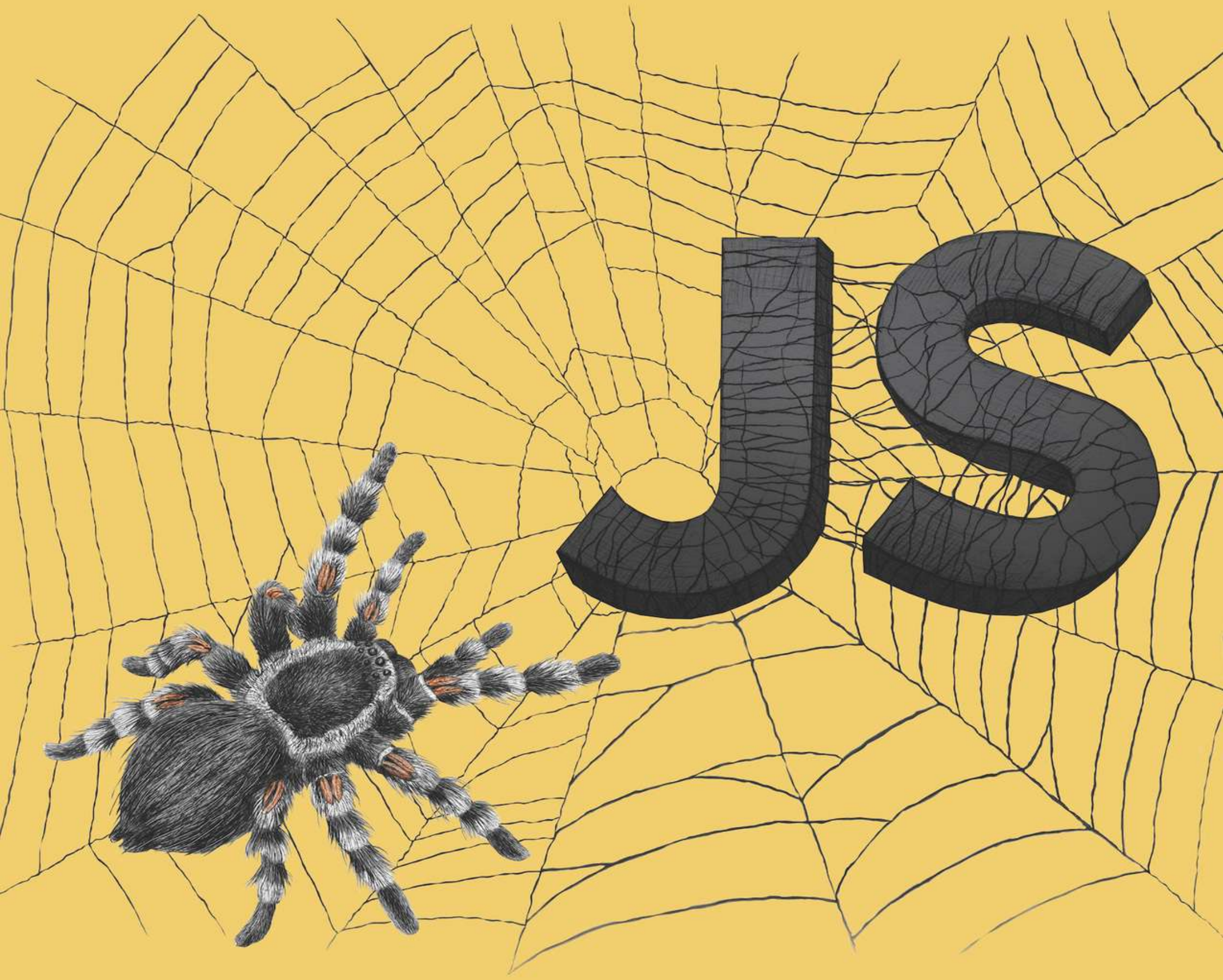


JavaScript
From Zero to Hero

The Most Complete Guide Ever,
Master Modern JavaScript Even
If You're New to Programming

Rick Sekuloski

JavaScript From Zero to Hero: The Most Complete Guide Ever, Master Modern JavaScript Even If You’re New to Programming

Rick Sekuloski

ISBN:

VARIABLES AND ASSIGNMENT

DECLARING A VARIABLE

INITIALIZING A VARIABLE

VAR, LET, AND CONST

LET

CONST

NUMBER LITERALS

STRING LITERALS

TEMPLATE LITERALS

ARITHMETIC OPERATORS

STRING CONCATENATION +

BOOLEAN VALUES

NULL AND UNDEFINED

COMPARISON AND LOGICAL OPERATORS

OBJECTS

CREATE OBJECTS USING NEW KEYWORD

CREATING OBJECTS USING OBJECT.CREATE()

PRIMITIVES PASSED BY VALUE

ARRAYS

ARRAY LITERALS

CREATE ARRAYS USING NEW ARRAY ()

SPREAD OPERATOR

ACCESS ARRAY ELEMENTS

CONDITIONAL STATEMENTS OR BRANCHES

IF-ELSE STATEMENT

ELSE IF STATEMENT

CONDITIONAL (TERNARY) OPERATOR

SWITCH STATEMENT

ASSIGNMENT OPERATOR

OPERATOR

WHILE LOOP

DO WHILE LOOP

FOR LOOP

FOR/OF LOOP WITH OBJECTS

OBJECT.KEYS() – FOR/OF

OBJECT.ENTRIES() – FOR/OF

FOR/IN LOOP

Preface

Welcome to my second JavaScript book. In the first book, I explained the basic concepts that everyone should know, and I also discussed why those features are crucial to understand. JavaScript today is one of the most popular web programming languages, and that is the reason why I'm writing this book. This book is more about intermediate to advanced features, but I will also include two extra chapters for those that want to learn the basic JavaScript concepts. This book is for everyone passionate about learning JavaScript, but with that being said, I would not start covering the basics of JavaScript in the first section. If you are new to JavaScript programming or need to refresh your memory, I recommend that you skip the first three advanced chapters of this book and read chapters four and five, where I will cover the basics of JavaScript. The idea of this book is to give you in-depth knowledge of advanced features, but as a bonus, I want to give everyone an equal chance. That is why I included the basics in the later chapters. If you already know the basics, then please start from chapter one. This book will help you master this amazing web language through many examples.
The book will include longer and smaller chapters, but I promise that they will be full of theory and examples that you will enjoy.
If you are looking for extra reference material, I recommend visiting the MDN website.
You can open the MDN website by visiting the following URL:
https://developer.mozilla.org/en-US/

I would also like to hear from you, so if you need to contact me, please reach out through some of my social media accounts and consider leaving a review with your comment.

My social media accounts:

- Twitter: **https://twitter.com/Rick29702077?s=09**
- LinkedIn: **https://www.linkedin.com/in/rick-sekuloski**
- Facebook:**https://www.facebook.com/theodorecodingwebdevelopment/**
- YouTube:**https://www.youtube.com/channel/UCQanUcCNaBg-IM-k0u8z0oQ**

Who is this book for?

This book is for:

- Students
- Anyone that is considering learning JavaScript for the first time
- JavaScript programmers with prior programming experience
- Anyone that is seeking to gain a deep understanding of the client and server-side APIs available to JavaScript

How to Get the most out of this book?

To get the most from this book, you will need the following tools if you are using a computer or tablet:

- A text editor of your choice, here I will use Visual Studio Code (VsCode).
- An up-to-date browser such as Google Chrome, Firefox, Edge, or Safari.

If you are using an e-reader, you can sit back and relax because I will include many examples so that you do not miss much coding. But if you to go through the examples, please use your computer.

Download the example code files

You can download the entire code by visiting my GitHub repository page using the following link:

https://github.com/RickSekuloski/rick-javascript-book2

There you will find all of the materials (code examples and exercises) that I used in this book.
Once the file is downloaded on your computer, you will need to unzip the content. Please don't open or run the code while it's inside the zipped folders/directories. Ensure that you extract the folder into your desired destination, such as the desktop.
You can Unzip the files using the following programs:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac

- 7-Zip/PeaZip for Linux

Obtain the images you will need

Once the downloaded files are extracted, you can find all of the images I use in this book in the ‘color-images’ folder. If you read this on an e-reader, some images might be blurry due to compression and resizing.

Chapter 1 – Strings and Regular Expressions

In this chapter, you will learn about the regular expressions and more about Strings. We will also cover the methods that we use to process the strings. As you know, the **JavaScript** types can be divided into two categories: primitive and non-primitive (object type). The Strings, just like **Numbers** and **Booleans,** are considered as **JavaScript** Primitive types. **Strings** are a series of characters enclosed by single or double-quotes.
This is an example of a basic string:

let playerName = 'Cristiano Ronaldo';

In JavaScript, the **text** is considered to be from a type string. The string is a sequence of Unicode characters.

Unicode

Why do we need to use **Unicode** characters in the first place? Let me put it simply like this: the computer machine does not understand English letters, but they do understand the sequence of characters. That is why we need to use **Unicode** because it will provide a list of character sets and assign each character a unique code point. **Unicode** is a universal character set, and it provides a unique number for every character. **Unicode** version 1.0 was released in 1991, and the latest up-to-date version was released in 2021, and it includes codes for 144,667 characters. That is a lot of codes, and without Unicode programming, everything we do will be very difficult.

Characters and Code points

I already mentioned that **Unicode** assigns a unique code point for each character. A code point is a number assigned to a single character. These numbers can range from **U+0000** to **U+10FFFF**. As you can see, we need to use **U+,** a prefix that stands for **Unicode,** and after the plus, we have the **<hex>,** which stands for a hexadecimal number. And there you have it. The Unicode manages this code point and tides them with a specific character. Another important part for you to understand is that JavaScript uses **UTF-16** encoding of **Unicode** character set. According to the **ECMAScript** specification, the strings are:

*" The String type is the set of all ordered sequences of zero or more 16-bit unsigned integer values ("elements"). The String type is generally used to represent textual data in a running ECMAScript program, in which case each element in the String is treated as a **UTF-16 code unit value**. "*

Before I confuse you, even more, let us go over one simple example:

```
> const message = 'Hello';
  console.log(message.length);

  5

< undefined
```

In the above example, we can see that in the variable **message,** we have a string that consists of 5 characters. Now we always associate the strings as a sequence of visible characters because we count the letters right, numbers, or even punctuation marks, if any? This approach will work if we use simple characters known as **ASCII** characters and belong to the Basic Latin block.
Let us consider the following string example:

```
const smile = '😈';
console.log(smile.length);

2
```

Wait, the length is 2, and I can only see one emoji but not two. What is happening here? Well, emojis are more complex characters, and the length property will give us back 2, meaning we can no longer rely on visual characters.

The JavaScript will consider this string as a sequence of two separate code units.
As I mentioned earlier, each character is assigned a special code point right, so the first example with the message variable in the background will be processed as this sequence of **UTF-16** code units:

```
> const message = '\u0048\u0065\u006C\u006C\u006F';
  console.log(message === 'Hello');
  console.log(message.length);
  console.log(message);

  true

  5

  Hello
```

For testing purposes, I use the basic **Google Chrome Developer Console**. Here is a link if you do not know how to use the console for simple testing:

https://developer.chrome.com/docs/devtools/console/

If you use a different web browser like **Firefox**, the console is very similar. Still, you will need to find that information on the internet because it is very easy to understand and is bit different for each browser.
Let us go back to strings, and so far, we know that **JavaScript** string is a sequence of **UTF-16** codepoints, and we can find out the exact number of those units when we use the **string.length** property. But here, we might have one particular problem, what do you think will happen if the codepoints we are talking about do not fit in the 16 bits, and this is possible? We need to use a rule known as surrogate pair of two 16 bits values. In simple words, if we have a string with a length of 2, it does not mean we have two separate Unicode characters, but it can be one complex character as the emoji example.
Please check out this example:

```
let euroSymbol = "€";
let smileSymbol = "😀";
console.log(`Euro symbol has a length of: ${euroSymbol.length}`);
console.log(`The smile symbol has a length of: ${smileSymbol.length}`);

Euro symbol has a length of: 1

The smile symbol has a length of: 2
```

I hope by now you understand how JavaScript, **Unicode**, and **UTF-16** codepoints are working together to achieve the result we expect to have.
Finally, I can now explain a method called **fromCodePoint.** This method that belongs to the String class can assemble a string from one or more code points. These code points we pass to the method as parameters. JavaScript methods, as we know, can take a list of parameters, and here as parameters, we need to use a sequence of code points.
The syntax of the static **String.fromCodePoint()** is:

```
String.fromCodePoint(num1)
String.fromCodePoint(num1, num2)
String.fromCodePoint(num1, num2, ..., numN)
```

The arguments **num1,….,numN** is a sequence of code points we already discussed. I said **static method** because this method must be invoked directly from the **String** constructor object, not by an instance of the **String** class.
The return value of this function will be a string that will be created by using the specified sequence of code points.
Please take a look at this example:

```
console.log(String.fromCodePoint(9731, 9733, 9842, 0x2F804));
```

The output should be ☃ ★♲ ??. But sometimes, you might have a huge array of codepoints, and you want that array to be used as an argument in our method. We can easily solve this if we use the famous three dots '…' spread operator.
Here is one example about it:

```
//array of codepoints
const myCodePoints = [9731, 9733, 9842, 0x2F804];
//spread operator
const stringOutput = String.fromCodePoint(...myCodePoints);
console.log(stringOutput);
```

The output should be exactly the same as the previous example. You can find this code in the file called **lecture1.js** inside the **chapter1** folder that you can find in the downloaded files. You can copy and paste the code into your browser, and please observe the output. You are free to use different code points and play around because even if you make a mistake, you will learn how to fix it and why it happened. And as I mentioned in my previous books, even if you edit the original code, it will be no problem because you can always download it again and start it from scratch.
Let us get back to work, and one interesting example I want to share is when we have a string, and we want to know the codepoints for each character. How can we achieve this? Well, we can use the good old **for loop** and traverse the code points of a string just like in this example:

```
const message ='Hello World!';
for (let i = 0; i < message.length; i++) {
    let codePoint = message.codePointAt(i);
    console.log(codePoint);
}
```

```
72
101
108
111
32
87
111
114
108
100
33
```

As you can see from the example the string ‘Hello World!’ have some interesting codepoints. Instead of printing them like this we can change our code and store the codepoints inside an empty array:

```
const message ='Hello World!';
const codePointsArray = [];
//we can traverse through our newly created string with a for loop
for (let i = 0; i < message.length; i++) {
   let codePoint = message.codePointAt(i);
   //console.log(codePoint);
   codePointsArray[i] = codePoint;
}
//this should return back 'Hello World!'
console.log(String.fromCodePoint(...codePointsArray));
```

String methods you should know about

A string is a sequence of unsigned 16-bit values and what we have not mentioned so far is that the string is an

immutable sequence. What does this mean? Well, it means only one thing: the methods we are using on strings will not change the contents of a given string. An empty string is a string with length zero ‘0’. There are many methods of the String class, but here I will list a few of them that I believe are worth mentioning. The examples are included in the same folder, under the name **stringMethods.js**.
The first one is the **repeat** method and is useful when we want the same string to be repeated several times.
Example:

```
//1) repeat method
const message = 'W';
const repeatIt = message.repeat(3);
console.log(repeatIt);// 'WWW'
```

Another useful method is the **trim** method. This method will remove whitespace characters from the start and the end of the string. Interesting to know is that this method must be invoked by an instance of the String class.
Syntax:

```
string.trim();
```

You should not supply any parameters in this method, and you should also know that this method will not change the value of the original string.
Let us take a look at the following example:

```
> const beforeAndAfter = '   Hi There   ';
  console.log('1 - trim method: '+ beforeAndAfter.trim());
  console.log('2 - does not change the original content: '+beforeAndAfter);
  1 - trim method: Hi There
  2 - does not change the original content:    Hi There
<· undefined
>
```

Another two methods for trimming the white spaces are **trimStart()** and **trimEnd()**. The **trimStart()** method will remove the leading white spaces, and the latter will delete the trailing white space. I will not test this two because you can try them in your free time. After all, they are very basic.
In JavaScript, not only the regular space character **‘\u0020’** is considered as white space but also recognizes newline, tab, carriage returns, nonbreaking space **‘\u{00A0}’** as white spaces.
There are other String methods in JavaScript will do the opposite of what the trim methods are doing, and that is to add space characters, which is very useful as well. So we can use **padStart()** and **padEnd()** to add spaces before and after, but keep in mind that the current string will be padded until the resulting string reaches the given length.
Here is one example where I have added only 5 space characters from the start of the current string:

```
const example = 'Hello';
const newString = example.padStart(10);
console.log(example);
console.log(newString);
Hello
     Hello
undefined
```

If we use the length property now, we will see that the example will have a length of 5, but the **newString** variable will have a total length of 10, not 15. So be very careful with adding the space characters:

```
console.log(example.length); // 5
console.log(newString.length); // 10
```

I have not shown you the syntax of the pad methods, but now is the right time to do this:

```
padStart(targetLength)
padStart(targetLength, padString)
```

As you can see, we already did an example where we supplied only one parameter, the **targetLength.** Still, the

second parameter is optional and nice because we can define our padding style.
Check out this example:

```
const newString1 = example.padStart(10, '#');
console.log(newString1); //  #####Hello
```

You can test out for **padEnd()** method because it is completely the same with only one difference, and that is, it will add a space at the end of the current string.
When we start writing programs in JavaScript, we would like to convert the string characters into upper or lowercase most of the time. For example, a person is trying to submit a registration form on our website, but our policy is to have all the names stored in the database in lowercase. It is a bad idea to write this on the form itself, giving direction to the users on how they need to write their first or last names using lowercase. This is not a good approach or the best user experience, right? So what we can do is leave the user to fill out the form with its details, and then later at the backend, where we are getting the form data, we can convert all of the required form fields into lowercases. This was just an example of why we might use these new methods, but I assure you there are many more. Finally, these two methods are called **toUpperCase** and **toLowerCase**.
Example:

```
const firstName = 'Andy';
const lastName = 'Garcia';
console.log(firstName.toLocaleLowerCase());//adny
console.log(lastName.toUpperCase());//GARCIA
```

Another method that we kept using but never discussed was the length of a string or the length property.

```
const fullName = 'Andy Garcia';
console.log(fullName.length);// result: 11
```

The Strings in JavaScript behave the same as arrays because they are arrays of characters, but they are not mutable; please note that. Therefore JavaScript strings are zero-based, the same as we have in arrays. We always start from index/position zero. So, the first 16-bit is in position 0, and the second 16-bit value is located at position 1, and so on.
Because we have indexes now, we can retrieve any specific character from a string if we use the square bracket notation []. The square bracket notation is trading marks of arrays, but we can also now use it on strings ‘because basically, strings are an array of characters remember.’
Example:

```
console.log(fullName[0]);//A
```

Remember, with arrays and strings, we start from 0, not 1, and if we want to get the last character from any string, we can use the length property minus 1.
Please take a look at the following example, and everything will be clear (I use **fullName** string from the previous example, and it contains this string ‘Andy Garcia’):

```
console.log('Length of the fullName string is: ' + fullName.length);
console.log('Last character is: '+ fullName[fullName.length-1]);
```

Output:

```
Length of the fullName string is: 11
Last character is: a
```

Instead of using this, we can use **charAt()** method that will return a new string consisting of only one single **UTF-16** code unit located at that position.
The **charAt** method takes only one parameter, the index value.

```
const sentence = 'I want to be a developer!';
const index = 7;
console.log(`The character at index ${index} is ${sentence.charAt(index)}`);// t
```

With our knowledge, we can use this method to get us the last character.
Here is the code that will do just that:

```
const lastChar = sentence.charAt(sentence.length-1);
console.log(lastChar);//!
```

One of my favorite String methods is the includes() method. This method takes only one parameter: the substring we want to search the current string. For example, we want to see if we have a particular substring inside the current string and if the method finds it, it will return true or false otherwise.
Here is one example:

```
const occupation = 'Web Developer';
if (occupation.includes('Dev')) {
  console.log(`Yes, it does!`);
} else {
  console.log(`Nope I can't find Dev here!`);
}
```

Output:

```
Yes, it does!
```

What if we want to extract that substring from the original string? Well, to achieve this, there is another method called **slice().** The slice method takes up to two parameters or two indexes. The first one will tell us where to start the extraction, and the second were to stop the extraction. You need to know that the index positions we refer to are not included in the extracted substring.
Let us try this new method:

```
const MyOccupation = 'Developer';
console.log(MyOccupation.slice(0, 3)); // "Dev"
```

Here we got ‘Dev’ back, but what if I don’t supply the second index or where the extraction needs to stop. What do you think will happen? Let us find out:

```
console.log(MyOccupation.slice(2));//2
```

This will return ‘veloper’ because the character at position 2 is ‘v’, and because there is no second parameter, the substring will return the remaining characters from the original string.
If we want to replace a substring inside a string with another substring, we can use the **replace**() string method. This method takes two parameters, the first one will be the substring we are searching to replace, and the second is the new string we want to replace it with.
Example:

```
const originalString = 'mozilla';
const updatedString = originalString.replace('mo','God');
console.log(updatedString); // "Godzilla"
console.log(originalString); // "Mozilla"
```

We will do another example of replace method later when we talk about regular expressions.
There is another very important method called **split()** method. This method will split the strings into an array of substrings. Again, same as the other methods, the split will not change or alter the original string. The split operator takes two parameters. One is the separator, and the other is the limit. These two parameters are optional. If the first operator is not listed, it will return the original string. The second parameter is the limit, which tells the method the number of splits.
Syntax:

```
string.split(separator, limit)
```

An example:

```
let text = "Hello World";
const splittedArray = text.split(" ");
console.log(splittedArray);
```

```
▼ (2) ['Hello', 'World'] i
    0: "Hello"
    1: "World"
    length: 2
  ▶ [[Prototype]]: Array(0)
```

In my first book, where I covered the basics of JavaScript, I explained that if we want to concatenate two strings, we can use the plus ‘+’ operator. Because these are strings and not numbers, the plus operator will concatenate the two strings into one. There are cases where this becomes tricky, but I will not include them here because that is not the goal of this book. Okay, let us discuss a new method called **concat()**, and it does pretty much the same thing as the ‘+’ operator.

Here is one example:

```
const message1 = 'Ana';
const message2 ='maria';
const newMessage = message1.concat(message2);
console.log(newMessage);//Anamaria
```

We have not covered regular expressions yet, but there are two methods that are used for searching a string. The first method is called **match()** method, and it searches the string against a given regular expression. If true, it will return the matches in an array; otherwise, it will return null if no match is found.
Let us do a global search for the substring '**xpr'**:

```
let text1 = "Regular expression";
const result = text1.match(/xpr/g);
console.log(result);
```

output:

```
['xpr']
```

The next method is **matchAll()** method that we can discuss after covering the next important section, the regular expression.
In **ES6** or **ES2015,** a new method was added to normalize the strings. This method will return the string normalized according to one of the four forms that we can pass inside the method as a parameter. This method is known as **normalize**(), and it can take one parameter called 'form', but this is optional. If the from parameter is omitted, the 'NFC' form is used, one of the four main normalization forms. This method will return the **Unicode** normalization form of a given input. But if the input we supply is not a string for some reason, it will be converted into a string.
As I mentioned, if the parameter is omitted, then **NFC** is used as default. The parameter can be from different types:

- **NFC**: Normalization Form Canonical Composition.
- **NFD**: Normalization Form Canonical Decomposition.
- **NFKC**: Normalization Form Compatibility Composition.
- **NFKD**: Normalization Form Compatibility Decomposition.

Now, this might confuse you, but here is one example that I hope will clarify all of your doubts.
As we know, for each character, **Unicode** assigns a unique numerical value. This was called codepoint, remember. Sometimes, a character can be represented by more than one code point.
Have a look at this example that I got it from **MDN** page:

```
let string1 = '\u00F1';
let string2 = '\u006E\u0303';
console.log(string1);  // ñ
console.log(string2);  // ñ
```

We have the same output, but string1 and string2 are not the same because they have different code points. When we compare the two strings, we will get false because of their different lengths.
Here is how we can test two strings using the strict equality ' === ' operator. The strict equality will return **Boolean** if the two operands are equal and it will consider if the operands are from different type:

```
console.log(string1 === string2); // false
console.log(string1.length); // 1
console.log(string2.length); // 2
```

Here is why we need to use the normalize method to convert the string into a normalized form. We can use **NFD** or **NFC** to produce a form that will produce a string that is canonically equal.
Here is the code that will make the two stings equal and will return Boolean true:

```
string1 = string1.normalize('NFD');
string2 = string2.normalize('NFD');
console.log(string1 === string2); // true
console.log(string1.length); // 2
console.log(string2.length); // 2
```

We have covered the most important String methods, and now is the time to focus on something harder called **regular expressions.**

Regular Expressions

This is very interesting but not a favorite topic for most people because it can be hard and confusing, and it takes a

lot of time to learn them. We use Regular expressions to find character combinations in strings that match a particular pattern. Regular expressions are very useful, not just for JavaScript but also for other programming languages. In JavaScript, we have **RegExp** class that represents regular expressions. So the **RegExp**, similar to the String class, has useful methods that will help us perform simple to complex pattern matching activities. **RegExp API** is hard to use if we do not know the regular expression grammar. This syntax/grammar is a complete language of its own. So we first need to understand the grammar, and only after that can we start writing regular expressions. We can construct a regular expression in two ways. The first way is to use the regular expression literal and the second way is to call the constructor function of the **RegExp** object. These objects can be created if we invoke the **RegExp()** constructor. What you will see in practice is that regular expressions are often created using the expression literal syntax. I hope that by now you know that strings literals are created when we have a character or set of characters enclosed within quotation marks. The regular expression literals use a pattern enclosed between slashes '/'.
Now, let us create a **RegExp** object and assign its value to a variable called **myPattern**.
Example:

```
let myPattern = /a$/;
```

The above example creates a new **RegExp** object and assigns its value to the variable **myPattern**. As I explained, the expression literals are delimited by slashes. The literals are instances of the **RegExp** class.
We can achieve the exact same result by calling the RegExp() constructor function.
Here is how that looks in practice:

```
let myPattern = new RegExp('a$');
```

This example will match any string that will end with the letter 'a'. The regular expression can be composed of simple characters or can be much more complex. When we use simple characters to build a pattern, we look to find the exact or direct match. For example, a simple regular expression pattern can be /abc/, and when we use this pattern, we want to find the exact sequence of 'abc' in the strings. For testing and creating regular expressions, there are different ways, there are many websites that offer this functionality free of charge, and the one website I mostly use when I'm in a hurry is called **regexr**. You can write and test different regular expression patterns. There are many more websites like this one, and I'm sure you can find them very quickly because it is pointless to mention them here since every year there is a new one coming out. Okay, now let us test some simple patterns like this:

```
const myString = `Hi, do you know your abc's`;
const regex = /abc/;
//const regex = new RegExp('abc');
console.log(regex.test(myString));
```

In this example, the pattern is very simple and composed of simple characters like 'abc', and as you can see, I have used both ways to create a regular expression. The first one is the literal way, and that is the top one, and the one that is commented is creating regular expression using the function constructor. You will need only one, so that is why the constructor function line is commented, and this way, it will not cause any errors or confusion for us. The string 'myString' contains the exact pattern 'abc' that we are trying to match. If you run this in your browser console, it will return true because the pattern we are looking for is matched. Here we are using the **test()** method that executes a search for a match between the regular expression and the specified string. This method will return true or false (we will talk more about methods later on). Now I want now to test the same example with a different pattern:

```
const myString1 = `Hi, do you know your abc's`;
const regex1 = /ac/;
//const regex1 = new RegExp('ac');
console.log(regex1.test(myString1));//false
```

Why I have renamed my variables, well, because some of the readers will try to run the examples twice in the browser, and it can happen that most of the time, you can get an error because we are using the same variable or we are trying to redeclare the same variable twice. You should always make sure that you refresh your browser to clean the memory before trying new examples or rename the variables, same as I did in the example above, and you will never run into this issue. Okay, the test method will return false this time, but why do we still have the pattern 'ac' in our string? We have 'ac' in the string, but it does not contain the exact substring 'ac', therefore we do not have a match.

Literal Characters

When it comes to alphabetic characters and digits, they all match themselves literally in regular expressions. But sometimes, we need to use nonalphabetic characters as well, and we can use this because JavaScript regular

expression can also support these. Still, we need to use them in combination with a backslash (\). Here is the entire table of these characters:

Character	Matches
\t	Tab
\n	Newline
\v	Vertical tab
\f	Form Feed
\r	Carriage Return
\xnn	Latin character; example \x0A is same as \n
\0	The Null character

Flags/modifiers in Regular Expressions

Let us go quickly over three different flags we can use in regular expression:

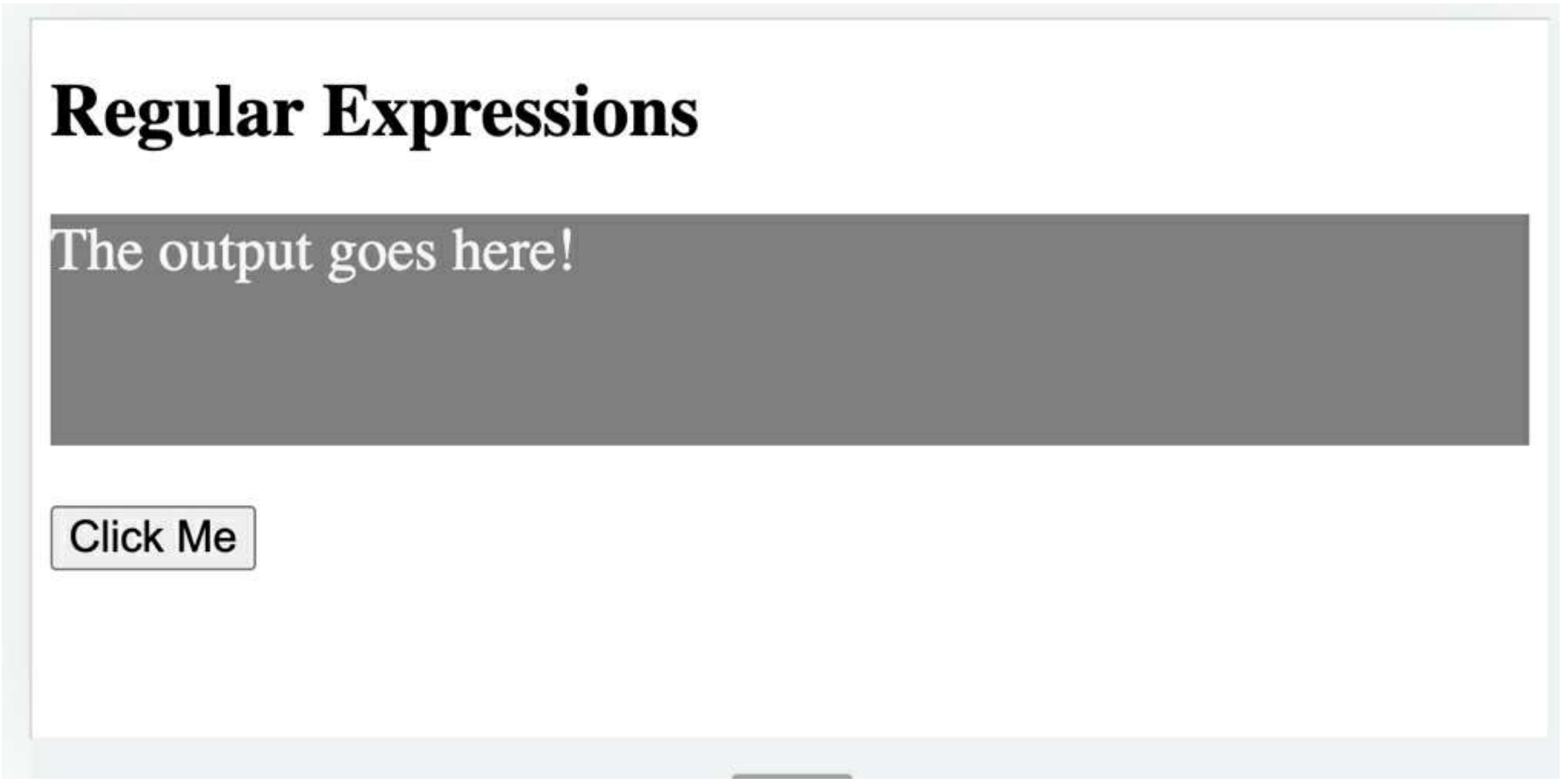
- standard
- g – global
- i – case-insensitive matching
- m – this performs multiline matching
- u – Unicode
- y – sticky

The standard flag/modifier is something that we have already seen in our regular expression pattern, and when we are creating the pattern, we do not need to specify anything. For example, **/abc/** is an example of a regular expression with the standard flag. The next flag is called the global flag and is described with the letter **g**. For example, **/abc/g**, as you can see here, we have used the **g** letter after the slashes to indicate the global mode. This means do not stop at the first match in the document or the string, but go through the rest of the document/string and find all relevant matches. The next flag is the '**i**' flag. This flag will perform case-insensitive matching. For example, our patterns contain only lowercase letters, but what will happen if we have this pattern: /Abc/. This will not result in any match because this is nowhere in our string; therefore, we can use the 'i' flag or /Abc/i to make it insensitive to lower and uppercase letters. The next flag is called '**m'** and stands for multiline. If we have multiple lines of text, it is good to use '**m**' flag to get multiple matches. We have a sticky or **'y'** flag that tells the regular expression to look for a match only at the last index, not anywhere before in the string. The '**u**' flag is another interesting flag introduced in **ES6**, and what it does is recognize the **Unicode** characters in the regular expression. To test out this flag, I have created a new file called 'testRegex.html', and inside this file, I have embedded JavaScript code. I know that some of you will say, oh no, he is using the embedded JavaScript code in the advanced JavaScript book, but I do this only because I will need fewer screenshots and fewer files created at the end. Here is the entire **HTML** markup and JavaScript code:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Regular Expressions</title>
</head>

<body>
   <h1>Regular Expressions</h1>

   <p class='result'
style="color:whitesmoke;font-size:26px;background-color: gray; min-height: 100px;">
   The output goes here!
  </p>

   <button id='clickBtn'
style="color:black;font-size:19px;">Click Me
  </button>

<script>
const btn = document.querySelector('#clickBtn');
const pTag = document.querySelector('.result');
```

```
const myString = `Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects`
                    const regPattern = /ions/;
                    btn.addEventListener('click',(e)=>{
                    const result = myString.match(regPattern);
                    pTag.innerHTML = result;
                    });
   </script>
</body>
</html>
```

If you open this in your browser, it should look like this:



The example is very simple I have **h1** tag, **p** tag, and button in the **HTML** markup. The button has its **id** to be easily targeted in the JavaScript code. The output of what our **JavaScript** code will produce will go inside the **p** tag, which has a class called **result**. I can use class or **ID's** to target these **HTML** elements inside our JavaScript code. What we are interested in here is the code we have in the JavaScript file:

```
const btn = document.querySelector('#clickBtn');
const pTag = document.querySelector('.result');
const myString = `Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects`;
const regPattern = /ions/;
btn.addEventListener('click',(e)=>{
  const result = myString.match(regPattern);
  pTag.innerHTML = result;
});
```

We can see that our regular expression pattern is a very simple combination of characters **/ions/**. Here we have to use the **match()** method, which is very popular because it retrieves the result of the matching a strong against the regular expression pattern we defined. Please do not worry at this stage about methods because they fall into two categories. Ones of them belong to the String class, and the others belong to the **RegExp** class. So, the /ions/ is exactly matched in **mySting** three times, and these are those exact words (express**ions**, combinat**ions**, express**ions**). But because our pattern does not have a modifier, it is a standard one. It will return the first match it will find. If we click on the button on our web page, this will be the output:

Regular Expressions

We can test the ‘g’ global modifier and see what will return after clicking the button. First, we need to add the flag ‘g’ at the end.

```
const regPattern = /ions/g;//global
```

After clicking on the Click Me button, it will return ‘ions, ions, ions’. This is because the global modifier will look into the entire text and find all matches. Let us test the ‘i’ modifier, and here is the regular expression pattern with a small change:

```
const regPattern = /Ions/i;//insensitive
```

As you can see, I have used capital letters, and if now I click on the button, I will still get the output ‘ions’ because of that ‘i’ flag (as you can see, I use two terms here the ‘flag’ and ‘modifiers’, but they are both terms used in regular expression literature). This is happening because of that ‘i’ flag because it makes the matching insensitive to lowercase and uppercase letters.
Let us do even some more interesting examples. For example, we want to find all the words that start with the letter ‘M’ and end with ‘M’. For this example, I have created another file called **testRegex1.html,** it is almost identical, but I changed these two lines:

```
const myString = `Mam, Mom, Mum`;
const regPattern = /M/g;//global
```

Now, if we test it like this, we will get ‘M,M,M’, but I want to have the three words matched, and as we can see, only the middle letter is different for the three words. So, we can use a single dot to replace a letter from a word. I know that sounds confusing but check this example:

```
const regPattern = /M.m/g;//global
```

The output is the three words ‘**Mam**, **Mom**, **Mum’**. So, the single dot will replace one character only, but if we add a longer word like ‘Magm’ into the **myString** and test this out, it will not work.

```
const myString = `Mam, Mom, Mum, Magm`;
```

If you run this, it will not work because the dot will replace only one character from the word and try to do the matching. Obviously, we have more letters here. So, the dot will match any character except the line break. An interesting example is when we are dealing with amounts in dollars. For example, if I input 5.00 dollars like this. Then the regular expression does not know that we use this dot as a decimal point.
I have changed just these two lines in the code, and the previous ones are commented in the file:

```
const myString = `5.00, 510, 570`;
const regPattern = /5.0/g;
```

This will return **‘5.0 ,510 ,570’**, which we don’t want to happen. To fix this, we will need to use backslash ‘\’ or an escaping metacharacter. The character after the backslash will be ignored. Then this will fix our problem. Let us try again and see if the **5.00** only be matched.

```
const regPattern = /5\.0/g;
```

We can test this out if we click on the ‘Click Me’ button, and it seems to be working perfectly. But then someone will ask me, well, what about the quotation marks in the text? They do not need to be escaped because they are treated as regular characters.

RegExp Class

We already know that we can create a regular expression using the constructor function and using a regular expression literal. We have already covered many examples of how we can use the regular expression literal, but now is the time to create a few using the constructor function of the **RegExp** object.
Here is one example:

```
const regExpression = new RegExp(‘ab+c’);
```

Since **ECMAScript 6**, we can have another additional argument that we can pass in the constructor function, and that argument will be for defining the flags, as we mentioned before.
So here is an example:

```
const regExpression = new RegExp(‘ab+c’,’i’);
```

Please remember that we need to separate the arguments with a comma. So the first argument is the regular expression literal, and the second argument is the flag, which is optional. Also, with the **RegExp** class, we can use two methods **test()** and **exec()** method. These two methods will be explained in detail later in this chapter.

Character Classes together with brackets in Regular Expressions

Let us look at this regular expression /**[abc]/**. We can see that we use the square brackets in this regular expression. What do these brackets even mean? Here we are talking about character classes. This gives us the power to combine individual literal characters into classes. Therefore, the regular expression above will match any of the characters between the brackets, meaning any letters a, b, or c will be matched. The opposite of this is to use the caret symbol together with the brackets like this **[^abc].** This means that we are trying to match any character that is not between the brackets or that is not a, b or c. Another famous expression is this [0-9], which means it finds any number/digit between the brackets. The hyphen indicates that we are specifying a range of characters. Opposite of this will be this expression **[^0-9]**, meaning it will find any non-digit character, not included in the brackets. For example, we can use the hyphen to match any lowercase character from the Latin alphabet like this **/[a-z]/**. We can use the same logic to match any uppercase from the Latin alphabet using the **/[A-Z]/**. We can move one step further and match all lower and uppercase characters from the alphabet using this regular expression: **/[a-zA-Z]/**. If we want to match all of the digits and characters from the alphabet, we can use this regular expression: **/[a-zA-Z0-9]/**.
Let us see some code, and hopefully, this will become very clear. If you want to find the exact code, please look into the file called **testRegex2.html -** (I used the same HTML5 markup when we tested flags in the previous section, but we will make some minor changes in the JavaScript code).
I have changed only these two lines:

```
const myString = `Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects`;
const regPattern = /[J]/g;//global
```

What this means, try to match the letter ‘J’ and include only that letter in the result. So, if we run this by clicking on the button, it will indeed give the single letter ‘J’ back. But if we try to match a set of characters like this:

```
const regPattern = /[JaS]/g;//global
```

then the result will be something like this:

```
a,a,a,a,a,a,a,J,a,a,S,a,a,a
```

This is happening because it matches the letters we provide between the brackets. And if we want the result to be with all of the letters from the **myString** except the [JaS], then we can use the caret symbol like this:

```
const regPattern = /[^JaS]/g;//global
```

The output will be:

```
R,e,g,u,l,r, ,e,x,p,r,e,s,s,i,o,n,s, ,r,e, ,p,t,t,e,r,n,s, ,u,s,e,d, ,t,o, ,m,t,c,h, ,c,h,r,c,t,e,r, , , , , , , , ,c,o,m,b,i,n,t,i,o,n,s, ,i,n, ,s,t,r,i,n,g,s,., ,I,n, ,v,c,r,i,p,t,,,
,r,e,g,u,l,r, ,e,x,p,r,e,s,s,i,o,n,s, ,r,e, ,l,s,o, ,o,b,j,e,c,t,s
```

This is interesting, right. This is the same thing for the numbers, but I will not include examples for those.

Regular Expression character classes

Here is a table of the most important special metacharacters:

\t	It matches the tab character
\v	It matches the vertical tab character
\r	It matches the carriage return character
\f	It matches the form feed character
\b	It finds a match at the beginning or the end of a word
\B	It tries to find a match but not at the beginning or the end of a word
\s	It matches a whitespace character
\S	It matches a non-whitespace character
\d	It matches any digit character, any ASCII digit, or same as **[0-9]**
\D	It matches a non-digit character
\w	Tries to find any ASCII word character, same as if we wrote [**a-zA-Z0-9_]**
\W	Tries to find a non-word character or same as [^**a-zA-Z0-9_]**
[…]	Matches any character between the brackets
[^…]	Matches any character that is not in between the brackets
\uhhhh	Matches a UTF-16 value WITH four hexadecimal digits.

I have already mentioned the escape backslash character. Well, to test out these special characters, you need to use a backslash in front of them. To test these special characters, I have created a file called **testRegex3.html**, and I added these lines of code there:

```
const myString = `Regular expressions are patterns used to match character combinations in strings 100% true. In JavaScript, regular expressions are also objects!`;
const regPattern = /\w/g;//global
```

As you can see in **myString**, I have added ‘100%’ and exclamation ‘!’ mark at the end of the string. If you look at the table, the lowercase ‘\w’ tries to match all of the word characters. And if we click on the button, we will get the following output – sorry it is so long – only without the percentage ‘**%**’ sign and exclamation mark.

```
R,e,g,u,l,a,r,e,x,p,r,e,s,s,i,o,n,s,a,r,e,p,a,t,t,e,r,n,s,u,s,e,d,t,o,m,a,t,c,h,c,h,a,r,a,c,t,e,r,c,o,m,b,i,n,a,t,i,o,n,s,i,n,s,t,r,i,n,g,s,1,0,0,t,r,u,e,I,n,J,a,v,a,S,c,r,i,p,t,r,e,g,
```

As you can see in the output, the whitespaces are not included. Let us now try to use the capital letter W and observe the output:

```
const regPattern = /\W/g;//global
```

Output:

```
, , , , , , , , , , , , , , , , , , ,%, ,., , ,,, , , , , ,!
```

So it returns all of the non-word characters like ‘% and !’. Please do not get confused. It does not return a comma, but it returns the whitespaces in between. I will not test each of them but let us test what will happen if we use the lowercase ‘d’:

```
const regPattern = /\d/g;//global
```

The output includes all of the digits it can find, and that is the ‘100 percent I used’:

```
1,0,0
```

Finally, what if we want to include a backslash character in our regular expression? Well, if we want that, we must escape even the backslash with another backslash. This will be the regular expression that matches any string that contains a backslash / \\ /.
And that is all for this section. Please feel free to test out what will happen with the other special characters that I have included in the table, but if you can’t, it should be self-explanatory if you only read what they do.

Unicode: flag "u" and class \p{...}

Since **ES2018,** if we want to handle Unicode characters correctly in our regular expressions, we can use the **u-flag**. If we use the u-flag, then the character classes **\p{..}** and the negation classes **P{…}** are also supported and available. Every character in **Unicode** has properties that are defined by the **Unicode** standard. For example, if the character has a Letter, it means that the character can belong to any alphabet. But if the property is a **Number, it means that it can be a digit and** belong to the Arabic or Chinese alphabets. Remember when we use the **\p{…}** classes, then the regular expression must also include the '**u**' flag at the end like this example:

```
/\p{L}/gu
```

The **Number** and **Letter** properties have their own aliases, so the single letter **L** will stand for **Letter,** and **N** will be for Number.
Here is one example so you can understand how you can create regular expression using u-flag and p classes:

```
let mixedString = "Hi ১৪ 必 ㅂ ㅅ ";
let regex1 = /\p{L}/gu;
let regex2 = /\p{L}/g;
console.log( regex1.test(mixedString));//true
console.log( regex2.test(mixedString));//false
```

The example above has 4 kinds of letters from the English, Gregorian, Chinese, and Korean alphabets. As you can see in the first test, the result we are getting is true because we are using the \p{…} regular expression and u flag at the end. But the second console log will give us false because we are trying to search \p{ …} without the **u** flag that enables the support of **Unicode** in regular expressions. We saw from the previous table that \d character class will match any **ASCII** digits, right? Now, if we want to match a decimal digit in any language, we can use the **Decimal_Number** property like: **/\p{Decimal_Number/u**. Remember, we can use the capital \P{…} to achieve negation, meaning it will not match any decimal digit but will match any other character in any language. For example, we want to target/match letters from any language or even the Chinese hieroglyphs. We can use a special Unicode property called **Script**. This property refers to the writing system and can take different values like Chinese, Cyrillic, Greek, Arabic, etc. For the Script property, the alias is **sc**, and then we need to use a value. Here is one example for the Macedonian language that uses the Cyrillic alphabet:

```
let mixedString = "Истражувањето на компанијата";
let regex1 = /\p{sc=Cyrillic}/gu;
console.log( regex1.test(mixedString));//true
```

Using the English letters inside will give us false results because the writing system detects the Cyrillic alphabet. Another interesting example is when we want to use foreign currency characters like $, €, ¥, then we can use another Unicode property called **Currency_Symbol**, and the short alias is \p{Sc}.
Here is the example I like you to consider, but please take a note that I use **\d** character as well because I want to include digits together with the currency symbol:

```
let  mixedString = `$5, €10, ¥109`;
let regex1 = /\p{Sc}\d/gu;
console.log( regex1.test(mixedString));//true
```

Quantifiers in Regular Expressions

In this section, you will learn about the quantifiers used in regular expressions. The quantifiers indicate the number of characters that can be repeated in a string. For example, with the regular expression syntax we know so far, we can create a pattern to match four-digit numbers like this: /\d\d\d\d/. But if we need a regular expression to specify how many times an element should be repeated, we could use special characters known as quantifiers. This table you can also find on the MDN website.

+	This means that it will match one or more occurrences of the item
*****	This means that it will match zero or more occurrences of the item
?	This matches zero or one occurrence of the item
{N}	It matches the exact N number of occurrences of the specified item
{N,}	It matches the N or more number of
{N,M}	It matches any string that contains N number of

	occurrences, but it can be no more than M

Let us write some code and test the **RegExp** quantifiers. The first one in the table is the +, and it will match the preceding item one or more times. For example, let us test to see how many times the character ‘a’ will be matched in the string:(the code you can find in file **testRegex4.html**)

```
const myString = `Regular expressions are awesome!`;
const regPattern = /a+/g;
```

The output is (a, a, a) because the item ‘a’ is three times present in the **myString** above (Regul**a**r expressions **a**re **a**wesome). Let us see what will happen if we replace the ‘+’ symbol with ‘*’ in the pattern.

```
const regPattern = /a*/g;
```

The output in this case is bit strange:

```
,,,,,a,,,,,,,,,,,,,,a,,,,a,,,,,,,
```

This is because in the final result, not only the character ‘a’ is included but the spaces as well that precede them. From the table, you can see that this quantifier will find all occurrences of the character and where it is also positioned, but it will look for zero or more occurrences. Try this pattern to test for the character ‘d’ and observe the result.
This is the code and output:

```
const regPattern = /d*/g;
```

```
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
```

So there is no character ‘d’ in the string, but it will still return the rest of the characters like whitespaces or whatever precedes them, and it is expected to match nothing. Whenever using the * and ?, you should know that can match the zero instances and whatever precedes them. Okay, the next quantifier ‘?’ works similarly to the ‘*’, because it will look for zero or more occurrences. For example, if we want to match the **be**, **bee,** and **bees** simultaneously, we can construct this regular expression pattern: **/be+s?/**
Okay, let us move forward, and please take a look at this new example:

```
const myString = `Regular expressions are awesome!`;
const regPattern = /s{2}/g;
```

This quantifier will match the exact number of occurrences of the item/character we are looking for. For example, **/s{2}/** does not match the ‘**s’** in ‘awesome!’ but it matches the two occurrences of the character ‘**s’** in ‘expre**ss**ions’. This is happening because we put number two inside of the curly braces, and if we change this to **/s{3}/,** it will not work because there is nowhere in the string this sequence of three ‘sss’ letters. For some reason, if we want to match all of the occurrences of the character ‘**s’,** then we can do this **/s{1}/,** and it will give us all of the occurrences. An interesting point to note is that the quantifiers like ***** and **+** are known as ‘**greedy’,** which means that they will try to match as many characters of the string as possible. When we combine these with the ‘?’ quantifier, it will no longer be treated as greedy because it will stop as soon as it finds a match. Let us go even one step further. For example, let us write an expression to match between three to four digits:

```
let numbersString = "1 12 999 34 9888 687665";
let regex1 = /\d{3,4}/g;

console.log(numbersString.match(regex1));
```

```
▶ (3) ['999', '9888', '6876']
```

Here I have used the **match()** method to match a string against a regular expression we created. The matching string is **numbersString,** and the regular expression is **regex1** in this example. Another example I want to show you is when we need to match a particular word. For example, it can be JavaScript in some long text, and we need to match it with one or more spaces before and after the word. Take a look at the step-by-step process (‘you can find the code in **regex2.js** file’):

1) We first do an exact word match like this:

```
let longString = `As we know, JavaScript is a scripting language that enables you to create dynamically updating content,
control multimedia, animate images, and pretty much....`;
let regex2 = /JavaScript/g;
```

```
console.log(longString.match(regex2));
//Output:['JavaScript']
```

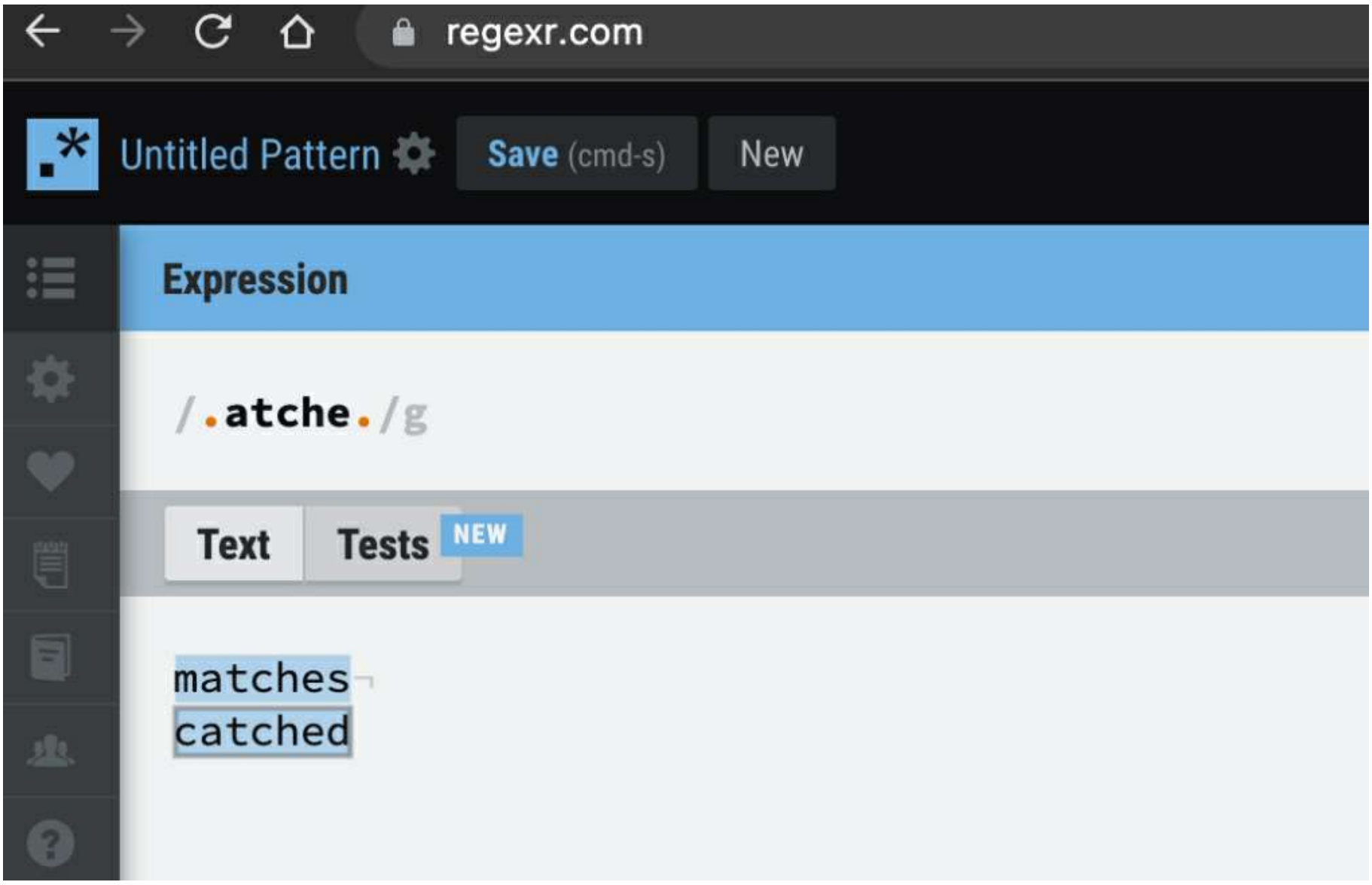
2) The second step is to include the whitespaces before and after. The special character for white spaces was 's' if you look at the previous table, and we need to combine it with the '+' symbol so we can specify that we are saying it has to have at least one whitespace before after the JavaScript.

```
let longString = `As we know JavaScript is a scripting language that enables you to create dynamically updating content,
control multimedia, animate images, and pretty much....`;
let regex2 = /\s+JavaScript\s+/g;
console.log(longString.match(regex2));
//Output:[' JavaScript ']
```

Okay, let us summarize what we know so far about regular expressions. In a regular expression, we have a few reserved characters that they have special meanings like:

- .
- +
- *
- ?
- { }
- |
- ( )
- [ ]
- \
- ^
- $

The symbol matches any one single character. For example, if we want to match these two words 'matches, cached using one single pattern, we can use this regular expression **/.atche./**. The + symbol means the repetition can be 1 or more times and the * symbol indicates 0 or more occurrences of the character.
For this example, I did not show you how we can do it inside JavaScript. It is simple, and all you need to do is create regex literal and use the test method I used in the previous example. If the test method returns true, the substring we are looking for exists in the larger or original string. But if you just want to test if your regular expression matches the substring, you are looking you can simply use the **regexr** website I mentioned before. In this website, if we have a correct regular expression, then the substring will be colored in blue like in this example:



Alternatives

Regular expressions have special character when we want to specify alternatives. I like to think about regular expression alternatives as to the logical ‘or’ operators because they give us the option to choose. The character we use to separate alternative is the well-known ‘|’. Please take a look at the following example (the same code is in the **regex3.js** file):

```
let alphabetString = "a b c d e f";
let regex1 = /a|c|m/g;
console.log(alphabetString.match(regex1)); //Output:  ['a', 'c']
```

The regular expression above will try to match the string ‘a’ or the string ‘c’ or ‘m’. As you can see, it will only match the letters ‘a’ and ‘c’ but not the letter m because it is not included in the list. The matching will start from left to right. But there is a problem with the alternatives. For example, having more complex matching, meaning two or more letters to match, will not always produce the result you expect.

Check out the following code:

```
let alphabetString1 = "a b ac d ae f";
let regex2 = /a|ac|ae/g;
console.log(alphabetString.match(regex2)); //Output:  ['a']
```

As you can see from the output, it only matched the first letter ‘a’, and the right ones are ignored even though we wanted to be included in the result. So, the matching starts from left to right, and as soon as it finds the first one, the rest are ignored. So please take this into a consideration when using alternatives.

Grouping

When we want to treat multiple characters as a single unit, we can use grouping. The grouping in the regular expressions can be achieved with parentheses. Any subpatterns inside the parentheses will be treated as a group. Let us have a look at this example that will be able to match a website domain:

```
let domainList= "google.com apple.com apple.com.au support"
let regexp1= /(\w+\.)+\w+/g;
console.log(domainList.match(regexp1));
```

The output will be:

```
['google.com', 'apple.com', 'apple.com.au']
```

As you can see from the output, the grouping example works, but this is a very simple domain matching a regular expression. For example, it will not catch a domain containing a hyphen, such as this URL '**myer-online.com**', because the hyphen is not included in the pattern. If you want to include those domains, we need to replace the \w with the **([\w-]+\.)**. The groups are very important for regular expressions because of these two things:

1) They will allow us to get part of the match and store it as a separate item in the final results array.
2) If the parentheses are combined with quantifiers, then those quantifiers will be applied to all of the items in the parentheses

Now, what is the connection between parentheses and arrays? Well, when we use the method **match(),** then we will get back an array that contains something like this:

1) At the position/index 0, it will be the entire match string
2) At position/index 1, it will be the contents of the first parentheses.
3) At position/index 2, it will be the contents of the second parentheses
4) …
5) …and so on…

So the groups are numbered by their opening parentheses, and they are numbered from left to right. Therefore the group matches are placed as separate items inside the array. Okay, as an example, let us match HTML5 tags. We know HTML5 tags are enclosed in angle brackets like this: < >. So, if we want to get the entire tag with the brackets plus the inside content, we can write a regular expression like this:

```
let html5Tags = '<h1>';
let tagResults = html5Tags.match(/<(.*?)>/);
console.log(tagResults);//Output: ['<h1>', 'h1', index: 0, input: '<h1>', groups: undefined]
console.log(tagResults[0]);//Output: <h1>
console.log(tagResults[1]);//Output: h1
```

As you can see, the tag content h1 in our case is now enclosed by parentheses and will be treated as a separate variable.

Nested Groups

You should know that we also have situations where the parentheses are nested inside each other. To explain how nesting works, I will keep working with the **HTML5** tags. We know that each **HTML** tag has content, and we can specify many things there, like classes and ids. Check out the example for defining the **HTML5 h1** tag with its name and the class attribute:

```
<h1 class='headingOne'>
```

Ok, let us go step by step and create separate parentheses for each of them (name and class attribute):

1) For h1, we can write the regular expression: **([a-z\d{1}]+)** – we can see one pair of parentheses used here
2) For the class=**'headingOne',** we can write the regular expression: **([^>]+)** – this means match any character except the '>' **,** and then we have another set of parenthesis
3) Let us combine points one and two and add include the whitespace between them:
   ([a-z\d{1}]+)\s*([^>]+)
4) The final step is to match the whole tag content adding one more pair of outer parentheses, we also can add the < > to match the angle brackets:
   <(([a-z\d{1}]+)\s* ([^>]+))>

Here is now the entire code for matching the h1 tag:

```
let html5Tags1 = '<h1 class="headingOne">';
let regexp2 = /<(([a-z/d{1}]+)\s*([^>]+))>/;
let tagResults1 = html5Tags1.match(regexp2);
console.log(tagResults1[0]);
console.log(tagResults1[1]);
console.log(tagResults1[2]);
console.log(tagResults1[3]);
```

And this is the output:

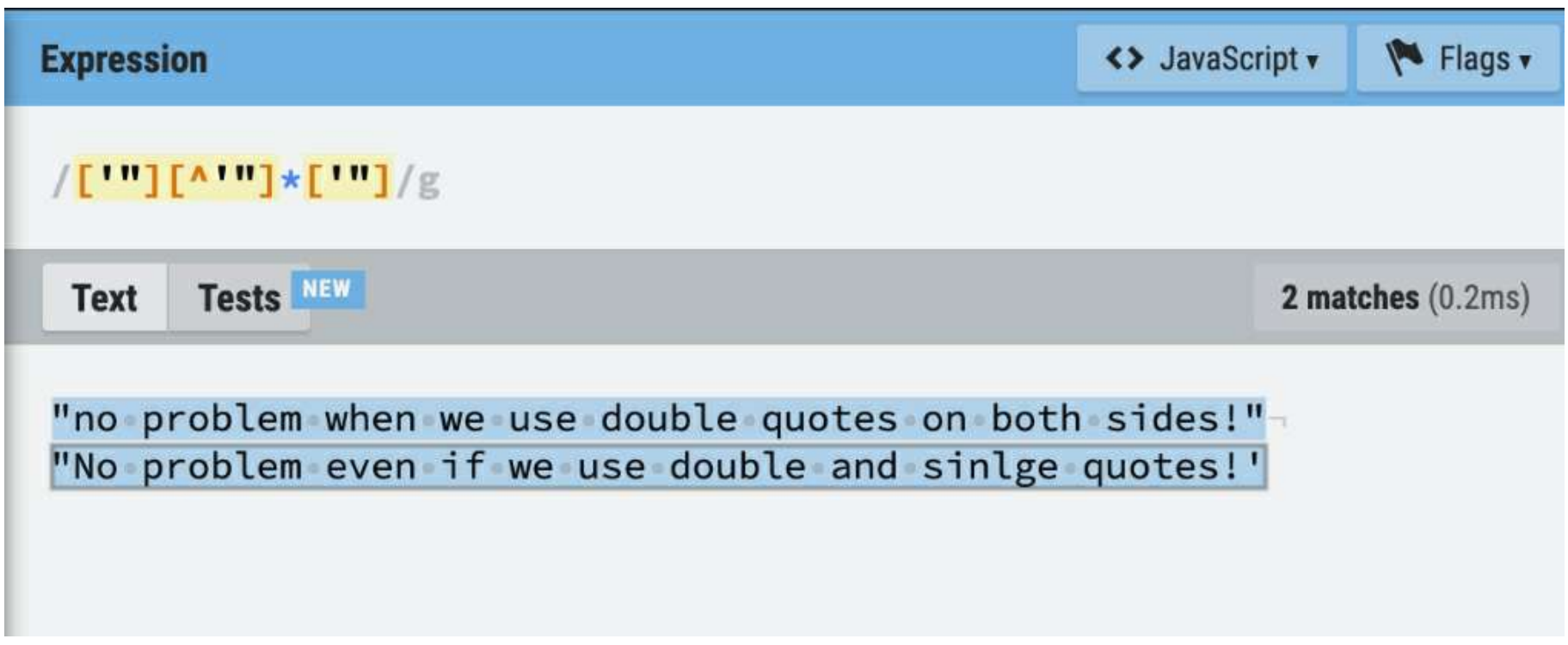
```
> let html5Tags1 = '<h1 class="headingOne">';
  let regexp2 = /<(([a-z/d{1}]+)\s*([^>]+))>/;
  let tagResults1 = html5Tags1.match(regexp2);
  console.log(tagResults1[0]);
  console.log(tagResults1[1]);
  console.log(tagResults1[2]);
  console.log(tagResults1[3]);
```

```
<h1 class="headingOne">                                    VM515:4
h1 class="headingOne"                                      VM515:5
h1                                                         VM515:6
class="headingOne"                                         VM515:7
```

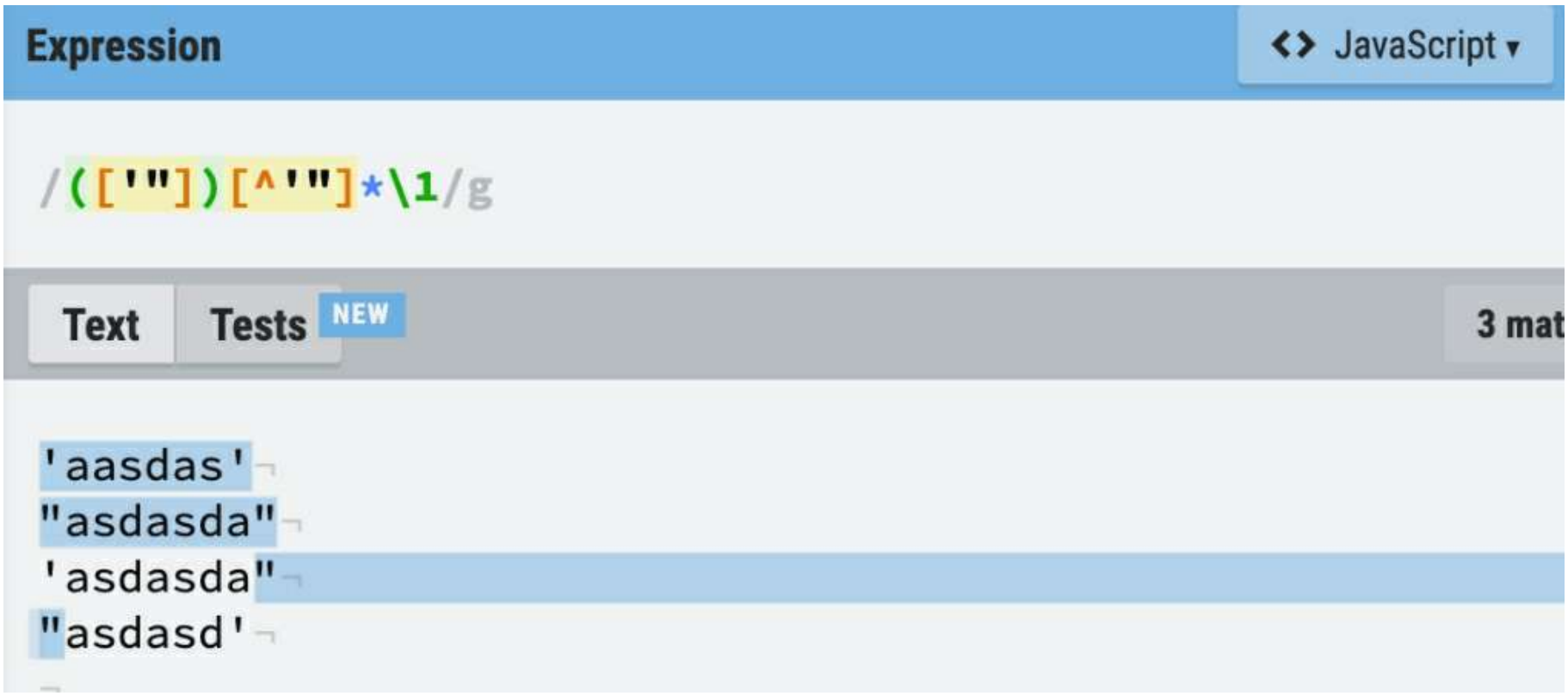
From this example, we can clearly see 3 groups of parentheses, one is for the entire tag, and then we have two separate, one for matching the h1 and one for matching the class attribute and the content in between the double-quotes. Before we move on something else let us do one example that includes the single and double-quotes. If we want our regular expression to match zero or more characters within a single or double quote, then we need to write it like this:

```
/['"][^'"]*['"]/;
```

But there is one problem with this regular expression, and the problem does not care if we open the string with double quotes and if we close it with a single one. It will be easier to explain this example if I use the **regexr** website.



See the example above. It will match both of the texts. If we want both quotes to match, we need to make one small change: adding \1 in the regular expression. This will ensure that the closing and opening quotes match.



Named groups

Imagine if we have a more complex pattern where we must keep track of all of our parentheses. Then this will be an extremely difficult and pointless process, but here is an option to fix this by giving the parentheses their names. It is a good idea to name your parentheses, and please use meaningful names because after a while, when you come back to your own code, you will not know what those names mean.
The syntax for naming parentheses is by adding question mark and meaningful name, like this '?<**name**>'. This feature was standardized in **ES2018**, and it helps the developers now to have an easier way to express and understand the regular expression patterns they build. This feature was not working properly until recently, but since 2020 is part of every modern browser and Node. Let us create a regular expression for a particular date in this format: 'date-month-year'. Please check out the following example:

```
let dateExpression = /(?<day>[0-9]{2})-(?<month>[0-9]{2})-(?<year>[0-9]{4})/;
let dateString = "29-09-2022";
let theGroups = dateString.match(dateExpression).groups;
console.log('The Year Is: ' + theGroups.year); // 2022
console.log('The Month Is: ' + theGroups.month); // 09
console.log('The Day Is: ' + theGroups.day); // 2022
```

This is the output:

```
The Year Is: 2022
The Month Is: 09
The Day Is: 29
```

As you can see, we created a regular expression to match a particular date in some format. All of the groups are accessible in the property called '**.groups**'. With this approach, there is only one small problem. If the string we are working on contains only one date, this approach will work fine, as we already saw in the previous example, but it will not get all of those dates if the string contains more than one date. So to fix this, we need to use the 'g' flag, which stands for global, to look into the entire string. We also need to use **matchAll**() to get the full matches with the corresponding groups. So **matchAll**() method will return an iterator of results, and we can iterate this result with a simple **for** loop.
Okay let us do this example and we are done with parentheses:

```
let dateExpression1 = /(?<day>[0-9]{2})-(?<month>[0-9]{2})-(?<year>[0-9]{4})/g;
let dateString1 = "29-09-2022 19-11-2023";
let theGroups1 = dateString1.matchAll(dateExpression1);
for(let theGroup of theGroups1){
   let {year, month, day} = theGroup.groups;
   console.log(`The Year Is: ${year}`); // 2022
   console.log(`The Month Is:  ${month}`); // 09
   console.log(`The Day Is: ${day}`); // 2022
}
```

And the output is:

```
The Year Is: 2022
```

```
The Month Is:  09
The Day Is: 29
The Year Is: 2023
The Month Is:  11
The Day Is: 19
```

Methods of the RegExp Class

So far, we have used some methods, but I wanted to summarize what methods belong to the **RegExp** class. One of the methods we have used in the beginning was the **test()** method. This method would return true if there were a match.
Here is one example of the test method:

```
/([0-9]{2})/.test(‘super 08’); //true
Or
/(^[0-9]{2})/.test('super 08'); //false
```

Another method called **exec**() would return an array containing the first matched subexpression or null if there were no matches.
Here is one example:

```
const result = /[0-9]+/.exec('super 08 12'); //true
undefined
result
▶ ['08', index: 6, input: 'super 08 12', groups: undefined]
```

From the output, we can see that we have an array of one element, which is ‘08’. We also have two properties like index and input. The index is where the matching happened. Remember, the matching process started from left to right and index zero. Here is a table that will help you figure out how we got the index to be six.

String	s	u	p	e	r		0
Index	0	1	2	3	4	5	**6**

The next property called input will hold the entire argument we are passing to the function exec and, in our case, was ‘**super 08 12**’.

String Methods on Regular Expressions

So far, we have seen the grammar and syntax we need to create regular expressions, but now we should move forward and see how regular expressions can be used in everyday JavaScript. We have only used a few of the strings methods until now, but we need to go deeper in the **RegExp API**.

Search Method – search ()

This method is by far the simplest one you can use. If you search for this method online, you will find more theory than practical examples and we as developers always want to learn from examples. So, simply put, this method performs a search to find a match between a regular expression and the String object. This method will give you the first position of the character where the matching happened. If there is no match it will return a value of -1.
Here is the syntax:

```
search(regexp)
```

We can see the search method takes only one parameter, a regular expression object. If a non-regular expression object is passed, it will be implicitly converted into a regular expression object with the help of the constructor function ‘new RegExp(regexp)’.
Please consider the following example:

```
let simpleString = "my name is Jack Rayan!";
let regExp = /[A-Z]/g
console.log(simpleString.search(regExp));
```

In this example, the output will be 11 because the regular expression says find where we have used the uppercase letter in the **simpleString**. So, the character **J** from **Jack** is located in the 11th position; therefore, we have the output 11. Remember, you need to count the whitespaces as well.
What do you think will happen if we change the same example:

```
let simpleString = "my name is jack rayan!";
let regExp = /[A-Z]/g
console.log(simpleString.search(regExp));
```

This will return -1 because there are no uppercase letters in the simpleString.

Replace Method – replace ()

The **replace**() method, similarly to the search method, performs a search, but when it finds the match, it will also do a replace operation. This replacement method will replace only the first match. To perform multiple replaces, you need to use the global 'g' flag. The first argument of the replace method can be a string instead of a regular expression. If we use a string, then the method will search the entire string literally, and it will not convert the string argument to a regular expression as we had this in the search method. Remember, this was done automatically by the **search**() method, but it will not happen in the **replace()** method.
Finally, here is **replace**() method example, using a Regular expression to match:

```
> let simpleText='I love javaScript, but javaScript can be
  difficult to learn!';
  let output = simpleText.replace(/javaScript/g, "JavaScript");
  console.log(output);
  I love JavaScript, but JavaScript can be difficult   VM1465:3
  to learn!
< undefined
```

As you can see from the example, the replace method returns a new string with the value(s) replaced. I want you to note that it will not change or affect the original string stored in **simpleText**. As discussed earlier, the replace method can literally replace the value in a string with another string value.
Here is the example of **replace**() using a string to match:

```
> let sampleText = "Hi my name is Andy!";
  let result = sampleText.replace("Andy", "Thomas");
  console.log(result);
  Hi my name is Thomas!
< undefined
```

In this example, we are replacing **Andy** with **Thomas**. Okay, let us do one more example where we want to perform replace at multiple places, and that can be achieved because of the global flag we already discussed:

```
const regExp3 = /\d{4}/g
const myText = "I was born in 1990. Do you know anyone that is born in 1990?";
const output1 = myText.replace(regExp3, "1989");
console.log(output1);
```

The output will be:

```
I was born in 1989. Do you know anyone that is born in 1989?
```

Let us cover something even more interesting and useful. Imagine that you want to replace 2 different substrings within the existing string with a new string. Well, someone will say this is easy. We can use two replace methods on that string, one after the other. But, instead of doing that, we can do a method chaining. We can achieve this with a single line of code like in this example:

```
let testString1 = "This James Bond movie was great. I love watching James Bond movies with my brother.";
let output2 = testString1.replace(/James Bond/gi, "Star Wars").replace(/brother/gi, "girlfriend");
console.log(output2);
```

The output will be:

```
This Star Wars movie was great. I love watching Star Wars movies with my girlfriend.
```

Another advanced feature is that the replace method can accept a replacement function as a second parameter. You should know that the function's job is to return a value, and that value will be used as a new string that will replace the matches. Here is an example of a replacement function:

```
> const testString2 = "I was born in 1990. Do you know anyone
  that is born in 1990?";
  const regExp3 = /\d{4}/g;
  function replacerFn() {
    return "1987";
  }
  console.log(testString2.replace(regExp3, replacerFn));

  I was born in 1987. Do you know anyone that is born  VM522:6
  in 1987?
```

But this does not end here. The replacement function can take a few more arguments. Here is the table of arguments that we can pass into the function:

Parameters	Description
match	This is the string that was matched by the regex pattern
P1, P2	The matches of all groups
offset	The offset of the match
string	The entire string

The first example will show you how the matched string is replaced using the replacement function:

```
const testString3 = "I hate JavaScript and I hate RegEx as well!";
const regExp4 = /hate/g;
function replacerFn1(match) {
  console.log(match);
  return `love`;
}
console.log(testString3.replace(regExp4, replacerFn1));
```

Output:

```
2 hate
I love JavaScript, and I love RegEx as well
```

The next example will show you the captures of the capturing group by our regular expression:

```
const testString4 = "I hate JavaScript and I hate RegEx as well!";
const regExp5 = /hat(e)/g;
function replacerFn2(match,p1) {
   console.log(`The matched string ${match}, capturing ${p1}`);
  return `love`;
}
console.log(testString4.replace(regExp5, replacerFn2));
```

Output:

```
2 The matched string hate, capturing e
  I love JavaScript and I love RegEx as well!
```

This following will show you the offset of the matches, and we have two in our case. One of the matches is located in index 2, and the other match that will occur for the substring ‘hate’ is located on index 24:

```
> const testString5 = "I hate JavaScript and I hate RegEx as
well!";
const regExp6 = /hat(e)/g;
function replacerFn3(match,p1,offset) {
    console.log(`The matched string ${match}, capturing ${p1}
at this index ${offset}`);
    return `love`;
}
console.log(testString5.replace(regExp6, replacerFn3));
```

```
The matched string hate, capturing e at this index 2   VM167:4
The matched string hate, capturing e at this index 24  VM167:4
I love JavaScript and I love RegEx as well!            VM167:7
```

Finally, here are all of the parameters we can use in the replacement function, including the entire string:

```
> const testString6 = "I hate JavaScript and I hate
RegEx as well!";
const regExp7 = /hat(e)/g;
function replacerFn4(match,p1,offset,string) {
    console.log(`The matched string ${match},
capturing ${p1} at this index ${offset} in
${string}`);
    return `love`;
}
console.log(testString6.replace(regExp6,
replacerFn4));
```

```
The matched string hate, capturing e at this   VM224:4
index 2 in I hate JavaScript and I hate RegEx as well!
The matched string hate, capturing e at this   VM224:4
index 24 in I hate JavaScript and I hate RegEx as
well!
I love JavaScript and I love RegEx as well!    VM224:7
```

Summary

Congratulations! This was the first chapter, and I hope you enjoyed reading it. This chapter started with methods we use on Strings. We ended up learning a ton of information about regular expressions, their syntax, grammar, how to create them, and what kind of methods we can use in different scenarios. I know that the RegExp part can be quite challenging, but If you did not get it on your first reading, you could always return and read it once more.

Chapter 2 - Asynchronous Programming

In the past, most computer programs ran continuously without stopping until they got the result. Today, the way we write these computer programs has changed, and most of these programs are executed ***asynchronously***. The difference between ***asynchronously*** and ***synchronous*** is evident. In this chapter, I will explain where those differences are, and hopefully, you will develop a deep understanding of the power of asynchronous programming. Basically, with ***asynchronous*** programming, we focus on a task or set tasks that need to be executed at some particular point in the future. JavaScript is a single-threaded language, and when we have single-threaded language, it means that the tasks are performed in a sequence or in order. Before the compiler can start executing a new task, the previous one must return or must finish. And here is where the problem is, imagine if something happens in the code like an error or bug and we need to wait for the response to come in, then the rest of the code/tasks needs to wait until the current one is resolved and this is known as **code-blocking**. In this chapter, we will learn in-depth about ***Promises***, ***async***, ***await***, and ***for/await***. Why is asynchronous programming so important? As you know, JavaScript is an event-driven programming language, meaning it waits for the user to take some action or action. Therefore, the servers keep waiting on user requests before processing those requests and sending the responses back. Let us learn all of this in the following sections. All of the files and examples for this chapter are in the downloaded files, **chapter2** folder.

Callbacks

Let's start this chapter by explaining what callbacks are. We can achieve asynchronous programming in JavaScript if we use the callbacks. A callback is a function that is passed as an argument to another function. The idea of the callback function is ‘I will call you back later’. You should know that the functions in JavaScript are executed in the sequence they are called, not in the sequence they are defined in. Okay, without confusing you, I will start from the beginning, why sequence control is so important in JavaScript. The sequence of control allows us to control the sequence when a function needs to be executed. For example, let us say that we want to create a function that will do some basic arithmetic operation like the sum of two integer numbers. After the function returns the result, we want to call another function to display the result to the user. So now we have two functions, one mainly for calculation, and the other is to display the user the result. Therefore we know their sequence of execution. Of course, you cannot start showing the results without first calling the calculation function.
Here is the entire example including the HTML5 markup and the JavaScript Code (code in **sequence.html** file) :

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Sequence control in JavaScript</title>
</head>
<body>
   <h1>Sequence control in JavaScript</h1>
   <h2 id="output">Output goes here:</h2>

   <form name="myForm" action="#">

      <label name="firstOperand">First Operand</label>
       <input type="text" name="firstOperand" id="first"><br><br>

       <label name="secondOperand">Second Operand</label>
       <input type="text" name="secondOperand" id="second"><br><br>

       <button id='calculate'>Calculate</button>

   </form>
   <script>
      let calcBtn = document.querySelector('#calculate');
calcBtn.addEventListener('click',(e)=>{
e.preventDefault();//to prevent the form to submit
let a = parseInt(document.querySelector('#first').value);
let b = parseInt(document.querySelector('#second').value);

                function printThis(theSum){
                        document.getElementById("output").innerHTML = theSum;
 }

        function calculateTwo(a, b) {
           let sum = a + b;
           return sum;
        }

        let result = calculateTwo(a, b);
        printThis(result);
      });

   </script>
</body>
</html>
```

If we open this file in our browser, we can type 5 and 9 in the two input fields, then we can click on the calculate button, and the result should be something like this:

Sequence control in JavaScript

14

First Operand 5

Second Operand 9

Calculate

Okay, let me explain what is happening in the code between the script tags. First of all, when we have a form like this with the submit button, the form's default behavior is to be submitted somewhere, usually another page. We usually specify attributes in the form tag like **action** and **method,** and for the action attribute, we usually define a path that will go to some file. In our case, we do not need such behavior, so I do not have any attributes in the form except a form name. This example is not designed to prevent users from submitting empty fields or to check if the user typed letters instead of numbers; that is not the idea here. Therefore, please do not expect a mechanism to check for user mistakes. If you test the same example, please use numbers in the fields. The question here is how will I be sure when the button of the form is clicked? I'm using an event listener that will listen when the user clicks on the button to catch this. I will explain how this works in the future. Just please follow me. Okay, imagine that the button is clicked, and the form will try to use its default behavior, and that is to submit; therefore, the first thing I should do is turn off this default form behavior using the **e.preventDefault()** method. This will not allow the form to be submitted, and now I can get the values of the first and second operand and store them in variables. Remember that even if we put numbers in the fields in the backend, we still will get strings. To convert the string to a number, I use the **parseInt** function that will parse the string argument and return an integer, and that is something we need to work on.

```
let a = parseInt(document.querySelector('#first').value);
let b = parseInt(document.querySelector('#second').value);
```

Okay, now these two values I passed them into the **calculateTwo** function. This function takes two parameters, the two values that I previously parsed to integers. The function then does the calculation and returns the sum. The value of the sum will be stored in a variable called the **result**. So, whatever numbers we use, the **calculateTwo()** function will always calculate their sum and store the value into the variable. Finally, we call the **printThis** function that takes only one parameter: the value we have stored in the variable result. This function will select the **h2** tag from the **HTML5** markup and change its inner **HTML** with the result of the calculation, and that is it. We are using the **HTML DOM getElementById**() method to select the **HTML** tag **h2**.

```
document.getElementById("output").innerHTML = theSum;
```

So the sequence again was:

1) Enter a number in the first operand field
2) Enter the second number in the second operand field
3) Click on the button
4) Get the values of the two fields and convert them into integers
5) Call the **calculateTwo** function and pass in the two arguments we converted to integer values
6) Call the **printThis** function with the result that we got from the **calculateTwo** function
7) Display the result back for the user to see

This was the sequence we wanted to happen, right? This approach is ok, but we can tweak it even more because we are making two separate functions to display the result. This is the reason why I have made some modifications to the JavaScript code (the code is in the **sequence1.html**)

```
<script>
 let calcBtn = document.querySelector('#calculate');
```

```
calcBtn.addEventListener('click',(e)=>{
    e.preventDefault();
    let a = parseInt(document.querySelector('#first').value);
     let b = parseInt(document.querySelector('#second').value);

         function printThis(theSum) {          document.getElementById("output").innerHTML = theSum;
          }

          function calculateTwo(a, b) {
             let sum = a + b;
             printThis(sum);
          }

          calculateTwo(a, b);

  });

</script>
```

In this case, we called the **calculateTwo(a,b)** function first, and we let the calculator function call the **printThis(sum)** function. This is good, right we are making some progress, and the output will be no different compared to the first example. The second approach is not perfect either because we cannot prevent the **calculateTwo** function from calling the **printThis** function. This is why we need to use the callback function. Finally, we are in a position to start explaining the callbacks. So, according to the definition, a callback is a function that is passed as an argument to another function. To show you how this is done, I need to change the code that is between the script tags again (the file is called **callback**.html):

```
<script>
   let calcBtn = document.getElementById('calculate');
   calcBtn.addEventListener('click',(e)=>{
      e.preventDefault();
      let a = parseInt(document.getElementById('first').value);
      let b = parseInt(document.getElementById('second').value);

      function printThis(theSum) {
         document.getElementById("output").innerHTML = theSum;
      }

      function calculateTwo(a, b, theCallback) {
         let result = a + b;
         theCallback(result);
      }

      calculateTwo(a, b, printThis);
   });
</script>
```

As you can see from the code above, we are using a callback as the third argument when we call the **calculateTwo** function. Then we let the calculator function run the callback after the calculation is finished. We can see that **calculateTwo** has 3, not 2 arguments now. The first two are the integer values we already know about, and the last one is the function's name called **printThis**. Every time we pass a function as an argument, we pass only the function's name like in our example, but we do not put the () – brackets. So please do not pass a function as an argument into another function like this:

```
calculateTwo(a, b, printThis()); //this is wrong

calculateTwo(a, b, printThis);//this is the way
```

This is the basic example of callback functions, but in reality, the callback functions are used with asynchronous functions. You will see what this means in the next section when we use timers.

Timers

Remember when I said that asynchronous programming is all about task/tasks that need to be executed at some time in the future. If we want to execute some code after a certain time in the future, we can use the **setTimeout**() function.
The syntax of **setTimeout** function is this one:

```
setTimeout(function[, delay, arg1, arg2, ...]);
setTimeout(function[, delay]);
```

```
setTimeout(code[, delay]);
```

From the syntax, the first argument is a function, and the second argument is the delay timer measured in milliseconds. The timer shows how much the function should wait before being executed. The **arg1**,…,**argN** are optional. The last syntax of the **setTimeout**() function uses code instead of the function, and here we can add a string that will be compiled and executed when the timer expires. So why is the **setTimeout** an asynchronous function? This function is asynchronous because you can specify when the callback function needs to be executed in the future based on the value of the timer. To be clear, the timer is in milliseconds, and one second has one thousand milliseconds.
Example:
1s = 1000ms
2s = 2000ms
Okay now we need to test everything that we had discussed, and bellow I will give you the entire code you will need (the actual code is in **setTimeOut.html** file):

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>setTimeout Function in JavaScript</title>
</head>
<body>
   <h1>setTimeout function in JavaScript</h1>
   <h2 id="output">Output goes here:</h2>

   <form name="myForm" action="#">

      <label name="firstOperand">First Operand</label>
       <input type="text" name="firstOperand" id="first"><br><br>

       <label name="secondOperand">Second Operand</label>
       <input type="text" name="secondOperand" id="second"><br><br>

       <button id='calculate'>Calculate</button>

   </form>
<script>
function displayResult1() {
   document.getElementById("output").innerHTML = 'Loading.....';
}
function displayResult2() {
   document.getElementById("output").innerHTML = 'You will see this message in 10 seconds';
}
setTimeout(displayResult2, 10000);
setTimeout(displayResult1, 5000);
</script>
</body>
</html>
```

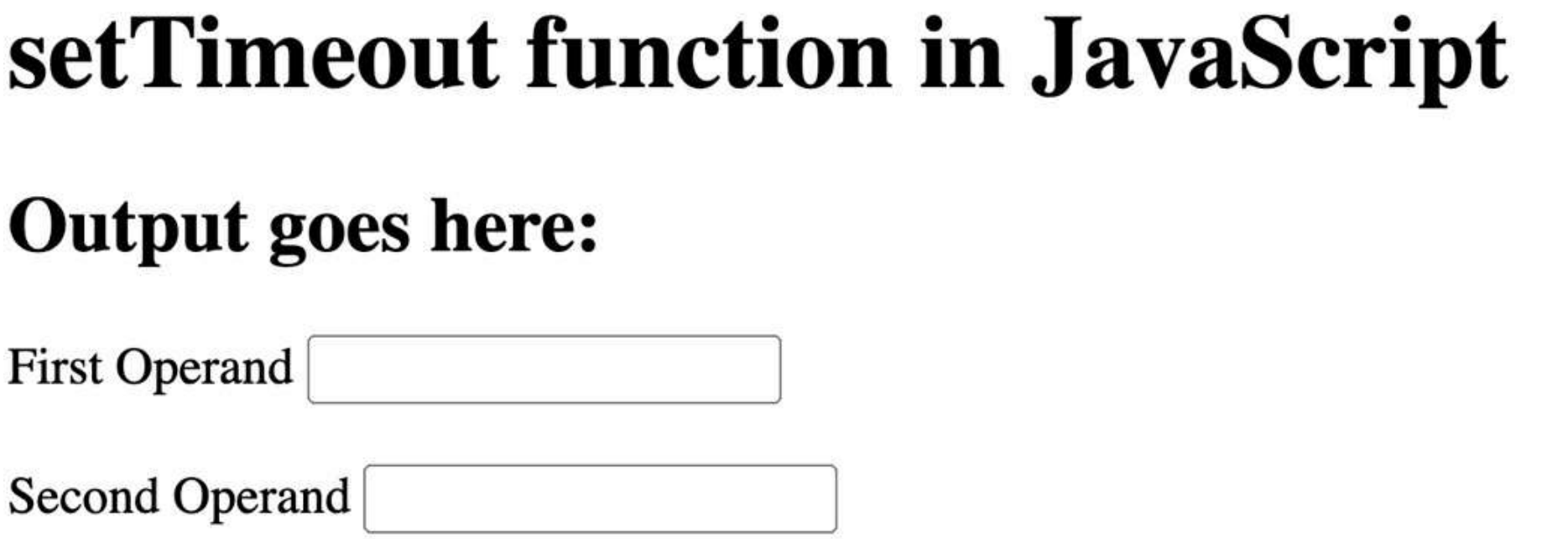
Before I show you the output, let me tell you what the code is doing. As you can see from the example above, we have two **setTimeout** functions. Let's start with this one:

```
setTimeout(displayResult1, 5000);
```

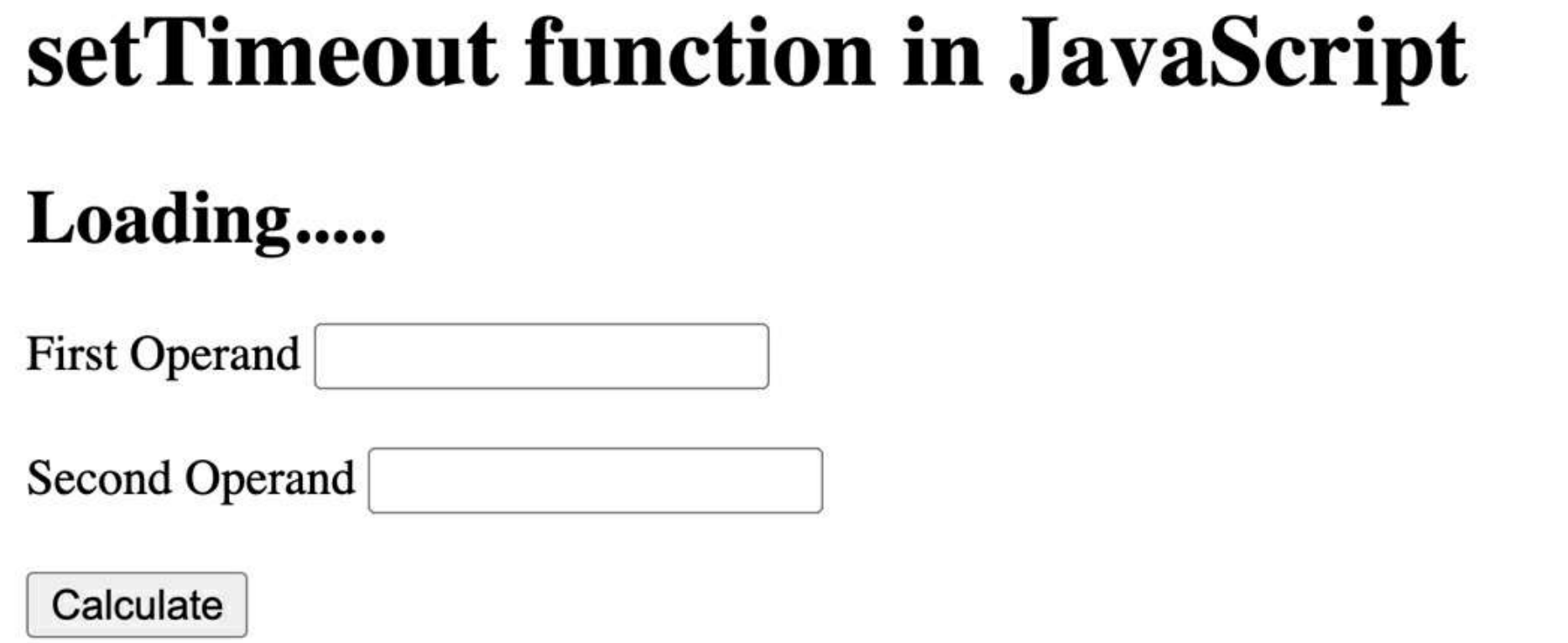
In the code above, **displayResult1()** is used as a callback function, and we pass this function as an argument inside our **setTimeout**, but please make sure you notice that here we are using the function name without any brackets. The second argument of the **setTimeout** function is the timer, which is set to 5000 milliseconds or 5 seconds before time-out. Therefore, the **displayResult1()** will be called after 5 seconds. The **displayResult1** function is very basic and what it does is change the content of the HTML tag h2. The original content is h2 tag is '**Output goes here'**, which we are trying to change inside our function. We can accomplish this by using one of the simplest DOM manipulations ever:

```
function displayResult1() {
document.getElementById("output").innerHTML = 'Loading.....';
}
```

So the **innerHTML** will change the content of the h2 tag after 5 seconds.
The first screenshot is taken when we load the file in our browser, and the h2 tag will still have the original content because the **setTimeout** function will is going to be called after 5 seconds:



As you can see, our logic is working so far, and we still have the '**Output goes here'** as h2 content.
The second screenshot is taken after 5 seconds, and then **displayResult1** function will be executed, and change the h2 tag content with new content '**Loading…**':



Now we have '**Loading….**' as h2 content, which is great because it means that everything is working as it should.
The final screenshot is taken after 10 seconds, but why did I do this? So, the second **setTimeout** with have the function **displayResult2** executed after 10 seconds. I will not explain it again because the **displayResult2** is identical with **displayResult1** with only one difference, and that is the timer that I have changed from 5 to 10. Here is the final screenshot with new updated h2 content after 10 seconds:

setTimeout function in JavaScript

You will see this message in 10 seconds

First Operand

Second Operand

Calculate

I hope you now understand how the **setTimeout** functions are working. Another interesting point is that instead of passing a name of a function as an argument we can pass the entire function inside the setTimeout, just like I did in this example (the code is in **setTimeOut1.html** file):

```
setTimeout(function () {
   document.getElementById("output").innerHTML = 'You will see this message in 7 seconds';
}, 7000);
```

Canceling setTimeout using clearTimeout function

Now you should know that the **setTimeout** is a function that returns an identifier known as timer id. We can use this timer id so we can cancel the execution.
Please check out the following example (code in: **setTimeout.js**):

```
let timerId = setTimeout(() => console.log("It will never be printed"), 2000);
console.log(timerId);
clearTimeout(timerId);
```

If you run this in the browser, you will never see the output. Please note that we can use arrow functions with the setTimeout function. In short, the function we have in setTimeout will never be executed because the timer is set for two seconds, and we use the **clearTimeout** function to cancel the **setTimeout** function before those 2 seconds. This is very useful because we might change our minds, and we do not want the setTimeout function to be executed if something else happens in our code. Again, the clearTimeout takes one argument: the timer Id and this id we get from the **setTimeout** function.

setInterval function

The setInterval function has the same or identical syntax as **setTimeout** but with one notable difference. The **setInterval** will run the function regularly after the given time interval.
Here is an example:

```
setInterval(() => console.log('tick tock, tick tock'), 3000);
```

I let the code run in my browser console for around 18 seconds, which happened. The same function was executed 6 times, or after 3 seconds as the interval we provide, therefore 6 * 3 = 18. This is very good when we want a function to run repeatedly. For example, we need a function that will check for updates regularly.
Here is the output of my function:

```
> setInterval(() => console.log('tick tock, tick tock'), 3000);
<· 2
6 tick tock, tick tock
```

If we look at the number six, that is one left side of the ‘tick tock, tick tock’ that will tell you how many times this

function has been executed before I shut my browser.
Same as the **setTimeout** we have a way to clear this interval using the **clearInterval**() function. This function takes one parameter, and that is the intervalId.
Here is the code:

```
let intervalId = setInterval(() => console.log('tick tock, tick tock'), 3000);
console.log('No more tick, tock because we will clear it with clearInterval fn')
clearInterval(intervalId);
```

I think we are done with timers, so let us focus on one very important topic called JavaScript events. I did include them in my first JavaScript book and I also include them in the last two chapters of this book. I think it is important to briefly mentioned them again as they are crucial for understanding how the whole process of asynchronous programming works. We have already used event listeners in our examples, but it will not hurt if we go over them once more.

JavaScript Events

In JavaScript, we are always focused on events. For example, we are waiting for the user to take some actions. Then, we respond to the user requests. So, the client-side programs are mostly event-driven programs. But where do these events come from? The browser generates those events every time a user clicks on a button, types on the search bar, moves the mouse, or even touches the screen. If you go back in the previous sections, you will see that I have used some event listeners in the exercises without explaining much about them 'because I assumed you already know these things'. Every time a specific event occurs, a callback function is created, and these functions are called event listeners, or in some literature, they are called event handlers. Here is one example of how event listener will look like and this is not a real example but just to show you the syntax:

```
let calculateButton = document.querySelector('.calculate');
//function
let doSomething = (e) =>{
    console.log(e.target);
}
calculateButton.addEventListener('click',doSomething);
```

So this is what is happening. Using the CSS selector we are selecting a button from our HTML document:

```
<button id='calculate' class='calculate'>Calculate</button>
```

The button has two attributes, ID and class. We can use the CSS query selector to target this button using the class or the id attribute. Only one is enough.
Here we used the class attribute to select the button:

```
let calculateButton = document.querySelector('.calculate');
```

But we can also achieve the same result if we target the id like this:

```
let calculateButton = document.querySelector('#calculate');
```

Then we add an event listener that will listen for the user to click on that button. Every time the user clicks, the new event will be added, and inside this event, we have the **doSomething**() function, a callback function.
The callback function **doSomething** is very simple. It will only console log the element that triggered that specific event. The output should be the same **HTML5** button tag because 'e.target' will get us the target:

```
<button id='calculate' class='calculate'>Calculate</button>
```

So after we select the HTML5 button element using the **querySelector()** we can register our callback. In the **addEventListener**, the first argument specifies what event we are interested in. In our case, the event occurred when the user clicked on the button; therefore, we must specify the 'click' as an event. The second parameter will be the **doSomething** function which will be the callback function, this is the function that the browser will invoke, and it will pass an object that will include the details of that event. That is why we can use **e.target** in the function to see the exact element that triggered this event.

Network Events: XMLHttpRequest, Callbacks

As I mentioned at the beginning of this chapter, JavaScript is a single-threaded language, and the tasks are performed in a sequence or in order. Now image we need data that comes from external API. Normally, we send a request, and we are getting a response back with that data, but what if there is some kind of problem and we are still waiting for the data to be available for us. So, we are still waiting for the data, and the rest of the tasks are blocked

and cannot be started. This is the same as when I'm on some website, and I click on a button to read further in the article, and as I'm waiting for the text to load, the website is completely frozen, and I cannot do anything. This is bad, and that is why we are learning about the concept of asynchronous JavaScript. The asynchronous programming will allow us to create multiple threads, meaning that we can continue executing the rest of the tasks while waiting for something to happen. Someone will ask me then what is the connection between asynchronous and **XMLHttpRequest**. If you are already familiar with **XMLHttpRequest**, you will know that it supports both ways, synchronous and asynchronous.
Okay, now let me talk you through the steps when we try to get data from an external API using the asynchronous function:

1) The compiler reaches a function that will get the data from external API
2) Then we go inside the function where we are using callback functions so we can create a second thread enabling the rest of the code to run freely
3) Until we are waiting for the data to be available, the rest of the tasks are being executed, and as soon as we get back the data then, the callback functions are started their execution

In the following sections, we will try to do all of the steps we outline above.

Create XMLHttpRequest

We can create **XMLHttpRequest** object by calling the **XMLHttpRequest** constructor. **XMLHttpRequest** objects interact with servers and retrieve data from specified **URLs** without a page refresh. This was a big deal in the past because the pages continuously did a page refresh after each request, which was time-consuming and not a good user experience. Now, with the **XMLHttpRequest** object, we can do a partial page update without a full-page refresh and destroy the user experience.
We can create **XMLHttpRequest** like this:

```
const request = new XMLHttpRequest();
```

After this, we can use the open method, which will be available on the 'request' to set up a request. The open method takes three arguments. The first one is the method or the type of request. The second is the **URL** or the endpoint from where we will retrieve data, and the last is a Boolean value, which is optional. If the third value is true or omitted, the request will be considered asynchronous. I will use the **JSON** placeholder **API** for this example, a free fake **API** that developers use for testing and prototyping. I will grab the user's data from there.
Here is the link to this website (you can read more about **jsonplaceholder** on their site):

https://jsonplaceholder.typicode.com/

So let us create this request now (the first way):

```
const request = new XMLHttpRequest();
request.open("GET", "https:// https://jsonplaceholder.typicode.com/users/
request.send();
```

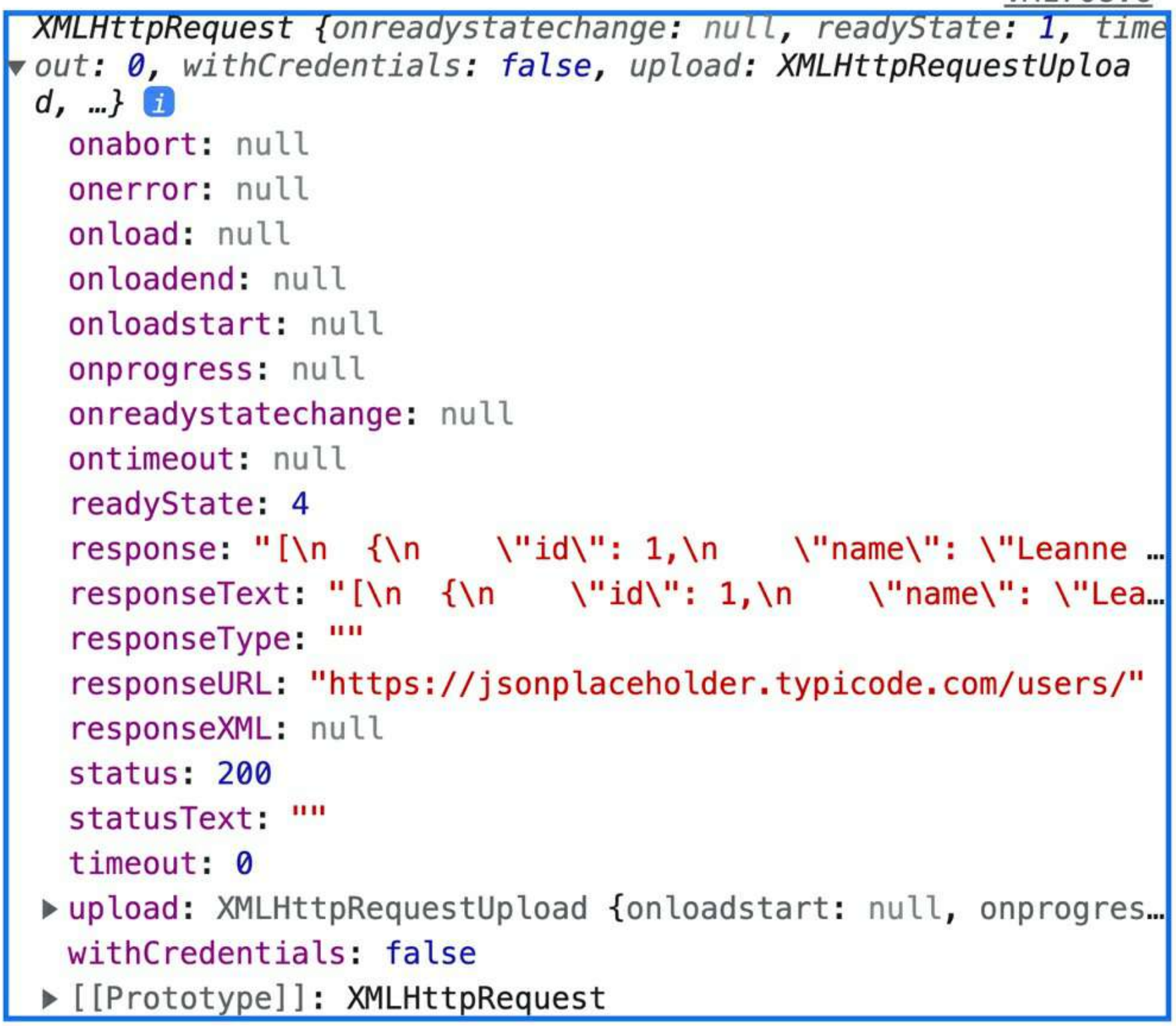
The second way we can achieve the same result and this is what I prefer is like this (the entire code will be in **networkEvents.js** file):

```
const request = new XMLHttpRequest(),
method = "GET",
url = "https://jsonplaceholder.typicode.com/users/";
request.open(method, url, true);
request.send();
console.log(request);
```

As you can see in the above code, we are creating the XMLHttpRequest object and storing it in a variable called 'request' to reuse the same variable later in the code. This variable has access to the same methods that we can use to retrieve and manipulate the data that comes over the wire.
Okay, what you can do next is copy this code and paste it into your browser and see if we are getting some data from the requested API (if you are reading this on your tablet or Kindle, then I will provide screenshots. So, you can sit back and enjoy).
This is the output:

```
const request = new XMLHttpRequest(),
method = "GET",
url = "https://jsonplaceholder.typicode.com/users/";
request.open(method, url, true);
request.send();
console.log(request);
```

VM1708:6



We are getting a lot of data, but please do not be afraid. Not everything listed here we will need to know. Among the properties, there is one particular property we need: the state property called **readyState**. The value of this property can be between zero and four. What do these values mean?

- 0 - This is the state before initialization of the request
- 1 - This is the state when the request has been initialized or after using the open method
- 2 - This is the state that shows the request has been sent
- 3 - This is the state when the request is being processed
- 4 - This is the state when the request is completed – we can get back data or an error

Now we can keep track of these states, but we can perform some actions on the received data as soon as we hit the number four. We can use this **readyState** attribute and call an event as soon as the state attribute changes. From here, we can use the **XMLHttpRequest.onreadystatechange** property. This property contains an event handler, and this handler will run every time the **readyState** changes its value.
Here is the syntax we can use:

```
XMLHttpRequest .onreadystatechange = callback;
```

So let us add more code that will summarize what we talked so far:

```
const request = new XMLHttpRequest(),
method = "GET",
url = "https://jsonplaceholder.typicode.com/users/";
request.open(method, url, true);
request.onreadystatechange = function () {
if (request.readyState === 4 && request.status === 200) {
   console.log(request.responseText);
  } else if (request.readyState === 4) {
   console.log("Oh no! There has been an error and could not fetch the data!");
  }


};
request.send();
```

In the callback function, we have one if statement with two basic conditions inside. Please make sure that only when both conditions are true will we be able to console log the data using the 'request.responseText'. Basically, we print the data when the **readyState** value will be 4 and only when the status of the request is 200. If there is some kind of error, then the else if statement will be executed and it will console log this text " **Oh no, There has been an error and could not fetch the data!"** And that is pretty much what this code is doing, and we need that else statement because the URL endpoint we are trying to fetch can be spelled with a mistake, and the request will still go through the necessary steps, but the responseText will be empty. If we run the same piece of code in our browser, we will see the following output:

```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
      "geo": {
        "lat": "-37.3159",
        "lng": "81.1496"
      }
    },
    "phone": "1-770-736-8031 x56442",
    "website": "hildegard.org",
    "company": {
      "name": "Romaguera-Crona",
      "catchPhrase": "Multi-layered client-server ne
      "bs": "harness real-time e-markets"
    }
  },
  {
    "id": 2,
    "name": "Ervin Howell",
    "username": "Antonette",
    "email": "Shanna@melissa.tv",
    "address": {
      "street": "Victor Plains",
      "suite": "Suite 879",
      "city": "Wisokyburgh",
      "zipcode": "90566-7771",
      "geo": {
```


This is not the entire output, but we did get all of the users from the fake API. If we look at the code, it is not ideal and not reusable. We can get the entire code and put it in a function, and then call that function as many times as we want in the future.
Step 1 will be to create a function called getUsers so we can wrap the entire code inside:

```
const getUsers = () =>{
const request = new XMLHttpRequest(),
method = "GET",
url = "https://jsonplaceholder.typicode.com/users/";
```

```
request.open(method, url, true);
request.onreadystatechange = function () {
if (request.readyState === 4 && request.status === 200) {
    console.log(request.responseText);
  } else if (request.readyState === 4) {
    console.log("Oh no! There has been an error and could not fetch the data");
  }

};
request.send();
}
```

Step 2 is to call the function and see if everything is working fine, and we are calling the function by its name:

```
getUsers();
```

Now you can run the same function in your browser, and you should get the same result. But how can we be sure that this is done asynchronously instead of synchronously? How can we be sure if we are not blocking any code while we wait for the response from the servers with the requested data? We could write some code before and after this getUser function, which will tell us if we wrote asynchronous code. I will not do anything special. I just added two console logs before and one after the function. Each of them will have a number so we will know the order they are executed. Also, I will add a statement inside our onreadystatechange function to test if it will be consoled after the first two.
Here is the entire code again:

```
console.log('1) This text should be first and before the getUser function');
console.log('2) This text should be after the first console log');

const getUsers = () =>{
const request = new XMLHttpRequest(),
method = "GET",
url = "https://jsonplaceholder.typicode.com/users/";
request.open(method, url, true);
request.onreadystatechange = function () {
if (request.readyState === 4 && request.status === 200) {
    console.log('3) This text should be thrid if this is synchronous programming but will it be?')
    console.log(request.responseText);
  } else if (request.readyState === 4) {
    console.log("Oh no! There has been an error and could not fetch the data");
  }

};
request.send();
}
getUsers();

console.log('4) This text should be last or the getUser function, if this is synchronous, but will it be?')
```

Here is the output:

```
1) This text should be first and before the getUser
function
```

```
2) This text should be after the first console log
```

```
4) This text should be last or the getUser function,
if this is synchronous, but will it be?
```

```
undefined
```

```
3) This text should be thrid if this is synchronous
programming but will it be?
```


```
[
  {
"id": 1,
"name": "Leanne Graham",
"username": "Bret",
"email": "Sincere@april.biz",
"address": {
  "street": "Kulas Light",
  "suite": "Apt. 556",
  "city": "Gwenborough",
  "zipcode": "92998-3874",
  "geo": {
    "lat": "-37.3159",
    "lng": "81.1496"
  }
},
"phone": "1-770-736-8031 x56442",
"website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server neu
    "bs": "harness real-time e-markets"
  }
},
{
```


As you can see, console logs 1, 2, 4 are printed before console log 3, which is happening because of the asynchronous function. So the compiler started to work from top to bottom, and the first console log was executed, then it went to the second one, and console logged the content. After finishing the first two, it went to the function, but it skipped the content of the function and went to the line of the function call or **getUser**(). So, the compiler, went back to the function, started working on the request, and instead of waiting for the callback function to finish and get the data, it skipped that function and went outside of the function and printed the console log number 4. Now the idea of a callback is that I will call you back as soon as I have information for you, right? So there are no more lines of code in our program, so the compiler is waiting for the callback function to say, hey, I got the data

please finish what you have started. Finally, the data is there, and that is why it goes inside the if statement and prints the console log number 3 and the entire response.

Callback Hell

The example we did in the previous section where we got the user's data from the fake API was fairly simple and easy to understand. Still, in reality, the tasks we need to do will include creating more callbacks functions, and we usually need to nest them. For example, one callback to the webpage that contains the data we want to retrieve and another callback that will scan the page for particular user info. Please note that this needs to be done asynchronously, and each data retrieval means more error handling, which will create even more callbacks. This is what we call "**callback hell**" or deeply nested callbacks that are hard to understand and maintain. Callback functions are very common nowadays, but they usually create a problem for us when we have deeply nested callbacks that depend on each other.
Here is one example that of callback hell:

```
const transformBtn = document.querySelector('.change');
const el = document.querySelector('.output');
const changeMe = (e) =>{
   setTimeout(function() {
   el.classList.add('redOutput');
      setTimeout(function(){
         el.classList.remove("redOutput");
         el.classList.add('greenOutput');
         setTimeout(function(){
            el.classList.remove("greenOutput");
            el.classList.add('blueOutput');
               setTimeout(function(){
                  el.classList.remove("blueOutput");
                  el.classList.add('yellowOutput')
               },800)
         },1500);
      },3000)
  }, 5000);
}
transformBtn.addEventListener('click', changeMe);
```

The code font on the exercise above is still small, but you can see the shape of the code and what nesting looks like. This code example is not very nice looking, and it tends to move towards the right because of the indentation levels, so it is similar to a pyramid, or some developers even say it looks like a Christmas tree. Let us summarize, the callback-hell is when we have multiple callback functions nested inside each other, and their call depends on the previous callback outcome. You can find the same code in **callBackHell HTML** file, and if you open the file in your browser, you will see a button that if you lick and each of the **setTimeout** functions will start executing as soon as it reaches the time specified, and it will change the background color of the square.
This is what you will see before we click on the button:

CallBack Hell

Output goes here:

Transform Me!

This is the output after 5 seconds:

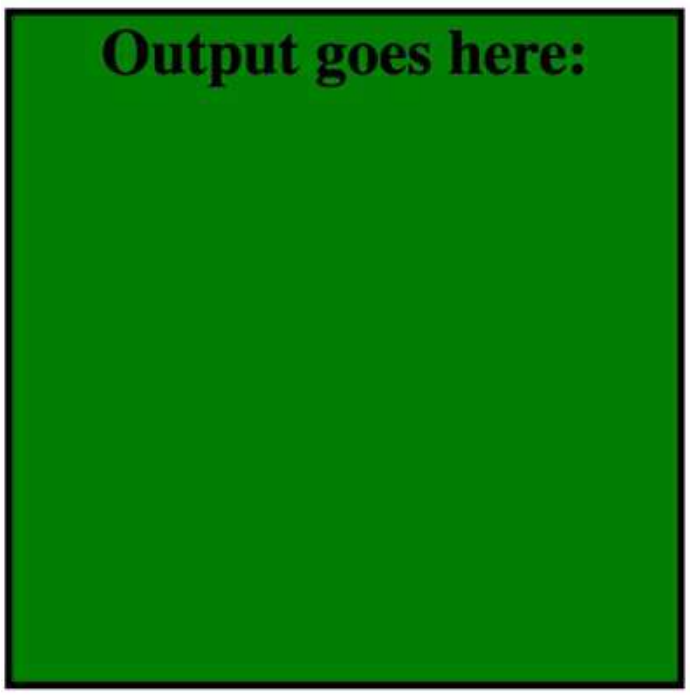
CallBack Hell

Output goes here:

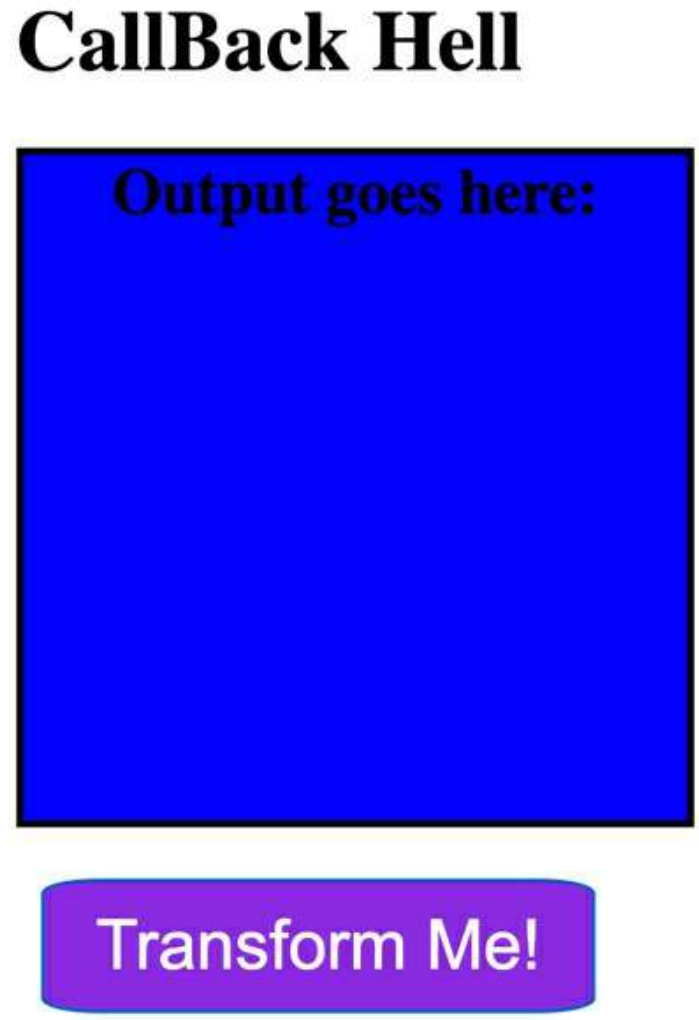
Transform Me!

This is the output after 3 seconds:

CallBack Hell

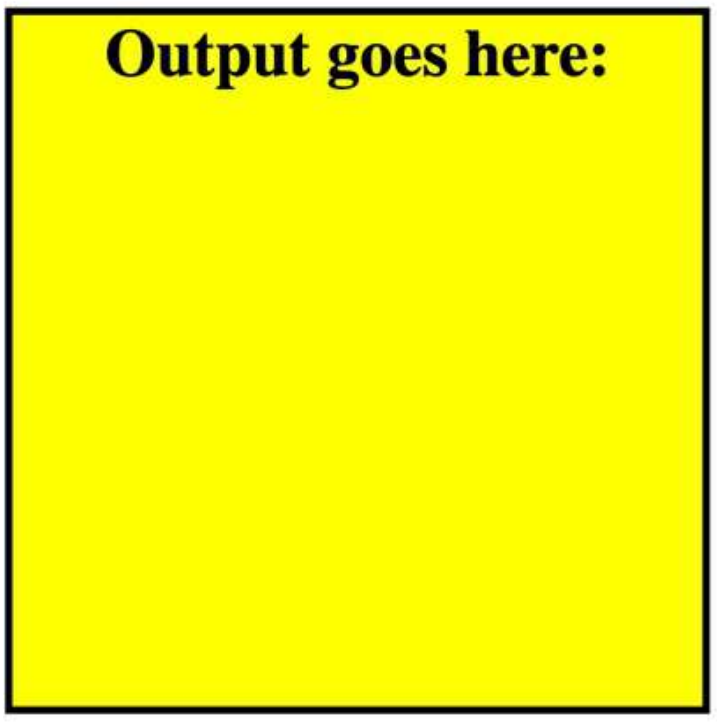


This is the output after 1.5 seconds:



This is the output after .8 seconds:

CallBack Hell



Today, Modern JavaScript has a new concept known as Promises to deal with and prevent the callback hell like the one I showed you in the above example. In the following section, we will learn about **Promises**, how to use them, and what is their syntax.

Promises

In this section, we will learn how to create Promises. As you already know from the previous examples, asynchronous programming can be very complicated. With this new feature of JavaScript, you can get rid of callback hell.
So, what is a **Promise**? A **Promise** is an object that represents the result of an asynchronous operation. The result may be ready or may even be a failure. Is there a **Promise** with synchronous functions? The answer is no because the Promise can call the **callback** function only when the value is ready. In the simplest terms, the **Promises** are a way to work with callbacks without creating the callback hell.
So you can understand the **Promises** like this: in our everyday lives, we are making many promises either to ourselves or to someone we care about. For example, I promise I will lose some weight in the future. So, this can have two possible outcomes: I will keep the promise and lose weight, or I won't keep my promise at all. There you have it; this is how I understand promises.
Here is the syntax for creating a **Promise**:

```
let promise = new Promise(function(resolve, reject) {
  // Here is the body of executor function
});
```

As you can see, the Promise is created using the new keyword and call of the **Promise** constructor. Inside the promise, we have a function that takes two arguments. This function is known as '**executor function**'. When we have a new Promise created, the executor function will run automatically. In this function, we have two arguments called resolve and reject. These are callbacks or handlers for success and failure outcomes. The body of the function is where we put our logic that will produce the desired result. One of the two handlers will be called when the result is ready.
So when we have a value, one of the two callbacks will be called:

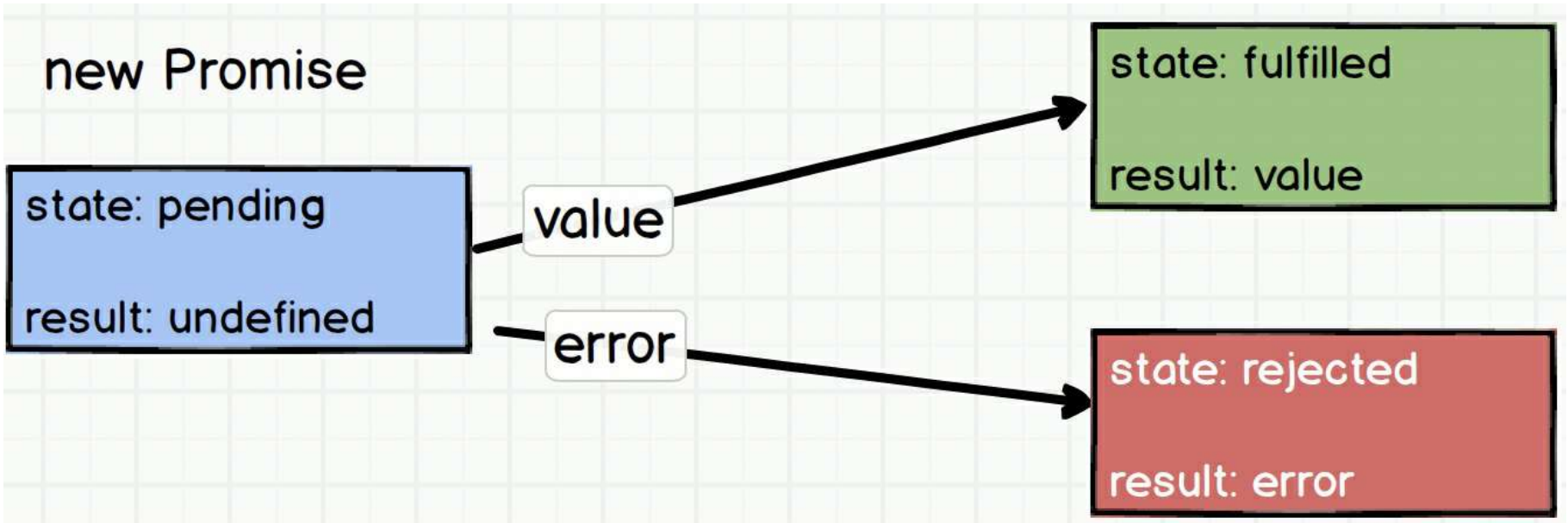
- resolve (value) - if there is an outcome from the finished job with some result value
- reject (value) – if there is an error, then what is the reason for failure

Ok, to summarize, when we create a new Promise using the Promise constructor, the executor function will run automatically. When the function has a value, it will call one of the resolve or reject handlers. The resolve if there was a success or reject if there was an error.
A Promise can be in one of these states:

- pending: this is the initial state, neither fulfilled nor rejected.

- Fulfilled: meaning that the Promise was completed successfully because the resolve was called.
- Rejected: meaning that the operation failed, there was an error, and reject was called.

The result is initially undefined, then it will change its state to resolve (value) or to an error when reject (error) is called



Let us do a very simple Promise using the **setTimeout** function (code in **promise.js**):

```
> let promise = new Promise(function(resolve, reject) {
    // the function is executed automatically when the
promise is constructed
    setTimeout(() => resolve("done"), 2000);
  });
<· undefined
> promise
<· ▼Promise {<pending>} i
    ▶[[Prototype]]: Promise
     [[PromiseState]]: "fulfilled"
     [[PromiseResult]]: "done"
```

This example is very simple and has a simple executor function that will produce results via the **setTimeout** function after two seconds. So the executor function is called immediately by the new **Promise**. In the function, we have two arguments, and we call one of them because the setTimeout function will always produce a result. In our case, we tweaked the function to resolve because we know the outcome of the setTimeout. So after two seconds, the executor calls the resolve with the value ‘done’ and, therefore, will produce the result. Now the state of the promise will change from pending/undefined to fulfilled/done.
This is an example just to show you how promises are successfully resolved.
Now let us create an example where we will reject a promise with an error:

```
let promise = new Promise(function(resolve, reject) {
  // after 1 second signal that the job is finished with an error
  setTimeout(() => reject(new Error("Error happened!")), 2000);
});
undefined
▶ Uncaught (in promise) Error: Error happened!
    at <anonymous>:3:29
promise
▼ Promise {<rejected>: Error: Error happened!
      at <anonymous>:3:29} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "rejected"
  ▶ [[PromiseResult]]: Error: Error happened! at <anonymous>:3:29
```

From the example above, we call the reject handler, and therefore the promise object will be in a rejected state.
The important thing to note is that the function can call only one resole or reject handlers. Any state is considered final, and if we have any other calls of resolve and rejected, they will be ignored.
Finally, a resolved or rejected promise is called a **settled** promise.
Here is one more complex example: we use promise to load images on a web browser.

```
function getImage(url) {
 return new Promise((resolve, reject) => {
   let img = new Image();
   img.addEventListener('load', e => resolve(img));
   img.addEventListener('error', () => {
     reject(new Error(`Failed to load image's URL: ${url}`));
   });
   img.src = url;
 });
}
```

In this example, we have a combination of callbacks and a Promise. The same example could have been done differently, but we will not discuss this here. The function **getImage** takes only one argument: the URL endpoint of an image that sits somewhere online, and we need to pass this endpoint anytime we want to use the **getImage** function. This function will return a promise that will be either resolved or rejected. Inside we have created a variable called **img.** This variable will hold the data from the actual image resource. We have used the **Image API** because we want to attach an event listener every time the state changes either to “load” or to an “error”. When the image has finished the request and produced the data, the event listeners will kick in. If we get the data back without any errors, we can call the resolve event with the ‘img’ as a value. If there is some kind of error, it will be passed on to the reject handler. Lastly, we set the ‘**src source’** property that returns the image's value or URL. This example will not work at this stage because we need to learn how to deal when a Promise returns. Please do not get confused at this stage because I have written the Promise constructor using simple arrow function syntax, but basically, everything is the same.

Immediately Settled Promises

If we want to immediately settle or fulfill a promise with a given value, we can use Promise.resolve(value). You should be aware that the value this method will return is a promise object with a given value.
Please take a look at this example:

```
> const loadImage = url => {
      if (url === undefined) return Promise.resolve('value');
  }
  loadImage();
<· ▶ Promise {<fulfilled>: 'value'}
```

If the value you are passing to the Promise is a simple plain value or even a promise, this method will return a promise as in the example above.
If we use the Promise. reject(error), then this promise will be immediately rejected with the given error.
Please check this code:

```
const loadImage1 = url => {
    if (url === undefined) {
        return Promise.rejected(Error('There is no URL provided'));
    }
}
loadImage1();
```

```
▶ Uncaught TypeError: Promise.rejected is not a function
    at loadImage1 (<anonymous>:3:24)
    at <anonymous>:6:1
```

Consuming Promises

Now, we know how to construct promises, but we also need to figure out how to get its result. Once the promise has settled, meaning it returned a result or an error, it is time to use some methods. Consuming functions/methods are

- .then
- .catch
- .finally

The fundamental one is the '.then' method and here is the syntax:

```
promise.then(function(result) {
 /* handle a successful result */
},function(error) {
/* handle an error */
}
);
```

So as you can see, that .then method will specify an action that should be done after the promise is resolved. The first argument is a function passed, as you can see from the syntax. This function will run when the promise is resolved and when it receives the result. The second argument is a function that will run when the promise is rejected. I will create two examples, one for the successfully resolved promise and one for the rejected promise. Check out the first example (the same code in **consumingPromises.js**):

```
let promise = new Promise(function(resolve, reject) {
    setTimeout(() => resolve("Resolved!"), 3000);
});
promise.then(
    result => console.log(result), //"Resolved!" after 3
seconds
    error => console.log(error) // it will not run
);
```

```
▶ Promise {<pending>}
Resolved!                                          VM166:
```

As you can see from this screenshot, the initial state at the beginning was 'Promise{< pending>}', this is happening because of the delay timer we set to be three seconds, and after those 3 seconds, we got the resolved value back.
This example will never be rejected because we know that the **setTimeout**() function will always resolve after three seconds.
Here is one example when we have an error:

```
let promise1 = new Promise(function(resolve, reject) {
    setTimeout(() => reject(new Error("Error happened!")),
3000);
});

promise1.then(
    result => console.log(result), // it will not run
    error => console.log(error) // "Error: Whoops!" after
1 second
);
```

```
▶ Promise {<pending>}
```

```
Error: Error happened!                                    VM177:7
    at <anonymous>:2:29
```

Important to remember is that Promise.then can take one argument as well. For example, if we are looking for promises that are always resolved successfully, then we can use the first argument only:

```
promise.then(
   result => console.log(result), //"Resolved!" after 3
);
```

Before explaining the other two methods, I would like to discuss a very important topic called promise chaining.

Promise Chaining

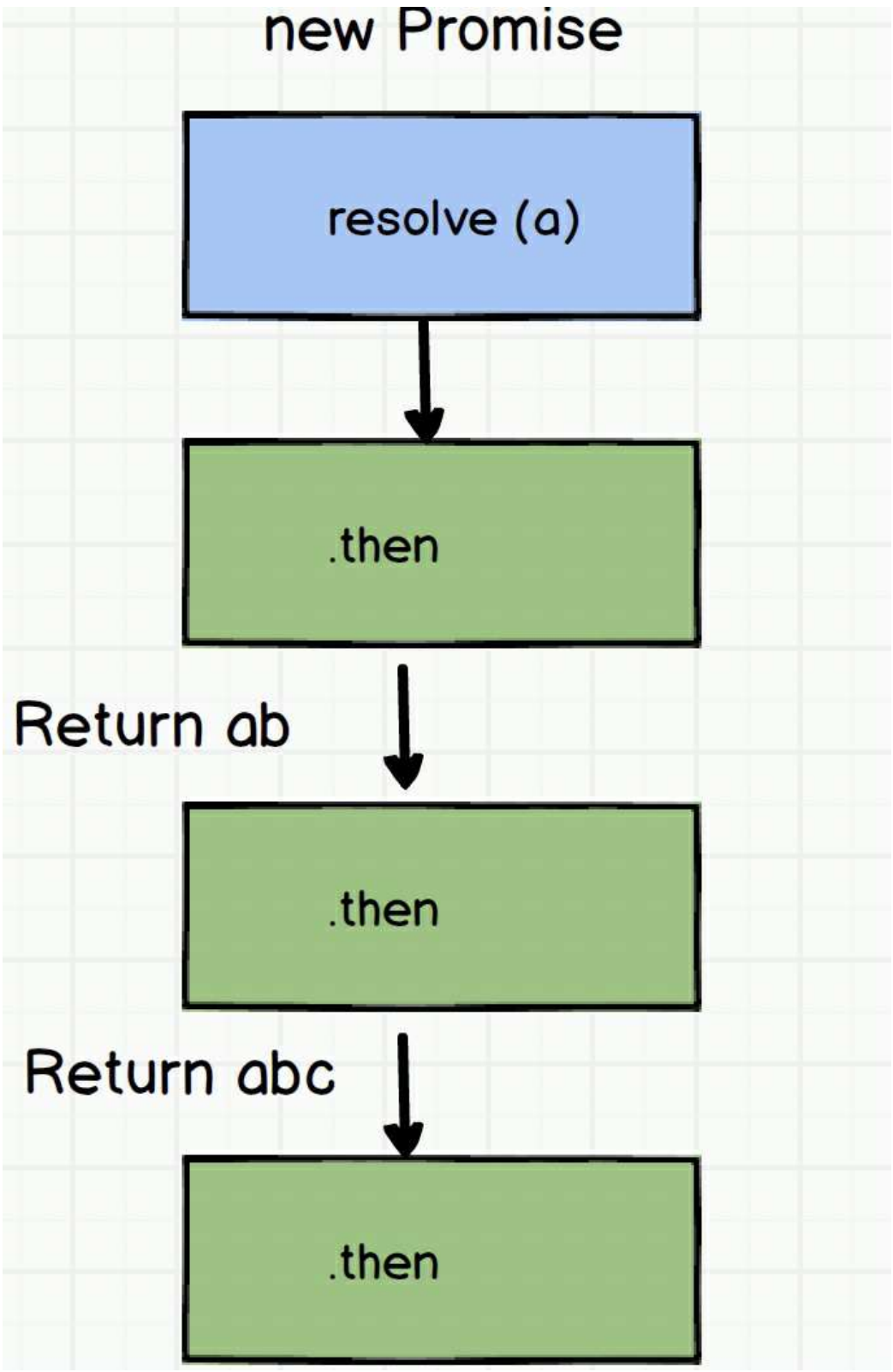
Remember when we talked about how difficult it is to work with a sequence of asynchronous tasks executed one after another? With the promises, we can fix those things and chain multiple promises together.
Have a look at the following code (code in **promiseChaining.js** file):

```
let myPromise = new Promise(function(resolve, reject) {
   setTimeout(() => resolve('a'), 1000);
 }).then(function(result) { //
   //console.log(result); // a
   return result + 'b';
 }).then(function(result) {
   //console.log(result); // ab
   return result + 'c';
 }).then(function(result) {
   //console.log(result); // abc
   return result +'d';
 });
myPromise.then((result) => console.log(result));
```

This example shows that the result from the initial promise is passed down through the chain of '.then' handlers. If you run this in your console, you will see the output like this: '**abcd**'.
Here are the steps:

- The initial promise will resolve after one second delay with a value 'a'
- When the '.then' handler is called, it will create a new promise that resolves with the concatenation of the two strings a + b which will give us the value 'ab'.
- Then the next '.then' handler will get the result from the previous and add the letter 'c' and pass the value to the next handler and so on

Now you know what promise chaining is and how we can chain the handlers together, and this thing works because '.then' will always return a new promise and therefore we can call the next .then handler on it.
Here is the flow:



From this example, you have learned that we do not need to save each promise in a separate variable. Instead, we can use a chain of promises. Here is the same example where we need to save each promise in a separate variable 'this is how we are not supposed to do, although it is not wrong and is usually preferred by the newbie coders':

```
let myPromise1 = new Promise(function(resolve, reject) {
   setTimeout(() => resolve('a'), 1000);
});
let myPromise2 = myPromise1.then(function(result) {
  //console.log(result); // a
  return result + 'b';
  });
let myPromise3 = myPromise2.then(function(result) {
  //console.log(result); // ab
  return result + 'c';
});
let myPromise4 = myPromise3.then(function(result) {
  //console.log(result); // abc
  return result +'d';
});
myPromise4.then((result) => console.log(result));
//Output: abcd
```

Error – Rejection Handling

In the scenarios/examples we have done so far, we were focused on how promises can be created and how they can be consumed. In all of the tasks, the promises have been mostly resolved successfully, but now is the time to see what will happen when we have errors. We have already seen that we can use the rejection handler as a second argument when using the .then method, but that is not how we should catch the errors. We usually need to use the .catch method to handle errors.

Here is one example:

```
getUserByUserId('abc')
   .then(user => console.log(user.username))
   .catch(err => console.log(err));
```

In the example above we handle the promise using both ‘.then’ and ‘.catch’ method. If any error then it will be caught inside the ‘.catch’ handler. Let us have a look at the following example:

```
let isAuthorized = false;
function getUserByUserId(id) {
   return new Promise((resolve, reject) => {
      if (!isAuthorized) {
         throw new Error('Unauthorized access!');
      }
      resolve({
         id: id,
         username: 'admin',
         role: 'administrator'
      });
   });
}
getUserByUserId(1234756525)
   .then(user => console.log(user.username))
   .catch(err => console.log(`Caught by .catch ${err}`));
```

Inside our promise, we are throwing an error, which will generate an error object with some message. Obviously, here in this example, no matter what kind of value we pass into the **getUserByUserId** function, it will always throw a new error because we set the **isAuthorized** variable to be false but this is just to demonstrate how things are done. This will not happen in everyday scenarios, but it is a good example of using the throw statement to throw an exception and then handle it using the ‘.catch’ method. So the throw statement will let us create our custom errors, which we want in our case.
We can also use the **reject**() handler, which will have the same effect as throwing an error.
Check the following example where I use the predefined reject handler explicitly instead of throw statement:

```
let isAuthorized = false;
function getUserByUserId(id) {
   return new Promise((resolve, reject) => {
      if (!isAuthorized) {
         //throw new Error('Unauthorized access!');
         reject('Unauthorized access!');
      }
      resolve({
         id: id,
         username: 'admin',
         role: 'administrator'
      });
   });
}
getUserByUserId(1234756525)
   .then(user => console.log(user.username))
   .catch(err => console.log(`Caught by .catch ${err}`));
```

I want to show you how we can catch an error that happened outside the promise, and for this, we need to use the try and catch block.

Try and Catch statement

The throw statement we have already seen allows us to create a custom error. In the literature is known as throwing an exception or throwing an error. So the exception that we throw can be JavaScript Number, Boolean, Object, or a String. Now, **throw** can be very powerful when combined with try and catch statements because we can control the program flow and create meaningful custom error messages. The try statement allows us to create a block of code that can be tested for errors during its execution. The catch statement would allow us to catch the error if that error happened somewhere in the try block during its execution. Please do not confuse the try and catch statement with the ‘.catch’ method we use in promises, although their purpose is to catch the errors.
Try and catch statement come in pairs:

```
try {
    //Block of code to try
}
catch(err) {
```

```
//Block of code to handle errors
}
```

So in the newer JavaScript syntax, we do not need to pass the error inside the catch, but it is good to know that the old syntax is still working just fine. The catch(err) has an exception variable that holds the exception value, and therefore we can retrieve its message. But if we do not need the exception value to be included, then it can be simply omitted like in this example:

```
function isValid(text) {
  try {
    JSON.parse(text);
    return true;
  } catch {
    return false;
  }
}
```

Now you know how the try and catch statement works and whatthey do so we can create example when error happen outside the Promise:

```
function getUserByUserId1(id) {
  if(typeof id !=='number' || id <= 0){
    throw new Error('This is not valid user id!');
  }
  return new Promise((resolve, reject) => {
    resolve({
      id: id,
      username: 'admin',
      role: 'administrator'
    });
  });
}
try{
  getUserByUserId1('AndyPeterson')
  .then(user => console.log(user.username))
  .catch(err => console.log(`Caught by .catch ${err}`));
}
catch(e){
  console.log(`Caught by try/catch statement: ${e}`);
}
```

Output:

```
Caught by try/catch statement: Error: This is not a valid user id!
```

Great now we know how to catch an error outside the Promise using the try and catch block. So if you throw an exception inside the promise, then the .catch method will handle this error, not the try/catch block.
Another interesting scenario is when we have a chain of promises. The question here is where we should place the catch method to handle the errors that occurred in any of the chained promises? The answer is to put the catch method at the end to ensure we catch all of the errors that happened anywhere.
Please consider the following example where I return **ab + ‘c’:**

```
let myPromise = new Promise(function(resolve, reject) {
    setTimeout(() => resolve('a'), 1000);
  }).then(function(result) {
    //console.log(result); // a
    return result + 'b';
  }).then(function(result) {
    //console.log(result); // ab
    return ab + 'c';
  }).then(function(result) {
    //console.log(result); // abc
    return result +'d';
  }).catch(error => console.log(error));
```

```
undefined
ReferenceError: ab is not defined
```

This will create a reference error in the second '.then' method because we have used the 'ab' + 'c' illegally, causing to throw a reference error that was caught successfully, as you can see from the output.

Promise all

Promise.all() is a handy method, and we can use it when we have multiple promises and want all of them to be resolved. This method takes an iterable input of promises, and the iterable means an iterable object such as an Array. In the end, you will get a promise that is resolved when the rest of the promises in the iterable are resolved. The iterable contains the values of the individual promises in the same order as the promises themselves.
Please check out this example (the code you can find in **promiseAll.js** file):

```
const promise1 = Promise.resolve('first');
const promise2 = new Promise((resolve, reject) => {
    setTimeout(resolve, 3000, 'second');
  });
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, 'thrid');
});
const promise4 = new Promise((resolve, reject) => {
    setTimeout(resolve, 2000, 'forth');
  });
let promiseArray = [promise1, promise2, promise3, promise4];
Promise.all(promiseArray).then((values) => {
  console.log(values);
});
```

```
▶ Promise {<pending>}
▶ (4) ['first', 'second', 'thrid', 'forth']
```

As you can see, the order is important, and it does not matter when the original promises are resolved. If we look at the individual promises, we can see that the promise2 will be resolved at the end, but we still got an array with the correct order. This proves that **Promise.all()** is a method that will not run tasks in parallel where each task is executed in a single thread and sequence.
An interesting scenario is when some of the promises we pass into **Promise.all()** rejects, then the whole **Promise.all()** will reject as well, which is not good because it might be only one of ten promises. Here is an example:

```
> var p1 = Promise.reject('fail');
  var p2 = 1337;
  Promise.all([p1, p2]).then(values => {
    console.log(values);
  });
<· ▶ Promise {<rejected>: 'fail'}
⊗ ▶ Uncaught (in promise) fail
```

What is the solution for this? Well, we can add a catch statement to each of the promises we pass to the **Promise.all()** method. So if any of the promises are rejected, their error will be resolved through the catch statement we chain to the promises.
Please consider the following example:

```
var p1 = Promise.reject('fail').catch((err)=>'error happened');
var p2 = 1337;
Promise.all([p1, p2]).then(values => {
  console.log(values);
});
▶ (2) ['error happened', 1337]
```

If we want to use a more refined method to settle all promises, you can use **Promise.allSettled()** method. This is a good method for performing async tasks parallel and same as **Promise.all()** method it takes an array as an argument or any iterable of promises. The **Promise.allSettled()** will return a Promise that will asynchronously fulfill once every input Promise has settled.
Here is one example with **Promise.allSettled(),** and please check out the output we have got back:

```
const pr1 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('First Promise after 1 second');
    }, 1000);
});
const pr2 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('Second Promise after 2 seconds');
    }, 2000);
});
const pr3 = new Promise((resolve, reject) => {
    setTimeout(() => {
        reject('Reason is because I wanted to be rejected :) ');
    }, 3000);
});
Promise.allSettled([pr1, pr2, pr3])
    .then((result) => {
        console.log(result);
});
▶ Promise {<pending>}
                                                    VM272:18
▼ (3) [{…}, {…}, {…}] i
  ▶ 0: {status: 'fulfilled', value: 'First Promise after 1 second'}
  ▶ 1: {status: 'fulfilled', value: 'Second Promise after 2 seconds'}
  ▶ 2: {status: 'rejected', reason: 'Reason is because I wanted to be rejected :)…
```

As you can see from the output, we have 3 objects back, and we know it is an array of objects because of the curly brackets. So the first promise will resolve after 1 second with a value of ‘First Promise after 1 second’, and the same goes for the second promise. If we look at the third one, we can see that the status is different from the rest of them, and it is rejected with the reason why it happened.

Promise Race

So far, we have seen different methods of settling promises, but most of them have something in common, and that is we won’t be able to see their results until all of them are resolved. Here comes another method called Promise.race(iterable) that will change it all. We can use this method if we want to run multiple promises in parallel but stop as soon as the first one is completed or resolved. That is why its name is Promise.race because multiple promises are racing, and whoever settles first will be returned, and the rest will be ignored. You should know that even if the first promise rejects, it will win the race again because it is still considered a first. The rest of the promises are discarded even though most of them might produce the result. To summarize, the ‘race’ method returns a promise that fulfills or rejects as soon as one of the promises in an iterable produces a result.
Please have a look at the following screenshot (code in **promiseRace.js** file):

```
const promise1 = new Promise((resolve, reject) => {
    setTimeout(resolve, 3000, 'one');
  });

const promise2 = new Promise((resolve, reject) => {
    setTimeout(resolve, 1000, 'two');
});

const promise3 = new Promise((resolve, reject) => {
    setTimeout(resolve, 3000, 'three');
});

Promise.race([promise1, promise2, promise3]).then((value) => {
    console.log(value);
  });
```

```
▶ Promise {<pending>}
```

```
two
```

All three promises will resolve, but the second one will win the race because the timer on the setTimeout function is one second, and that is why the output will be the resolved value ‘two’.

Promise Any

Promise.any similarly like the race method will take an iterable of Promise objects. So both these methods accept an iterable object, but what is the difference? This method will return a single promise that resolves with the value from that promise as soon as one of the promises we feed in fulfills.
In order for us to understand the difference we need to look at the following examples (code in **promiseAny.js** file):

```
const promise1 = new Promise((resolve, reject) => {
   setTimeout(resolve, 3000, 'Hans Solo');
 });

const promise2 = new Promise((resolve, reject) => {
   setTimeout(resolve, 1000, 'Mandalorian');
});

const promise3 = new Promise((resolve, reject) => {
   setTimeout(resolve, 2000, 'Luke Skywalke');
});

 Promise.any([promise1, promise2, promise3]).then((value) => {
```

```
  console.log(value);
});
```

The output will be the first promise that is resolved, and that is the promise with the shortest setTimeout:

```
Mandalorian
```

Ok, now lets us try the same example with a rejection:

```
const promise1 = new Promise((resolve, reject) => {
    setTimeout(resolve, 3000, 'Hans Solo');
  });

const promise2 = new Promise((resolve, reject) => {
    setTimeout(reject, 1000, 'Mandalorian');
});

const promise3 = new Promise((resolve, reject) => {
    setTimeout(resolve, 2000, 'Luke Skywalke');
});

Promise.any([promise1, promise2, promise3])
.then((value) => {
console.log(value);
})
.catch((error)=>console.log(error));

▶ Promise {<pending>}

Luke Skywalke
```

And here is the difference between **race** and **any** method. Finally, let us reject all of the promises and see what will happen:

```
const promise1 = new Promise((resolve, reject) => {
    setTimeout(reject, 3000, 'Hans Solo');
  });

const promise2 = new Promise((resolve, reject) => {
    setTimeout(reject, 1000, 'Mandalorian');
});
const promise3 = new Promise((resolve, reject) => {
    setTimeout(reject, 2000, 'Luke Skywalke');
});
Promise.any([promise1, promise2, promise3])
.then((value) => {
console.log(value);
})
.catch((error)=>console.log(error));

▶ Promise {<pending>}

AggregateError: All promises were rejected
```

As you can see, the output for the example above will be an aggregate error where all of the promises were rejected. So this method is opposite to the Promise.all method and the AggregateError is a new subclass of Error, and it tends to combine errors into one group.

Async function

This topic is fundamental for every JavaScript programmer that wants to be a serious developer. This feature was introduced in ES8 or **ECMAScript 2017**. So far, we have seen how we can create promises, chain them, get back their resolved or rejected values, and handle the errors that can happen inside the promises. We also looked at a few important methods that work incredibly well with promises. Now is the time to learn new special syntax that will allow us to work with promises more naturally. This syntax is called **'async/await',** and if you understand the promises, this will be fairly easy to understand. Remember I have said that it is never good to wait for a function to return a value because this might take a long time, and we are blocking the code? That is why we said the solution for this is to switch from synchronous to asynchronous programming. The async/await cannot be used on normal functions like we are used to writing up until this point. The functions must be tagged with the async keyword at the beginning to use the await operator in the function body. **Async/await** functions allows us to write completely synchronous-looking code while performing asynchronous tasks and calls behind the scene. Let me illustrate this through a few examples:

```
async function f() {
    return 'Async Function';
}
f().then(result => console.log(result));
```

```
Async Function
```

```
▶ Promise {<fulfilled>: undefined}
```

Here we have used the async keyword right before creating the function called '**f**'. This will mean that we deal with an async function that will return a promise. This function will fulfill with a value of 'Async Function'. This is how we can create async functions, but now we should focus our attention on the next keyword, 'await', that works only if we have an async function.

The syntax for the await is:

```
let value = await promise;
```

As we already discussed, the await function makes the JavaScript compiler wait until the promise is settled and returns a value.

```
const myFirstAsyncFn = async () => {
    let promise = new Promise((resolve, reject) => {
      setTimeout(() => resolve("Resolved in 2s!"), 2000)
    });
    let result = await promise;
    console.log(result);
  }

myFirstAsyncFn();
```

```
▶ Promise {<pending>}
```

```
Resolved in 2s!
```

I have explicitly created a promise from the above example inside the function. So the function will pause in the line where we have the await keyword. The variable called 'result' will not have a value until the promise is resolved. The delay timer on the **setTimeout** function will be two seconds. After the promise settles, the result variable will have this value '**Resolved in 2s!**'. The await keyword will stop the function from execution until the promise settles, but the function can continue its normal execution after the promise resolves. With other functions, we are wasting time. This is valuable because the JavaScript engine will not block the code while waiting for the result. It will continue executing the rest of the tasks until the promise settles and this, to be honest, is a very smart way of running tasks. If you compare the promise syntax with the await syntax, you will see that async/await syntax is easier to follow, no more .then methods. Now our code looks cleaner and organised. In one async function, multiple awaits are ok, and loops work fine. You can use the async keyword with the following:

Arrow functions:

```
async firstFn => { ... }
async (param1,param2) => { ... }
```

Anonymous and named functions:

```
async function(param1) { ... }
async function firstFn(param1,param2) => { ... }
```

Object methods:

```
myObj = {
   async myFirstAsyncFn(ulr) { ...},
   ...myFirstAsyncFn.
}
```

Methods:

```
class User {
   async authorized(data){ . . . }
}
```

It is important to mention that the browser support for async/await in JavaScript in 2022 is 95.61%. Notable exceptions are internet explorer IE11 and opera mini, with no support whatsoever for async/await.
By the time you are reading this book, things might have improved, and that is why you can visit the website 'can I use' and type async/await on the search bar, and you will see the browser support.

https://caniuse.com/?search=async%2Fawait

Promise-based Fetch API

Previously we have already seen how we can create an **HTTP** request using the **XMLHttpRequest** object. This is an oddly named object and is old compared to the newer Promised-based fetch **API**. The **Fetch** will allow us to make **HTTP** requests and handle the responses much easier than its older brother - **XMLHttpRequest(XHR).** Therefore, the biggest difference is that the **Fetch API** uses **Promises** and provides a nice way to avoid callback hell. In simple words, the newer **HTTP API** is just a function called **fetch**(). Inside we can pass the **URL** endpoint as a parameter, and at the end, it will return a promise that resolves to the response of the request we sent. I know you want a real example to see the power of promises and Fetch API, so I decided that now is time for you to work on a very interesting example. I will use a public API containing Star War movies information for the following example. The name of the API is called '**swapi'**, and it contains all of the Star Wars data that you will ever need. I'm a very big fan of Star Wars movies, and that was the reason why I decided to use this API, which is free, to create an awesome example. Please read the full documentation here:

https://swapi.dev/

If you are using your computer, you can start working on this example with me from scratch, but if you are just reading this, please sit back and enjoy it because I will document each step. So, let us begin by creating a **fetchApi.html** file that will contain some basic HTML markup. We will use this empty skeleton file to populate with data from the API that we will fetch. Here is the entire HTML5 code for this example:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
 <title>Using the Promise-based Fetch API</title>
</head>
<body>
   <h1>Fetching data from SWAPI</h1>
   <h2>The Star Wars API</h2>
   <p> All the data is accessible through
      our HTTP web API.
   <br>
      Consult our documentation if you'd like to      get started.
        Helper libraries for popular programming      languages are also
<br>
      provided so you can consume swapi in your      favourite programming language,
     <br>
      in a style that suits you.
     <br>
        Read more Here :
           <a href="https://swapi.dev/documentation">
              Swapi Documentation
           </a>
   </p>
   <div id="output">
     <p>Name:<span class="name">Name</span></p>
     <p>DOB: <span class="dob">Brith</span></p>
     <p>His movies:</p>
      <ul id="movieList">
      </ul>
   </div>

   <script>
      //Code goes here
   </script>
</body>
</html>
```

As you can see from the HTML markup, which is very basic, I have left a few HTML tags empty. They are sitting there without content. Where would I get this data from? We can populate this markup, or we can easily make this static HTML page dynamic with data coming in from fetched swapi API. For this example, I would collect data for one special character in Star Wars: Darth Vader. I strongly advise reading the documentation of swapi because you can get data for any specific spacecraft, vehicles, planets, people, and so on. So after we fetched the data, we would not use everything from it; instead, I would only use the name, the birth year, and the list of movies he is in. For the movies part in the markup, I have created one unordered list <ul> tag that is empty. You should know that each <ul> tag can have a list of elements called <li>. I do not have any of these elements in the markup because I will be creating them inside the promise. This means that I will get a lot of data and an array of movies from the fetched API. This array will be an array of URL endpoints to each specific movie. What is the best way to get all the info from an array? Well, we can use the ‘for loop’ to iterate through each item. We can make separate fetch API calls for each iteration to get the data back for each movie. At first, this may look very difficult and confusing but let us start working, and you will see that it will not be that hard to implement all of the mentioned steps.
If you open the fetchApi HTML file in your browser, you would see this:

Fetching data from SWAPI

The Star Wars API

All the data is accessible through our HTTP web API.
Consult our documentation if you'd like to get started. Helper libraries for popular programming languages are also
provided so you can consume swapi in your favourite programming language,
in a style that suits you.
Read more Here : Swapi Documentation

Name: Name

DOB: Brith

His movies:

So we would not make any further changes to this file, but we will change its content when we have the data from the specified URL endpoint.
You can create an external index.js file in the same directory where you have the HTML file and reference the file inside our HTML file like this:

```
<script src="index.js"></script>
```

You should know that I would use inline JavaScript because it is much easier to have HTML and JavaScript in one file. The JavaScript code will be between both <script> tags. Okay, let us start working on fetching the data, and the first thing we want to do is use the fetch() method with the URL pointing to the swapi.
This is the URL endpoint we need to pass in as an argument inside our fetch method:

https://swapi.dev/api/people/4/'

```
const fetchPromise = fetch('https://swapi.dev/api/people/4/');
console.log(fetchPromise);
```

If you open the fetchApi file in your browser, you should check out the browser console. You should see something like this:

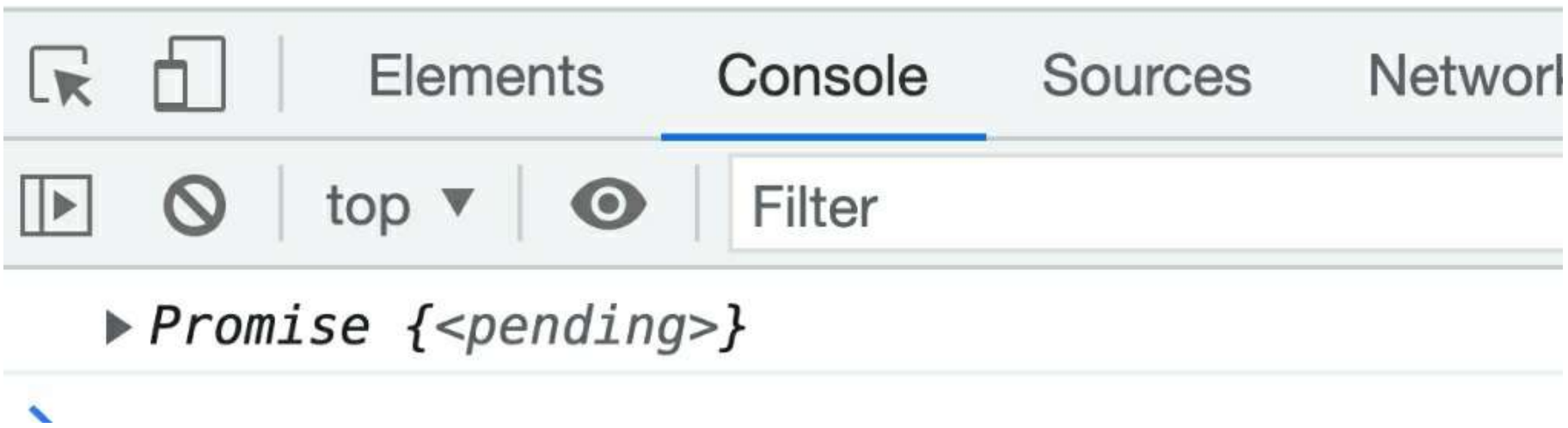
The Star Wars API

All the data is accessible through our HTTP web API.
Consult our documentation if you'd like to get started. Helper librar
provided so you can consume swapi in your favourite programming
in a style that suits you.
Read more Here : Swapi Documentation

Name: Name

DOB: Brith

His movies:



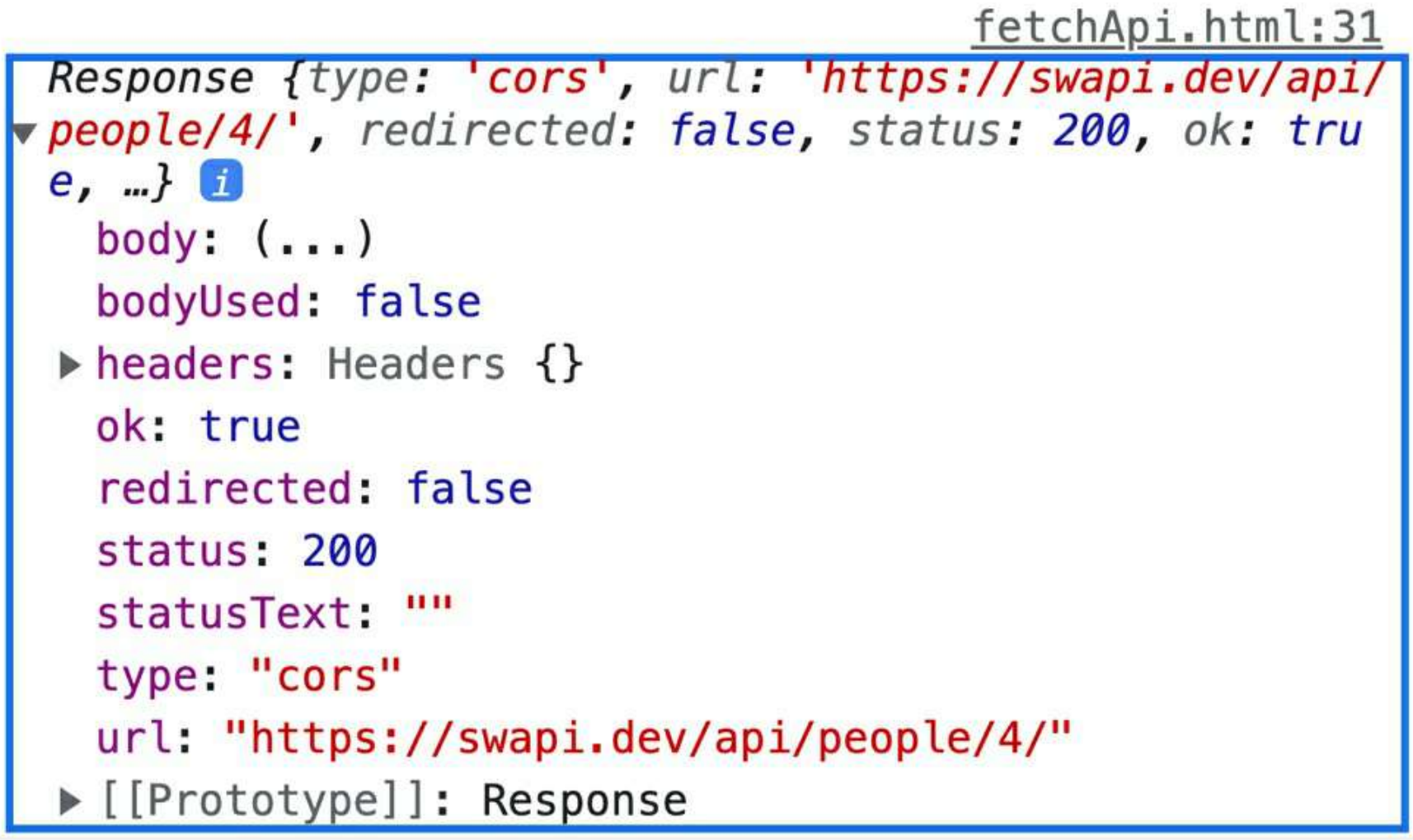
Please be patient. We need to write a few more lines of code until we get the real data. When we make an asynchronous HTTP request, the fetch method will not return real data but will return a promise. This promise that we got back is in the pending stage, and it means that the HTTP response will resolve eventually in some time in the future. From this stage, the promise can progress into the next stage that can be fulfilled if everything is ok or rejected if there is some error. Once the promise is settled, the state will be permanent. It will no longer be able to change. After a few seconds later, the promise state is now fulfilled:

How will we get data from this promise? To get data, we need to use Promise.prototype.then method to attach a callback since the promise was fulfilled.

```
const fetchPromise = fetch('https://swapi.dev/api/people/4/');
    fetchPromise.then(response =>{
```

```
        console.log(response);
});
```

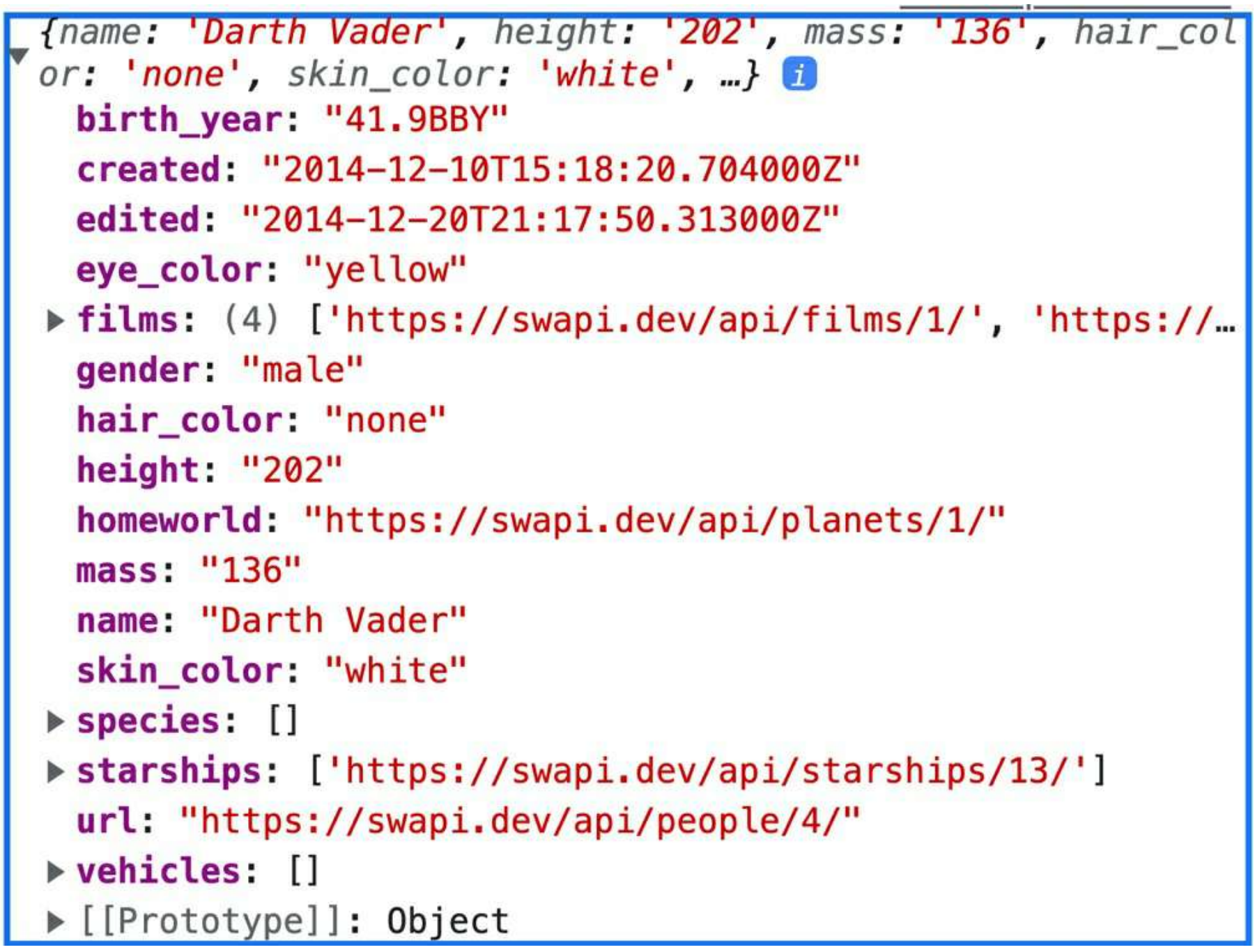
We console log the response to see what information we got back from the API. If we go back to our browser console and refresh the page again, we should have a different response that includes the headers, type, body, and status code as well:



This tells us that our API response is working as it should, but we do not want to stop here. We need more specific data. To get the response body in a JSON format, we can call the json method. JSON is a very simple format and became very popular when sending data over the network. JSON syntax is derived from JavaScript object notation, but you should know that JSON format is text only. JSON format is almost identical to JavaScript objects we know, and this similarity allows us to easily convert JSON data into native javascript objects. The operation we will perform is asynchronous because the json() method returns a Promise; therefore, we can create a Promise chain.

```
const fetchPromise = fetch('https://swapi.dev/api/people/4/');
      fetchPromise.then(response =>{
         return response.json();
      }).then(data =>{
         console.log(data);
});
```

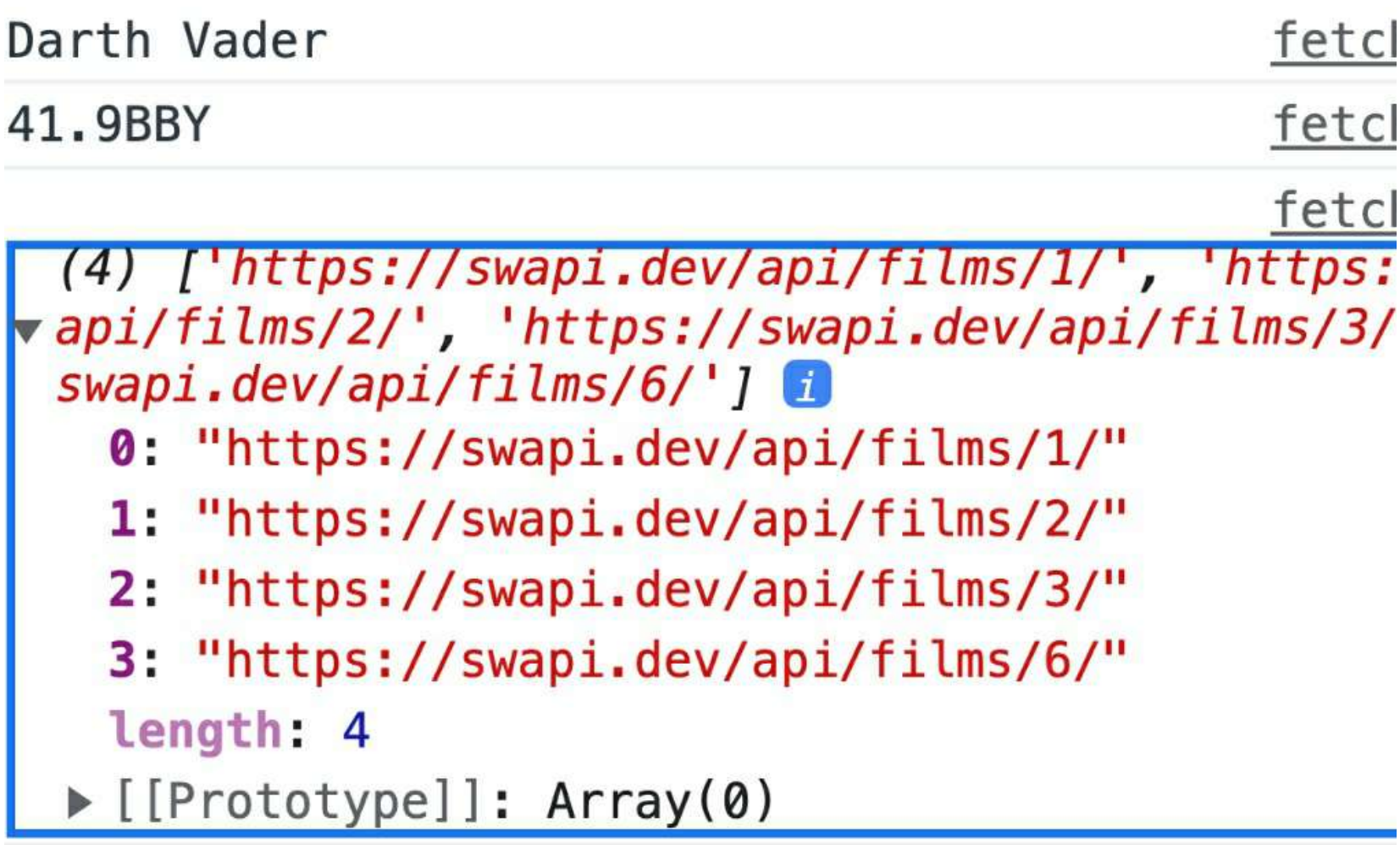
As you can see, we no longer console log the response, but we return a new Promise because response.json() will return another promise on which we can use another .then method. We passed the value we returned from the first Promise, and we just passed it in the new .then method as a **data** object. Refresh the browser and check the logs to see a new object containing all of the data about Darth Vader. Here is the output:



From the screenshot above, we have more information than we need for this exercise, and let us write a couple of lines of code to focus on displaying his name and birth year at first. We can use the object and the dot notation to grab the attributes we are interested in, like the name, birth year, and the movies or films as we have them in our data object. Now instead of consoling log everything, let us console log only the data we need:

```
const fetchPromise = fetch('https://swapi.dev/api/people/4/');
      fetchPromise.then(response =>{
         return response.json();
      }).then(data =>{
         console.log(data.name);
         console.log(data.birth_year);
         console.log(data.films);
});
```

Here is the output now:



This is great we have achieved what we wanted. We have the data now, but how will we render this data back to our HTML markup?
Looking at our HTML5 file, we can see that for the name we use the span tag that has class attribute **'name'** and for the birth year we have another span element with class '**dob'** f. So we can target these or any IDs or classes we want using the **CSS** selectors. Usually, you need to target these tags at the beginning of your **JavaScript** code, and here is how it will look:

```
const output = document.querySelector('#output');
const name = document.querySelector('.name');
const dob = document.querySelector('.dob');
const ul = document.querySelector('#movieList');
```

I have used the **querySelector** to target specific **HTML5** elements by their **IDs** and class names, and I have used both **IDs** and classes to show you how it can be done. We need to append the data fetched to the specific **HTML5** tags using the **innerHTML** property. The **innerHTML** element property will allow us to set new content of an **HTML** element.

```
const fetchPromise = fetch('https://swapi.dev/api/people/4/');
   fetchPromise.then(response =>{
      return response.json();
   }).then(data =>{
      const films = data.films;
      name.innerHTML = data.name;
      dob.innerHTML = data.birth_year;
});
```

This code should be sufficient to see new changes in our browser, so refresh and see if we can successfully render the data on our page:

Fetching data from SWAPI

The Star Wars API

All the data is accessible through our HTTP web API.
Consult our documentation if you'd like to get started. Helper lib programming languages are also
provided so you can consume swapi in your favourite programm in a style that suits you.
Read more Here : Swapi Documentation

Name: Darth Vader

DOB: 41.9BBY

His movies:

Great now only one thing left for us to do and that is to get all of the movies and render them inside our <ul> tag. This is where it gets a bit more complex but let me to add few more lines of code and after that we can discuss about it:

```
for(const film of films){
        const fetchMovie = fetch(film);
        fetchMovie.then(response =>{
           return response.json();
        }).then(movie => {
           console.log(movie);
        });
}
```

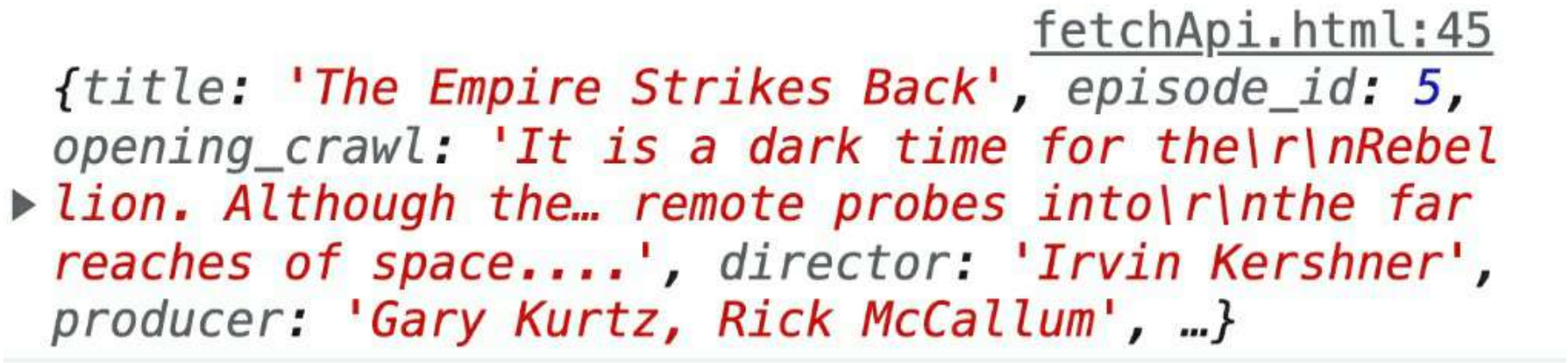
As you can see, we loop over the array of **URLs** using the **for/of** the loop. The **for/of** statement will iterate over iterable objects, including Arrays, Strings, or array-like objects.
To refresh your memory, the **for/of** syntax looks like this:

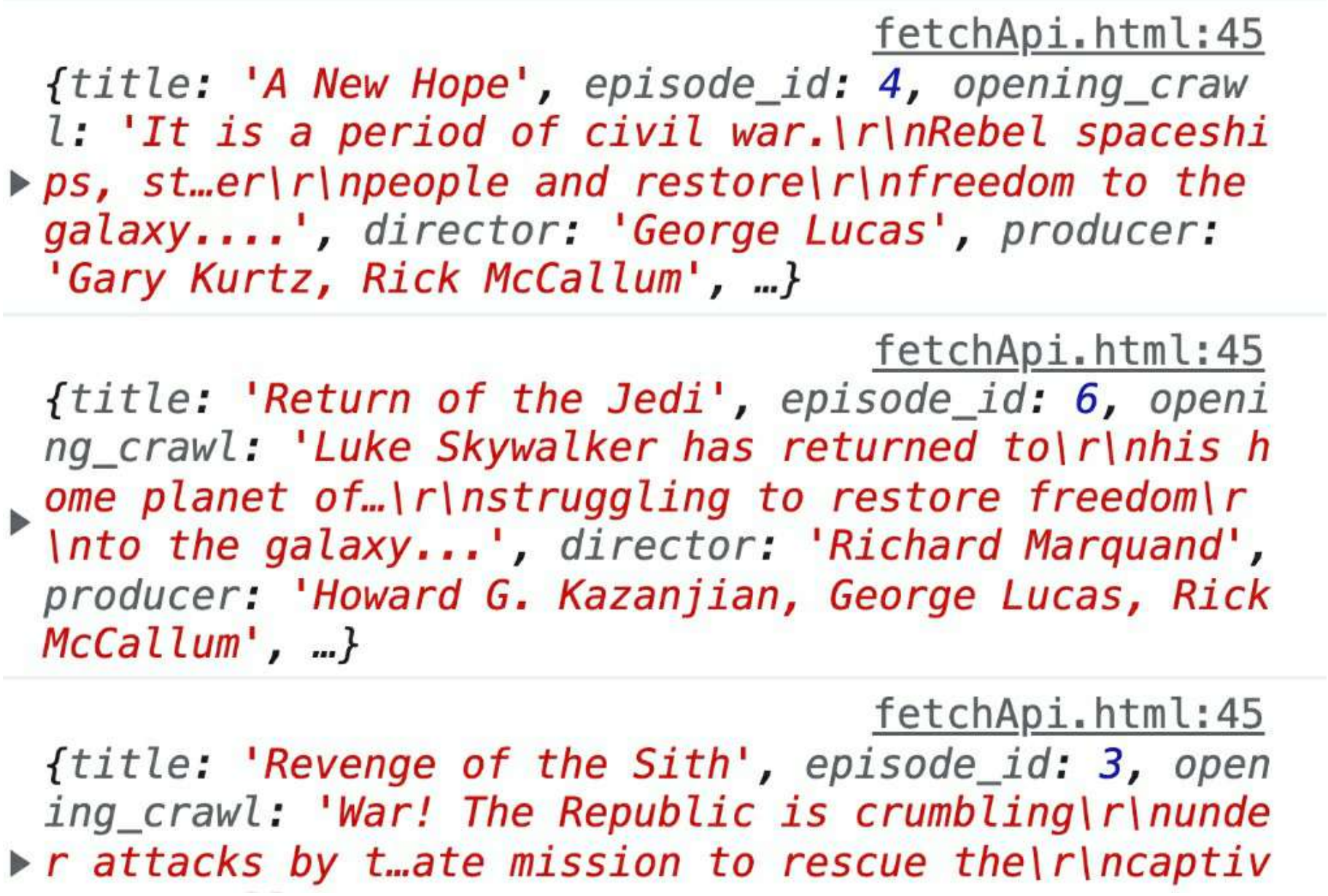
```
for (variable of iterable) {
  statement
}
```

Let me explain what variable and iterable means from the syntax above:

- **Variable** – on each iteration new value is assigned to a variable, and the variable can be declared as const, let, or even the old school var.
- **Iterable** – this is the object we want to loop over, which will be the entire film array.

Now you know how the for/of loop works, and in each loop, the variable **film** in our **for/of** loop will have a different URL address from the array. Therefore, we can use this variable and pass it to the fetch method.
Here is the output that we got:





Great now we see that our logic is working, and we can see that we got 4 successful responses back, and from these responses, we need to use the title of the movie only.
The rest of the steps we need to write are the same as we had in the first fetch call when we got back Darth Vader's data; therefore, I would not waste your time explaining the same code again. The only difference is that our **<ul>** tag is empty, and we need to populate with the data stored in list elements. But how are we going to achieve this? Check these few lines and see the explanation below:

```
for(const film of films){
        const fetchMovie = fetch(film);
        fetchMovie.then(response =>{
           return response.json();
        }).then(movie => {
        let li =document.createElement('li');
        li.innerHTML = movie.title;
         ul.appendChild(li);
        });
      }
```

Luckily, we can use the **createElement() HTML DOM** method to create an **Element Node** with a specified name. In our case, we want to create a list element <li>, and therefore, we need to pass ‘li’ to **createElement()** method. After creating this **HTML** element, we can use the **appendChild** method to append the already created element as the last child of the node.
Refresh your browser, and you should see the following:

Fetching data from SWAPI

The Star Wars API

All the data is accessible through our HTTP web API.
Consult our documentation if you'd like to get started. Helper librarie
popular programming languages are also
provided so you can consume swapi in your favourite programming l
in a style that suits you.
Read more Here : Swapi Documentation

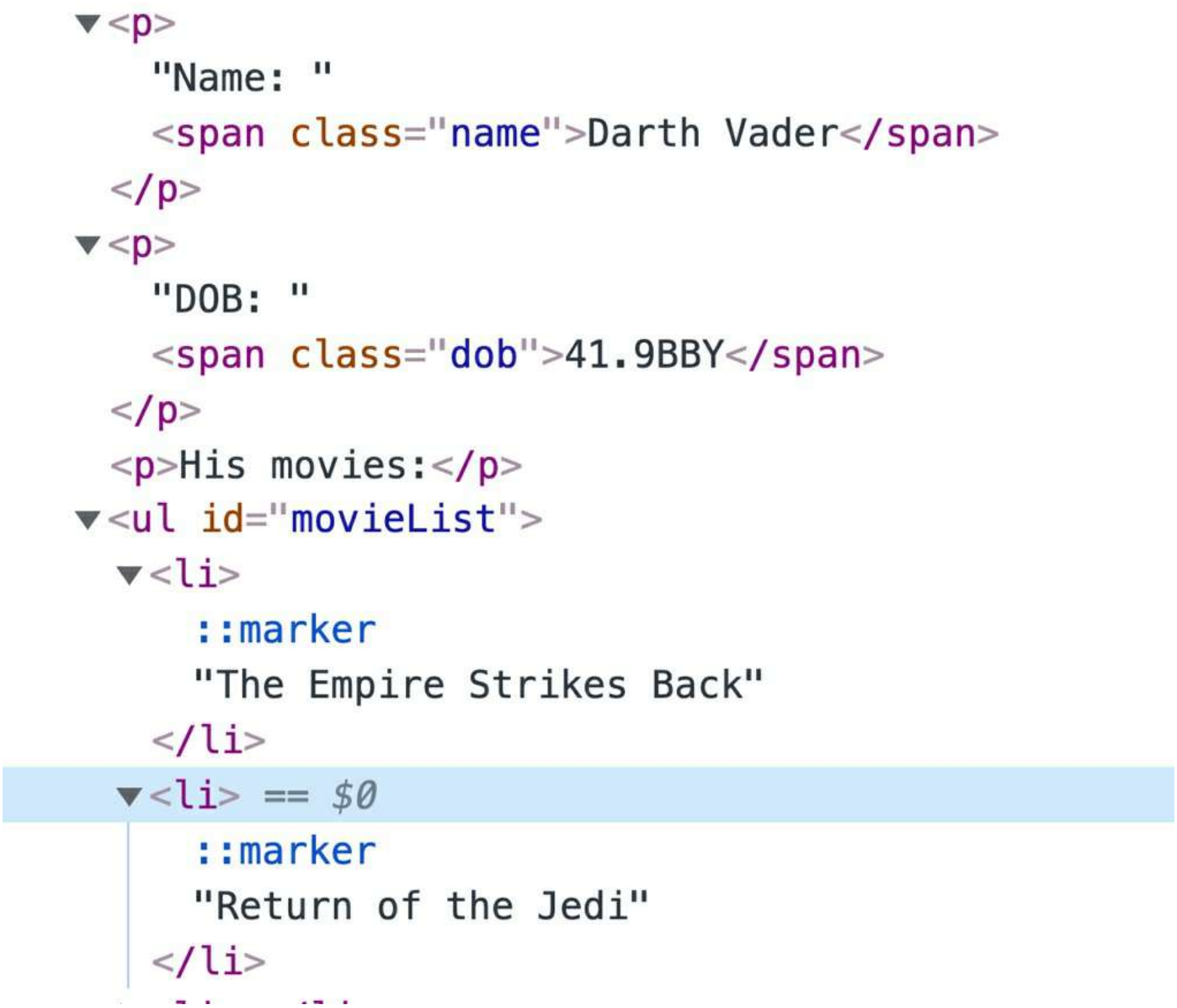
Name: Darth Vader

DOB: 41.9BBY

His movies:

- The Empire Strikes Back
- Return of the Jedi
- Revenge of the Sith
- A New Hope

If we inspect our page in our web browser, we will see the old markup we had plus the new markup with all of the new data and elements we created using the fetch and DOM manipulation techniques:



Congratulations now you know how to use the **Fetch** promised based **API** and here is the complete code:

```
const output = document.querySelector('#output');
const name = document.querySelector('.name');
const dob = document.querySelector('.dob');
const ul = document.querySelector('#movieList');


const fetchPromise = fetch('https://swapi.dev/api/people/4/');
fetchPromise.then(response =>{
  return response.json();
}).then(data =>{
  const films = data.films;
  name.innerHTML = data.name;
  dob.innerHTML = data.birth_year;
  for(const film of films){
     const fetchMovie = fetch(film);
     fetchMovie.then(response =>{
        return response.json();
     }).then(movie => {
        let li = document.createElement('li');
        li.innerHTML = movie.title;
        ul.appendChild(li);
     });
  }
});
```

But we are not finished yet. The code above is not reusable at all. For example, look at the part where we have the **for/of** loop. Can you spot the problem? Here we have too much code in one place, and we cannot reuse this code at all. The one improvement I suggest is to create a function that will deal with the code where we loop over the list of movies. This function we can use later as many times as we need. The new function can have any name, but I chose this '**listOfFilms'**, and it will take only one parameter that will be the array of movies.

Here is the entire code and see how better it looks now because it is much cleaner and more readable compared with the previous one:

```
const output = document.querySelector('#output');
const name = document.querySelector('.name');
const dob = document.querySelector('.dob');
const ul = document.querySelector('#movieList');


const fetchPromise = fetch('https://swapi.dev/api/people/4/');
fetchPromise.then(response =>{
   return response.json();
}).then(data =>{
   const films = listOfFilms(data.films);
   name.innerHTML = data.name;
   dob.innerHTML = data.birth_year;


});
function listOfFilms(films){
   for(const film of films){
      const fetchMovie = fetch(film);
      fetchMovie.then(response =>{
         return response.json();
      }).then(movie => {
         let li = document.createElement('li');
         li.innerHTML = movie.title;
         return ul.appendChild(li);
      });
   }
}
```

Perfect, now we have refactored some of our old code, and we learned how to make our new code much more user-friendly and reusable.
Finally, I would like you to think about how we can make our function **async/await** function? If you still cannot figure it out, it is very simple, and all we need to do is make our function asynchronous by adding the keyword async in front of it, and we also need to use the await keyword after each fetch response.
Here is how we can turn the **listOfFilms** from normal to **async/await**:

```
async function listOfFilms(films){
   for(const film of films){
      const fetchMovie = await fetch(film);
      const movie = await fetchMovie.json();
      let li = document.createElement('li');
      li.innerHTML = movie.title;
      ul.appendChild(li);
   }
}
```

And this is how we rewrite **listOfFilms** to be async/await. Make sure you comment the previous function because you cannot have two functions with the same name. As an exercise, try to create another function that will load the character, you can name it as you want, and this function should take one argument as well, the **URL** endpoint we used for when we fetched the **Darth Vader** data.
Here is my solution (the same code will be in **fetchApiFullAsync.**html file):

```
<script>
const output = document.querySelector('#output');
const name = document.querySelector('.name');
const dob = document.querySelector('.dob');
const ul = document.querySelector('#movieList');
const URL = 'https://swapi.dev/api/people/4/';


loadCharacter(URL);


async function loadCharacter(url){
   const fetchPromise = await fetch(url);
   const data = await fetchPromise.json();
   name.innerHTML = data.name;
   dob.innerHTML = data.birth_year;
   const films = listOfFilms(data.films);
}
async function listOfFilms(films){
   console.log(films);
   for(const film of films){
      const fetchMovie = await fetch(film);
```

```
      const movie = await fetchMovie.json();
      let li = document.createElement('li');
      li.innerHTML = movie.title;
      ul.appendChild(li);
   }
}
</script>
```

You can open the same file in your browser but remember to give it some time until it loads all the data completely.

Async/await Error Handling

We have seen how the async/await functions are created and how they are working, but I have never discussed the error handling mechanism. The error handling is done completely synchronously, and because of this, we need to use the good old try/catch statements.
Here is one example that will help you understand how we can handle errors (the code is in **asyncAwaitErrorHandling** file):

```
function trueFalse() {
   return new Promise((resolve, reject) => {
     const result = Math.round(Math.random() * 1); // it will be 1 or 0
result ? resolve('True  ') : reject('False  ');
   });
 }

 async function randomMessages() {
   try {
     const whatWillBe = await trueFalse();
     console.log(whatWillBe);
   } catch(err) {
     console.log(err);
   }
 }
 for (let i=0; i < 10; i++){
   randomMessages(); // Lucky!!
}
```

So the function trueFalse will probably reject a few times, and my screenshot here will be different from yours if you test the same code in your browser (because of the *Math.random* function):

```
False 👎
4 True 👍!!
2 False 👎
2 True 👍!!
False 👎
```

Here I got lucky, and I have 4 rejections, which is not bad. As you can see, **randomMessages** is an async function where I use the try and catch block to catch if an error happens, but since the async function returns a promise to us, we can use the catch statement as well.
Here is how we can achieve this:

```
> function trueFalse1() {
    return new Promise((resolve, reject) => {
      const result = Math.round(Math.random() * 1); // 1 or 0
      result ? resolve('True 👍') : reject('False 👎');
    });
  }

  async function randomMessages1() {
    const whatWillBe = await trueFalse1();
    console.log(whatWillBe);
  }
  for (let i=0; i < 10; i++){
    randomMessages1().catch(data => {
      console.log(data);
    });
  }
```

```
4 True 👍
6 False 👎
```

for-await-of

According to the MDN website, the '**for-await-of'** statement creates a loop iterating over async iterable objects as well as on sync iterables, including built-in Strings, Arrays, TypedArray, Map and Sets.' You should know that **for-await-of** cannot work on iterators that are not async iterables.
Syntax:

```
for await (const variable of iterable) {
  statement
}
```

From the syntax, the variables can be declared as const, let, or var, and on each iteration, a new value will be assigned to the **variable**. On the other hand, the **iterable** is an object whose properties are iterated over.
Before I explain about 'for-await-of' loop, I would like to talk about
async iteration and generators. We need asynchronous iteration when we have data that comes asynchronously, and we want to iterate over that data. I will start with a basic example about iterables and how we have a simple object and want to add iteration ability. This will not be an asynchronous iteration example, so let's start with a simple object that we want to iterate over.
For example, we have this object (code will be in lecture **for-await-of** file):

```
let pages = {
  from: 1,
  to: 10
};

for(value of pages){
  console.log(value);
}
```

If you run the same code in your browser console, you will get the output: '**TypeError: pages are not iterable'**. If we want to add this iteration ability to this object, we need to use **Symbol.iteraton**. This is not the only step we need to do because it gets more complicated. If we want to be able to use for-of loop on this object called 'pages', then we need to write this code:

```
let pages = {
  from: 1,
  to: 10,

  [Symbol.iterator]() {
    return {
```

```
    current: this.from,
    last: this.to,

    next() { // method is called in every iteration, to get the next value
      if (this.current <= this.last) {
        return { value: this.current++, done:false };
      } else {
        return { done: true };
      }
    }
  };
 }
};

for(let value of pages) {
  console.log(value);
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

On the other hand, when we have an asynchronous iteration, the values come asynchronously. This can happen if we have used the **setTimeout** method or when we are requesting data using the **fetch()** method. It takes more time until we get the requested data. How are we going to make an object iterable asynchronously?
At first, we need to make a few changes to our previous object example to return values asynchronously or, let's say that we want to have a value after two seconds. This way, the object will return values asynchronously, one after every two seconds.
Here is the example:

```
let pages = {
   from: 1,
   to: 10,

   [Symbol.asyncIterator]() {
     return {
       current: this.from,
       last: this.to,

       async next() { // method is called in every iteration, to get the next value
const result = await new Promise(resolve => {
setTimeout(() => resolve(this.current), 2000);
         });

         if (result <= this.last) {
           return { value: this.current++, done:false };
         } else {
           return { done: true };
         }
```

```
      }
    };
  }
};
const getResult = async () => {
  for await(let value of pages){
    console.log(value);
  }
}
getResult();
```

We have used regular iterators from the previous example, but we have to make a few adjustments to get the results back. If we want to make an object asynchronously iterable, we need to use **Symbol.asyncIterator**. The **Symbol.asyncIterator** is called only once by the for-await-of loop at the start, and after that, we will use the **next()** method for the values. Compared to the previous example, the next method will return a promise, which needs to be fulfilled with the next value. The next method in our case is async, but it does not have to be async, but because I wanted to use await for the results, it was logical to create an async/await function. As you can see, the delay timer is set for two seconds in **setTimeout** function. If we want to iterate over this object, we need to use for-await-of loop instead of for-of loop. This is very easy because we only need to include the 'await' keyword after the 'for' statement.
Hope you understand how we can make an object iterable asynchronously. Now let us talk about two very important topics recall generators and async generators.

Recall and async generators

In real life, when we want to make an object iterable, we will use generators as they are much more convenient and make our code look cleaner and less confusing. In simple words, the generators are functions that will yield results. Take note of the special keyword I'm using, 'yield' or generate/produce results. You will know the code uses generators because the functions have a star like this: **function***.
Example for regular generator (code in generators.js):

```
> function* myFirstGenerator(i) {
    yield i;
    yield i + 1;
    yield i + 2;
    yield i + 3;
    yield i + 4;
  }

  for(let value of myFirstGenerator(1)) {
    console.log(value);
  }
1
2
3
4
5
```

We can rewrite the example with pages because it is very common for **Symbol. iterator** to return generator. This will make our code shorter and cleaner:

```
let pages = {
  from: 1,
```

```
  to: 10,

  [Symbol.iterator]: function*() {
    for(let i = this.from; i <= this.to; i++) {
      yield i;
    }
  }
};

for(let value of pages) {
  console.log(value); //1, 2, 3 … 10
}
```

See, the code is much shorter and will yield the same result. Now, if we want to use async generators, we need to make a few changes. The reason why we need to use async generators is that the regular generators do not support the use of the await keyword. The syntax for async generators is a function* and then the async keyword. Instead of using a normal for-of loop, we need to use the for-await-of syntax.
Let us use the previous example with a few modifications:

```
//async generators
let pages = {
  from: 1,
  to: 10,

  [Symbol.asyncIterator]: async function*() {
    for(let i = this.from; i <= this.to; i++) {
      await new Promise(resolve => setTimeout(resolve, 2000));
      yield i;
    }
  }
};

const getResult = async () => {
  for await(let value of pages){
    console.log(value);
  }
}
getResult();
```

```
▶ Promise {<pending>}
1
2
3
4
…
```

The output will be up to a value of 10.

Exercise

Rewrite the same example using async/await (you can test out the same example:

```
async function test() {
   const arrayOfPromises = [
      Promise.resolve('first'),
      Promise.resolve('second'),
      Promise.resolve('third'),
   ];
for await (const promise of arrayOfPromises) {
      console.log(promise);
```

```
    }
}
test();
```

The output will be:

```
first
second
third
```

```
function loadUrl(url) {
    const result = fetch(url)
      .then(response => {
        if (response.status == 200) {
          return response.json();
        } else {
          throw new Error(response.status);
        }
      });
      return result;
}
//You can test out with this url as well:
//https://jsonplaceholder.typicode.com/todos/1
loadUrl('https://jsonplaceholder.typicode.com/todos/as')
.then(result => console.log(result))
.catch(data => console.log(data)); // Error: 404
```

Solution:

```
async function loadUrl(url) {
    try{
        const result = await fetch(url);
        if (result.status == 200) {
          let json = await result.json();
          console.log(json);
        }
        else{
            throw new Error(result.status);
        }


    }
    catch(error){
        console.log(error);
    }
  }
//loadUrl('https://jsonplaceholder.typicode.com/todos/1');
loadUrl('https://jsonplaceholder.typicode.com/todos/as');
```

Summary

Congratulations! This very important chapter taught us the difference between synchronous and asynchronous programming and why it is so important to use asynchronous functions as much as possible. We also learned how to create Promises, consume promises, and work with different promise methods. We also learned about async/await, how we can handle rejections and how to use for-await-loop. This was another chapter packed with information, and I hope you understand all of the new concepts with the examples I have provided.

Chapter 3: JavaScript Modules

If we want to create a code that many others will use, we can declare that code as a public one. The idea is to have two separate parts, the public that others can access and the private one that will contain classified information and be accessed only by a few developers. This can be achieved in two ways, the first one that developers thought was to use classes introduced in **ES6**. The classes are the main feature of any object-oriented language, not just JavaScript. The important concept of the classes is data encapsulation. This data encapsulation helps us hide the values or state of a structured object inside a class. This will stop someone from accessing the data directly. The second way to have private and public parts are to split our application into different files called modules. The modules can have classes or functions that serve specific purposes. At the very beginning, the JavaScript files were very small, and there was no need to have modules. Still, as JavaScript applications grew bigger, the need for modules became apparent, and therefore, we need to learn about modules and their rules/syntaxes. As I mentioned earlier, the modules can have classes and functions to control our application. In 2015, a simple module system was introduced, and since then, is very popular. This module system is currently supported by most modern web browsers and **Node.js**. This short chapter will focus on teaching you what JavaScript modules are.

What are modules?

As I mentioned in the introduction, the modules are nothing but files. For example, each of the scripts we have created so far is considered a module. We have features like classes, functions, and values inside each module. These features can be exported and become available in other modules. The features that are not exported are private to that module, so now we can have private and public features, which was the goal. A single module can depend on many other modules; therefore, we need to specify which other modules are needed so the JavaScript runtime engine can load the dependent modules before execution starts. We need to use two specific keywords when discussing modules: export and import.

- **Export keyword** – allows us to export variables and classes to other modules outside the current one
- **Import keyword** – allows us to import some functionality from other modules.

What is interesting with modules is that it prevents to have name conflicts. This means we can use the same names for private fields in different modules because the private properties stay hidden for the outside modules. Please note that the modules and classes are different, so do not confuse them. A single class can have many instances, but when it comes to modules this is not possible. You should consider modules as one big container with classes, functions, variables, and values.

Exports and Imports

As you know, we have two important keywords for modules, and they are import and export. As a developer, you will use both of them, but if you are working in a team, you will most likely consume the modules features because someone else will already write them for you. Therefore, the most common tasks you will perform are to import features from other modules and figure out where you can use them. What can we import from these modules? You will most likely import functions and classes, but you can also import arrays, primitive values, and objects. This means that if you can import these features, you can export them. In JavaScript, the modules have different types of imports and exports, but we will start with the export features first because we cannot import something unless we export it first. The folder for this chapter called **chapter3** will have another folder called '**modules**'. Inside, I will have all of the module files that we will work on. I will also have an index **HTML** file that will sit outside the modules folder. Let us start with the **module1.js** file. Inside this, we can write functions, classes, or primitive values to export them to other modules/programs.
There are two types of exports:

- **Named Exports** (we can have zero or more exports per module)
- **Default Export** (Only one default export per module)

```
//module1.js
//exporting individual features
export let firstName = 'Tom';
export let lastName = 'Holland';
```

As you can see from the code above, I have the export keyword before the variable name, and this will allow us to export individual features from this module.
Let us have a look at the **index html** file where we should have very basic **HTML5** markup and where we will import the exported features:

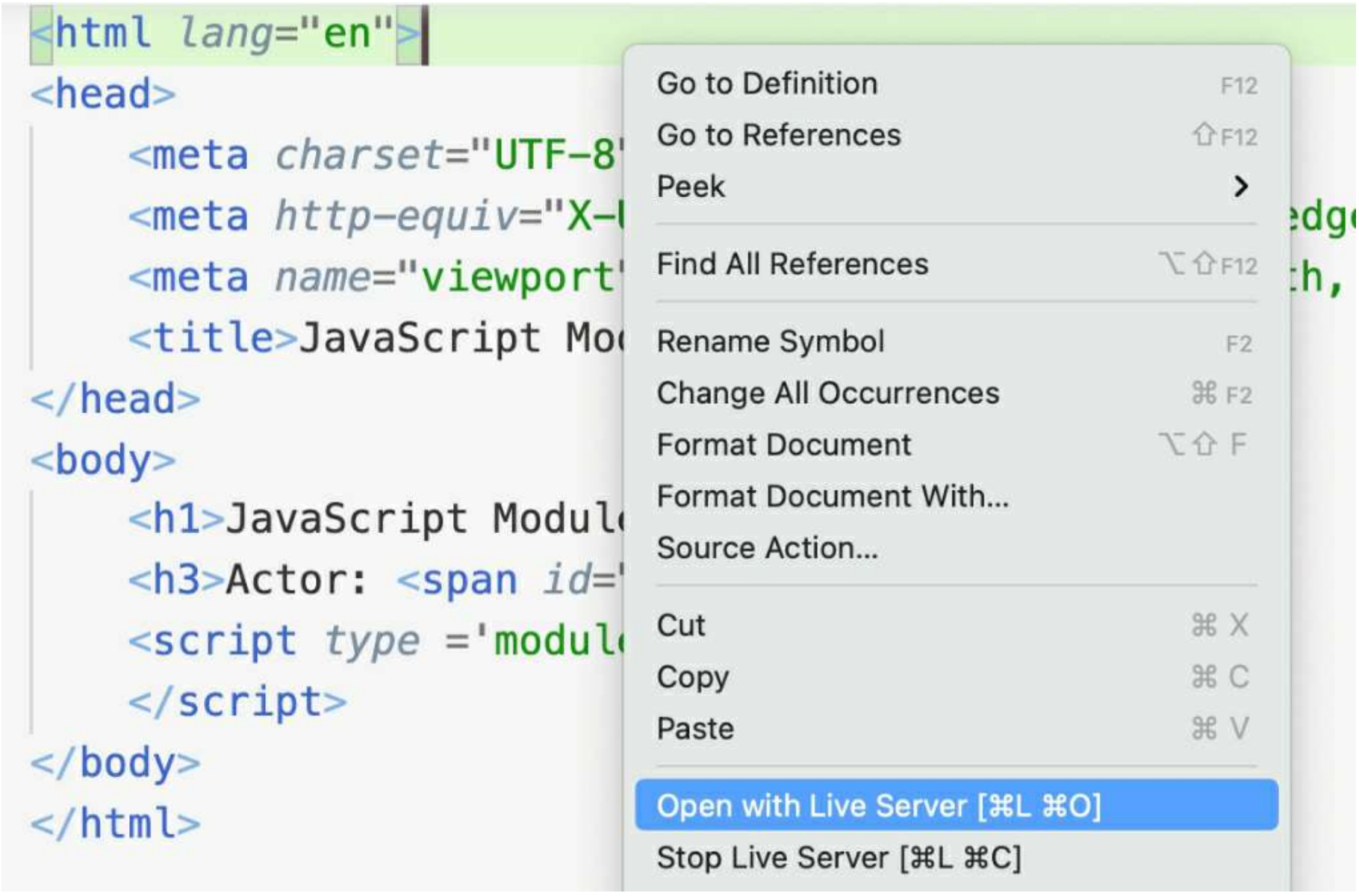
```
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>JavaScript Modules</title>
</head>
<body>
   <h1>JavaScript Modules</h1>
   <h3>Actor: <span id="actorName"></span></h3>
   <script type ='module'>
   </script>
</body>
</html>
```

As you can see here, I have used a script tag that has a type attribute with the value 'module'. When you work with modules, you need to tell the browser that the script you are trying to load inside your **HTML** file is a module, and

it is not just plain JavaScript code. If we want everything to work, we need to use the attribute **type='module'** inside our script tag. I'm using a VS Code editor, and I have installed an extension called '**Live Server Extension'**. If you are using the same text editor as me, please install it. Here is a link where you can read more about this extension:

https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer

Modules will not work locally, but they work via **HTTP(s)**; therefore, if you try to open this file via '**file://'** protocol, you will see that the **import/export** directives will not work. For this reason, I'm suggesting you use the local **web-server** or the '**live-server**' I just mentioned. If you are using other code editors, I'm sure you will find something similar to the *VS Code Live Server Extension*, and you can successfully test modules. Let us start exploring the modules, and first thing first, you can open the index **HTML** file with Live Server extension



If you check your browser, nothing will happen, the file will load with the current markup, but we haven't written any JS code between the script tags, so nothing will happen. So let us import whatever we exported from 'module1.js'. As I mentioned, we will go through different imports and export through this chapter, and just be careful because you need to get used to this new syntax. We need to do several things when importing a single export from a module. First, we need to know the name of the feature we exported explicitly, and the name was **'firstName'**. This variable we need to import in our **HTML** file, so we need to store it under some name. It is mandatory to use the same name of the corresponding feature we exported in our **HTML** file, and the name should be between two curly brackets:

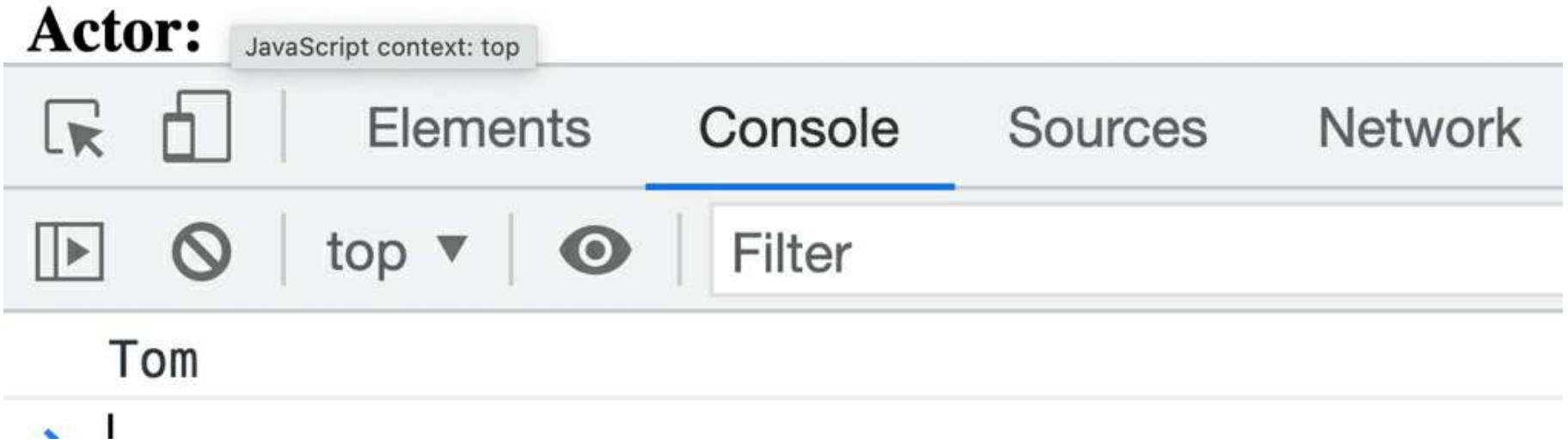
```
import {firstName} from './modules/module1.js';
console.log(firstName);
```

So we exported **firstName** variable from module1 file, and we imported in our **HTML** file using the import keyword and the same name we exported in curly brackets.
As you can see, the import statement loads the module if we provide the actual path relative to the current file:

```
'./modules/module1.js'
```

Then it will assign the exported variable firstName to the variable with the same name in the curly brackets. If we run this example and open the browser console, then you should see this:

JavaScript Modules



Great the export and import statements are working as intended. Next in the module1 file I have another export and that is the lastName variable. So, we can import that one as well:

```
import {firstName} from './modules/module1.js';
import {lastName} from './modules/module1.js';
console.log(firstName);
console.log(lastName);
```

Sure, this will get us the result we want, but it is obvious that we have code repetition importing two named exports coming from the same file. Luckily for us, we can import multiple exports from a module with only one import line just like this:

```
import {firstName,lastName} from './modules/module1.js';


console.log(firstName);
console.log(lastName);
```

We can do the same thing in the ‘module1.js’ we have two separate exports for the two variables; therefore, we can write one single export that will export those two features:
Check this example:

```
//exporting list
let list1 = 'list1';
let list2 = 'list2';
let list3 = 'list3';
export {list1, list2, list3};
```

In this example, we first declare the variables and then export them as a list of variables.
What else can we export from modules? Well, as I mentioned before, we can export a lot of things. Let us go through this example:

```
// export an array
export let daysArray = ['Monday', 'Tuesday', 'Wednesday','Thursday', 'Friday', 'Saturday','Sunday'];


// export a constant
export const URL = 'https://developer.mozilla.org/';


// export a class
export class User {
  constructor(name) {
    this.name = name;
  }
}
export function userDetails(user) {
    return user;
}
```

As you can see from the example above, we can export a lot of different features, and when we use the export keyword before a class or a function, this does not mean we are trying to make a function expression. Please make sure you do not use semicolons after the function and class declaration:

```
// export a class
export class User {
  constructor(name) {
    this.name = name;
  }
}//; no semicolons please after class declaration
export function userDetails(user) {
    return user;
}//; no semicolons please after function declaration
```

Our **module1** file contains a lot of data and a lot of exports but you do not need to be worried because this is just an example, and it will be pointless to create separate files for all of the exports we will go through. I also mentioned that we can have named exports as many as we want per module, but we can have only one default export per module. Let us see how we achieve this. I will put a default export in the same **module1** file, but in real life, it is not good to mix both named and default exports in one file, so either the module will have one default export or multiple named exports. It is up to you. In our case, it is totally fine because we are learning how things work. Inside the **module1** file, I will have a class called Person, and I will put export default before the class name:

```
// export a class
export default class Person {
    constructor(name) {
      this.name = name;
    }
}
```

You should know that we can import the export default features, but we need to do it differently from the named exports. If you look back in the previous import statement, you will note that I used curly brackets for importing the named exports, and it was mandatory to use the same name in the import as the name in the export. Here is a bit different. The default export can be imported by any name we want, and we do not need to use curly brackets.

```
import Person from './modules/module1.js';
const tom = new Person(firstName);
console.log(tom);
```

Even if we use different name, it will still yield the same result:

```
import P from './modules/module1.js';
const tom = new P(firstName);
console.log(tom);
```

Output:

```
Tom
Holland
▶ Person {name: 'Tom'}
>
```

It is very important to understand that we can rename an export when importing it. I know you thought there must be a way how to rename the named imports, and you were right there are. The renaming of the exports in our index file can be done using aliases. For example, we have this in our **module1** file:

```
let list1 = 'list1';
let list2 = 'list2';
let list3 = 'list3';
export {list1, list2, list3};
```

Instead of importing it like this:

```
import {list1, list2, list3} from './modules/module1.js';
```

We can use aliases and give names that will be different from the ones we exported:

```
import {list1 as l1, list2 as l2 , list3 as l3} from './modules/module1.js';
```

```
console.log(l1);
console.log(l2);
console.log(l3);
```

You can also use the same method to rename the exports we have in the module files, and by doing this, you will avoid naming conflicts when importing them:

```
//rename named exports
function userDetails1(user) {
   return user;
}
export{userDetails1 as currentUser};
```

We export the **userDetails1** function under the name of the **currentUser**. When we import this, we need to use **currentUser**, but we can again rename it to be even shorter, so in our index file, we can have something like this:

```
import {currentUser as c} from './modules/module1.js';
const user = c(firstName);
console.log(user);//Output: Tom
```

At the moment, our **module1** file exports to many things, and we import them under different names in the index file. Instead of doing this, what we can do is we can include all of the exports into a single object using the '* as':

```
import * as module1Obj from './modules/module1.js';
```

This import statement will import everything that we exported from **module1** file, and they are now stored in the **module1Obj**. But how can we now access the different exports we exported in our index file? For example, how we can access the **lastName** variable we exported from **module1** file? We can use the dot notation to access the exported features and their names. Please make sure you understand that module1Obj contains all of the exports from the module file so using the dot notation we can get specific feature from the module1 file:

```
console.log('We can get last name like this: ' + module1Obj.lastName);
```

We can import the entire module without specifying what to import if this is something we need to do. This will run the module in the global scope, and it will not be able to import any values because of it:

```
import './modules/module1.js';
```

Exporting features without a name

We can choose not to name the entities we are trying to export from our modules, but you should know that this can be achieved using the default export only. We can do this because there is always one export default per module, and therefore the import without curly brackets will know exactly what we are trying to import.
Here are few examples where we have no name default export:

```
// Examples only – one export default per file
export default class {// no class name provided
   constructor(name) {
      this.name = name;
   }
 }
//Second Example:
export default function(user) {// no function name provided
   return user;
}
```

If we try to do this for the named exports, we will run into an error because non-default export needs to have a name.

```
//This is not going to work
export default class {// Error!
   constructor(name) {
      this.name = name;
   }
 }
```

Default keyword as reference

Imagine we have a function like this in our module file:

```
function greetings(name) {
```

```
 return `Hi there: ${name}`;
}
```

We want to ‘export default’ this function without writing export default during the function declaration. This is not very common, but I still believe it is useful to know:

```
export {greetings as default};
```

This is basically as we wrote ‘export default’ before the function declaration. As I mentioned is not a good idea to have mixed exports, named and default export in one single file, but if we do, how can we catch all of these exports with one single import line? We have already done this for the named exports but see the following example. Here is what we want to export from our **module2** file:

```
function greetings(name) {
 return `Hi there: ${name}`;
}
export function cube(x) {
   return x * x * x;
}
export class Person {
   constructor(name) {
     this.name = name;
   }
 }

export {greetings as default};
```

Bellow you will see how we are going to import all of the named exports and the single export default in one line in our index html file:

```
//index.html

import {default as greetings, cube, Person} from './modules/module2.js';
console.log(cube(3));
console.log(new Person('Rick'));
console.log(greetings('Rick'));
```

Here is the output:

Re-exporting

Imagine that we have multiple files inside a single package, and we have many things exported. The packages are maintained and managed by the package managers like the NPM, but this is not the case here. I just want to provide an example for re-exporting. So each of these files has important exports, but imagine if someone else wants to work with us, so he/she needs to go into each of our files and select what to import. This problem can be solved using the re-export feature. This means combining several exports from various modules and creating only one central export module. Now the users will need to look at one single file and import the features they need. Let us do some very basic examples so you can understand this new concept:

```
//module3.js file


const calcAddition = (n,m)=>{
   return n + m;
}
const calcSubstraction = (n,m)=>{
   return n - m;
}
const calcMultiplication = (n,m)=>{
   return n * m;
}
```

```
const calcDivision = (n,m)=>{
   return n / m;
}

export {calcAddition, calcSubstraction, calcMultiplication, calcDivision};
```

In this file, we have four functions that are doing very basic arithmetic operations, and we export all of them. In another module, we have a single function that will return whatever we pass into, but we will use it to return our name only:

```
//module4.js file

export function tellMeYourName (name) {
   return name;
}
```

The last step is to combine the exports from the two files into a single module and then re-export them from the new file. I know it sounds confusing, so here's how the **main** file looks:

```
//main.js file

export * as Calculate from './module3.js';
export {tellMeYourName as name} from  './module4.js';
```

In the first line, we are re-exporting the exports from the module3 file, and the second line is where we re-export the exports from the module4 file. The syntax for re-exporting is similar to the import one, but we are certainly re-exporting them. The advantage here is that any user can import only this file 'main.js' that aggregates multiple exports. Let us see how we can import the re-exported features in the index file:

```
//re-export in index.html file


import * as Main from './modules/main.js';
const n = 12;
const m = 4;
const name = Main.name('Andy');
console.log (`Hi ${name} these are the calculations:`);
console.log(' + => ' + Main.Calculate.calcAddition(n,m));
console.log(' - => ' + Main.Calculate.calcSubstraction(n,m))
console.log(' * => ' + Main.Calculate.calcMultiplication(n,m));
console.log(' / => ' + Main.Calculate.calcDivision(n,m));
```

If we look in our browser console this is the output:

```
Hi Andy these are the calculations:
 + => 16
 - => 8
 * => 48
 / => 3
>
```

Dynamic Imports

So far, we have worked with imports/exports statements that were both static. We need to follow the rules and their syntax to get a result. The question is how we can import a module dynamically or on-demand.
If we want to load a module dynamically, we need to use the **import()** expression. This expression will return a promise that will resolve into a module object. This module object will contain all of the exports.
The syntax is:

```
import(modulePath)
```

Before with the static statements, we have called all of the modules at the top of our code, but when we are dealing with dynamic imports, we can call them anywhere in our code.
Here are a few functions that I will export them first so I can import them dynamically later in the index file:

```
//module5.js file


function first() {
   console.log(`First`);
}
function second() {
   console.log(`Second`);
}
function third() {
   console.log(`Third`);
}
function fourth() {
   console.log(`Fourth`);
}


export {first, second, third, fourth};
```

Now in the index file, I will import them dynamically, and then we can even store the promise into a variable:

```
//index.html


let module5 = import('./modules/module5.js');
console.log(module5);
```

Here is the output:

If we want to have a result we need to consume the promise we got and I hope you remember how this was done:

```
//index.html


let module5 = import('./modules/module5.js');
//console.log(module5);
module5
.then(data => console.log(data.first()))
.catch(err => console.log(err));
```

This also works in async functions as well, but we need to use the **async/await** keyword:

```
//index.html


console.log(`********************
      Async import`);


async function load() {
   let module5Data = await import('./modules/module5.js');
   module5Data.first();
   module5Data.second();
   module5Data.third();
  module5Data.fourth();
}


load();
```

Please comment the previous code before you test this:

```
**********
      Async import
```

```
First
```

```
Second
```

```
Third
```

```
Fourth
```

Now in the **module5** file, let's add another function that will be export default.

```
//module5.js
export default function (){
   console.log(`Fifth`);
}
```

Here is how we can get the export default features in our async function:

```
//index.html

async function load() {
let module5Data = await import('./modules/module5.js');
      module5Data.first();
      module5Data.second();
      module5Data.third();
      module5Data.fourth();
      //calling the export default
      module5Data.default();
}
load();
```

Please note that dynamic imports are treated the same as the regular scripts, so it is not necessary to use the attribute:

```
script type="module"
```

Important to know!

So far, we know that the modules are treated differently from the JavaScript files. It is important to note that the code in the modules is always executed in strict mode. The module is processed only once regardless of how many times it is loaded. The modules are processed asynchronously, and each module can have imports and export statements. We can have multiple exports in each module but only one export default statement. The scope of the module starts from the top level and is different from the global scope that we have in JavaScript. When we load a module inside the HTML file, we need to use the script tag with a type attribute equal to a module. Working with Node.js, you probably have seen this file extension '.**mjs**'. These file extensions indicate that we are dealing with modules, not plain **JavaScript** files.

Summary

Congratulations! This was a short chapter compared to the previous ones, but it was identically important as the preceding chapters. In this chapter, we have learned a lot about modules how we can import and export certain features from them. We also learned about the syntax and rules we need to work with them. Finally, we have covered dynamic modules that are slightly different from static ones, but they are not that hard because you can easily handle promises.

Chapter 4: Basic to Intermediate JavaScript

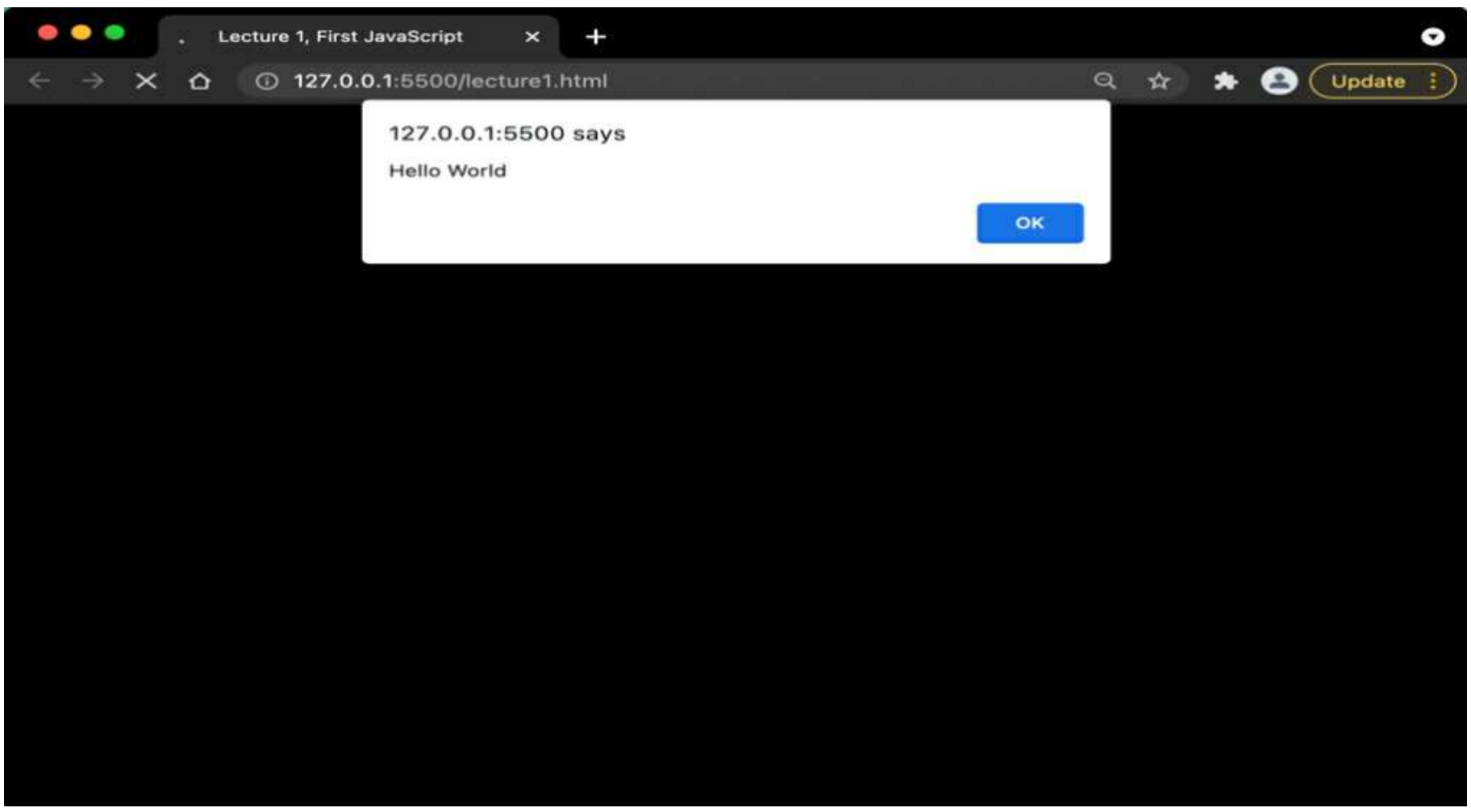
This chapter will cover the basics of JavaScript. During this book's planning stage, the basic to intermediate features were not considered, so you can treat this whole section as a bonus section. I promise it will not be boring because we will cover some intermediate JavaScript topics with examples. This chapter aims to equip you with enough knowledge to master the first three chapters. Who is this chapter for? This chapter is for everyone who wants to learn JavaScript basics before diving into the more advanced features. This chapter is also for readers who have already covered the basic concepts before but are still not confident enough to dive into the advanced features. JavaScript can easily be forgotten like any other programming language, so I strongly encourage you to read this crucial summary.

How to Run JavaScript?

In this section we are going to look into few ways how we can run JavaScript code. In this book I will be using a **VsCode** code editor, but any code editor will be okay. You can download the VsCode editor if you visit the following link:

https://code.visualstudio.com/download

Make sure you select the appropriate version for your operating system - the installation is straightforward, and you will not have any issues. If you are reading this on an e-reader, please sit back and enjoy because I will provide enough screenshots and code examples to understand what I'm trying to achieve.
So, if you open the downloaded files and go to the folder called **chapter4/1,** you will see inside an **HTML** file called **lecture1.html**. You can double click this file, and it will/should be opened inside your default browser – the result would be like this:



If you want to see the structure, **HTML** markup, and **JavaScript** code, then load the file into your code editor – this is how the file looks like:

As you can see in the example above, I have used window.alert method to display whatever I have stored inside the

helloWorld variable. Do not worry about methods and variables at this stage, but what is important for you is to understand how we can run JavaScript code.
We do not need a file like "**lecture1.html**" to write and execute JavaScript code. We can simply use our browser to run the code. Open the JavaScript console or the developer tools. For different browsers, the developer tools are displayed differently. Here are the steps you need to open the console in the two most common browsers, **Google** and **Mozilla**.

Google Chrome developer tools:

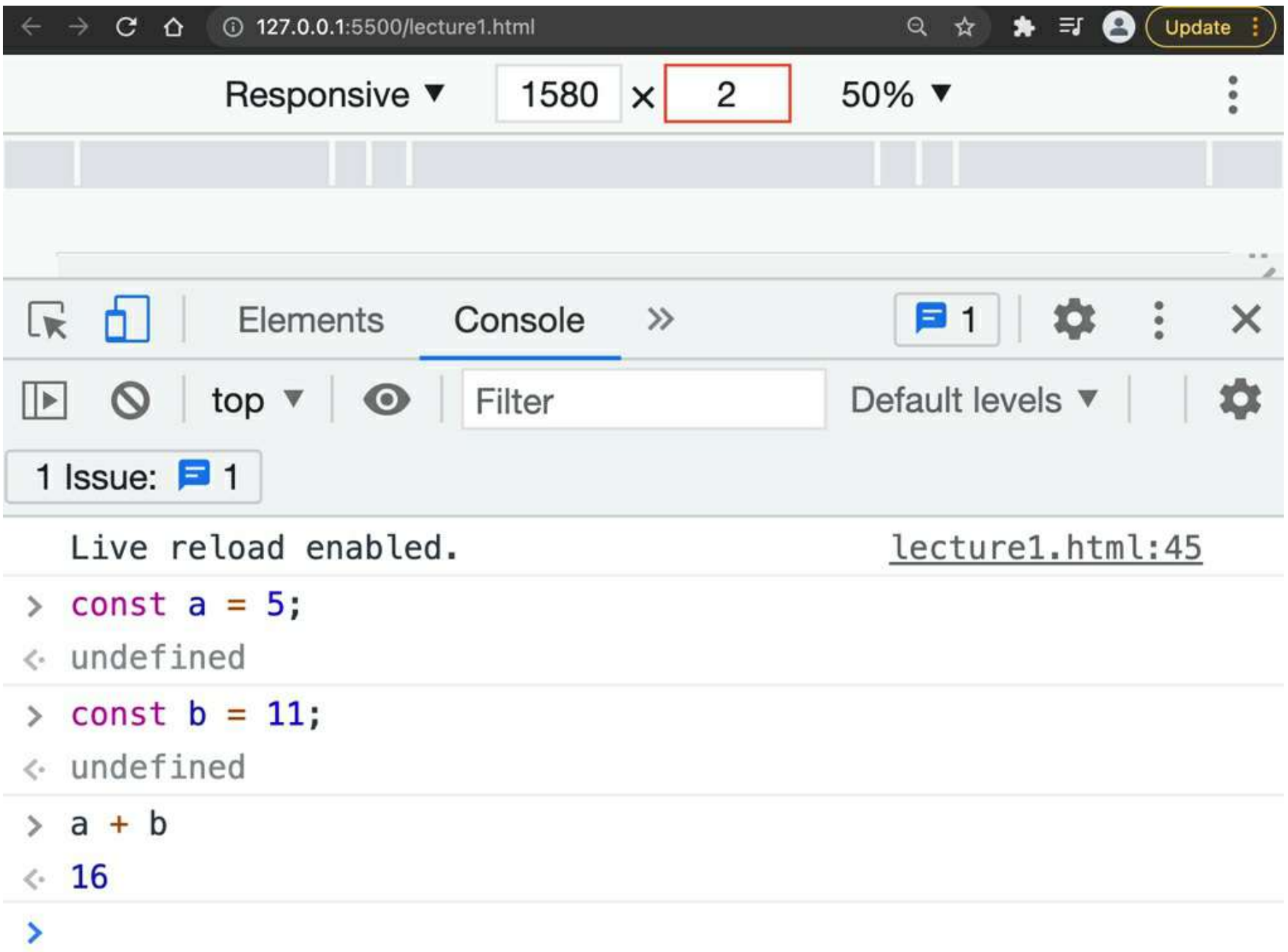
- Navigate to any desired page – like **google.com**
- For Mac press (CMD +OPT + J) – this should open a window
- Windows (CTRL + SHIFT + J) – this should open a window
- F12 sometimes works, sometimes not.

After you test some of the above steps, you should see the developer tools window, either will be below or on the right side. You can increase the height and width of that window because, by default, it will be very small. Please spend time getting the size of the console that is perfect for you. You can detach the console as a separate window if you like.
Lastly, in the developer tools window, please click on the console tab "*sometimes you might have some warnings and errors in the console, but you do not have to worry about those because they are not from your code*."

Mozila Firefox developer tools:

- Navigate to any desired page – like google.com
- For Mac press (CMD +OPT + J)
- Windows (CTRL + SHIFT + J) or (CTRL + ALT + I)
- F12 sometimes work

The rest of the steps are the same as for Google browser.
The last remaining step is to type in our JavaScript code – see the figure below:



Great! Now you know how to see and edit your code, and I have given you two options. The first one was to use the code editor, where you can write and edit the code. For this method, you will need an HTML file to load the file in the browser to see the result. The second method was to use the browser console where you can write and see the output of the code. There is another way how we can write/execute JavaScript code.
This method is more advanced. Let say that more experienced developers like this approach more. With this approach, you need to use the terminal and **Node.js.** Therefore, you need to download and install Node.js on your computer. You can download the Node from this website:
https://nodejs.org/en/
Once Node is installed and running on your system, you can open the terminal window and type **node**. This will lunch the **REPL** loop known as JavaScript "read-eval-print loop."
You can see my interaction with Node here:

```
Last login: Fri Jul 30 11:00:16 on console
rick@Ristes-iMac ~ % node
Welcome to Node.js v14.15.5.
Type ".help" for more information.
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the REPL
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file

Press Ctrl+C to abort current expression, Ctrl+D to exit the REPL
> const a = 5;
undefined
> const b = 7;
undefined
> a+b
12
>
```

Personally, I would like to use the code editor (to write and edit code) and load the same file in a browser to see the output. Another awesome feature of **VsCode** editor is that it supports the installation of extensions. For example, the "Live Server" extension will help you automatically load the HTML and JavaScript files. As soon as you make changes, it will listen for those changes and refresh the page to update the output. You can read more about this extension if you open the following link:

https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer

Let us see example how we can run JavaScript code from a file. To do this we need to have an **HTML** file and inside this file we will put a link to external JavaScript file. Take note that HTML files does have .**html** extension: This is the HTML markup (the same markup is in **first.html** file):

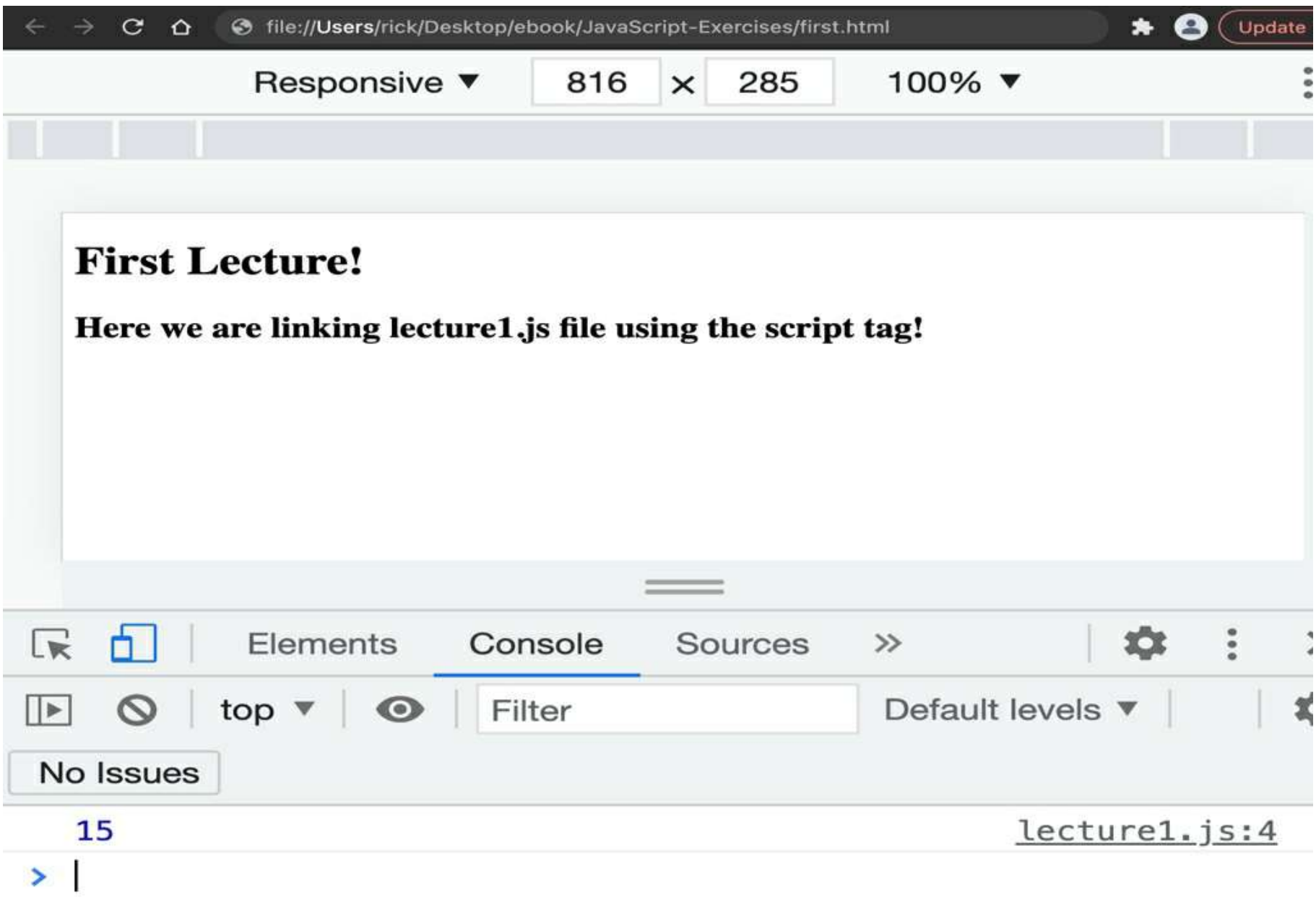
```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Lecture 1, First JavaScript</title>
</head>
<body>
   <h1>First Lecture! </h1>
   <h2>Here we are linking lecture1.js file using the script tag!</h2>


   <script src="lecture1.js"></script>


</body>
</html>
```

As you can see from the code above, the external JavaScript file is linked before the body tag is closed.

Now, if you haven't already downloaded the *live server extension*, do not worry; you can still load the file into your web browser. Locate where the actual file is (and the name is **first.html)** and double click on it. Eventually, you can load it into your web browser using a *file://* and URL, just like my one below:

file://Users/rick/Desktop/ebook/JavaScript-Exercises/first.html.

The result should be like this one:



Next is to open the browser developer tools, and as you can see, I already have opened them, and I can see the output is 15. Use the shortcuts to open the developer tools. How did we get the result? Well, the HTML file only contains the HTML tags, but it also contains a reference or a link to the external JavaScript file:

```
<script src="lecture1.js"></script>
```

This link connects the HTML file with the JavaScript file. And that is all you need to know how to link two files.

How to write comments in JavaScript?

In JavaScript, there are two types of comments. One type is the single-line comment, and we write the comment using //. The text that follows after the // is treated as a comment and is ignored by the JavaScript compiler. Another type of comment is the multi-line comment. Open the file called comments.js in **chapter4/2**, and you will see the following code there:

```
//Comments in JavaScript
//This is a Single Line comment
/*
Multi-line comment
Multi-line comments can also be used as single-line comments.
*/
```

But why do we need comments? Well, the comments are for us that we write the code and for other programmers that will read our program in the future, so they can understand what we have done or what was our logic behind it.

Identifiers

We use identifiers in JavaScript to name variables, properties, and objects, classes, and functions; please do not

worry about the terminology at this stage because it will be explained as we cover more material in the future. There are specific rules that we need to follow if we want to have legal identifiers.

The rules:

- JavaScript identifier must begin either with a letter, underscore ‘_ ‘ or a dollar sign ‘ $ ‘
- The first character must not be a number or digit
- The subsequent character after the first one can be letters, digits, underscore or dollar sign, and any combination of those.

Identifiers can also be single letters like a and b, but I always suggest using more descriptive and meaningful names.

Statements

In JavaScript we have statements. Here is an example with three statements:

```
let firstName = 'Rick ';
let lastName = 'Sekuloski';
alert(firstName + lastName);
```

Do not worry about the variable names and methods I have used in the example. Here I have three lines of code, and each of those lines is separated by a semicolon (;). This is required in most programming languages, but the semicolon at the end of the statement is not required in JavaScript. So, what is a JavaScript **statement**? The statements tell the computer program to do something. The first one is when I assign a value to a variable called **firstName**. The second is another assignment statement. The last one is an output statement that will output what the first and last variables contain. You can find the same code in lecture **statements.js**.
Another important rule is that we can group two or more statements in one line as long as semicolons separate them.

```
//grouping the statements
let number1 = 4; let number2 = 5;
```

Case Sensitivity

What you need to know is that JavaScript identifiers are case-sensitive. This means that if we name a variable only with a single letter like ‘**x’**, we can refer that variable in the program as ‘x’. The variables like ‘**x**’ and capital letter ‘X’ are two different variable names. Why? Because JavaScript is a case-sensitive language, all keywords or variables must be typed with consistent capitalization.

Primitive and Object types

JavaScript types can belong in two categories:

- **Primitive types**
- **Object types**

JavaScript primitive types can be:

- Strings
- Numbers
- Boolean values (Booleans for true or false values)
- Bigint
- Undefined and null are regarded as special values
- Symbol

Any value that is not a primitive value is considered to be an object. An object is different than the primitive values because each object can have multiple properties. Each property has a name and a value.
The difference between the primitive and object types is that the primitive types are stored by value, and the non-primitive (objects) are stored by reference.

Variables and Assignment

One of the most important and fundamental concepts in JavaScript is variable declaration and value assignment. A variable is a container that exists so we can store a value. Variables have names, and the names must follow the same rules we have explained about the identifiers. The most important part is to **declare** the variable before we can

use it.

Declaring a variable

So far we have used simple variable names but the more complex variables names are usually a composition of two words. When we have complex variables names we can use a convention called **camel case**. Here is one example:

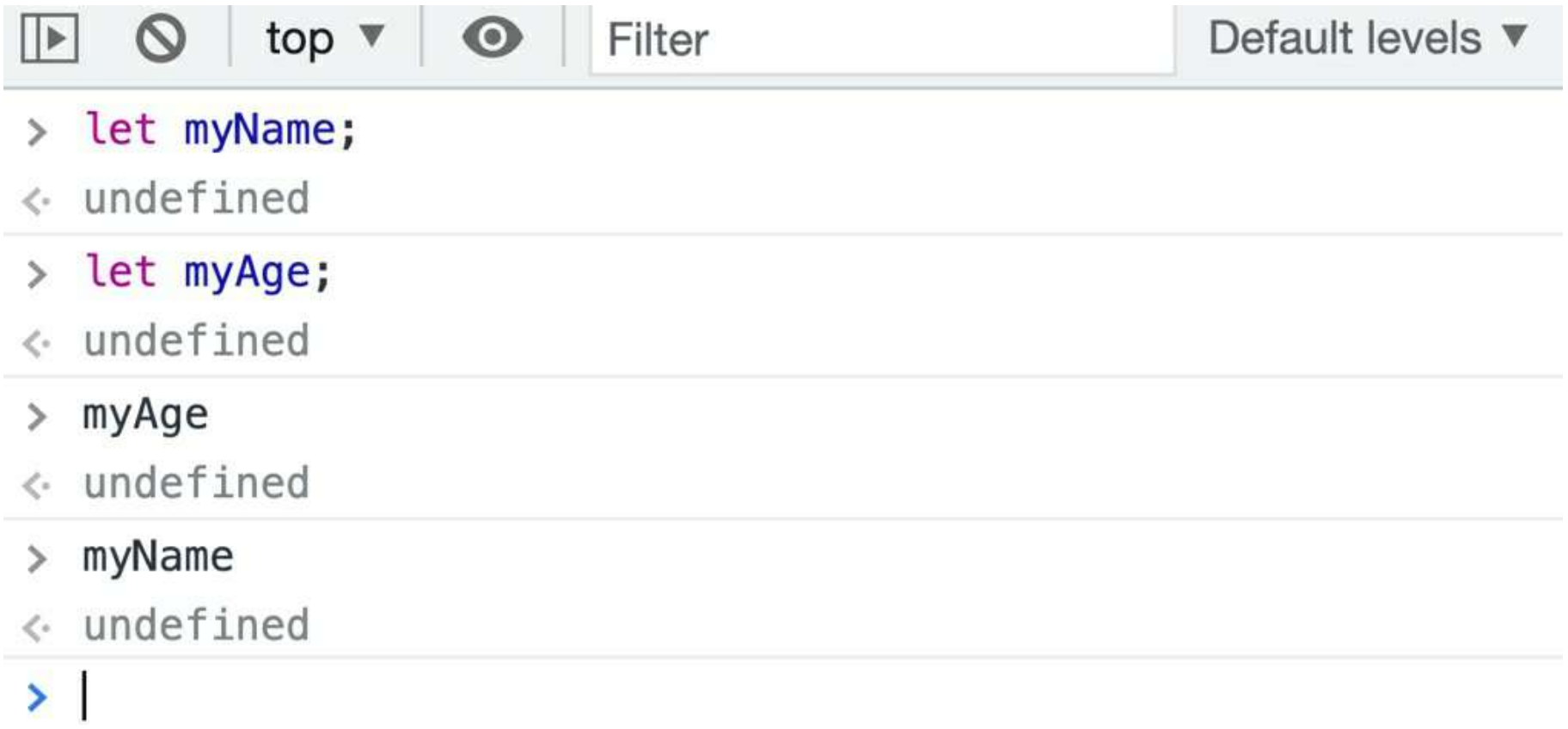
```
//camel case convention for user names
let userNames;
```

This is the preferred convention among developers.The variables also have a data type, and that data type is from its current value. In most programming languages, the data type is strongly fixed and cannot be changed after its declaration, but in JavaScript, this is not the case; we can set the variable to any type we want.

Let us create two variables, do not worry about the “let” keyword for now:

```
//Declare a variable
let myName;
let myAge;
```

Here I have two variables called **myName** and **myAge**. You can type these lines in your browser console:



Did you notice that I used the semi-colon (;) to mark the end of the instructions, try to get into this habit. I also tried to test whether these variables have values by typing **myAge** and **myName** into the console, but I got undefined back. They currently do not have any value stored in them; they are just empty containers. The **undefined** means there are no values present, but if you try to type a variable name like **myLastName** in your console you will get an error message because this variable is not even created/declared, and we are trying to call it. Therefore, please do not confuse variables created without values and variables that do not exist or are not being declared.

Initializing a variable

Once we have declared a variable, we can initialize it with a value. We can do this by typing the variable name on the left side, then this is followed by the equal sign (=), and finally, we provide the value we want to store in that variable. Check out this example:

```
//Initializing variable
myName = 'Andy';
myAge = 33;
```

Here the equal sign is used for assignment; so, we can give the variable value. If you go back to your console and type the same code, you can see they have values.

You can also declare and initialize a variable in one single line:

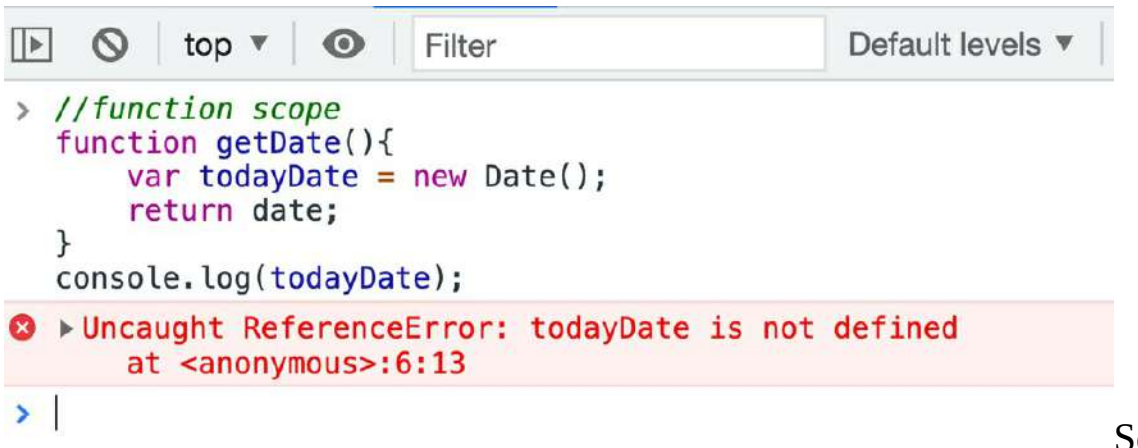
```
//Declare and Intiialize
let myLastName = 'Mcalister'
```

Var, Let, and Const

If you noticed before I declare the variable name, I have used the "**let**" keyword, but what does this keyword even mean? In the versions of JavaScript before the ES6, we used another keyword called "**var**" and you can still use this. In ES2015 or ES6, we have two new ways to create variables, and those are by using the "**let**" and "**const**" keywords. To understand why we use those keywords, I need to explain to you a little bit about the **scope**. The scope defines where variables and functions declared can be accessed. So, we cannot access something that is not in the current scope. In JavaScript, there are two scopes (code in variables.js file in **chapter4/3**):

- global-scope
- function scope

We haven't talked about functions yet but let me show you one simple example:



So the variable (**todayDate**) is declared inside the function **getDate()**. Same as variables, the functions have names as well. A function body is the code between the curly brackets **{},** and here we have declared and initialized the **todayDate** variable. But if we try to print out the result of that variable outside the function body, we will have a reference error. This is happening because the **todayDate** is only visible and accessible inside the function scope but not outside the curly brackets. That is why we are having this error. The scope outside the function is called **global** scope.

When we have a variable declared outside the function body, that variable belongs to a global scope. The global scope variables are part of the global object, and we can access that object using the **globalThis keyword**.You can type this in your browser console, and you will see that there is no issue with how we declare and initialize the

variable:

```
var a = 5; // this variable is outside function and belongs to the global scope
globalThis.a = 5; // we can achieve the same result if we use globalThis.a and set the variable
```

You should know that variables declared using the **var** keyword cannot be deleted by the **delete operator**. Another important point to understand is that you can declare the same variable multiple times using var, so re-declaring and re-initializing is not a problem with var, but this will be a problem if we use "**let**" and "**const**."

Let

A more modern way to declare variables is to use the "**let**" keyword. Unlike **var** keyword, the **let** allows declaring variables that are block-scoped. So, we can declare variables with the let keyword like this:

```
//let variable declaration
let actorName = 'Tom';
let actorLastName = 'Cruise';
```

It is recommended that you assign a value when declaring the variable name if that is possible. There are examples where we don't need to do this, and we will also cover those. Same as with **var**, the let keyword allows us to declare a variable without an initial value, and if we try to access that variable, we will have **undefined**. Variables declared with "**let"** and "**const"** keywords are block-scoped. This means they are visible and accessible within the block where they are defined. But we can declare them outside in the global scope, but unlike the **var** keywords, the **let** does not create properties of the window global object. So, what is the difference between the **var** and **let** variable then? They look pretty much the same, right? Let us check out the following code, and hopefully, this will help you understand why the **let** variables are blocked scoped. The functions are a very good example for explaining the let variable declartion:

First here is example with using the **var** keyword example**:**

```
//let vs var variable declaration
function first() {
   var x = 1;//declare and initialize variable x to 1
   {
     var x = 2;//same variable, just we assigned new value to x!
     console.log(x);// Now x is: 2
   }
   console.log(x);// We print the last value assigned to X and that is: 2
 }
```

Now here is the example using **let** keyword example:

```
function second() {
   let x = 1;//declare and initialize variable x to 1
   {
     let x = 3;//this is a different variable, this is because of the block scope, the above x is not same as this variable x because of the block scope
     console.log(x);// It will print out the value of x declared inside the curly brackets: this is 3
   }
   console.log(x);// Here we have access to the X variable that is declared on top. We don't have access to the X variable defined inside the second block, and that is why the output is 1
 }
```

So, the code above explains it all if you read the comments. The block scope is ignored with the var keyword, and changing the value will affect the other variables under the same name. With the let keyword, the variable created inside the block is treated as a different variable with a new value. The block variable is different than the variable outside the block. Therefore, we have two different results.

Another interesting feature is that we cannot re-declare the same variable because it will throw a syntax error.

Important to know about the **let** is that we cannot read or write to **let** variables unless they are being fully initialized; if no value is specified during the declaration, then the variable same as var will be given a value of undefined. Therefore, if we try accessing it will throw an **ReferenceErrror**.

Const

If you need to declare a **constant** variable, we can use **const keyword** instead of **let** or **var**. Be careful when you use this keyword **const** because you must initialize the constant during its declaration. Check out the following example:

```
> const fixedPrice = 9.99;
  const myName;
Uncaught SyntaxError: Missing initializer in const declaration
>
```

The first const variable declaration is okay,

fixedPrice is declared and initialized in one line, but the second will give us an error. If we try to create a constant without giving it value, we will have a syntax error, just as the example above. As the name suggests, constants cannot change their values, and if we try to modify the value, we will get a **TypeError**. It is good practice to write constant names using capital letters, for example, FIXED_PRICE. This way, you can easily distinguish them from the other variables in your code. The scope of the constant is the same as one of the let variables. You will get a syntax error if you use the same name more than once inside the same scope, so no re-declaration for const variables. Another important point is that JavaScript variables can have a value of any type. Check out the following example:

ItemPrice can have text and numbers as their values, and this is allowed in JavaScript.

The question you probably have is how to choose between let and const? From my experience, I can tell you that you only use constants when you know the value of the variable to remain unchanged. I have used const to declare variable values that have some sort of key as value. For example, a password or key that will not change in time. If you change the value stored in a const, then you will have another error called run-time error.

Number Literals

The primary numeric type for JavaScript is a Number. JavaScript numbers can be as large as $\pm 1.7976931348623157 \times 10^{308}$ and as small as $\pm 5 \times 10^{-324}$. All of the numbers are double-precision floating-point numbers. In short, this means you can use 2 or 2.0 without thinking about getting an error. Therefore, in the background, JavaScript treats all numbers as "**floating-point**" numbers. What are the minimum and maximum ranges for integer numbers in JavaScript? This is very easy to find out, and all you need to do is type in your browser console Number.MIN_SAFE_INTEGER and you will get: -9007199254740991, or this is equivalent to $-2^{53} + 1$. Same if you want to know the maximum safe integer, then you need to type: Number.MAX_SAFE_INTEGER and you will get: 9007199254740991, or this is equivalent to $2^{53} - 1$. If you use larger numbers than these, then you will lose precision, so be careful.

String Literals

Strings are nothing but a sequence of text enclosed by quotes. The quotes can be single or double, but remember we need to use the same quotes at the beginning and the end of each text.

Example:

```
"Hello World"
'Hello World'
`Hello World`
```

There is one problem with strings. What about if we want to use a quote inside a string. To do this, we will need to escape it with backslash. Have you noticed the last example? It is kind of different, right?

Template Literals

Since ES6, we can delimit strings with a backtick. The last example is according to the ES6 template literals. In this book, I will use single quotes or backticks as delimiters. In the past, the string's literals were written on a single line. If we wanted to use multiple strings, we should have concatenated the strings using the + operator (the plus operator will not mean we are performing an arithmetic operation on strings).

Here is an example:

```
'This is single line string ' + ', and is concatenated with plus operator';
Output:
'This is single-line string, and is concatenated with plus operator';
```

Since ES6, we can use backtick that is a template string and supports multi-line string and string interpolation. Because we are using `` backticks, we can include single or double quotes inside without any problems. Before ES6, we used the backslash \' escape, which represents an apostrophe or single quote.
Check out this example (you can find the same code in **lecture2.js**, copy and run it into the console):

```
> console.log('This is Line: 1\n' +
  'This is Line: 2\n'+
  'This is Line: 3');
  This is Line: 1                                          VM44:1
  This is Line: 2
  This is Line: 3
< undefined
>
```

Did you notice I used '\n' newline escape character so I can push the second line below? In the past, we used the newline character every time we needed the string to be moved to the next line. On the other hand, this is very simple with ES6 backticks (``). Check out the same code for strings using backticks:

```
> //Backtick ES6 feature
  console.log(`This is Line: 1
  This is Line: 2
  This is Line: 3`);
  This is Line: 1                                          VM67:2
  This is Line: 2
  This is Line: 3
< undefined
>
```

Using the backticks, we can easily stretch the string into multiple lines.
Another awesome feature is that we can use multiple single or double quotes inside the string with the backticks, and we don't need to use the escape character.
Check out the following example. The first line in the console bellow is the input, and the second line is output (You can try this in browser console):

```
> //Backtick with single and double quote
  `It's awesome`;
< "It's awesome"
>
```

Before ES6, we had to escape things to use single or double quotes inside the string. Check out the following example:

```
//Backslash Character before single-quote to ignore it
'It\'s awesome';
“It’s awesome”
//Backslash Character before double-quote to ignore it
"It\'s a \"double\" quote example";
“It’s a “double” quote example”
```

So, with template literals, we can safely use apostrophes and “quotes” in a template literal.

Arithmetic Operators

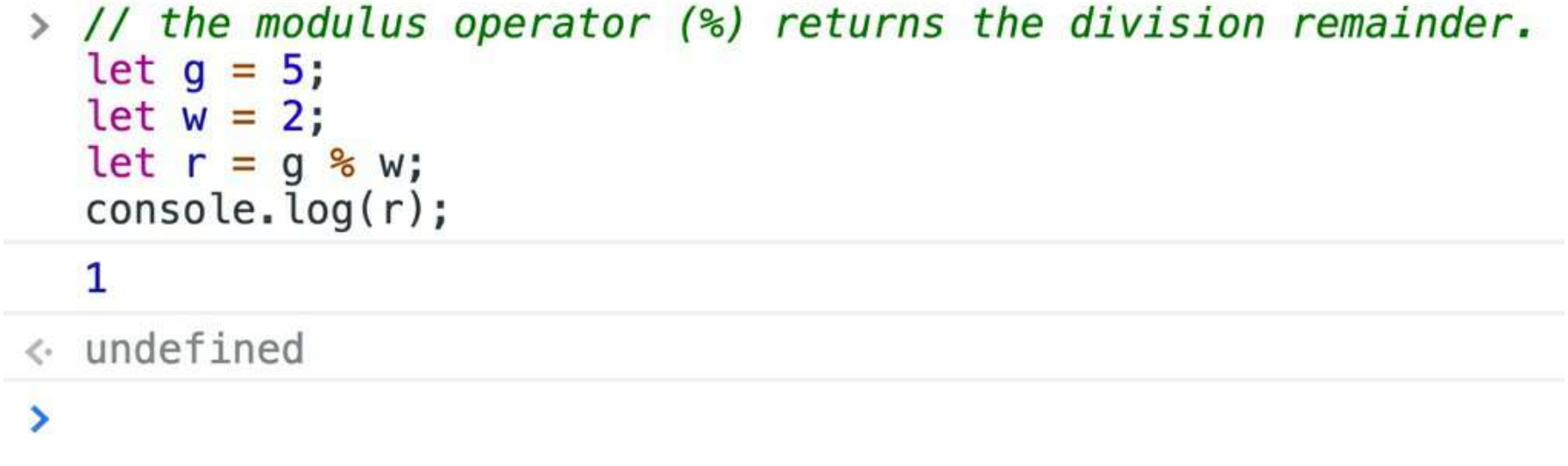
Same as in math, we can do simple math operations in JavaScript. JavaScript uses the usual operators:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- ++ increment (add 1)
- -- decrement (subtract 1)

Please note that when we use division on two integer numbers like **(5 / 2),** the result will be a floating point.
In the **lecture3.js** file you will find the same code as bellow so you can copy it and paste it straight to your console:

```
// + operator
let a = 5;
let b = 2;
let c = a + b;
console.log(c);
// - operator
let m = 5;
let n = 2;
let d = m - n;
console.log(d);
// * multiplication
let k = 5;
let j = 2;
let h = k * j;
console.log(h);
// / division
let l = 5;
let s = 2;
let q = l / s;
console.log(q);
// the modulus operator (%) returns the division remainder.
let g = 5;
let w = 2;
let r = g % w;
console.log(r);
//The increment operator (++) increments numbers.
let x = 4;
x++; // this should increase the value of x by 1 so now x will be 5
console.log(x);
//The decrement operator (--) decrements numbers.
let y = 4;
y--; // this should decrease the value of y by 1 so now x will be 3
console.log(y);
```

One interesting operator here is the modulus operator. Let us check the output of the same example:



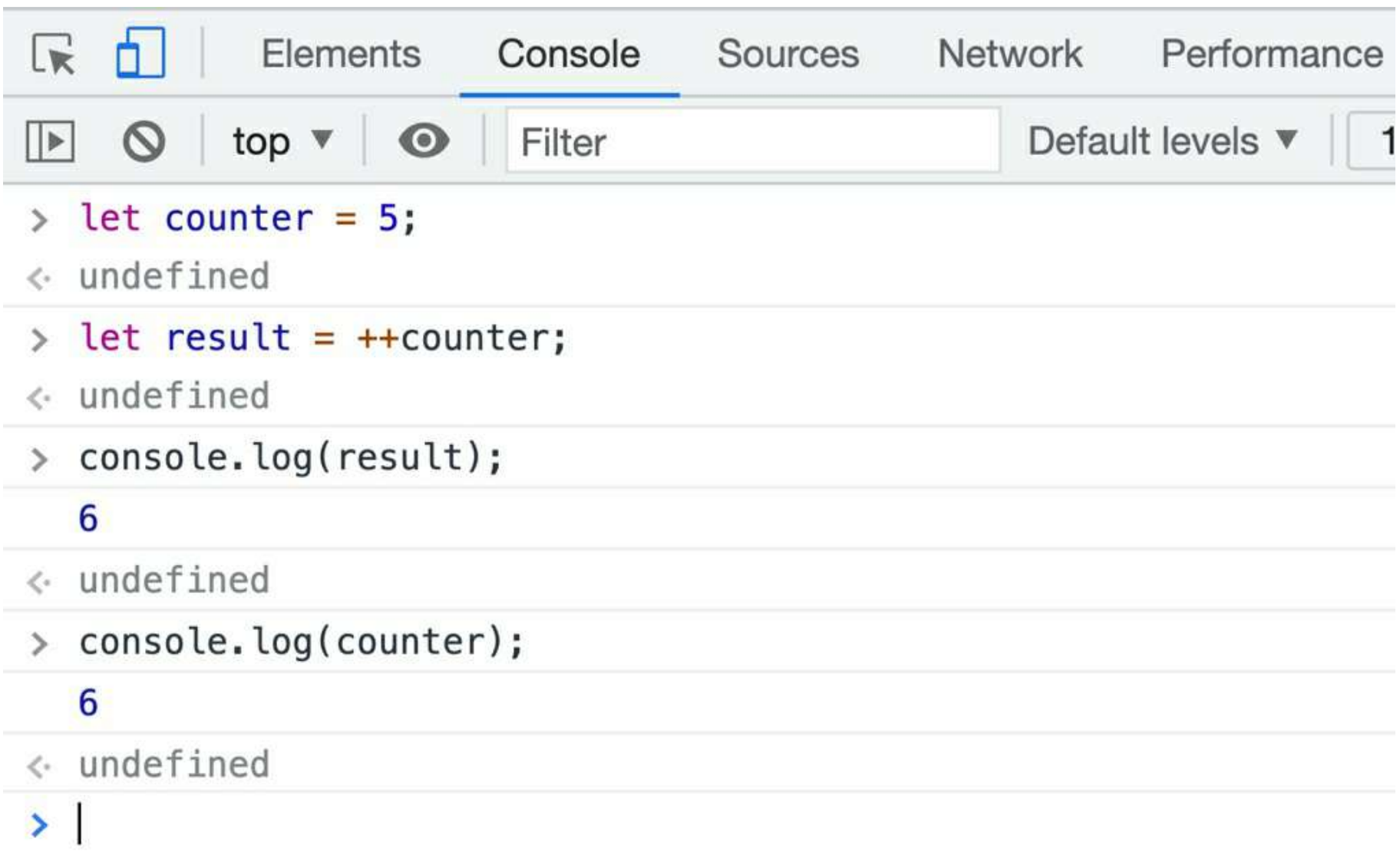
As you can see that the modules return the division remainder. So, 5 divided by 2 will give us the remainder that is left after the devision. And from here, the remainder after performing the division is 1; I hope this makes sense.
Since ES2016 or ES6, we have ** which means exponentiation.
The exponentiation operator (**) returns the result of raising the first operand to the power of the second operand. It is equivalent to Math.pow function that we have in JavaScript.

```
//** exponentiation
```

```
console.log(3 ** 4); // The output is 81
```

Finally, let us discuss the following operators ++ and --.

In the first line, we declare and initialize a variable called counter to have a value 4; The second line is where we assign the value of the counter to the result variable, but what this ++ operator is doing. If you have thought that the result will have value 5 then that is not how the postfix operator works. The first thing that is done is that the **result** variable is assigned to the value of the current **counter** variable, and that is 4; after the assignment process is done, the value of the **counter** variable is updated or incremented by 1. That is why the result variable holds the initial value of the counter variable, but when we try to print out the counter variable, the value will be the updated one.
There is another way we can use the ++ operator but not as a postfix but as a prefix operator.
Let us check the following example:



Here is opposite from the previous example, and first, it changes or increments the value to the **counter** and then sets the result of the new incremented value to the **result** variable. That is why both console logs produce the same result. You can use the same examples to test what the “--” decrement operator does.

String Concatenation +

I had mentioned before that if we wanted to use two or more strings as one, we could use concatenation. The addition operator + is used to perform concatenation on strings. This operator will join both of the strings together. That is why I had to explain the operators before showing how they work on strings. Check out the following example:

```
> let myName ="Rick";
  let myLastName = "Sekuloski";
<· undefined
> let myFullName = myName + " "+ myLastName;
<· undefined
> console.log(myFullName);
  Rick Sekuloski
```

If you can notice that I also use “ “ in the example above during the concatenation of the two strings. That means I concatenate the first name and last name with an empty string in the middle, and the result is the following: Rick Sekuloski. Otherwise, there would be no space in between the two strings. Another way is to include that space at the end of **myName** and it will look like this:

let myName = “Rick “;

As you can see I have single space left after Rick text.

Boolean Values

The Boolean type can have only one of these two values:

- True
- False

In JavaScript, we have reserved words, and true and false are reserved as Boolean values.
One most important thing here is that any JavaScript value can be converted to a Boolean value.
These values are considered false Boolean values

- **Null**
- **0**
- **-0**
- **NaN**
- **“” – empty string**
- **Undefined**

All other values we have, including objects and arrays, are converted to Boolean true values.

Null and Undefined

We have already mentioned that when we declare a variable but do not assign a value, that variable's value is **undefined** by default. The second value that indicates an absence of variable value is **null**. The **null** keyword indicates that there is “no value” for strings, numbers, and objects. Please note that **undefined** is not a reserved keyword, but **null** is. It is advisable not to name variables or constants as **undefined**. Both **null** and **undefined** are converted as **false** Boolean values, and they are both used to indicate the absence of a value. In my experience, I rarely have to assign a variable to return null or undefined, but if I had to, I would use null.

Comparison and Logical Operators

In JavaScript, we have operators called comparison and logical operators. These operators are used to test for **true** or **false** values. We use comparison operators in logical statements to determine the equality or difference between the values.

Let assume that we have a variable called “a” to hold a value of 5, so we can test it in the table below:

let a = 5;

Operator	Description	Comparing	Value
==	equal	a == 6	false
!=	not equal	a != 6	true
>	greater than	a > 4	true
<	less then	a < 4	false
>=	greater than or equal to	a >= 5	true
<=	less than or equal to	a <= 4	false

Logical or Boolean operators are used to examine the logic between the values.
Let us check the following table with logical operators.

let x = 5;
let y = 4;

Operator	Description	Example	Result
&&	**and**	(x < 6 && y >1) (x < 6 && y <1)	true false
||	**or**	(x>4 || y<5) (x>4 || y>5) (x<4 || y>5)	true true false
!	**not**	!(x == y) !(x>y)	true false

From the table above, the “&&” means ‘**and’**. Therefore, the && returns true if both operands are true and return false if only one of the operands is false. On the other hand, the “or ||” operator returns false if both of the operands are false, all other cases will return true. The last logical operator is “**not** !” and this operator reverses the value of the condition. For example, if the condition returns true, then the “not” operator will turn it to false and vice versa.

Objects

If you have learned other programming languages, then you would have knowledge about objects. Still, the objects in JavaScript are different, and they are one of the most fundamental datatypes together with arrays. In JavaScript, every object consists **name:value** pairs called properties. Objects are non-primitive data types, and if you remember, I mentioned them in the previous section. Most of the primitive parts we have already introduced earlier include: **numbers, strings, Boolean, unrefined and null values**. Everything else is considered as non-primitive or as an object.
Here is an example of a simple object

```
let user = {
   userName: 'Rick',
   age: 33
};
```

In the example above, in the variable “**user**” we store an object with two properties. This is how we create a JavaScript object with an object literal. In the object above, we have **name:value** properties. For example, we have age which represents the name and value of the age is 33. But how can we access those properties of that object? Well, we can access the object properties using the dot notation like in the example below:

```
//Objects
let user = {
   userName: 'Rick',
   age: 33
};
//We can get the object properties using the dot notation
let userName = user.userName; // also we can store these properties in a variables
let userAge = user.age;
//print out the object properties and values
console.log(userName);
console.log(userAge);
```

You can find the same example in the file called **objects.js**.
Not only can we access properties, but we can add new properties to an existing object, making the objects more

dynamic. Test the following code into your browser console:

```
user.occupation = 'Developer';
user.isWorking = true;
//print the entire object
console.log(user);
```

This is how it will look after we add new properties:

```
//adding new properties to the existing object
user.occupation = 'Developer';
user.isWorking = true;
//print the entire object
console.log(user);
Rick
33
▼{userName: "Rick", age: 33, occupation: "Developer", isWorking: true}
    age: 33
    isWorking: true
    occupation: "Developer"
    userName: "Rick"
  ▶[[Prototype]]: Object
undefined
```

Important to note is that we can also modify the existing value of some of the properties.

```
//modify the age property
user.age = 30;
```

Let us see if we changed the value of the age property:

```
▶{userName: "Rick", age: 33, occupation: "Developer", isWorking: true}
User Object after modifying the age value
▶{userName: "Rick", age: 30, occupation: "Developer", isWorking: true}
undefined
```

Another awesome feature with objects is that we can remove a property using the delete keyword:

```
delete user.occupation;
```

Create Objects using new keyword

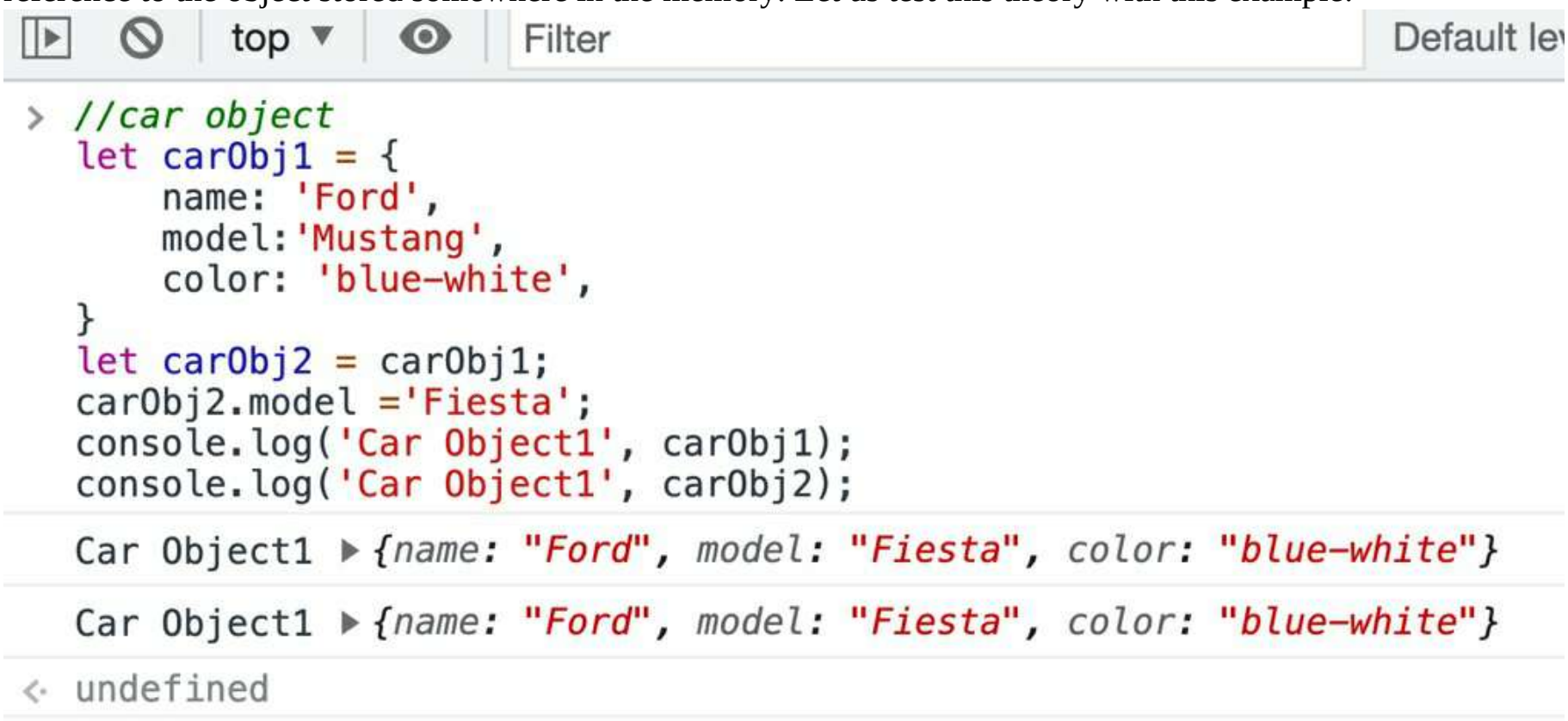
There is another way how we can create an object using the “new” keyword. This method will require more writing, but in the end, the result will be the same. Have a look at the following example:

```
top ▼   Filter   Default levels ▼   18 Issues: 18
> //Create an object using the new keyword
  const newUser = new Object();
  newUser.name = 'Rick Sekuloski';
  newUser.student = false;
  newUser.occupation = 'Web Developer & Programmer';
  newUser.students = 20000;
  console.log(newUser);
▼{name: "Rick Sekuloski", student: false, occupation: "Web Developer & Programmer", students: 20000}
    name: "Rick Sekuloski"
    occupation: "Web Developer & Programmer"
    student: false
    students: 20000
  ▶[[Prototype]]: Object
```

The first way of declaring an object as literal is more used among developers. We can also have an empty object, an object with no properties **name:value** or key:value pairs, and this is acceptable coding practice.

```
//empty obj
let studentObj = {};
```

From the examples above, you need to understand that **studentObj** or any other variable we create contains a reference to the object we are creating, not an exact copy of that object. So **studentObj** is a variable name that has a reference to the object stored somewhere in the memory. Let us test this theory with this example:



Here we have two variables pointing to the same object stored somewhere in the memory and only hold a reference. Therefore, changing the car model with the second object also affects the first object. No matter what object we are using, we are still changing the value of the object stored in the memory, and both of the objects have a reference to that object. That is why the value of the model is changed from ‘Mustang’ to ‘Fiesta’, and when we print out both object variables, we get the exact same result. If you remember, I previously stated that objects or non-primitive values are passed by reference, and primitive data types are passed by value.
If you have noticed, the object has a trailing comma at the end of the last property. This trailing comma is very useful when we add an additional property in our object, this way we are always sure that we can add new property without causing a syntax error.

Creating objects using Object.create()

Object.create() method creates a new object using its first argument as the prototype of the newly created object. This might be confusing but check out the following example:

```
//create object called person with one property set to false so we can change it later
const person = {
  isHuman: false,
  myDetais: function() {//myDetails is a function that will print all of the details
    console.log(`My full name is ${this.name} ${this.lastName}. Am I human? ${this.isHuman}`);
  }
};
//create a rick object using the Object.create but as an argument we pass the person object
const rick = Object.create(person);

rick.name = 'Rick'; // "name" is a property set on "rick", but not on "person" object
rick.lastName = 'Sekuloski';  // "lastName" is a property set on "rick", but not on "person" object

rick.isHuman = true; // Well isHuman is one inherited property from person object and can be overwritten by giving it new value

rick.myDetais();
// expected output: "My name is Rick Sekuloski. Am I human? true"
```

As you can see in the example, I have a person object that will be used as a first argument when we create a new object called **rick**.

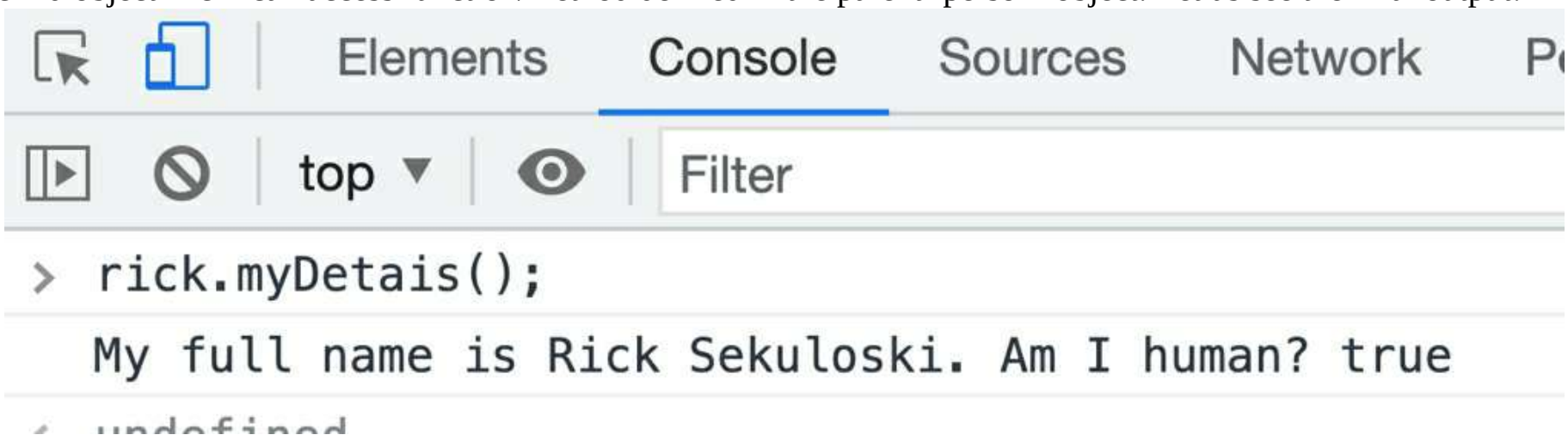
```
const rick = Object.create(person);
```

I have used the new method here called **Object.create** and passed the person object as a first argument. The person Object has two properties. The first property isHuman is set to Boolean value false, and also there is myDetails property that has a value of whatever the function returns. We have not covered functions yet, but that function will

print out the details from the object. What is strange here is that the function will return name, last name, and isHuman, but only the isHuman property is declared inside the person object's body. I know this is looking strange now, but it will be clear soon. So let us check out these two highlighted lines from the example:

```
rick.name = 'Rick';
rick.lastName = 'Sekuloski'
rick.isHuman = true;
```

As you can see, I set a name and last name to the “rick” object, and those properties belong to the ‘**rick’** object. Inheritance works like this, the parent can pass properties down to its children, and the children can add their additional unique properties, and that is what we have done here, we add two new properties to the ‘rick’ object. On the other hand, the father ‘person’ object does not have access to first and last name. To simplify, the child object can have access to all the properties and functions we have in the parent object plus its own, and it can also modify the parent properties like the isHuman property. And the rick is setting the isHuman to be true. Let us now test if the child object ‘rick’ can access function/method defined in the parent ‘person’ object. Let us see the final output:



Great! As you can see everything works as it should. You can also create a new object that does not inherit anything, which is bad because we still want to use some of the basic methods that **Object.prototype** offers. We can achieve this:

```
//creating new object that does not inherit from prototype
const obj2 = Object.create(null);
```

We can create an empty object same as the one above with the help of object literals:

```
//empty objects using object literals
const obj3 = {};
//empty object using Object.create, its same as the one above
const ob4 = Object.create(Object.prototype);
```

I think, for now, it is more than enough discussion about Objects. The inheritance will be clearer when we talk about classes in the next sections.

Primitives passed by value

Now is the time to explain how primitive passed by value are different from objects passed by reference. We have already seen an example with objects passed by reference and now will look into the primitive data types.
So, when we have a primitive type assigned to a variable, then that variable contains the actual primitive value, not the reference to the memory but the exact location to the memory. This is totally opposite from the objects because objects variables only hold a reference to where the object is stored in the memory but not the actual value.
Let me explain this with this awesome example (you can find the code in **passed-by-value.js** file):

```
let a = 5;
let b = 'name';
let c = null;
c = a;
a = 10;
console.log('Value of a:',a);// Output: 10
console.log('Value of c:',c);// Output: 5
```

If you run this code in your browser console, you will notice that at the end, the value of ‘a’ is different than the value of variable ‘c’. When we assign ‘a’ to the variable ‘c’, we copied whatever we have stored in ‘a’ to the variable ‘c’.

```
c = a;
```

After that, we changed the value of ‘a’ to be 10, and this will not affect the variable ‘c’ because they do not have a

reference relationship to each other like we had with the objects.

Arrays

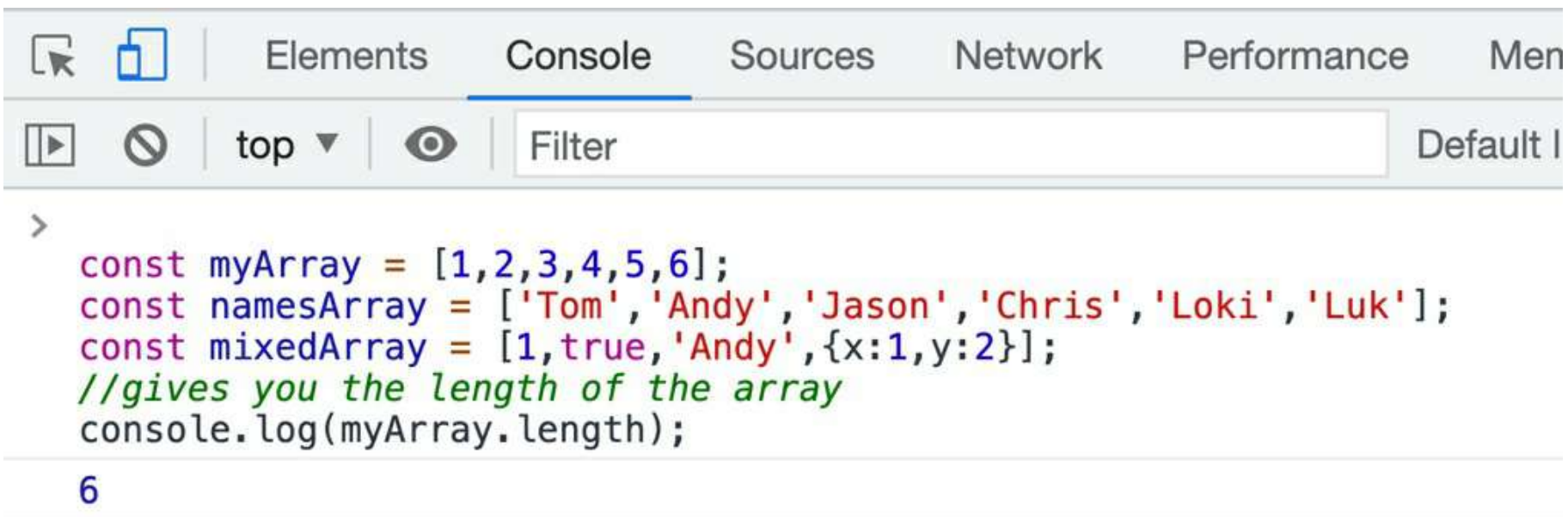
Arrays in JavaScript are the second most important datatype. Arrays are simply an object. We use Arrays so we can store multiple values inside a single variable or constant. An array is an object where the property names are string numbers like '0', '1', '2', '3', '4', and so on. Important to remember that property names are strings. They are not numbers. Someone will say why we are using strings when we can use numbers? The reason is simple; the property names inside an object cannot be from type numbers. Therefore, we must use strings. So, an array is an ordered collection of values where each value is called an element or item.

Array Literals

Let us declare an array using array literals:

```
const myArray = [1,2,3,4,5,6];
const namesArray = ['Tom','Andy','Jason','Chris','Loki','Luk'];
const mixedArray = [1,true,'Andy',{x:1,y:2}];
```

In the example above, we have declared and initialized three arrays. The elements of each array must be enclosed by square brackets. The square brackets are important, same as the curly brackets for the objects. JavaScript arrays are zero based. The first element starts from position zero, and why we start from zero, you will find out soon. As you can see, the elements of an array can belong to any data type; this is more obvious in the last **mixedArray** example. Array elements can be numbers, strings, Boolean values, objects, functions, or other arrays. JavaScript arrays are dynamic, just like the objects, so that we can add and delete elements. JavaScript arrays have access to **length** property, and with this property, we can get the actual length of the array as a number. Check out the following example where we print the array length:



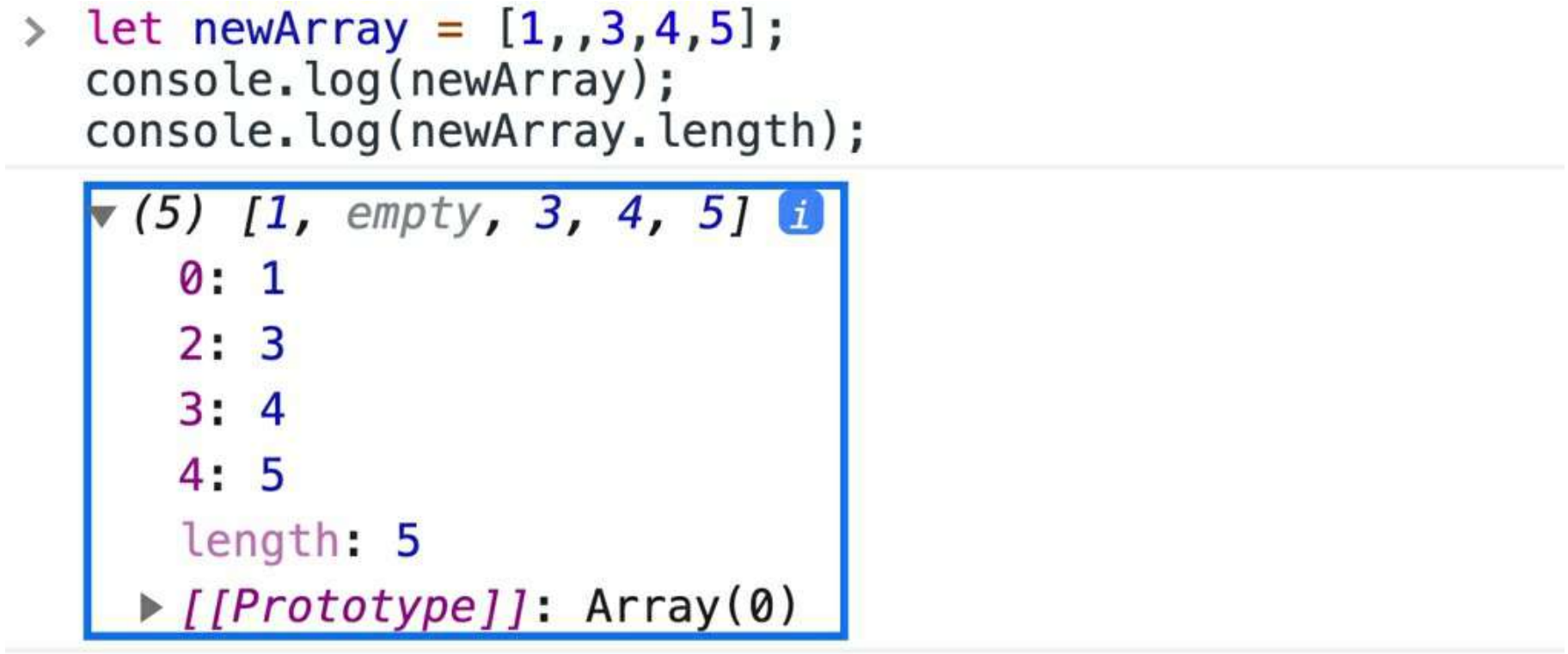
When we create a new array, as we did in the example above, we need to know that there are properties inherited from **Array.prototype**. This will allow us to use some array methods, or methods we can use out of the box without thinking too much about them.
Same as objects, we can create an empty array just like this:

```
//creating empty array using array literals
let emptyArray = [];
```

We can also use trailing commas in the arrays without any problems. Check out this example:

```
//trailing comma
let trailingComma = [1,3,5,];
console.log(trailingComma . length);//Output 3
```

But if we have multiple commas with no value assigned in between, something interesting will happen. Check out this example:



From the example, we can see that all elements have their own index starting from position zero. In position zero, we have a value of 1. In position 1, we don't have a value, and so on. When we print out the array, we have an "**empty**" slot because there is no value, but interesting it gives us a length of 5, including the empty slot, this is important so be careful using commas and want to find out the length.

Create Arrays using new Array ()

Another way to create an array is to use the **new** keyword. This will use the Array constructor, and we can call or invoke this constructor in three different ways. We have not discussed constructors yet, but please just focus on learning more about the arrays. You will learn the constructor, classes, instances, and all of these terminologies later because they are more advanced concepts.
But in the following example, everything will be clear about creating arrays using the new keyword.

```
//empty array
let array1 = new Array();//constructor without any arguments passed
console.log(array1.length);//0
//single argument constructor
let array2 = new Array('Tom');//constructor with one arguments called Tom
console.log(array2.length);//1
//multiple argument constructor
let array3= new Array('Tom','Jerry');//constructor with multiple arguments passed
console.log(array3.length);//2
```

Spread Operator

Since **ES2015** or **ES6,** we can use the '**spread operator'** to create a new array or even to copy array to another array. The spread operator syntax is three dots … and I know it looks funny.
Check out this example before I confuse you even more:

```
const firstArray = [1,2,3,4,5];
console.log('The elements from the first array '+ firstArray);
console.log('The length of the first is: '+ firstArray.length);
//Use the spread operator to create new array
const secondArray = [...firstArray];
console.log('The elements of the second array'+ secondArray);
console.log('The length of second array is: '+ secondArray.length);
```

In this example, I have done two things. First, we spread all of the elements from the first array into the second, and therefore we created a new array. The second important point that you need to learn here is that we made an actual copy from the first array because the array items from the first one we spread into the second one. If we add more elements into the second array, then this will not affect the first array.
Check out the output of the examples below:

```
> const firstArray = [1,2,3,4,5];
  console.log('The elements from the first array '+ firstArray);
  console.log('The length of the first is: '+ firstArray.length);

  //Use the spread operator to create new array

  const secondArray = [...firstArray];
  console.log('The elements of the second array '+ secondArray);
  console.log('The length of second array is: '+ secondArray.length);
  The elements from the first array 1,2,3,4,5
  The length of the first is: 5
  The elements of the second array 1,2,3,4,5
  The length of second array is: 5
```

Access Array Elements

To access elements from the array, you need to use the indexes. Remember I have told you we will cover array indexes, and now is the perfect time to explain why arrays are zero based. So, array elements always start from zero position/index, right? Let us create an array using array literals with five car names inside:

```
> //Accessing the elements of the array
  let carNames = ['Toyota','Cadillac','Porsche','Bentley','Ford'];
  console.log(carNames);
  let Toyota = carNames[0];//Output:Toyota
  let Cadillac = carNames[1];//Output:Cadillac
  let Porsche = carNames[2];//Output:Porsche
  let Bentley = carNames[3];//Output:Bentley
  let Ford = carNames[4];//Output:Ford
  ▼(5) ["Toyota", "Cadillac", "Porsche", "Bentley", "Ford"] i
      0: "Toyota"
      1: "Cadillac"
      2: "Porsche"
      3: "Bentley"
      4: "Ford"
      length: 5
    ▶ [[Prototype]]: Array(0)
<· undefined
> console.log(Toyota)
  Toyota
```

To access the last name of the array, you will need to use the name of the array. Then in the square brackets, you need to provide the index. So **carNames[4]** will print out the ‘Ford’. What do you think will happen when you try to access an element not present in the array? This can happen because you can make a simple mistake with the indexes. I know that the length of this array is 5, and most of the students will try to access the last fifth element, but this element is not there because there is no index 5; only the length is 5.

```
//access non-existent element
console.log(carNames[5]);//undefined
```

The result is the same as with objects we are trying to access a value not there, so it will return as **undefined**. And this led us to the point where we can answer how we can add new elements to the array. Simple we can use the last position and add new element there

```
//add new element on the last position
```

```
carNames[5]= 'Mercedes';
```

If you check the array length now, it should be increased by one new element, and ‘Mercedes’ car name should be included.

Conditional Statements or Branches

As the name suggests, the conditional statements need to have a condition to work. Based on the condition, we can either skip or execute a set of statements. These statements are also known as *branches*. In JavaScript, we have **if/else** and **switch** branch statements.
Example of if syntax (code inside the **chapter4/4** folder):

if (expression)
statement

The condition must be placed inside the parentheses. Please note that the condition will produce either true or false values.
Have a look into the following example:

```
let number1 = 1;
let number5 = 5;

if(number1 <= number5){
   console.log('The condition is true');
   let result = number1 + number5;
   console.log(result);
}
```

We start the if statements by writing the **if keyword**, then we have a condition inside the brackets where we compare if number1 is smaller or equal to number5. In our example, this condition evaluates to true because indeed 1 is smaller than 5. So, if the condition evaluates to true, we are executing the statements inside the **if block**. Otherwise, we skip all of those statements. Because the condition is true, then all of the statements below the **if statement** will be executed. In between the curly brackets, we can have one or multiple statements. If we have only one statement, we can even exclude the curly brackets, but if we have multiple statements, we need to use them. Multiple statements are referred to as a **statement block**.

If-else statement

What if you want to execute one set of statements if the given condition is true and another set if the condition is false. Well, if we want to do this, then there is another statement called **if/else.**

The syntax:

if (condition)
{
statement block 1
}
else
{
statement block 2
}

Example:

```
number1 = 10;
//else condition statements will be executed
if(number1 <= number5){
   console.log('The condition is true');
   let result = number1 + number5;
   console.log(result);
}
else{
   console.log('The condition is false');
   let result = number5 - number1;
   console.log(result);
}
```

If we look at the syntax above and ‘if’ the condition is evaluated to **true**, then statement/statements inside the block

1 will be executed, and statement block 2 will only be evaluated when the if conditions is **false**. In our example, the condition is false, so the statements inside the **if block** will never be executed, but the statements in the **else** block will be executed because that is the rule of the if/else.

Else if statement

But what will happen if we want to execute one of many pieces of code? Well, one way we can do this is to use the **else if** statement. This statement is very easy to understand if you did understand the previous two concepts because the only difference is that we have more than two choices.
Check out the following code:

```
//else-if
age = 18;
if(age <= 14){
   console.log('You are in the children group');
}
else if( age >= 15  && age <= 24){
   console.log('You are in youth group');
}
else if( age >= 25  && age <= 64){
   console.log('You are in Adults group');
}
else{
   console.log('You are in Senior group');
}
```

Based on the age, we can see in which age group we belong. For example, if we run the same code in our browser console, we will get that we are in the youth group because the **else if** condition is evaluated to true, the rest of them are being evaluated too false. In the end, you can see that we have a normal **else clause** that will be triggered if each of those previous conditions were false. If you want to test this out, I suggest running the same code for different ages and seeing which condition will be evaluated to true and what statements will be printed out.

Here is one example where we have single statements, and the {} braces are not necessary.

```
let isNumber = true;
if(isNumber)
   console.log('Inside the if clause!');
else
   console.log('Inside the else clause.')
```

Conditional (ternary) operator

Here we will discuss a topic that is a bit more advanced, and it might take some time until you get used to this operator. The **ternary** operator is like a shorthand version of the **if statement**. This conditional (ternary) operator is the only JavaScript operator that takes three operands.
Before you get confused, here is the syntax for this operator:

condition ? expressionIfTrue : expressionIfFalse

The first operand is the condition, an expression whose value is used as condition, and the second operand after the '?' is the **expression if true;** but what does this mean? If the condition evaluates to true or truthy value, then this expression will be evaluated next. The last operand is the **expression if false** after the **':'**. This expression is executed if the condition is false.
Let us check this example:

```
//Conditional (ternary) operator
let age = 26;
let whichBevarage = (age >= 21) ? "Beer" : "Juice";
console.log(whichBevarage); // "Beer"
```

In this case, I only save whatever the operator returns in a variable called **whichBeverage** and then print out the result with the console.log method. Obviously, because the age is 26, the condition (age > = 21) is true, the first expression is evaluated. That is why we are having 'Beer' stored as a value in the **whichBeverage** variable.

Switch Statement

So far, we have covered the **if**, **if-else**, and **else-if** statements to enable the program to execute a different block of statements based on the given condition or in multiple conditions as we have in **else-if** statement. However,

JavaScript has an additional statement that is very popular, and it is called **switch statement**. The switch statement is more complex to explain, but with examples, it becomes very easy. The switch statement is the same as the **switch** statements in many programming languages like **Java**, **C**, C++, etc. You can skip this part if you are familiar with **switch**. Basically, we use switch statement when we have a large number of cases we need to check. The syntax of this statement is:

```
let x = 6;
switch(x){
   case 1: // if x === 1
      //Execute the statements for case #1
      break; // Stop
   case 2: // if x === 2
      //Execute the statements for case #2
      break; // Stop
   case 3: // if x === 3
      //Execute the statements for case #3
      break; // Stop
   case 4: // if x === 4
      //Execute the statements for case #4
      break; // Stop
   case 5: // if x === 5
      //Execute the statements for case #5
      break; // Stop
   case 6: // if x === 6
      //Execute the statements for case #6
      break; // Stop
   case 7: // if x === 7
      //Execute the statements for case #7
      break; // Stop
   case 8: // if x === 8
      //Execute the statements for case #8
      break; // Stop
   case 9: // if x === 9
      //Execute the statements for case #9
      break; // Stop
   case 10: // if x === 10
      //Execute the statements for case #10
      break; // Stop
   default:
      //Execute the statements
      break;
}
```

From the example above, we can see that switch is comparing an expression with many possible values. We can achieve the same result using multiple **if/else** statements, but that will be too much code repetition. We start the switch statement using the keyword **switch**. The **switch keyword** is followed by an expression and block of code inside curly braces. Inside we have a sequence of **case clauses**. Each clause inside the **switch** statement has the **keyword case** followed by an expression and colon (**:**). But how does the switch statement work? For example, we are trying to find which number is stored in the variable '**x'**, and in our case, that is the number 6. Then the switch statement equals the value of the expression, and it tries to find the case label that matches the value from the expression. But what is **case label**? In our example, we are trying to find a particular number, right? That is why the case label contains numbers; 'case 1:', 'case 2:' and so on. But what will happen if the **x** variable has a value of 11; we don't have a case label to match the expression, right? When the **switch** statement cannot find a case with the matching value of the expression, it will execute the **default** statement. And that is why we need to have **default** at the end of the switch statement; consider this to be equal to the **else clause** in the **if/else** statement. If you noticed, we also have a **break** at the end of each clause. This will cause the JavaScript interpreter to break out of the switch statement and continue executing the statements outside the switch statement. So, the **break** statement at the end of each clause will ensure that the switch statement will be entered when the case label is matched and will continue with the code that follows. If you forget to put a break, then the switch will not stop its execution and will fall through to the next alternative. Another important thing to know is that the switch statement in the background uses the '===' operator, which will ensure the exact value be found in the case label.
Let us change the previous example with real code and run it into the browser.

```
let x = 6;
switch(x){
   case 1: // if x === 1
      //Execute the statements for case #1
      console.log('We have found the number ' + x);
      break; // Stop


   case 2: // if x === 2
```

```
   //Execute the statements for case #2
    console.log('We have found the number ' + x);
    break; // Stop

case 3: // if x === 3
    //Execute the statements for case #3
    console.log('We have found the number ' + x);
    break; // Stop

case 4: // if x === 4
    //Execute the statements for case #4
    console.log('We have found the number ' + x);
    break; // Stop

case 5: // if x === 5
     //Execute the statements for case #5
    console.log(x);
    console.log('We have found the number ' + x);
    break; // Stop

case 6: // if x === 6
    //Execute the statements for case #6
    console.log('We have found the number ' + x);
    break; // Stop

  case 7: // if x === 7
     //Execute the statements for case #7
     console.log('We have found the number ' + x);
     break; // Stop

  case 8: // if x === 8
     //Execute the statements for case #8
     console.log('We have found the number ' + x);
     break; // Stop

  case 9: // if x === 9
     //Execute the statements for case #9
     console.log('We have found the number ' + x);
     break; // Stop

  case 10: // if x === 10
     //Execute the statements for case #10
     console.log('We have found the number ' + x);
     break; // Stop

  default:
     //Execute the statements
     console.log('The number is not in the range between 1 - 10: and the number is: ' + x)
     break;
}
console.log(`This is printed after the break statement terminated the switch statement`);
//Output
We have found the number 6
This is printed after the break statement terminated the switch statement
```

From the example above, we can see that case 6 is being executed. After the break statement is reached, the switch statement is terminated, and the code outside the switch statement is being executed. We do not have a large code in this example, but simple console.log will prove that the switch statement is working as intended. As an exercise, try to initialize the x to a value or number not in the range between 1 to 10 and observe the output.

Assignment Operator

We have already used the assignment operator so far. For example, we use this operator when assigning a value to a variable. The assignment operator is this (=). We can check out the following example:
let result = 4 + 6;

Here we have an expression on the right side of the assignment operator, and this expression is evaluated. The value then is assigned to the variable on the left called ‘result’. This is a very simple and trivial example, but I also want to point out that we can combine this operator with other operators.

Here is a table of possible operator combinations

Operator	Description
+=	add and assign
-=	subtract and assign
*=	multiply and assign
/=	divide and assign

While Loop

While a condition is fulfilled (or true), the **while loop** repeatedly executes one or multiple statements. This is the **while statement** or **while loop syntax:**

```
while(condition)
{
    code block to be executed
}
```

Example:

```
//While loop
let i = 0;

while (i < 10) {
  i++;
  console.log(i);
 }
```

In this example, the code in the curly brackets will repeatedly loop until the condition is true. You should know that if the condition is false, the curly brackets code will never be executed. This is the output:

```
> //While loop
  let i = 0;

  while (i < 10) {
     i++;
     console.log(i);
    }
1
2
3
4
5
6
7
8
9
10
```

Do While Loop

The while loop will never execute its code if the condition is false, but there are situations that we need that code to be executed at least once, and for this, we can use the **do/while loop**.

Do/while syntax:

```
do{//statement
   //code
}while(condition)
```

Here is an example where I have declared another variable called ‘i’, but the value will be 11 in this case.
let i = 11;

```
let i = 11;
do{
   console.log('The value of i:', i);
}while(i < 10);
```

Output: The value of i: 11
As you can see, the loop executes the statement (it can be a block of statements as well) and then tests the condition. In our case, the condition is false because 11 < 10 will evaluate to false, but we still have the value of **‘i’** printed out at least once, which was the whole idea. This loop is still less common than the **while loop** we discussed earlier, but it is always nice to know it.

For Loop

One of the most used **loops** and favorited among developers is the **for loop**. The for loop have 3 separate statements. For example, we have: a counter or a variable that is updated after every iteration, condition test, and incrementing variable.
So general use of the **for loop** is to iterate through a set of elements, and we mostly use this loop to iterate through an array.

Here is the syntax of the for loop:

```
For (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

- Statement 1: sets a variable before the loop starts (let i = 1). In most programming languages, this variable is **‘i’** and always starts from zero. Remember the arrays are zero based index
- Statement 2: defines the condition for the loop to run (‘i’ must be less than 10). The loop needs to be true.
- Statement 3: increases the value of **i** (i++) each time the code block in the body of the loop has been executed.

Let us do the same example as the **while loop** where we to logged the numbers from 1 to 10.

```
for(let i = 1; i < 10; i++){
    console.log(i);
}
```

The output is the numbers printed from 1 to 10.
Let me give you another example where we loop through an array of numbers. This is where we use the **for** loops mostly:

```
let numbersArray = [1,2,3,4,5,6,7,8,9,10];
  for(let i = 0; i < numbersArray.length; i++){
   console.log(numbersArray[i]);
}
```

Why are we accessing the numbersArray[i]? To get an item or element from an array, you need to use the index of that element. We know that the array elements always start from position 0. Therefore we have the counter ‘i’ to start from zero and increment this ‘**i**’ in each iteration. Therefore we can use the variable ‘**i**’ to access all of the elements in that array. In the condition/test section, we compare if the ‘**i’** is smaller than the length of the entire array.
To summarise, the for loop has three separate sections (initialize, test and increment), and all of them are separated by a semicolon at the end.
The initialize expression is evaluated once before the loop starts, and we usually do a variable **declaration** and **initialization**. The section in the middle is the **condition test,** and it’s done before each iteration. If the test is evaluated to be true, the statements inside the body of the for loop are executed. Finally, the increment section is evaluated. Please note that we can use any of these two operators ++ and --.

for/of loop with objects

We have covered objects, and therefore I can show you how you can access both the key/value pairs with the **for/of** loop. The objects are not the same as the arrays, and we cannot traverse the objects the same as the arrays because it

will throw an error. To traverse as an object, we can use the following methods:

- Object.keys();
- Object.entries();

Object.keys() – for/of

Object.keys() is a method that returns an array with the property names.
Here is an example where I have one test object created:

```
let testObject = {
   a: 1,
   b: 2,
   c: 3,
   d: 4,
}
//Object.keys
console.log(Object.keys(testObject));
// Output: (4) ["a", "b", "c", "d"]
```

In the above example, we can see that output is an array with the property names from the provided object. So now we have an array of property names, and we can use the for/of loop over the array as we did in the previous section. Check out this example:

```
let propertyNames = "";
for(let names of Object.keys(testObject)) {
   propertyNames += names;
}
console.log(propertyNames); // => "abcd"
```

Object.entries() – for/of

With the **Object.keys();** method, we are getting the property names out of an object, but what if we need both the key/value pairs from that object. If we want something like this, we need to use **Object.entries().** This method will give us access to both key and value pairs.

Here is an example:

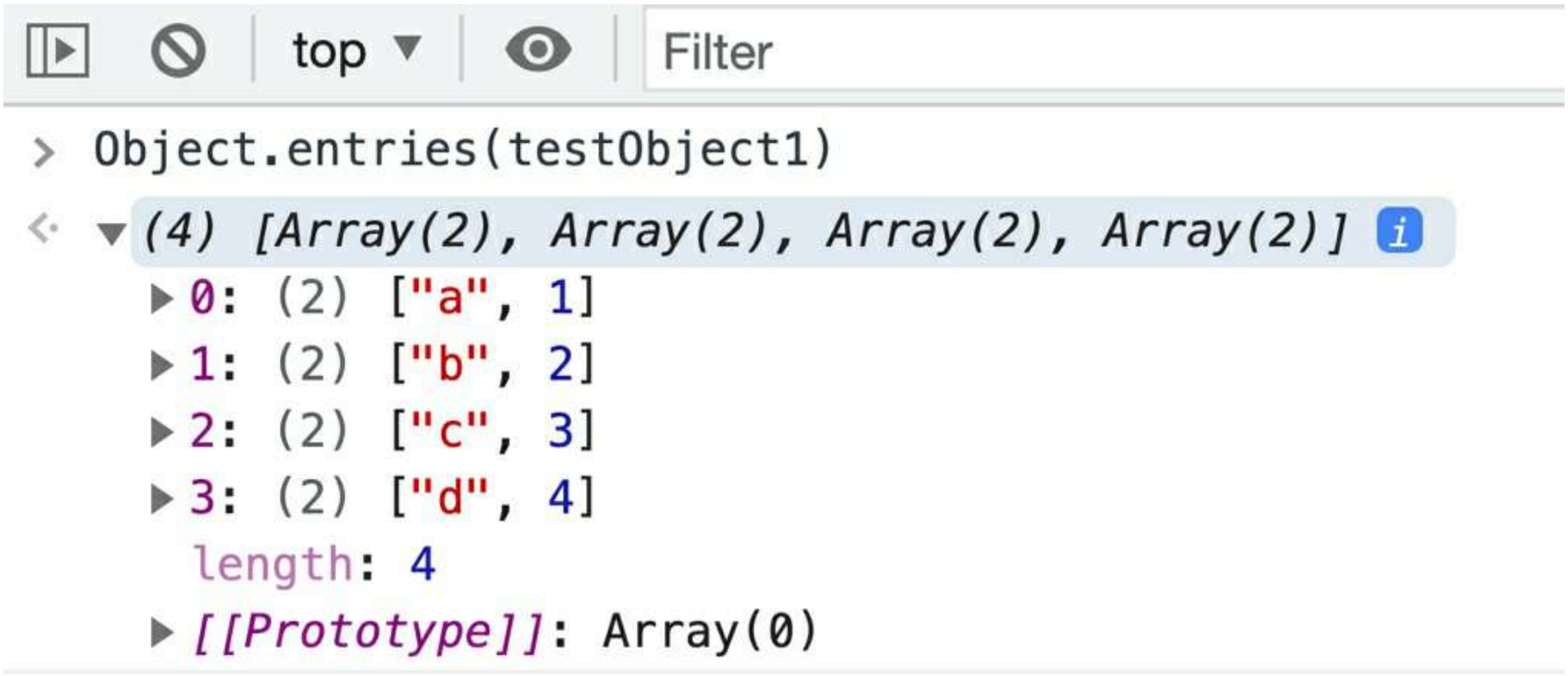
```
let testObject1 = {
   a: 1,
   b: 2,
   c: 3,
   d: 4,
}
for(let [k,v] of Object.entries(testObject1)) {
  console.log('Key: '+k);
  console.log('Value: '+v);
}
```

Output:
Key a
Value 1

Key b
Value 2

Key c
Value 3

Key d
Value 4
If we print out in console what Object.entries() method does, this will be the output:



As you can see, the **object. entries** return an array of arrays. In Our case, we have 4 inner arrays where each array contains two elements, the key and the value pair element from the object.

for/in loop

The **for/in** statement loops through the properties of an object. This loop is not a new feature of JavaScript because it has been part of JavaScript since its beginnings.
Syntax:

for (*variable* in *object*) {

statement

}

Let us go over the syntax really quick:

- variable - a different property name is assigned to the variable on each iteration.
- Object - object whose properties are iterated over.

Interesting is that the block of code inside the curly brackets will be executed once for each property. The JavaScript interpreter executes the body of the loop for each property of the object we provide.

Let us go through the following example:

```
let objPerson = { firstName: 'Denzel', lastName:'Washington',age: 67 }
for (const key in objPerson){
   console.log(`Key: ${key}, value: ${objPerson[key]}`)
}
```

Output:

```
Key: firstName, value: Denzel
Key: lastName, value: Washington
Key: age, value: 67
```

Here we have a **for/in** loop that loops through the property names of an object we provide. The variable name can be anything and does not have to be '**key'** like in our example; so, the variable is evaluated before each iteration and assigned the name of the property to it.

Functions

Another fundamentally important feature in JavaScript is functions. In this section, you will master functions, and I will help you learn how to write functions, call and execute them. The code for this section will be in **chapter4/6** folder**.**
Functions are subprograms that are written once, and they may be invoked or executed many times.
Here is one function that I want you to test out in your browser console:

document.write(‘Welcome’);

This is a function which purpose is to simply generate output. Most of the functions provide and return a value. Did you see that the text ‘Welcome’ is written on your browser? The document.write() method will write a string of text to the current document. This is just an example of a very simple function, but we will go deeper than this. Another function that I want you to test out is this one:

let getNum = prompt (‘Enter a number’);

Tip: if you copy and paste text from this document, please make sure you replace the quotes around the text because it might give you an error.

This function will prompt the user to input text. Even if you type 5 and hit enter/return, the prompt will close, but the value will be stored in the variable **getNum** and will be stored as a string, not as a number. Therefore the value of getNum will string ‘5’**.**
These functions are just to warm you up before we start writing our own functions. Let us practice how we can write functions in the next section.

Declaring Functions

In JavaScript, there are different ways to declare a function, and we will go through them. The most common way to declare the function is by using the function keyword followed by parentheses, where we write the names of the parameters. Finally, the function body is where we can have single or multiple statements. The body of the function is between the curly brackets, but if we have a single line of code, as you already know, we can omit them.
Function syntax:

```
function name (parameter1,…parameter2) {
  //function body
}
```

Simple example:

```
function total (x, y) {
  return (x + y);
}
```

Let me go over the syntax one more time. We start with the keyword function then we provide a name to that function. The name is known as function identifier. Inside the brackets, we provide parameters. In our case, we have two, but it can be many as long as they are separated by a comma. The types of the parameters are not important as well. Finally, in the body, we are returning their sum using the return statement. The statements inside are the body of the function, and they are executed only when the function is called or invoked.
The return statement in the function body yields the value that the function will return, and, in our case, it will be the sum of the values of the two x and y parameters. This is the statement I mentioned when we covered the break and continue statements.
We can also have the function without a parameter list, but we still need to provide the parenthesis. The number of parameters, as you can see, is not important to declare a function.

```
//function without a parameter list
function simpleFn(){
   return 'No Parameters';
}
```

Invoke Functions

Now you know how to declare functions, but it is time to call these functions.
To call a function, you need to pass the arguments and store the value inside the variable.

```
//call the total function
let storeResult = total(4,9);
console.log(storeResult);
```

Let me explain this: a function provides a list of arguments. Each of the arguments we pass to the function is matched to a parameter we specified in the function declaration or function definition. Therefore, the **x** will be matched with the number 4, and **y** will have number 9. Remember the parameters in the function do not have a data type. Still, the arguments we are passing during the function invocation/call must be from the correct type. For example, we need to do a simple mathematical calculation. We need to pass numbers, not strings. But what will happen if we pass strings? Will we get an error? Let's find out:

```
//lets pass two string arguments instead of numbers
let storeResult1 = total('4','9');
console.log(storeResult1);
```

The output will be the string '49' because the return statement in the function body thinks we are trying to concatenate two strings, '4' and '9'.
Let us discuss the **return** statement and what this statement does in our function. The **return** statement returns immediately, and it will **terminate** the function. The return statement will return the value to the caller. Imagine if we have more statements below the **return** statement. As soon as the function returns, the function is terminated, so the remainder of the code below the **return** keyword will never be evaluated.
Let us consider this scenario: We can have a function without a parameter list and no return statement:

```
//function without return
function noReturnValue(){
   console.log('No return statement');
}
let storage = noReturnValue();
console.log(storage);
```

In this case, if we print the storage variable, we will have **undefined** value. You need to understand that when we declare a function name, that name becomes a variable in which we store whatever the function returns.

Function Expression

In JavaScript, there is another syntax for creating a function, and this is called **function expression**.

Syntax for **function expression (anonymous)**:

let variableName = function (x, y) { statements... return (x+y) };

Here is an example:

```
let greeting = function(name,lastName){
   return `Nice to see you again ${name} ${lastName}`;
}
```

From the example and the syntax I provided, we can see that the **function expression** allows us to create an anonymous function that does not have function name. This is the main difference if we compare it to the first function declaration examples. The function expression is stored in a variable and can be invoked/called using that variable name.
We call it anonymous function because there is no name, so the variable name can access this function. JavaScript also allows to use names for function expression. Check out the following syntax:

Syntax for function expression (named):
let variableName = function functionName(x, y)
{
statements... return (x+y);
};

It is recommended that you use the **const** instead of let with function expression so you don't overwrite the function by accident because we cannot modify things that are declared as **const**. Remember this? Another important rule about functions expressions is that we cannot use them before they are created, meaning the function expression in JavaScript is not hoisted. What is hoisting? Well, hoisting in JavaScript means that variables and function

declarations are moved to the top of their scope before code execution. When using a normal **function declaration**, the function object is created even if the code that specifies its declaration has not been processed or evaluated. That is why we can call these normal functions before the code where we write the definition statement is evaluated. The story with function expression is a little bit different, they do not exist, and we cannot call them before the expression that declares them is evaluated. The function expression is assigned to a variable, so we can refer to that function. In short, the function defined as the expression cannot be called before it is defined, which is very logical.

Invoke Function Expression

Let us call the function that we have created using its variable name '**greeting'**:

```
let greeting = function(name,lastName){
   return `Nice to see you again ${name} ${lastName}`;
}
```

```
//invoke function expression
let theMessage = greeting('Luke', "Perry");
console.log(theMessage);
```

The output of this function is: '**Nice to see you again Luke Perry**! Now you know how simple it is to call a function expression in JavaScript.

Arrow Function

At the beginning of this chapter, I said there are different ways how we can declare a function, one of them was to use the **function** keyword, and the second one is to use the **arrow operator** '=>'. This is the ES6 feature and means a shorthand syntax for function declaration. Using an arrow function comes with advantages and some disadvantages as well. The arrow function expression is an alternative to the traditional function expression.
Here is an example that will help you understand more about arrow functions:

```
// Traditional Function
function addFun (a){
  return a + 100;
}
// Arrow Function
let addFun = a => a + 100;
```

The arrow functions are much cleaner and shorter, but let us go through its syntax. We provide the parameter variables on the left side of the arrow operator, and on the right side of the operator, we return the function value. If there is a single parameter 'a' like we have in the example above, we do not need to enclose that parameter in parentheses as we did in the traditional function declaration ' (a) '. We know that a traditional function can be without parameter/parameters right, and we can do the same thing with the arrow function, but we need to use the empty set of parentheses like in this example:

```
//arrow function with no parameters
let noParams= () => 'No Parameter function';
```

How can we invoke the arrow functions then? Well, we can simply use the name of that function.
Let first invoke the **addFun:**

```
let addFun = a => a + 100;
//call the arrow function
let result = addFun(10);
console.log(result);
```

We stored the return of the function into a variable called '**result'**, and we console log the output.

The no parameter's function invocation is like this:

```
let storeResult = noParams();
console.log(storeResult)
```

Let me explain that when we have a single line of code in the function body, we do not specify the return keyword. Check out the example without return keyword:

```
a + 100;
```

In this case, we do not need to write **return** keyword. But the same function will be a valid arrow function even if we include the return keyword like this:

```
let addFun = a => {
```

```
    return a + 100;
};
```

If the body requires multiple lines of code, then we need to use the curly brackets plus the return statement because the arrow function does not know what or when we want the function to return. Check out the following example:

```
// Traditional Function
function additionFn (x, y){
    let result = x + y;
    return 'This is the result :'+ result;
  }
let store1 = additionFn(3,4);
console.log(store1);

// Arrow Function
let additionFn1 = (x, y) => {
    let result = x + y;
    return 'This is the result :'+ result;
}
let store2 = additionFn1(3,4);
console.log(store2);
```

The output of both functions will be the same, and that is 7.

One important thing here is that we need to use the '=>' operator in the same line as the parameters. Otherwise, it will throw you an error:

```
//This is OK
let multiFn = (x, y) => {
    let result = x * y;
    return 'This is the result:'+ result;
}
//This will give you an error
let multiFn = (x, y)
=> {
    let result = x * y;
    return 'This is the result:'+ result;
}
```

Arrow function on arrays

So far to iterate over arrays we used some of the following statements like: **for**, **for/in** and **for/of** loops, but since we covered the arrow functions, we can use the **forEach** method to iterate through both elements and the indexes of the array. These are the different arrow function syntaxes that we can have for the forEach loop:

```
Arrow function syntax
// forEach((element) => { ... } )
// forEach((element, index) => { ... } )
// forEach((element, index, array) => { ... } )
```

Check out the following example:

```
//arrow functions and arrays
const carBrands = ["Saab", "Volvo", "BMW"];
console.log('Printing only the elements of the array');
carBrands.forEach(element => {
    console.log(element);
});
```

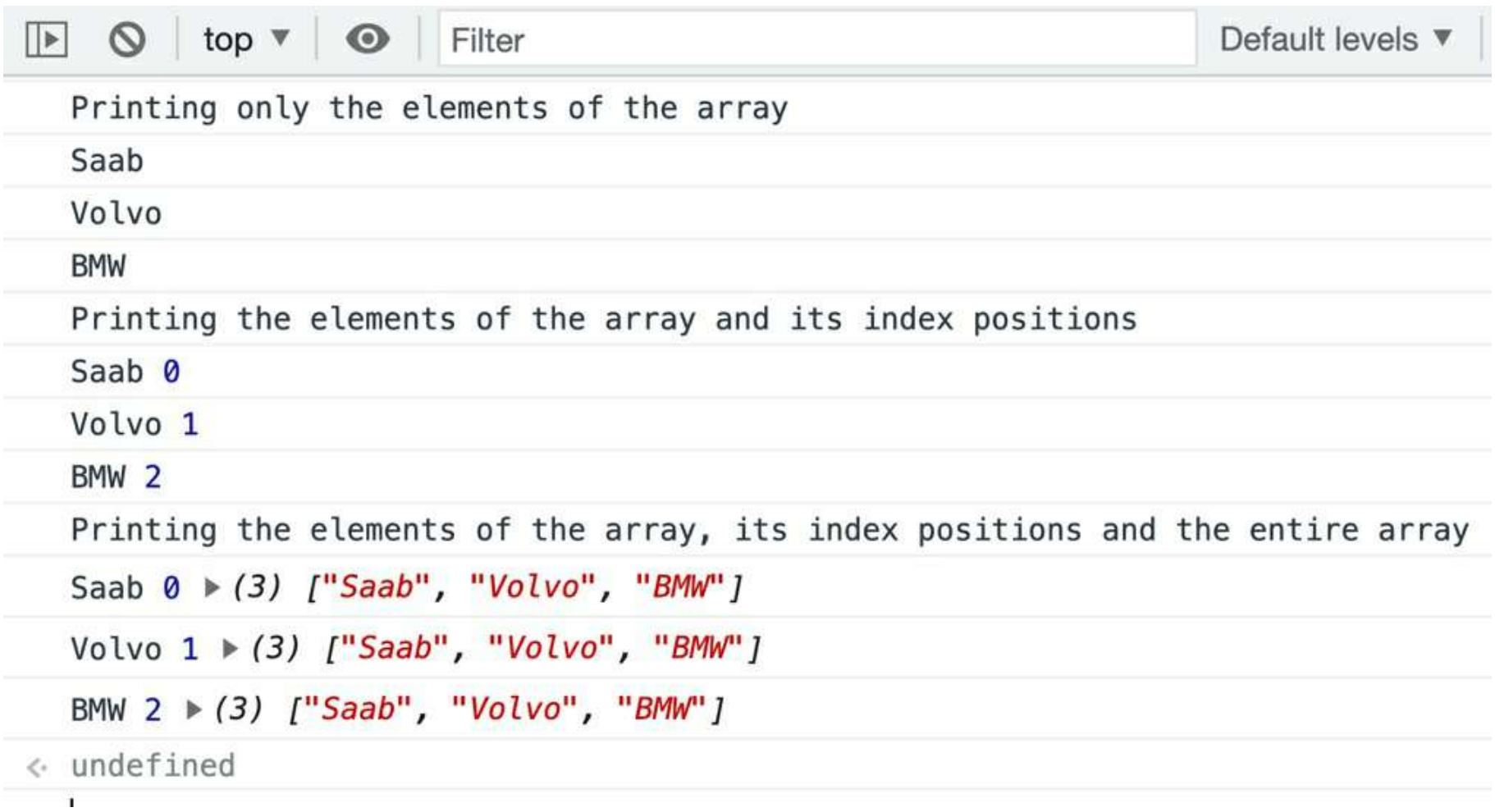
This is same forEach function but with extra keyword to get the indexes of the array items:

```
console.log('Printing the elements of the array and its index positions');
carBrands.forEach((element,index) => {
    console.log(element , index);
});
```

This the last forEach syntax where we can return the entire array if we need something like this:

```
console.log('Printing the elements of the array, its index positions and the entire array');
carBrands.forEach((element,index,array) => {
    console.log(element , index, array);
});
```

Output:



Therefore **forEach** method can call this function either for:

- Elements
- Elements and indexes
- Elements, Indexes and the entire array

As you can see, we can ignore indexes and the entire array if we don't need them in the final result.

Passing Arguments to functions

So far, we have learned a lot about functions. We know how to define/declare a function with specific parameters and invoke the same function in our program. Now let us imagine we have a function that will take two numbers and compare which one of them is bigger, and return that one:

```
let a = 5;
let b = 7;
function findMax(x,z){
   if(x > z){
      return x;
   }
   else{
      return z;
   }
}
let biggerOne = findMax(a,b);
console.log(biggerOne);
```

The output will be, of course, the 7. What do you think will happen if we don't supply the two arguments to the function? Will this cause an error? In JavaScript, something strange will happen. Therefore we can add more arguments to the function and still have an output.
We can declare another variable called **c** with a value of **8.**

```
//let biggerOne = findMax(a,b);
let c = 8;
let biggerOne = findMax(a,b,c);
console.log(biggerOne);
```

What do you think the output will be? I hope you guess it right; the result will be 7 because the last argument we supplied additionally to the function will be ignored.
Another scenario is when we supply fewer arguments into the function.
I will just edit the last line of the previous example:

```
let biggerOne = findMax(a);//only one argument and we need 2
```

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

What will be the output now, do you know? Well, the output will be undefined because this is equivalent to this:

```
let biggerOne = findMax(a,undefined);
```

Then in the function body, this comparison will happen:

```
if(5 > undefined)
```

This will give us a result that is not of any use, but we can fix this using default function arguments.

Default function parameters

It is very simple to provide default arguments in the function declaration.
This is the syntax:

function fnName(param1 = defaultValue1, ..., paramN = defaultValueN) { ... }

Check out the following example:

```
function multiplyFn(a, b = 1) {
  return a * b;
}
let result1 = multiplyFn(5,2);
console.log(result1);// output: 10

let result2 = multiplyFn(5);
console.log(result2);// output: 5
```

The default function parameters allow us to initialize the parameters with default values if **no** or **undefined** value is passed. Therefore, in the second function call (`multiplyFn(5)`), we pass only one argument only, and that is why the function will look like this:

```
function multiplyFn(5, b = 1)
```

This is how we provide the default function parameter, and please note we can also do this for the first parameter.

Closures

Closures in JavaScript are not that straightforward; therefore, we have many people struggling to get their heads around; to be honest, I was one of them when I started learning this language. I will explain as simply as I can what closures are using very simple code examples.
Check out the following example:

```
// JavaScript closure
function outerFun() {
   let x = 10;
   function innerFun() {
       let y= 20;
       console.log(x+y);
    }
  return innerFun;
}
```

From this example, you can see I have two functions, and they are called nested functions. The first function is called an **outerFun**, where we have declared a variable '**x**' and initialized it to 10. This function returns the result of the inner function.
Inside the outer function, we have an inner function called **innerFun**, which has another variable '**y**' declared and initialized to 20. But here, we return a result from both variables **x + y**; note **x** is defined outside the **innerFun** body.
Please do not forget that the variable **x** scope is limited to the outerFun, and the scope of variable **y** is also limited to the innerFun function.
Consider this piece of code now:

```
let firstOuter = outerFun(); //first invocation
let secondOuter = outerFun(); //second invocation
console.log(firstOuter);
```

We have the following output from the console.log:

```
f innerFun() {
   let y= 20;
   console.log(x+y);
}
```

What is happening here? Let us go step by step for you to understand this part. When the JavaScript compiler reaches this line:

```
let firstOuter = outerFun(); //first invocation
```

Then we are going into the body of the outerFun, the first line is to declare and initialize the variable x to value 10, but the next line is where there is a function declaration. The function **innerFun** is declared, and that is it. We do not have anything else for execution at this stage. The last line is to return **innerFun**, and the compiler is looking for a variable called **innerFun**, but there is no variable under this name in the body of the function. That is why we have an output that is the entire function body:

```
f innerFun() {
   let y= 20;
   console.log(x+y);
}
```

The last step is when the function finishes its execution, and all of the variables declared within the **outerFun** scope no longer exist. Now you know that whatever we declare inside this function lives during the function execution. In short, the variable '**x**' exists only in the **outerFun,** and when the function finishes its execution, the variable no longer exists.
What will happen if we execute the same function again and console it log the result:

```
console.log(secondOuter);
```

Again, the same process will occur, the function is invoked, the variable **x** is created and initialized, and when the function returns, the lifespan of all of the variables declared in **outerFun** is finished.
Let us look at this code again:

```
let firstOuter = outerFun(); //first invocation
let secondOuter = outerFun(); //second invocation
```

The variables **firstOuter** and **secondOuter** both contain the same function because the outerFun during its execution returned a function; therefore, the two variables are functions. We can confirm that the variables are functions using the **typeof** operator**.**

```
> console.log(typeof(firstOuter)); //firstOuter is of type function
  console.log(typeof(secondOuter)); //secondOuter is of type function
  function
  function
```

This means we have two functions stored in the two variables. In JavaScript we can always execute function if we add brackets () after the function name such as **firstOuter**() and **secondOuter**().

Let us run some tests and invoke the same function couple of times, just like the following code:

```
firstOuter();//we invoke the function using ()
firstOuter();//we again invoke the function
```

You can run this code in your console, and you will see that we have two identical outputs with a value returned of 30 because 10 + 20 is 30.

Okay, let us go step by step how we got this result:

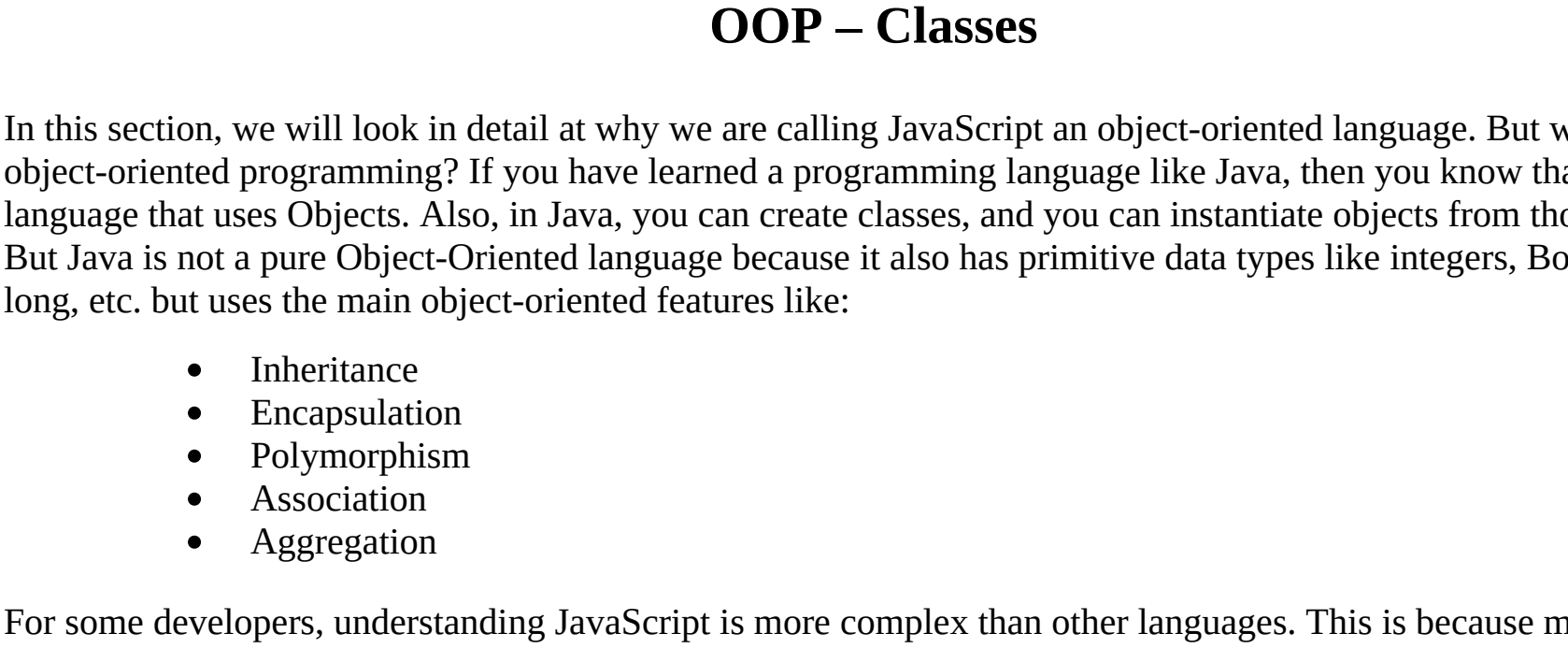
- In the outer function, the variable x is created, and its value is set to 10.
- In the inner function, the variable **y** is created and initialized to 20. In the next line, the JavaScript compiler is trying to execute **x + y**. But how is this possible? The variable **y** was just created, and JavaScript knows about its existence, but variable **x** no longer exists since it is a part of the outer function. Basically, the **x** would exist while the outer function is in execution. Because the function execution is finished, the variables declared within the scope of the outer function cease to exist, and hence variable **x** no longer exists.

There is where **closures** come into action. This whole concept is strange, but because of the closers, the inner

function will continue to have access to the variables declared in the enclosing function even after the enclosing function is finished. In other words, the inner function will preserve all the scope chain of the outer function created at the time of the function execution. Therefore the inner function can use all of those enclosing function variables. The inner function has access to the variable x declared and initialized in the outer function because the **closures** preserve the scope chain during the outer function execution.
If we want to see the Closure element in our console window, then instead of console log the result, we can use another method called **console.dir();**

```
console.dir(firstOuter);
```

This will return the inner function, but we can expand it and see what is happening inside. We need to use the browser console if you want to see the **Closure** element. In the console, you will see the [[Scopes]] and click on it to expand and see the Closure element. Notice that the value of x:10 is preserved in the Closure even after the **outerFun()** function completes its execution.



Now you know more about closures, and I hope it makes sense. The closures concept is not easy to grasp at first, but it will be clear that there is no other way after a while.

So, the inner function has access to **three scope chains**:

- Access to the variables declared in its own scope
- Access to the outer function variables because of the closures
- Access to any variables that are declared in the global scope. This means if we have variables declared outside of the two functions, the inner function would still use them.

OOP – Classes

In this section, we will look in detail at why we are calling JavaScript an object-oriented language. But what is object-oriented programming? If you have learned a programming language like Java, then you know that Java is a language that uses Objects. Also, in Java, you can create classes, and you can instantiate objects from those classes. But Java is not a pure Object-Oriented language because it also has primitive data types like integers, Booleans, long, etc. but uses the main object-oriented features like:

- Inheritance
- Encapsulation
- Polymorphism
- Association
- Aggregation

For some developers, understanding JavaScript is more complex than other languages. This is because most developers always complain that JavaScript is not a true OOP language or is not well structured and working with it is a pure mess, especially when working with objects and classes. As you already know JavaScript objects, are a bit different compared to objects that you have probably seen in another object-oriented programming like C++ or Java. It is confusing how you can have methods or classes or even inheritance in JavaScript. Classes syntax and definition are similar to Java but what happens under the hood is quite different.
I know you are eager to learn all of these new concepts so let’s dive into OOP. The code for OOP will be located in **chapter4/7**, there will be more files there compared to the theory we will cover in this section.

Classes

In 2015 JavaScript introduced a new construct known as **class**. Please know that the classes have been part of JavaScript since the early beginnings, but they finally have their official syntax in ES6. If you read the MDN website or other online resources, the classes are nothing but syntactic **sugar** over the prototypes. The JavaScript classes are nothing more than a bundle of constructor functions and prototype methods. A class in JavaScript is a set of objects that inherit the properties from the prototype object. Ok, let's rewrite the previous example but using a common class syntax:

```
//Classes in JavaScript
class Person {
   constructor(firstName, lastName, age, position, country){
      this.firstName = firstName;
      this.lastName = lastName;
      this.age = age;
      this.position = position;
      this.country = country;
   }
   printDetails() {
      console.log(`Hi! My details are:
      Name: ${this.firstName},
      Name: ${this.lastName},
      Name: ${this.age},
      Name: ${this.position},
      Name: ${this.country}
      `);
  }
}
```

In the class, we have a **constructor** (), and this keyword declares the body of the Person constructor function. The JavaScript classes are nothing but syntactic sugar for a constructor function and a prototype object. An important thing to mention is how we can instantiate an object from a class. Here is an example:

```
const samuel = new Person('Samuel','L Jackson',59, 'Actor','America');
samuel.printDetails();
```

Remember, since **ES6**; it's recommended to use classes syntax instead of the JavaScript prototype-based classes. Using classes is very easy. We need to remember only to use the class keyword followed by the name of the class, which should start with a capital letter. The class has its body, and that is marked with the curly brackets. Inside the class, we have a keyword constructor, which is the same since we are defining the constructor function. In other examples, you will find that we do not need to define constructor at all. This means we do not have variables that need to be initialized and given values, but because we omit constructor keyword that does not mean there is will not be a constructor for this class. An empty constructor function will be implicitly created for us in the background.

Inheritance

One of the hottest topics in any OOP programming language is inheritance. To talk about inheritance, we need to have a parent class, and we want the children of that class to get all of the functionality and properties that the parent class has to offer, but what is more important is that the children of that class can add its own new functionalities and properties. This is like in real life, all of us have parents, and since your birth, you have inherited something from your parents right, maybe that is the color of your eyes, hair, skin and etc. But you are a unique individual, meaning you have many more individual characteristics that define you as a unique person because we are not simply clones or copies from our parents. Now let us see how we are going to achieve inheritance in JavaScript through examples.
Let us say we have a class called **Animal**. Inside the class constructor, we will set and initialize two properties of the animal (name and speed). We will also have two methods that are very 'generic,' and every animal can do. Here is the Animal class example:

```
//JavaScript Inheritance
class Animal {
 constructor(name) {
    this.speed = 0;
    this.name = name;
  }
 startRunning(speed) {
    this.speed = speed;
    return `The ${this.name} runs with speed of   ${this.speed}km per hour.`;
  }
```

```
  stopRunning() {
     this.speed = 0;
     console.log(`The ${this.name} stopped running and now sits still.`);
    }
  }
let animal = new Animal("Animal");
```

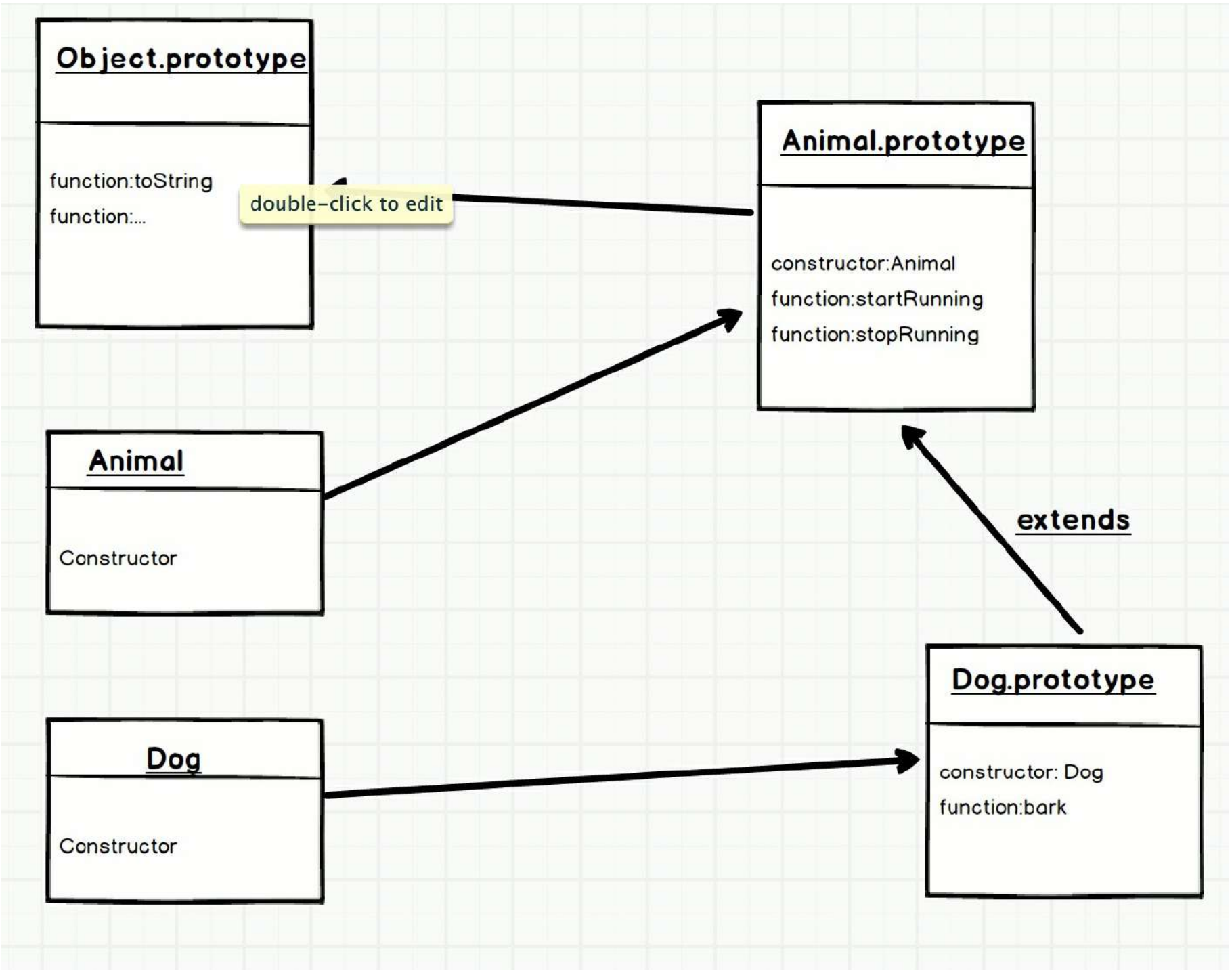
After we declare this basic Animal class, we can create another class, for example, **Dog**. Dogs are animals as well, and they should be based on Animal class, meaning they can access the generic animal methods and properties, but also the dogs can do something that other animals can't. To inherit properties from its parent, this is the syntax we should use:

```
class Child extends Parent
```

Here the keyword '**extends**' is very important, and now we can create the Dog class that will inherit or extend from the Animal class:

```
class Dog extends Animal {
   bark() {
     return `The ${this.name} barks!`;
   }
}
let dog = new Dog("Dog");
console.log(dog.startRunning(10)); // The Dog runs with speed of 10km per hour.
console.log(dog.bark()); //The Dog barks!
```

From this example, the object we instantiate from the Dog class has access both to Dog methods and Animal methods, such as startRunning and stopRunning.
Under the hood, the extends keyword works using the prototype, so it sets the Dog.prototype. [[Prototype]] to Animal.prototype. So, if the object requests a method that cannot be found in Dog.prototype, it will start looking into the from Animal.prototype.



From the figure above, you should understand how JavaScript inheritance is working in the background. In some literature, the Animal class is called a superclass, and the class that inherits from the superclass is called a subclass.

Setters and Getters

Setters and getters are methods that are very easy to understand. When we are using the getter methods, remember that they will always return us something. The getters methods do not take parameters, and they are declared with the keyword **'get'**.
Check out the following example:

```
class Person {
   constructor(firstName, lastName) {
      this.firstName = firstName
      this.lastName = lastName;
}
get fullName() { return `${this.firstName}, ${this.lastName}`;}
}
const actor = new Person('Tom', 'Hiddleston');
const actorName = actor.fullName;
console.log(actorName);//  Tom, Hiddleston
```

So the getter methods like the one in our example, **fullName**(), does not take parameters, and the same method can be called as we are accessing a property value:

```
actor.fullName;
```

On the other hand, the setters are used to set or assign a value to a property. We also need to provide that new value of the property as a parameter. Check out the following example:

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
  get fullName() {
    return `${this.firstName} & ${this.lastName}`;
}
  set fullName(name) {
    const parts = name.split(',');
    this.firstName = parts[0];
    this.lastName = parts[1];
    }
}
const haroldKumar = new Person();
haroldKumar.fullName = 'Harold,Kumar';
console.log(haroldKumar);
haroldKumar.fullName;
```

As you can see, the setters and getters methods are very easy, one is to set a value, and the other is to **get**/**return** value. You need to remember that **getters**/**setters** exist to define accessor properties as methods instead of simple data properties. The setter method is defined by the keyword **set** followed by a function named after the property that will take the property's new value as a parameter. The **getter** is defined by the keyword get and followed by a function named after the property. The getters take no arguments, but they return the value of the property.
We can also use setters and getters with objects as well.
Check out this example:

```
//setters and getters for objects
const testObj = {
  name: 'Andy',
  get userName() {
    return this.name;
  },
  set userName(newValue) {
    this.name = newValue;
  }
};
console.log(testObj.userName);
testObj.userName = 'Rick';
console.log(testObj.userName);
```

This example illustrates a getter and a setter used to define an accessor property called **username**.

Static properties and methods

In JavaScript, we can also assign a method to the class function, not to its prototype. These methods are called static

methods. To declare a static method, we need to use the keyword **static**. The static methods are used to implement functions that belong to the class but not to the object. Static methods were introduced in ES6.
Check out the following example:

```
class User{
   constructor(name){
      this.name = name;
   }
   static staticMethod(){
     console.log(this.name);
   }
 }
let newUser = new User('Simon');
```

As you can see, to declare a method to be static, we need to prefix it with the keyword **static** inside the class declaration.
As we mentioned that these methods operate in the class but not in the instances of the class, and we can call them like this:

```
//call a static method using the class name
 User.staticMethod();
```

From the example above, we can call the static method if we use the class name.
But using the static keyword, we can also define a static property of that class. Same as the static methods, the static properties will not be called from the instances of the class. So, in our case, we have **newUser** instance of the **User** class, and this instance cannot access the static property or method. Only the class **User** can. The MDN website mentions the static methods are used as utility functions like the ones we use to create or clone objects, and the static properties are used for data that we do not need to manipulate and replicate.
Check out the following example:

```
class User{
   constructor(name){
      this.name = name;
   }
   static staticPropery = 'Some Info';
   static staticMethod(){
     console.log(this.name);
   }
 }
let newUser = new User('Simon');
 //call a static method using the class name
User.staticMethod();
console.log(User.staticPropery);
```

As you can see, we can call the static property again using the class name.

Overriding methods

For this section I will use the same class example I had in the inheritance part. Here is the entire code:

```
//JavaScript Inheritance
class Animal {
   constructor(name) {
     this.speed = 0;
     this.name = name;
   }
   startRunning(speed) {
     this.speed = speed;
     return `The ${this.name} runs with speed of ${this.speed}km per hour.`;
   }
   stopRunning() {
     this.speed = 0;
     console.log(`The ${this.name} stopped running and now sits still.`);
   }
 }
  let animal = new Animal("Animal");
//let us create Dog class that will inherit from the Animal class
class Dog extends Animal {
   bark() {
     return `The ${this.name} barks!`;
   }
 }
let dog = new Dog("Dog");
```

```
console.log(dog.startRunning(10)); // The Dog runs with speed of 10km per hour.
console.log(dog.bark()); //The Dog barks!
```

Now we can move forward and override a method. Remember from the inheritance section that when the dog instances could not find the dog class method, they would start looking into the superclass or the parent class for that particular method. In our case, this was the **Animal** class.
So, if we add another method in the Dog class with the same name as it in the animal class, the process is called method overriding. Why? The name is self-explanatory, meaning we have a method with the same name in parent and child class. For example, let us create another method in the **Dog** class called **stopRunning** (), and this method will have the same name as the method we already have in the **Animal** class. Therefore, we performed a method overriding.
Check out the code:

```
class Dog extends Animal {
   stopRunning (){//this method will be used by the instances of the Dog class
    }
   bark() {
     return `The ${this.name} barks !`;
    }
 }
```

When we have new method declared in the **Dog** class, all of the instances that belong to the dog class will be able to use/access this method instead of looking into the parent class **Animal**. We do not want to replace the entire method in the parent class when we do method overriding. Still, we rather want the method in the child class **Dog** to extend the functionality inherited from **Animal** class. Now someone can ask this question "will we need to write again the same code we already have written in the parent method?". This question is quite legitimate, and it is important to remember that the answer to this question is to use the keyword 'super'. In JavaScript, we have another new keyword called 'super' to do all of that manual work for us, meaning we do not have to write the same code again and repeat ourselves.
Read this before we jump into the next example:

- super (…), we can use it to call the parent constructor (we need to use this keyword inside the child constructor)
- super.method (…), we can use this to call the method declared in the parent.

Ok, let us now try using the super keyword in our method. Here is the code:

```
class Dog extends Animal {
   stopRunning (){//this method will be used by the instances of the Dog class
      super.stopRunning();
      console.log('The dog is licking its paws');
    }
   bark() {
     return `The ${this.name} barks!`;
    }
 }
```

From the code above, you can notice that we called the parent method using the **super.stopRunning(),** and after that, we added extra code to represent our intentions for the Dog class. So this is like we are copying the content from the Animal method **stopRunning**() and putting it in the Dog method with the same name. The super keyword now makes our code much cleaner.
Let us test this in our console:

```
console.log(dog.stopRunning());
```

The output:

The Dog runs at a speed of 10km per hour.
The Dog barks!
The Dog stopped running and now sits still.
The dog is licking its paws.

Now you know how to override methods in the children class, remember they need to be with the same name as they are in the parent class.

Strict Mode

In JavaScript, we something called strict mode, and it was introduced in ECMAScript 5. This mode is the way to opt-in to the restricted version of JavaScript, and the sole purpose is to indicate that our code should be executed in

strict mode. With this strict mode, we can easily spot the hidden bugs and debug our code easily. For example, we can't use undeclared variables under this mode. As you can see, JavaScript has some unusual features that allow us to avoid getting errors or bugs in some cases. With this mode, we can control that behaviour, and we should always use the strict mode.
Okay, to enable this mode in JavaScript, all we need to do is to write this single line of code at the top of the file:

```
'use strict';
```

The single or double quotes can be used without any problems. If we want to use this mode in a **Node.js environment,** then we need to type this in the terminal:

```
node --use-strict
```

Benefits of using Strict mode:

1) Strict mode will ensure we are writing 'secure' bug-free code
2) Strict mode will resolve the errors by throwing new errors
3) Strict mode can be executed faster than non-strict code
4) Strict mode will prevent errors
5) Strict mode disables unusual JavaScript features

Important to note is that the strict mode can be applied to the entire script/code if we place the 'use strict' as a first line in the file or use strict mode inside individual functions.
Let us write the following code in our browser console:

```
> //Strict mode in JS
  'use strict';
  message = 'Hello my name is Andy, what is your?';
  console.log(message);
✖ ▶Uncaught ReferenceError: message is not defined
      at <anonymous>:3:9
>
```

As you can see from the example above, we cannot use the **message** variable because it has not been declared/defined before using it. Let us remove the 'use-strict' line and run the same code again:

```
> //Without Strict mode in JS

  message = 'Hello my name is Andy, what is your?';
  console.log(message);
  Hello my name is Andy, what is your?
<· undefined
```

The output is now printed without any errors.
As I mentioned before, we can use the strict mode inside functions as well. Here is one example where I will have two variables but without declaring them anywhere. One variable will be outside the function, and the other will be in the function body, using strict mode.

```
> a = 10;
  let addTwo = function(){
      'use strict';
      b = 15;
      return a + b;
  }
  let result = addTwo();
  console.log(result);
```

```
▶Uncaught ReferenceError: b is not defined
    at addTwo (<anonymous>:4:7)
    at <anonymous>:7:14
```

Here we have a reference error because variable **b** is not declared, but we are trying to use it in the code. The strict mode does not allow us to do this, and therefore it will throw a reference error. But why do we have only one reference error because the variable **a** is also not properly declared? The answer is simple because we use the strict mode only inside the function, so it did not catch the other errors.

'This' keyword–function context

When we do not use the strict mode then 'this' references to the global object if we call a function like this:

```
//function 'this' reference'
function message() {
   console.log(this === window); // true
 }
 message();// Output: true
```

In this example, the 'this' keyword references the global object, which is the window object in our web browser. We can confirm this because we can call the function **message** using the window object, and the result will be the same.

```
//function 'this' reference'
function message() {
   console.log(this === window); // true
 }
 message();//true
 window.message();// Output: true
```

From the example above, calling the message () function is the same as using the window.message().

When we use strict mode, then **this** keyword is set to undefined. Here is an example:

```
//function using the strict mode
'use strict';
function newMessage() {
   console.log(this === undefined);
 }
 newMessage(); //true
```

We can also use the strict mode inside the specific function. Check out the following code:

```
//strict mode inside a function
 function myMessage() {
   'use strict';
   console.log(this === undefined); // true

   function innerMessage() {
      console.log(this === undefined); // true
   }
   innerMessage();
}

myMessage();

Output:
```

```
true
true
```

The strict mode applies both to **myMessage** and **innerMessage** functions, which are set to **undefined** for both of them.

'This' keyword – method invocation

When we have an object, then 'this keyword' references to the object that owns that method.
What is a method?
JavaScript methods are the actions that can be performed on objects.
A JavaScript method is simply a property that contains a function definition.

Please consider the following example:

```
//method invocation
let animal = {
   type: 'Mammals',
   getType: function () {
      return this.type;
   }
}
console.log(animal.getType()); // Mammals
```

In this example, **this** object in the getType() method references the animal object.
Since this method is a property of the Animal object, you can store the value.

```
let animalType = animal.getType;
console.log(animalType());//undefined
```

If we run this code in the console, we will get **undefined** instead of **Mammals** because we called the method without specifying the object. Therefore, JavaScript will set the **'this'** keyword to the global object in non-strict mode and undefined in the strict mode. We can solve this if we use **bind** () method, where we can specify to which object the 'this' keyword belongs.
Here is how we can do this:

```
//bind method where we pass the animal object
let animalType = animal.getType.bind(animal);
console.log(animalType()); // Mammals
```

Summary

Congratulations in this chapter we have discussed about the basics of JavaScript. There are a lot more basic to intermediate features that I want you to learn and these are all covered in my first book. The link to my first book will be included at the end of this book together with some additional information you will need to practice JavaScript. In this long chapter we have covered JavaScript types, functions, arrays, loops, conditions, objects, classes, arrow functions and this keyword.

Chapter 5: Final Chapter
DOM – Document Object Model

The DOM or the **Document Object Model** is essential for making our website pages interactive and user-friendly. The **DOM** is not a programming language, but it is an interface that allows other programming languages to manipulate a website's content, structure, and style. We know that JavaScript is a scripting language that allows us to implement features on web pages. These features can be very simple or very complex. JavaScript today is everywhere. Whenever a user sees content being updated, or every time a user interacts with maps, or there is 2D or 3D animation in our pages, we are most likely talking about JavaScript events running in the background. In short, JavaScript is an amazing client-side scripting language that can connect to the DOM and manipulate the page content. This chapter will cover everything you need to know about DOM and make our web pages more dynamic using the document object.

Introduction

As I mentioned before, JavaScript is one of the most popular web programming languages. Every time a user clicks on the website, either a simple website menu clicks or even a click on a submit button from some form, then we can be certain that JavaScript processes are running in the background and they usually access and try to manipulate the **DOM**. Every website consists of **HTML** and **CSS**. The browser we are using is a simple program that interprets the HTML markup and renders the **CSS** style. **HTML** stands for **Hypertext Markup Language**. **CSS** stands for **Cascading Style Sheet**. **CSS** exists to format the content of the web page written in **HTML**. We cannot have a webpage without **HTML**. **HTML** is a very simple language that is consisted of elements. HTML describes the structure of a web page. **HTML** elements are very important because they tell the browser how to display the content.
Here is one example of very simple **HTML** document:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Document title goes here</title>
</head>
<body>
   <h1>H1 Heading</h1>
   <p>Paragraph</p>
</body>
</html>
```

In the first line of this example, we have the **DOCTYPE** element. This declaration tells the browser that we are dealing with an **HTML5** document. The **<html>** element is the root element of the HTML page. So, what is an **HTML** element? Well, an **HTML** element is defined by both a start and end tag. In between those tags, we have the element content.
Here is one example of **HTML** element:

```
<h1>H1 Heading</h1>
```

So the **HTML** element is everything from the start to the end tag. The end tag is the same as the start tag with only one difference, and that is the forward-slash must be included </>.
The **<head>** element or tag will contain all the meta tags for the required **HTML** page. There we have a title tag like this '<title>' shown in the browser title bar and represents the title of the entire **HTML** page. We have the entire document body in the body tag <body> to include many different **HTML** tags. Consider the body tag as a container where we put all visible page contents like headings, images, links, tables, lists, etc. In our example, we have **<h1>** and <**p>** elements, where **h1** stands for large heading, and p tag stands for a paragraph. This is the simplest **HTML5** explanation you will ever read, but I wanted you to know the very basics before we start working on the DOM.
Here is a list of important HTML elements that you can read about:

https://developer.mozilla.org/en-US/docs/Web/HTML/Element

What about the **CSS**? The existence of **CSS** allows us to have a separation of the content and the document format. You should know that we can have multiple **CSS** style sheets in our HTML document. We write **CSS** rules to style the **HTML** markup or the tags/elements. We can include these **CSS** style sheets in different ways. The first one is to use an internal stylesheet in our document. This one is placed inside the head of the document or to be more specific in the head tag (remember we need to use <style> opening and closing </style> tag). Here is an example of internal stylesheet where we style the paragraph text to be red and 18px in size. The 'px' stands for pixels:

```
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Document title goes here</title>
   <style>
      p { color: red; font-size:18px; }
   </style>
</head>
```

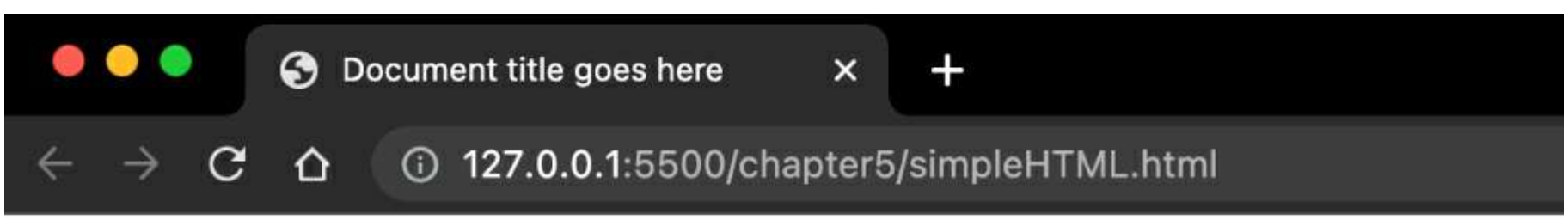
The second way to include the **CSS** styles is to use the inline method. As an example, we can make our paragraph text to be red again and with a font of **18 px** if we add this style attribute directly in our paragraph tag:

```
<p style="color: red; font-size:120%; ">
            This is second paragraph
</p>
```

As you can see, we can style the individual elements/tags directly in the tag declaration using the inline method. The third and the most popular way is to include **CSS** styles sheets in our **HTML** by importing them using the <**link**> tag inside the head element of the document. This is how we can link an external **CSS** file in our document:

```
<link rel="stylesheet" type="text/css" href="styles.css">
```

The href attribute is where we provide the path where the file is, so the path to the file and the name of the file. As you can see, our CSS files have a '**.css**' extension, and the **HTML** file has a **'.html'** extension.
If you open this document in your browser, it will look like this:
(the document I'm working on is in **chapter5** and the file name is **simpleHTML.html**):



H1 Heading

Paragraph

This is second paragraph

Now you know how **HTML5** and **CSS** files work together to create nice-looking web pages. Let me cover a few more basic things before working on the **DOM**. So each **HTML5** element or tag can have one or more attributes. We need these attributes so we can add more information about them. We need to specify these attributes in the start tag, not the closing tag. The attributes have name and value, for example, **href='https://www.google.com/',** where href is the name of the attribute, and the value is the entire Google link. Single or double quotes can enclose the values of the attributes.
Here is the full example of **HTML** link element that points to Google:

```
<a href="https://google.com/">Visit Google</a>
```

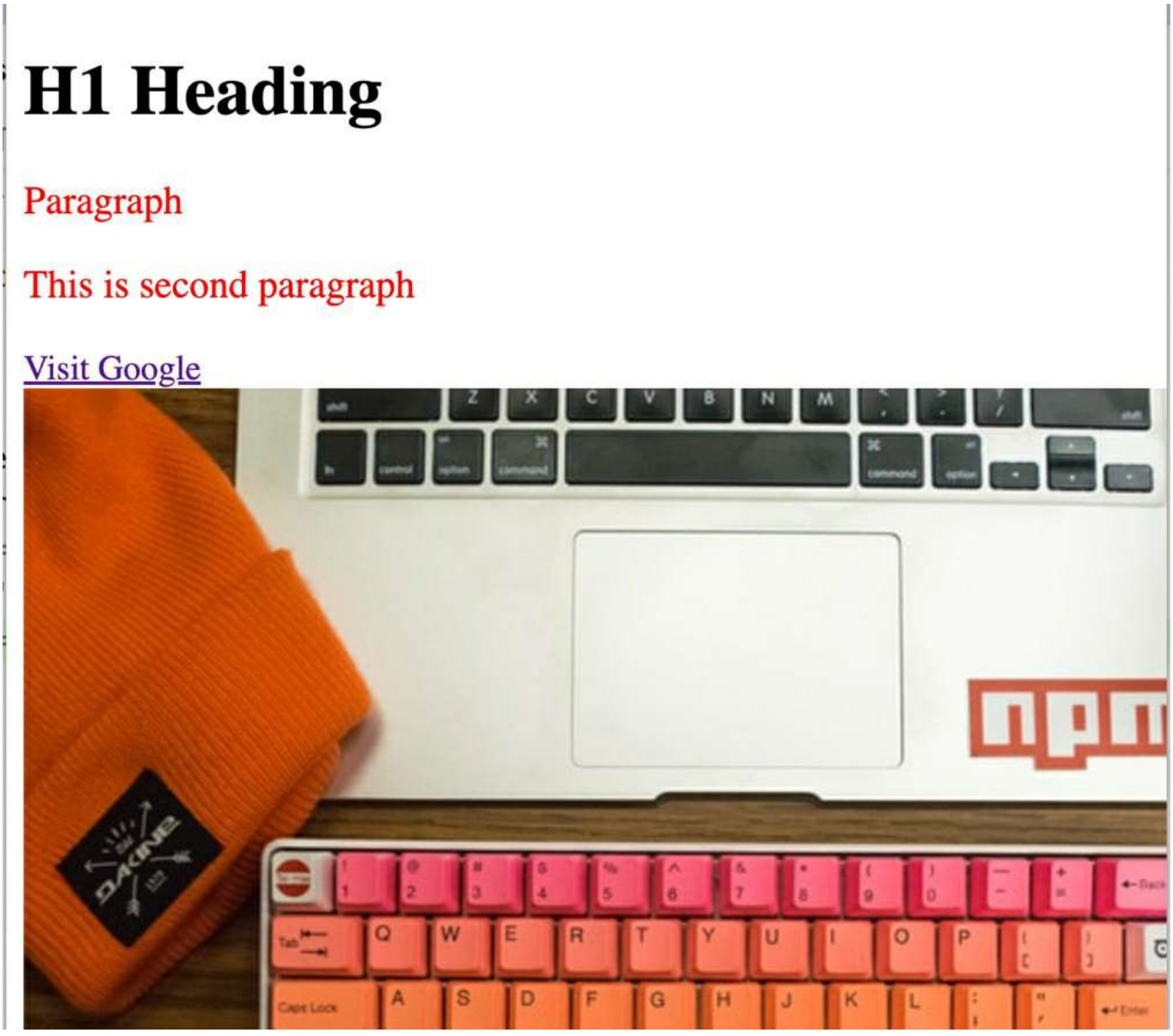
The **<a>** tag is an element that we use to define a hyperlink. This can go somewhere outside our document or link things in the same document. The href is an attribute where we put the endpoint or the URL as value.
Another interesting attribute is the **'src'** attribute. The 'src' stands for source and is used when dealing with images or video files.

```
<img src="./paul-esch-laurent-oZMUrWFHOB4-unsplash.jpg">
```

As you can see, that src attribute specifies the path to the image, and this path can be a relative **URL** like the one we have in the example above, or it is an absolute **URL** that will be a link like this:

```
<img src="https://example.com/hello.png">
```

If we open the document now, we have this:



As you can see, the image is too big, and it is not displayed properly. Well, the **img** tag can have two more attributes like width and height where we can specify what size the image should be:

```
<img src="./paul-esch-laurent-oZMUrWFHOB4-unsplash.jpg" width="300px" height="300px">
```

Now, if we open the browser, this will be the output:

Great, we have the image loaded, but the **img** tag can also have an ‘alt’ attribute that specifies the alternative text for an image. For example, imagine that the path we provided is broken and is not working, so we can use the alternative tag to tell us what the image above is. This text will be displayed only if the image cannot be found or loaded successfully.

```
<img src="./paul-esch-laurent-oZMUrWFHOB4-unsplash.jpg" width="300px" height="300px" alt="JS image">
```

So we have covered a lot of things, but they are very, very basic things, and now is the time where we need to start thinking about other **HTML** attributes that we can use in our document object model. For example, one such attribute is the **HTML ID** attribute. The **HTML5 IDs** are being used to specify the unique identifier for the **HTML** element. The rule is that we cannot have multiple elements in our document with the same id value. They should be unique or have unique values. We need this element **ID** for two things. We can either target that specific element and apply some specific **CSS** styles, or we need this attribute to perform JavaScript element manipulation. We can use the elements **IDs** in JavaScript to manipulate that specific element. That is why the **IDs** should be unique, so we know we are targeting the corresponding element. Here is one example of using the **IDs** to apply **CSS** styles. We can also use the **IDs** to do JavaScript manipulation, but we will cover that later. So we can target specific elements using its **ID** and apply some **CSS** styles:
The **HTML5** markup for the paragraph element:

```
<p id="paraID">
   Style this paragraph from our styles.css
</p>
```

CSS rules – inside the **styles.css** file

```
#paraID {
   background-color: darkblue;
   color: white;
   padding: 40px;
   text-align: center;
 }
```

So here we have a paragraph tag and our **HTML** markup with **ID** with value ‘paraID’. We can now target this id

inside our external **CSS** file if we use the hash character (**#**), followed by the name of the **ID**, and then we need to apply **CSS** properties or **CSS** rules inside the curly braces {}.
Here is how the paragraph now will look after we applied some **CSS** styles using the element id:

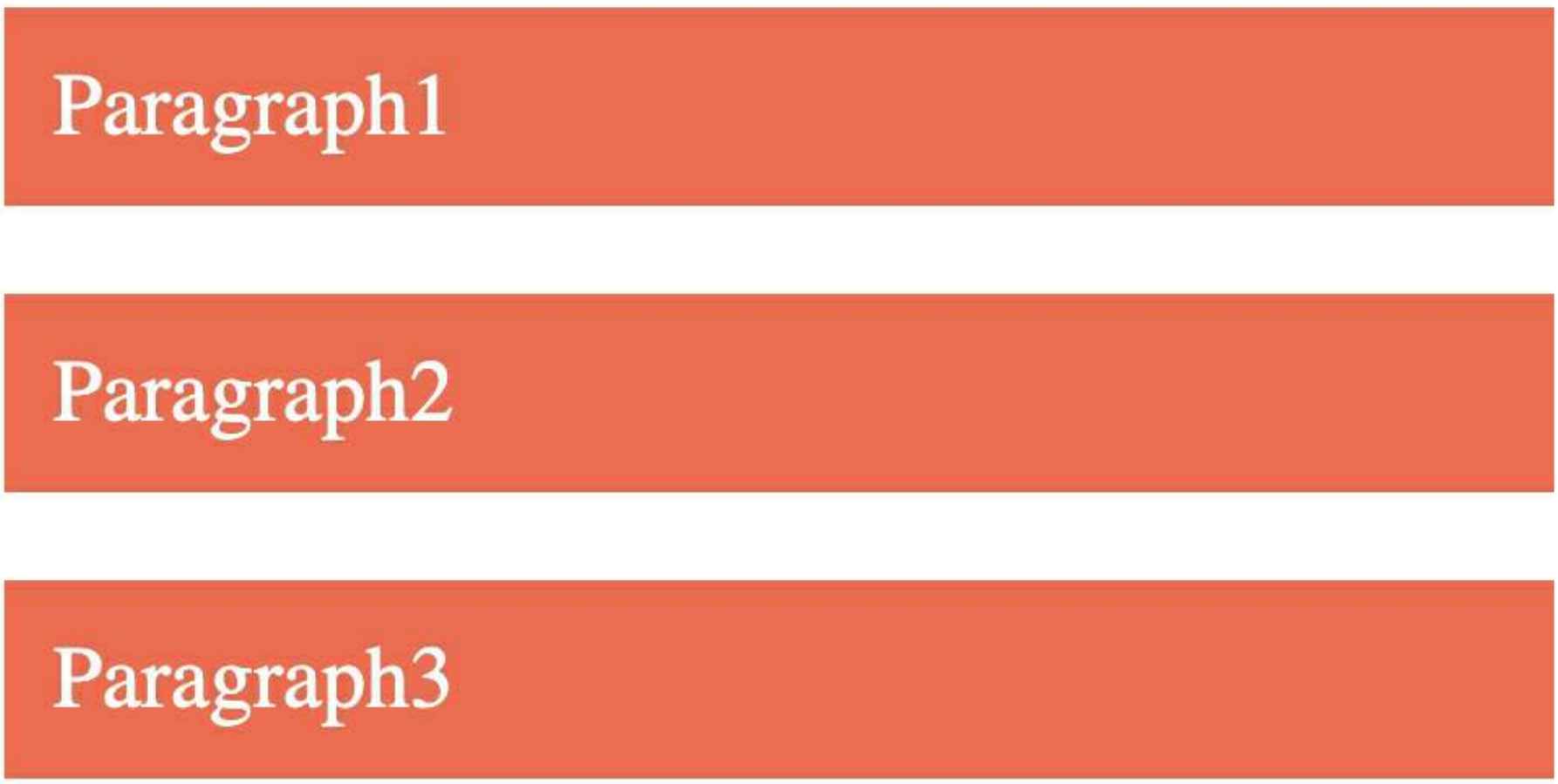
Another attribute that an element can have is called ‘class’ name. The classes can be used in many different HTML elements, and the ID must be unique for the entire document. Same as with IDs, we can use the class name of the particular element to apply CSS styles or to do JavaScript manipulation. Here is another example of how we can have multiple files with the same class name:

```
<p class="paraClass">
    Paragraph1
</p>
<p class="paraClass">
    Paragraph2
</p>
<p class="paraClass">
   Paragraph3
</p>
```

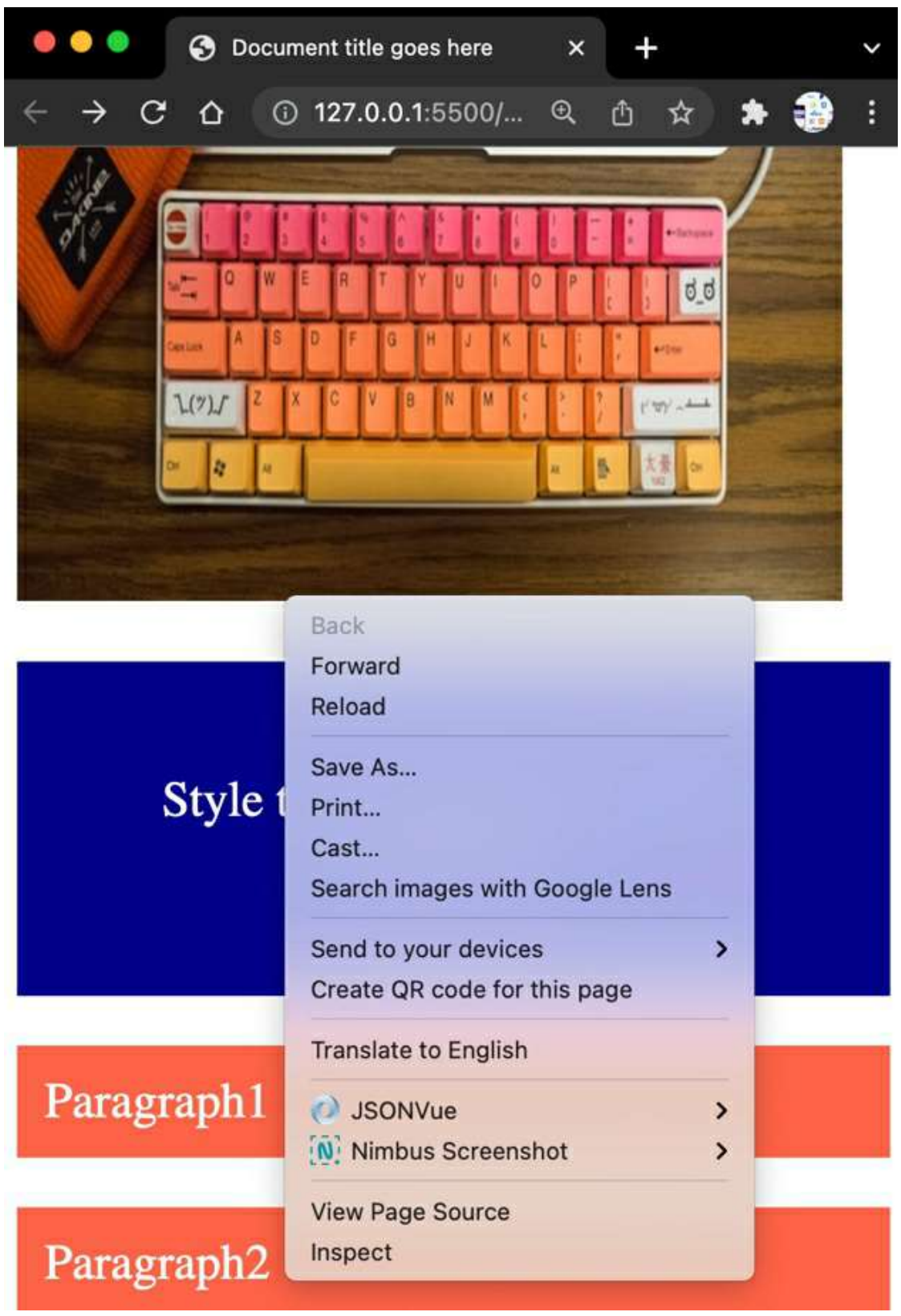
Here is how we can style elements by their class name (note that here we are using ‘.’ to access the class name instead of # symbol we used with IDs):

```
.paraClass {
  background-color: tomato;
  color: white;
  padding: 10px;
}
```

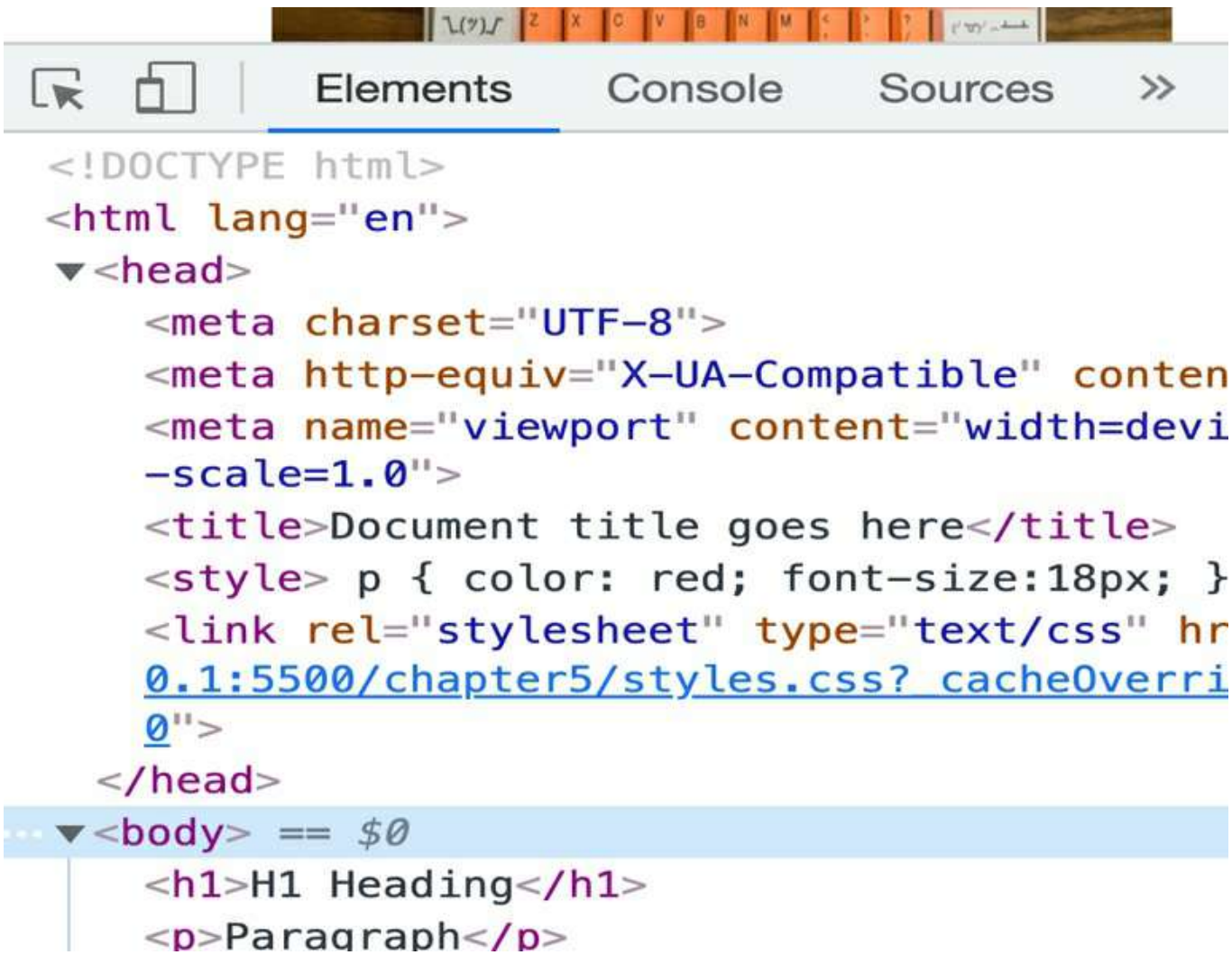
The output:



Now we know about **HTML** markup and how we can use the **CSS** style sheets, so you can style those **HTML** tags to look nice. When a browser loads the page, it creates a representation of that document known as the **DOM** or **Document Object Model**. JavaScript needs this model to access the **HTML5** tags or elements from this document as objects. JavaScript does not understand the **HTML5** markup elements or the <**h1**>, <**p**> tags, but it does understand objects. We know that in JavaScript, everything is an object. I will use the same **HTML** file that we were working up until this point. This **HTML** document is very simple but contains the essential elements that one page should have. I will use the Google Chrome web browser to explain how **DOM** and JavaScript work together. Like many other modern browsers, the Chrome browser will come with the Developer tools. There are different ways to open the Developer Tools, but I will show you the most basic one. So load the index file, right-click anywhere on that page, and select the ‘Inspect’ option; in my case and probably in yours as well, this option should be the last one.



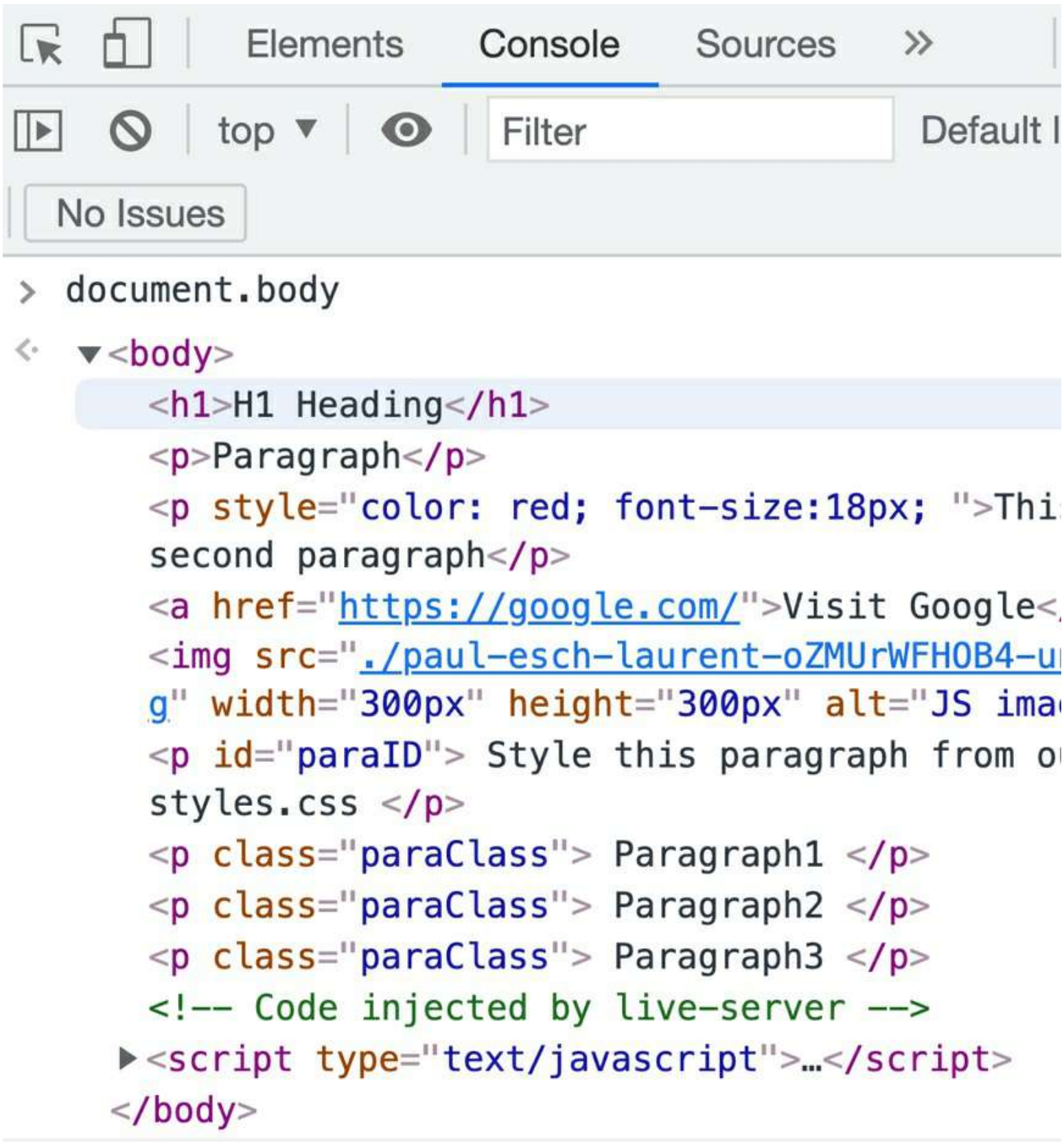
After clicking on the ‘Inspect’, you should see this window:



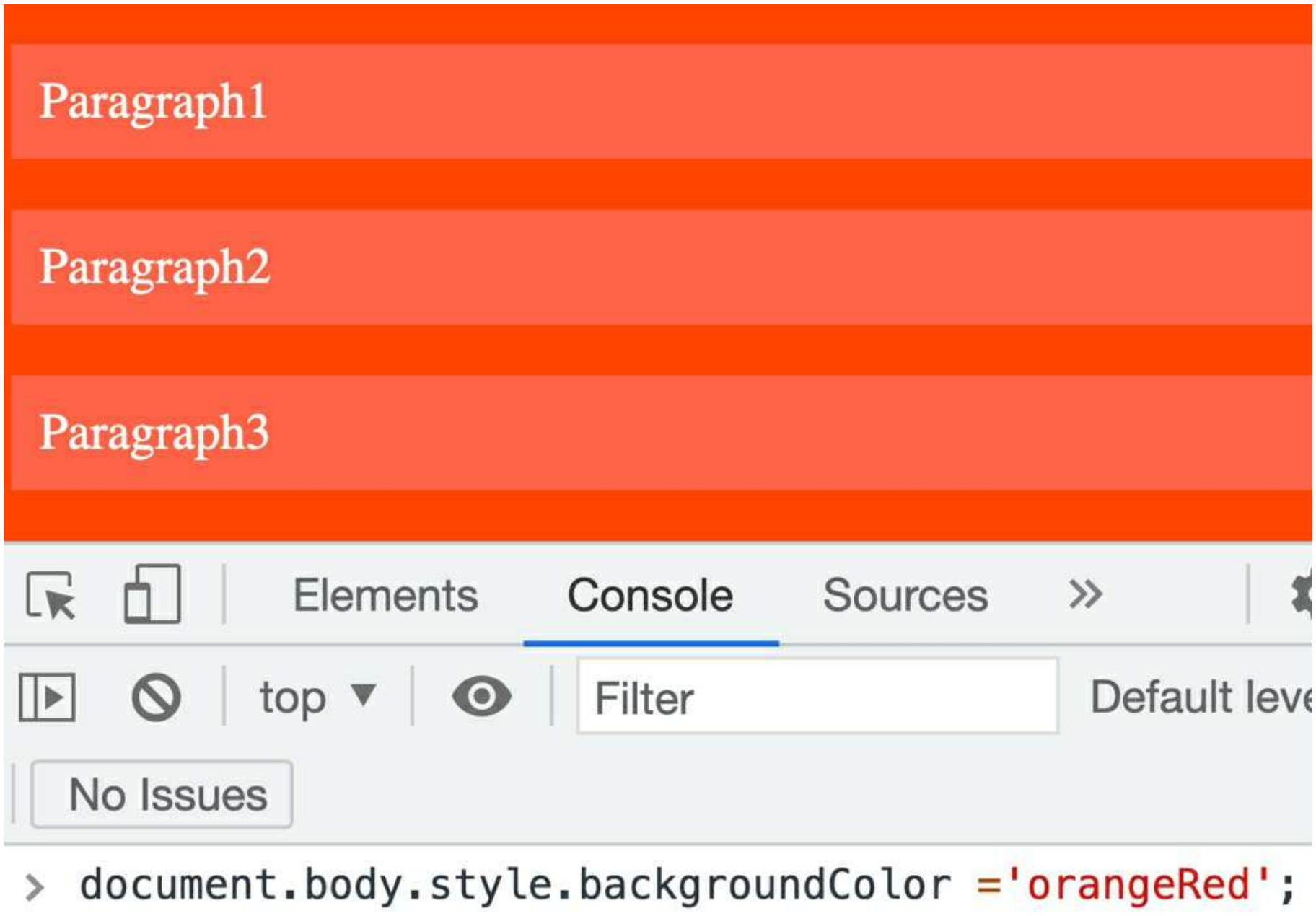
Under the ‘Elements’ Tab, we can see the **DOM**. If we expand all sections like head and body, it looks exactly the same as the **HTML** source code inside the **simpleHTML** file. If we hover each of the **HTML** elements with our mouse, it will highlight the respective element in the rendered page. As you can see, I have highlighted the body element from the **DOM**, and next to the body, we have a little arrow that we can toggle to expand or hide the rest of the elements. Please feel free and play around with the developers' tools. If you open them for the first time, then it can happen the entire developer tools window to be on the right side of the browsers. It is up to you to customize the developer tools window according to your needs and preferences.

DOM vs HTML Markup

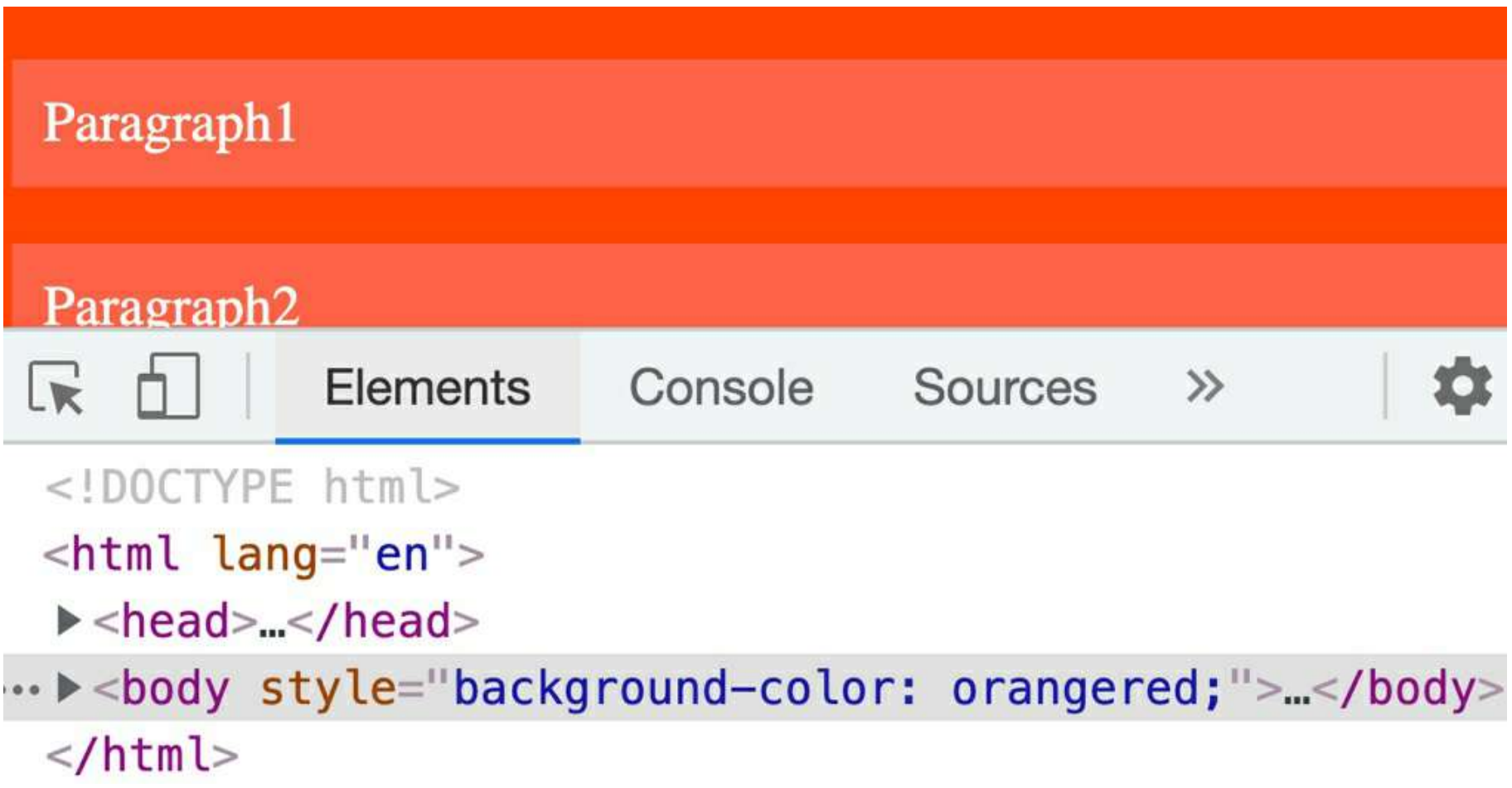
As we saw under the ‘**Elements’** tab in our Developer Tools, we have the **DOM**, and it looks completely identical to our **HTML** source code. But the browser-generated **DOM** is still different from the **HTML** markup because the JavaScript can modify the **DOM**. We need to run a small test to show you that JavaScript can access and modify the **DOM**. In the Developer tools, there is a ‘Console’ tab so you can select that one and write ‘document.body’ and hit enter or return:



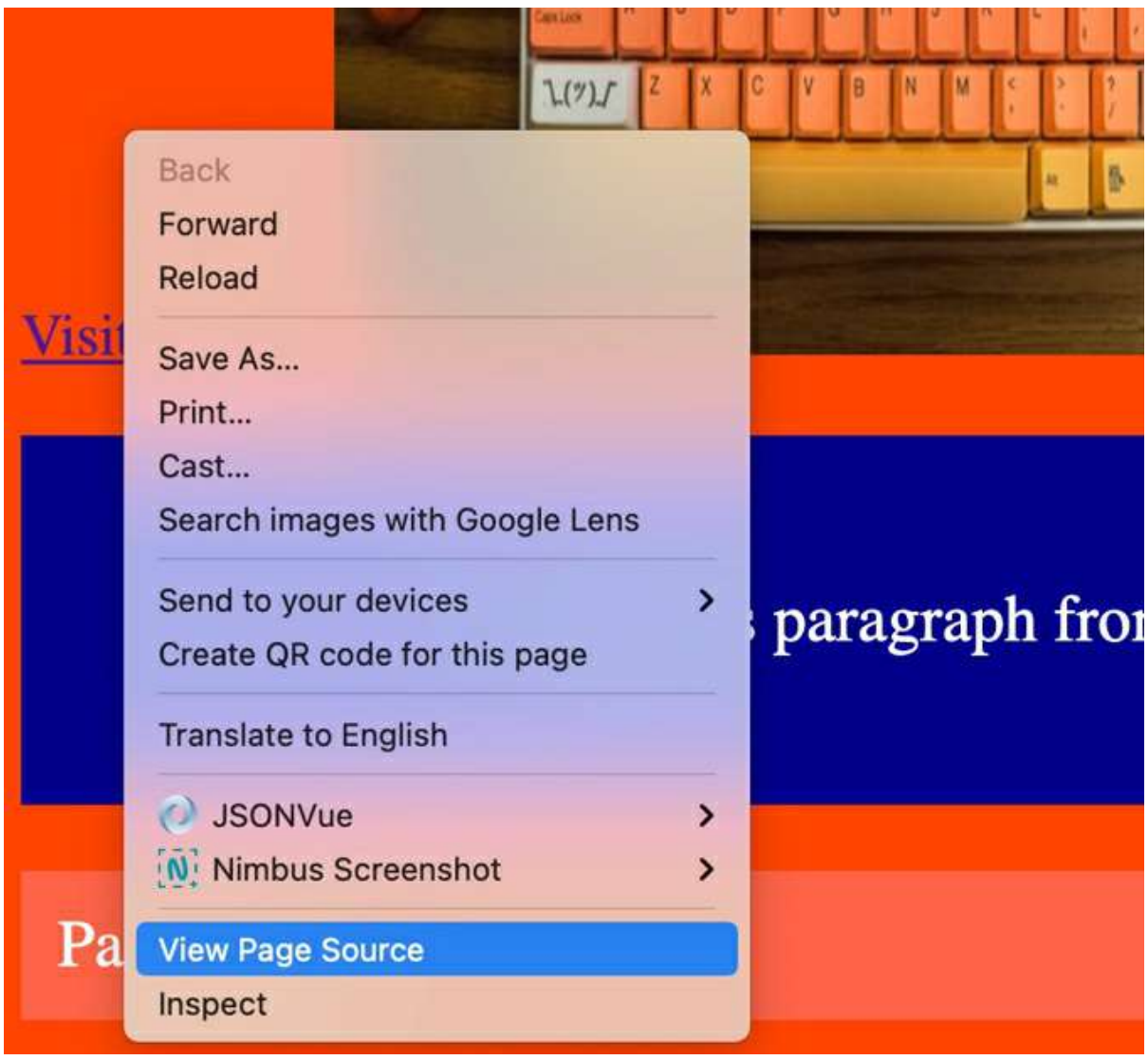
As you can see from the screenshot above, we got the whole body back. How is this possible? The document is a built-in object with methods and properties that we can use to modify the pages. That is why when we type ‘document.body’ in our console, we will get the entire page body because the ‘document’ is the object and the ‘body’ is a property of the object. Because we are dealing with objects, we can use the dot notation to access the properties. At the moment, we do not have any specific **CSS** styles for the body, but we can add some new properties to the current body. Let us say that we want the body background to be orange-red. We can add a style to our body element if we type this code in the browser console:



As you can see, we have done some modifications to our document body, and if you go back and look into the Elements tab, you will see that we have added a new style:



I want you to pay close attention to the style we added ‘background-color’. How come we have this property and we typed ‘backgroundColor’in the console? This is heppning because the properties in **CSS** have hyphens, but in JavaScript, we are using the camel case notation, and we omit the hyphens. We have managed to use the ‘document’ in-built object to modify the background color of our body element. Again, the body is a property of the document object. If we open the source code in your editor, you will not see any change we have made directly from the browser console. You do not need your code editor to see the source code because we can inspect the source code from the browser, so anywhere on the page we have opened in our browser right, click with your mouse and select the ‘View Page Source’.



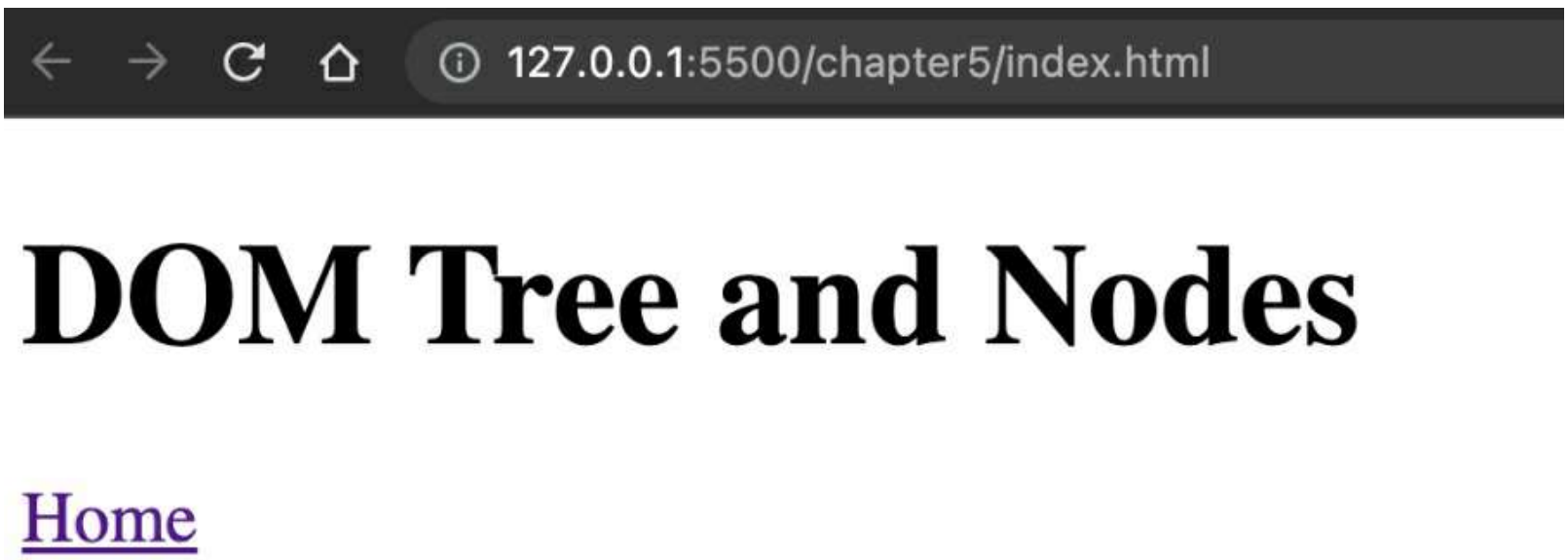
This will give you the same code that we have in **simpleHTML** file. This means that the source code that is coming from our **simpleHTML** file will not be affected by the JavaScript modifications we made from the browser console. If you are testing this example on your computer, you can refresh the page in your browser, and you will see that everything will be back to normal, meaning the styles we added through the browser console will be gone. This is happening because those changes we made from the console were never saved into the **HTML** file, and we were able to do those changes because the **DOM** gave us access to the page elements as they were objects.
Now you know how we can use the 'document' object to access the **DOM** elements straight from our console. JavaScript uses this object to update or modify the **DOM** properties. We have also learned that we will not change the original **HTML** source code when using the in-built object.

DOM Tree and Nodes

To understand how DOM works, you need to know the HTML terminology. At the beginning of this chapter, I included a few HTML elements so you can easily understand the things that I will explain in following section. Let us start with creating a new HTML file:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>HTML Terminology</title>
</head>
<body>
 <h1>DOM Tree and Nodes</h1>
 <a href='index.html'>Home</a>
</body>
</html>
```

If you open the file in your browser, you will see this output:



I have installed a **VsCode** extension called **liveServer**, and you can install the same extension or find similar so you can automatically open and load your **HTML** files in your browser. That is why my browser **URL** starts like this: '*127.0.0.1:5500/chapter5/index.html*'. The port number, in your case, might be different, but that does not matter.

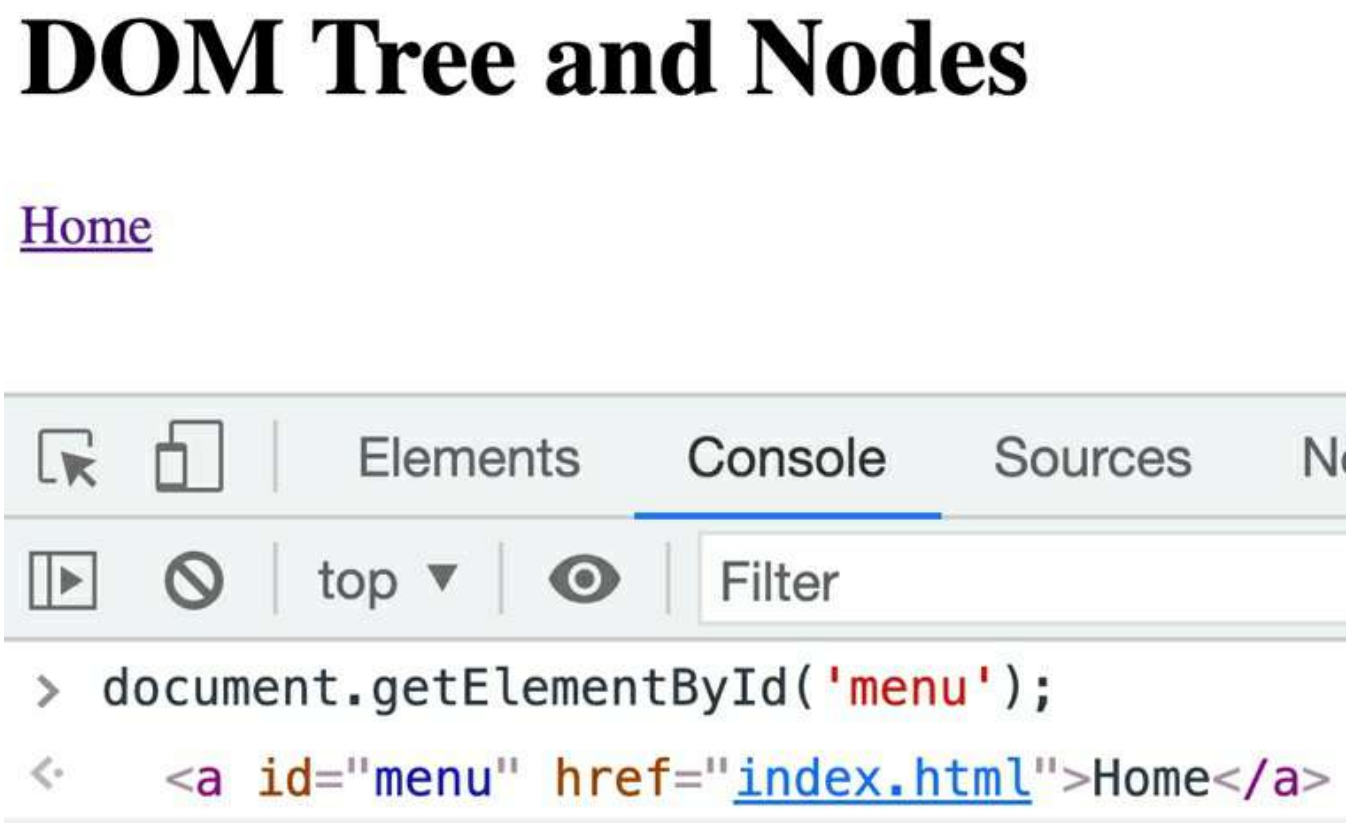
Let us have a look at this particular HTML markup:

```
<a href='index.html'>Home</a>
```

Here we have an anchor element that links to the same page. So we have <a> tag with href attribute. The 'index.html' inside the href is the attribute value. The 'Home' is the text that will displayed in the browser and that is why the end user will see. So far, we have learned that we can style this link from our external **CSS** file if we have an ID or class name associated with the corresponding **HTML** element. Let us add the **ID** to this <a> tag called 'menu':

```
<a id='menu' href='index.html'>Home</a>
```

Open the Developers tools and type this in the console 'document.getElementById('menu');':



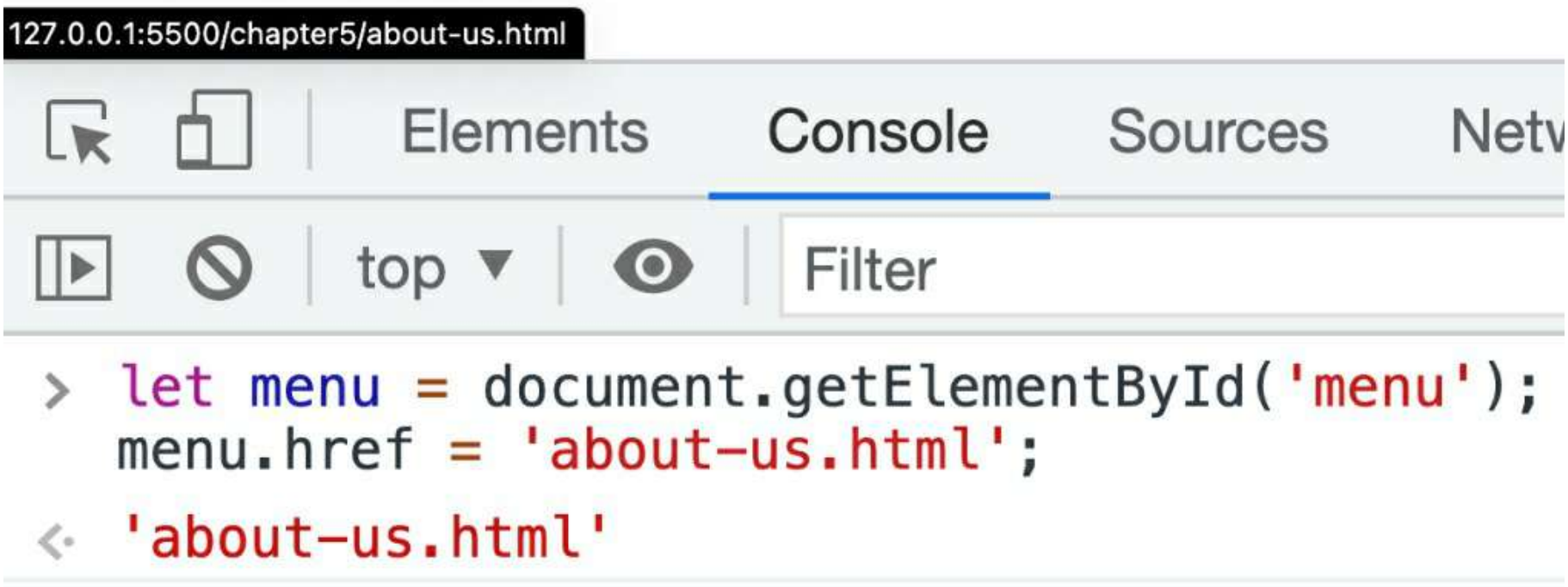
In this example, I have used one of the most common methods used for **DOM** manipulation, and this method is called **getElementById**(). This method will get information from an element by its unique **ID**. There are many other methods, but the **getElementById**() method will return one element from our document with the same **ID** we are passing into the method. As you can see, this method takes only one parameter, and that is the **ID** of the element. We should put the **ID** in single or double-quotes. We know that in JavaScript we love using variables, so we can store the result of what this method returns into a variable to use later when we need it.

```
let menu = document.getElementById('menu');
```

We have now successfully stored the entire anchor element in a variable called 'menu'. Because we have the entire element stored in the menu, we can use this variable name so we can easily modify the values and attributes. For example, let us replace the current attribute value 'index.html' to a different value like 'about-us.html'.

DOM Tree and Nodes

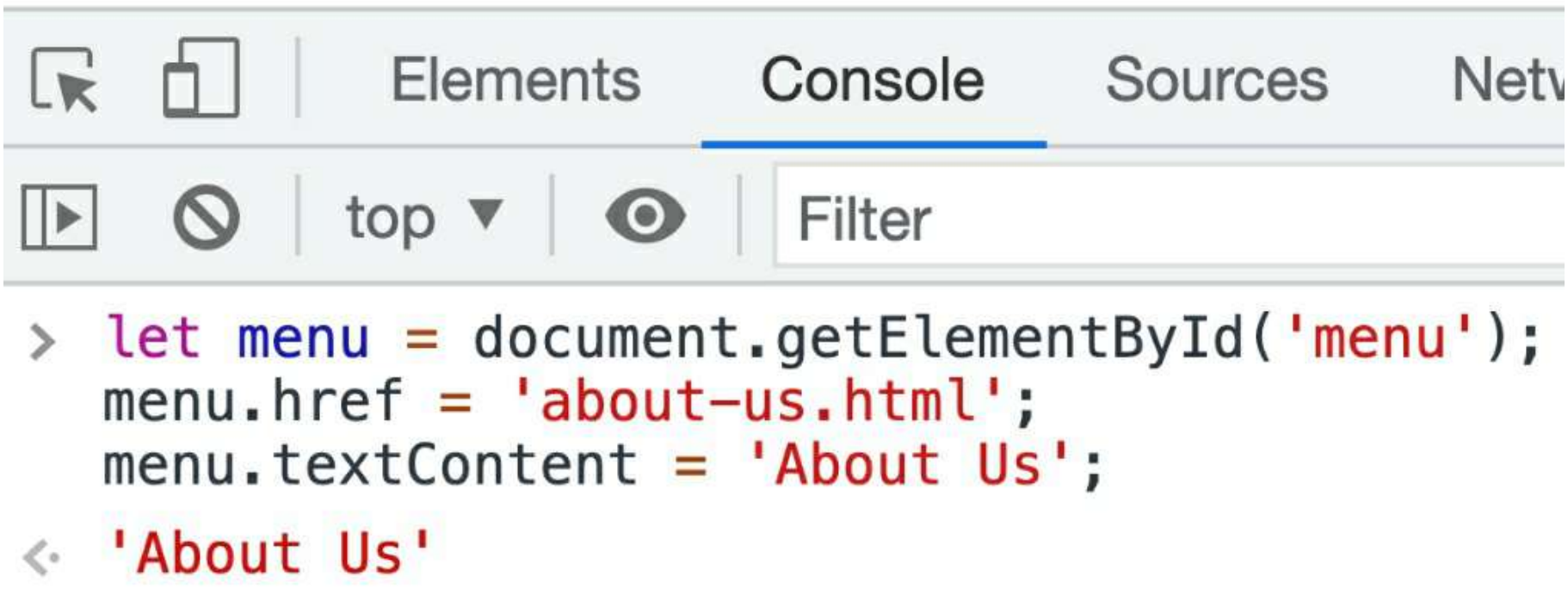
Home



As you can see, now I have hovered the ‘Home’ link in our browser, and now it gives us ‘about-us.html’ as a link value.
We can also change the text description from ‘Home’ to ‘About Us’ like this:

DOM Tree and Nodes

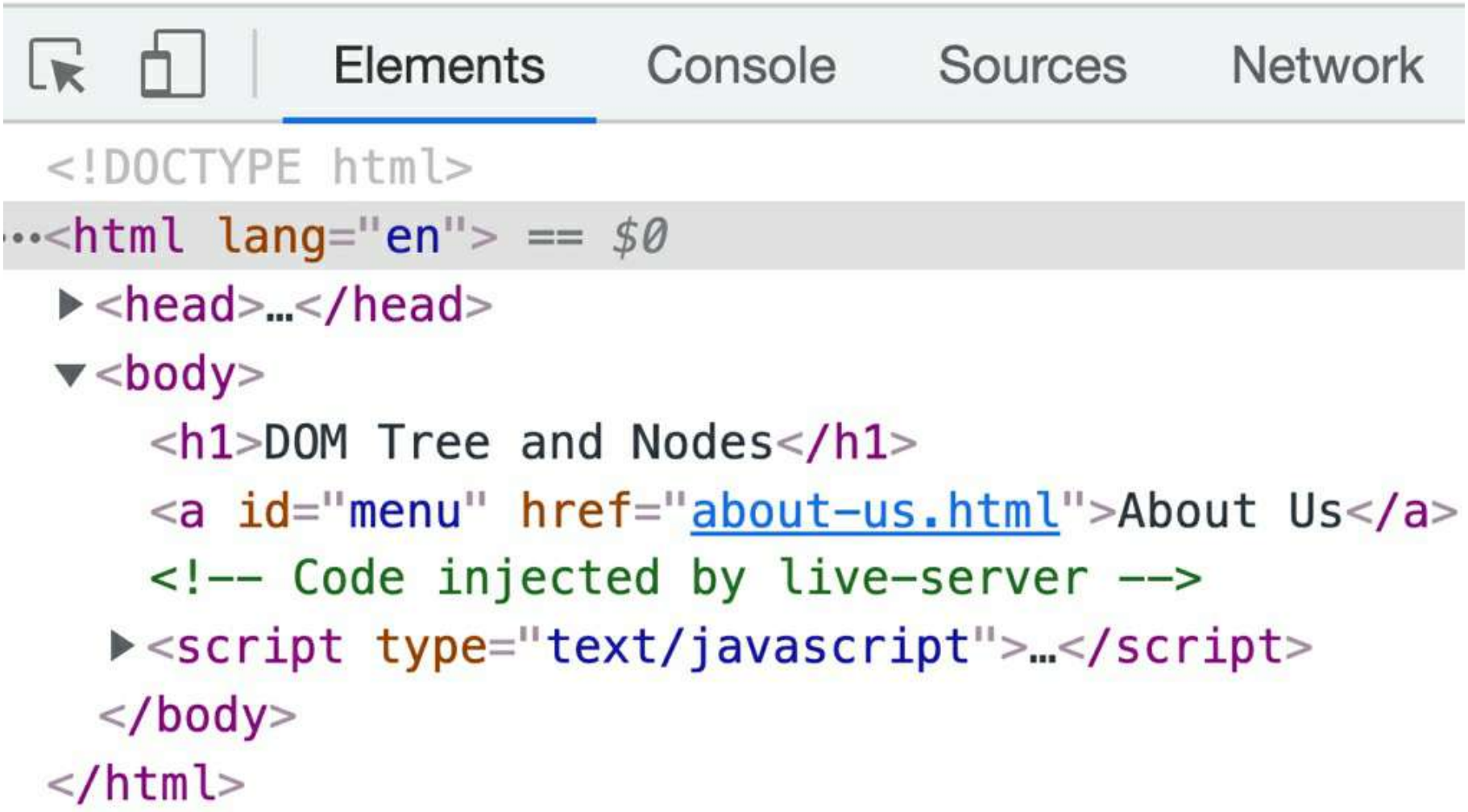
About Us



As you can see, the text is changed, and we have used the **textContent** property here. If you want to see the changes, you can click on the Elements Tab:

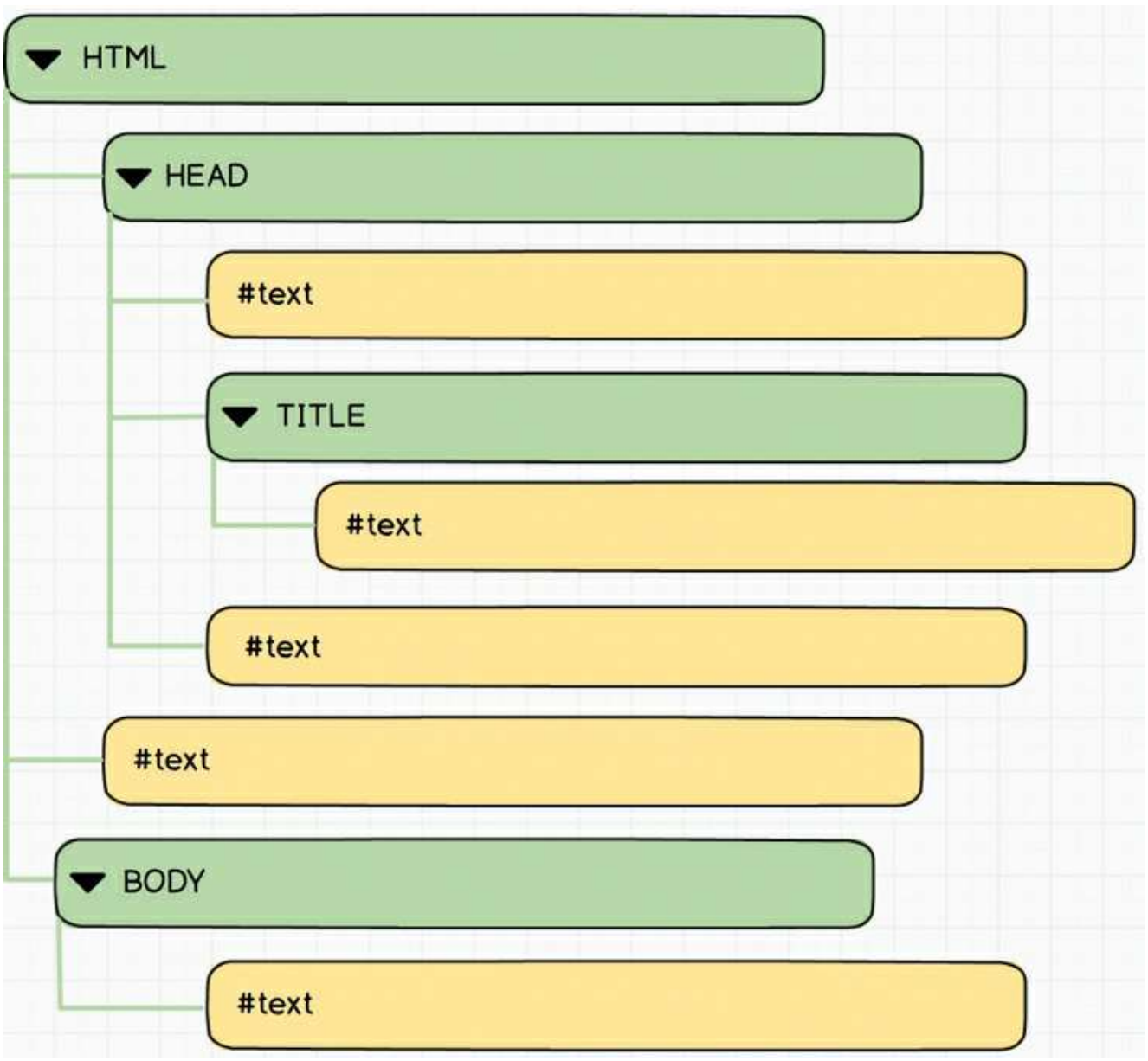
DOM Tree and Nodes

About Us



Great! The changes are reflected on the front end of the website. Remember, if you refresh this page, everything will be reverted as before we started the JavaScript manipulation. This was easy because, with the help of the in-built document object, we can get access to a particular element in our **DOM** and modify or alter the same element properties and values. This was the first step and let us focus more on **DOM** elements. All of the **DOM** elements are known as nodes. The **DOM** represents an **HTML** document as a tree structure. Each of the **DOM** items, such as elements, attributes, text, etc., is organized in a tree-like structure referred to as a **DOM** tree. As in every other tree, the **DOM** tree has branches, and each of these branches ends in a node. We have different types of nodes in the **DOM** tree, but mainly we need to know these three:

- Element nodes
- Text nodes
- Comment nodes



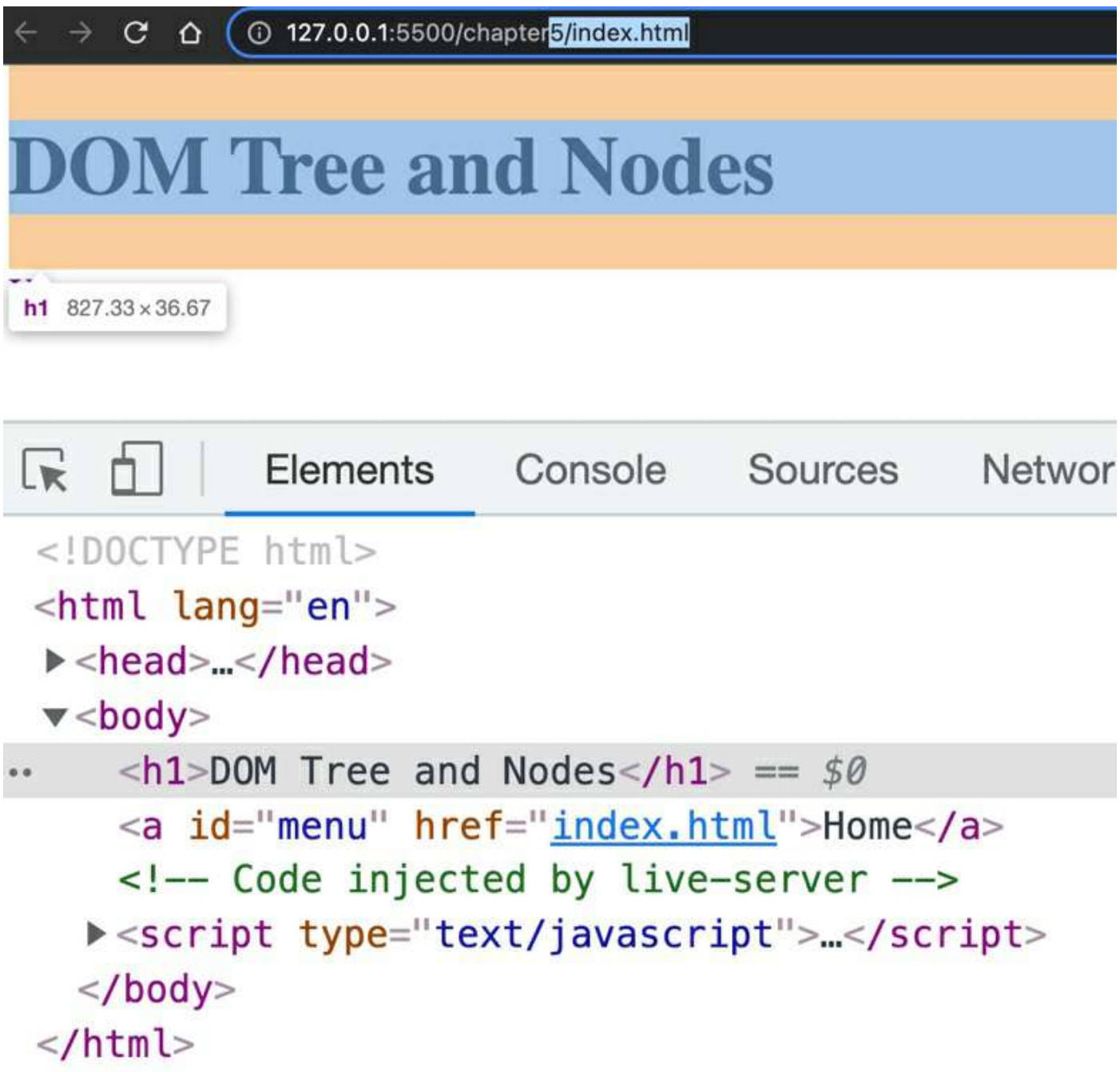
You should know that every tree node is an object. Each **HTML** element in the **DOM** is called an element **node**; therefore, the tags are element nodes or simply referred to as elements. The **<html>** tag is the root node or a parent node. After, the root **<html>** node we have the **<head>** and **<body>** tags which are siblings, children of the parent node. As we can see from the picture above, we have five yellow text rectangles in the document. These tags are known as text nodes. The text node contains a string only. It may not have any further children. Consider these nodes as tree leaves. What is also important is that the white spaces and newline characters are considered valid characters and, therefore, text nodes. When it comes to spaces and newlines, there are two exclusions only:

- The spaces and newlines are ignored if they are before the <**head**> node
- From **HTML**, we know that the body is a container where the rest of the elements should go, so everything declared outside the <**body**> will be added and included in the body. After the closing body tag </**body**>, the spaces we put will be ignored.

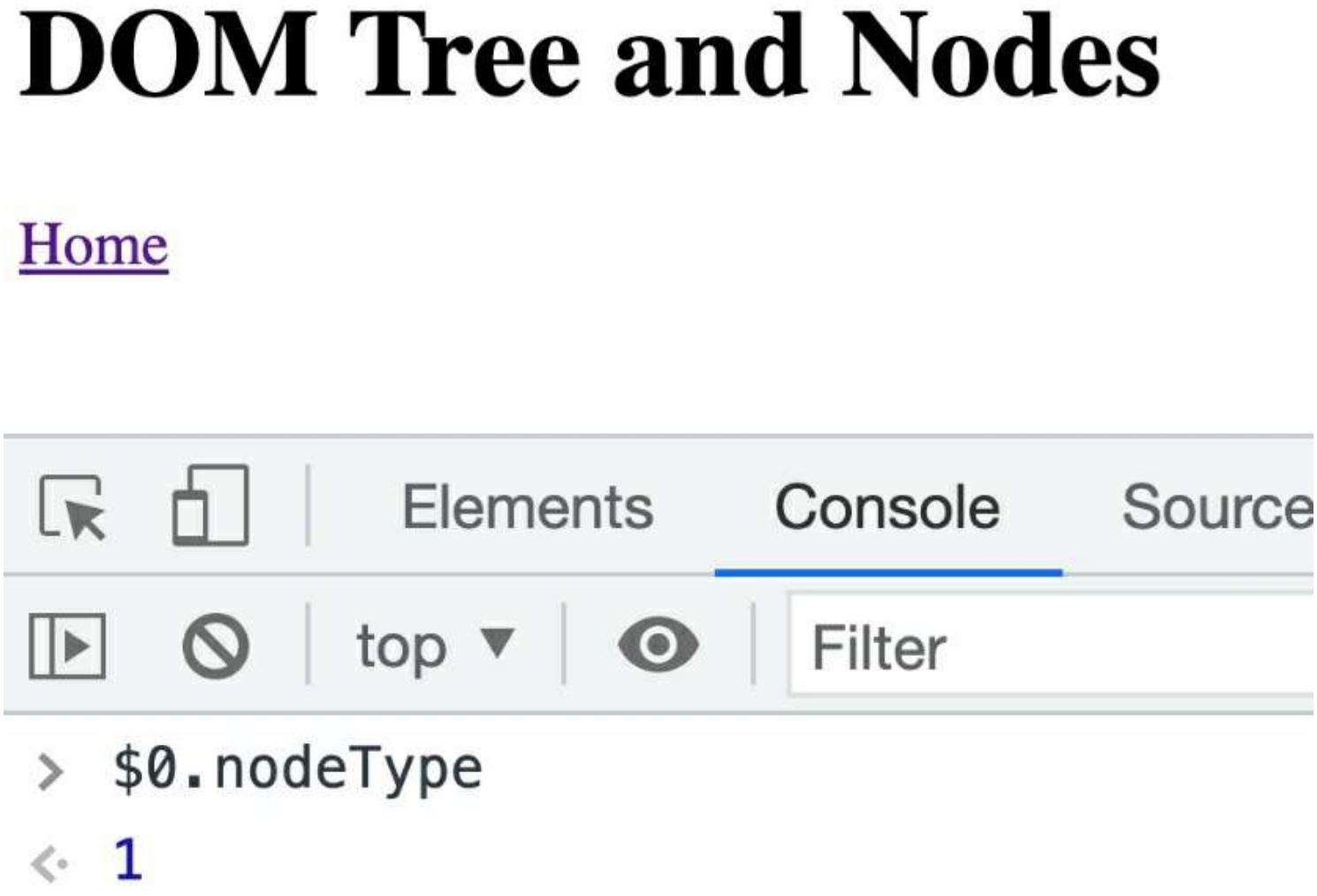
Every node in our **DOM** has a node type, and the **nodeType** property can help us discover the node type.
The **MDN** website has a complete, up-to-date list of all valid node types:

https://developer.mozilla.org/en-US/docs/Web/API/Node/nodeType

Okay, let us test the node types of some of our HTML elements. Open the ‘index.html‘ file in your browser, and in the Elements tab of the Developer tools, highlight the heading ‘h1’ element:



As you can see, I have highlighted the h1 element, then in the DOM, the value '==$0' will appear next to the element I select. This tells us two things: what element is being selected and how we can access this currently active element. Now, if we select the Console tab and type **$0.nodeType,** it will return value 1:



What does this value '1' even mean? Output 1 correlates to **ELEMENT_NODE**, and indeed our h1 tag is an element node. Here is a chart with the most common node types and their corresponding values:

Node Type	Value	Example

ELEMENT_NODE	1	<h1> or <body>
TEXT_NODE	3	Text
COMMENT_NODE	8	<!-- an HTML comment-->

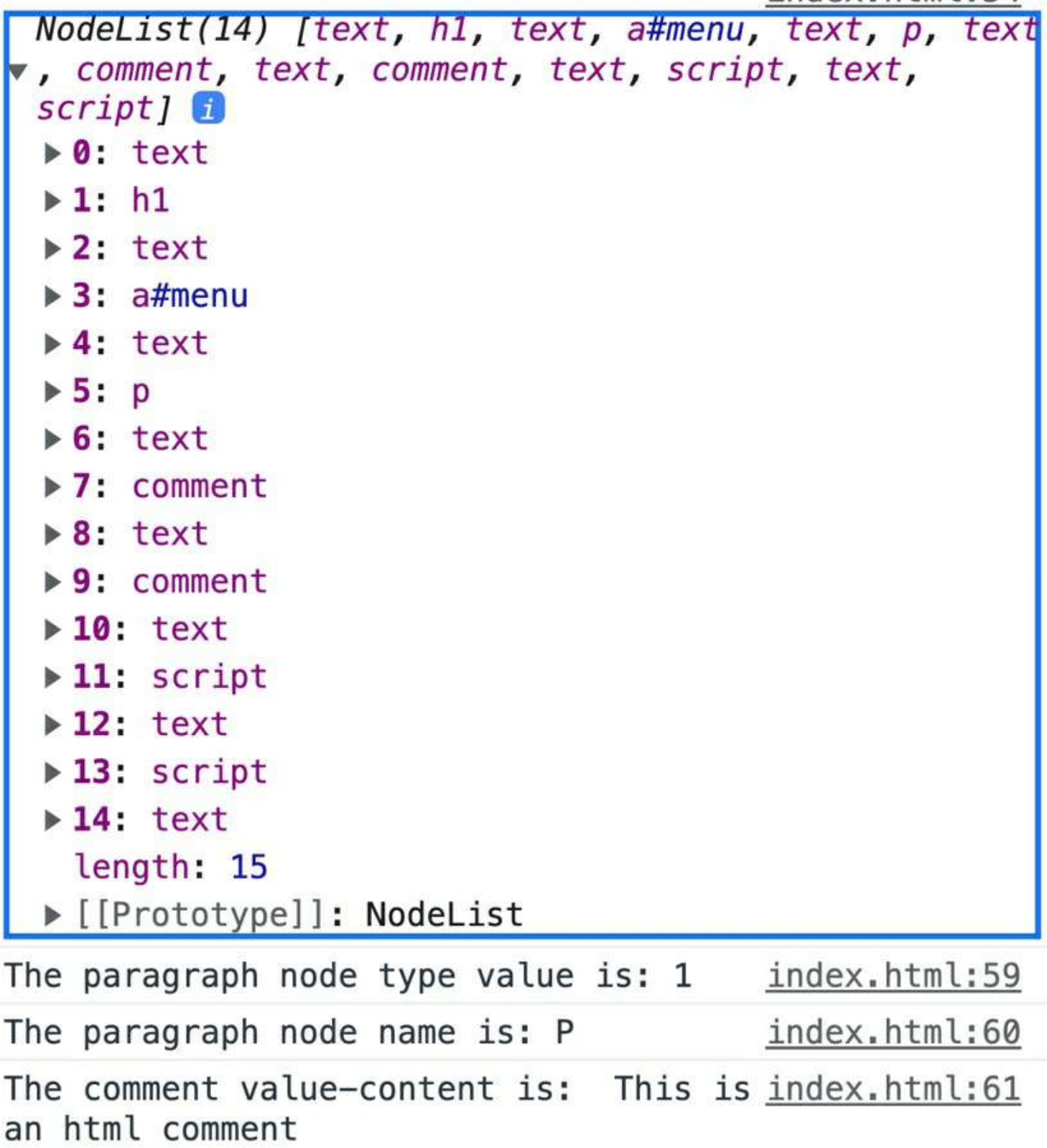
What if we want to get the name of the node or the actual value of that particular element? Well, we can use the **nodeValue** property to get the value of a text or comment node, and we can use the **nodeName** property to get the tag name of the element.
In our 'index.html' file we can add the following **HTML** markup:

```
<body>
  <h1>DOM Tree and Nodes</h1>
  <a id='menu' href='index.html'>Home</a>
  <p>This is a text</p>
  <!--This is an html comment -->
</body>
```

Imagine that now we want to get the comment value or the actual content of the HTML5 comment node, and from the paragraph <p> tag, we want to get the type and the tag name.
To achieve this, I would like to add the following JavaScript code just before the closing </body> tag.

```
<script>
 let allNodes = document.body.childNodes;
 console.log(allNodes);
 let pNode = allNodes[5];
 let pNodeType = pNode.nodeType;
 let pNodeName = pNode.nodeName;
 let commentNode = allNodes[7];
 console.log(`The paragraph node type value is: `+  pNodeType);
 console.log(`The paragraph node name is: `+ pNodeName);
 console.log(`The comment value-content is: `+ commentNode.nodeValue);
</script>
```

To know which node to select, I can get all of the **childNodes** I have in the body. We will talk about **childNodes** later in this chapter. So this will return a **nodeList**. In this example, we have 14 child nodes in our body so far. From there, I can use the node index and access the value, type, and node name. Here is the output I got from the **childNodes**:



Malformed HTML and DOM

So far, our **HTML** markup was perfect without any errors or missing the most needed elements. The **HTML** markup contains the essential elements and tags we need, and therefore, our browser can create a perfect **DOM**. But what will happen if our **HTML** file does not contain the necessary tags like head, **HTML**, or even the body tag? Then the browser will autocorrect those mistakes we have in the actual **HTML** file and try to create a perfect **DOM**. I have a piece of this code in the file called **autoCorrect**:

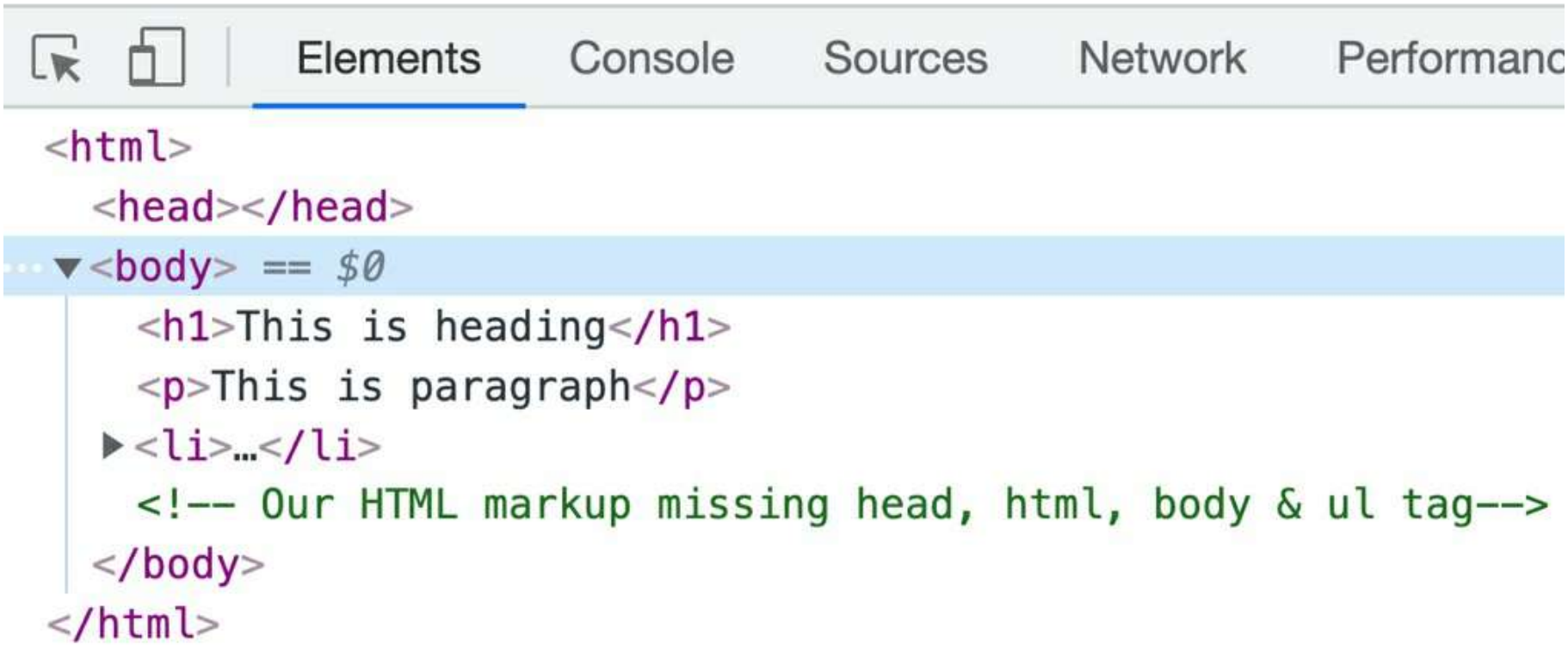
```
<h1>This is heading</h1>
<p>This is paragraph</p>
<li>This is list element</li>
<!-- Our HTML markup missing head, html, body & ul tag-->
```

From the code above, we have h1, paragraph, and comment. Each comment in the HTML file is a comment node in our DOM tree. Let us open this file in our web browser and check out the DOM.

This is heading

This is paragraph

- This is list element



From the picture above, everything looks perfect, the **DOM** contains all of the tags, self-closing tags, **HTML**, body, head, but we do not have the ul tag because the li or list element can exist even without this tag. So the browser will restore the missing parts and errors we have and generate a perfect **DOM** with all of the needed tags. So everything we have in the **HTML**, including the comments, is now added to the **DOM** tree. Even the <!**DOCTYPE**> directive that we have at the beginning of the **HTML** file is also a **DOM** node.
In the next section, we will learn more about methods to access individual **DOM** elements.

Access the DOM Elements

So far, we have learned that the **DOM** is structured as a tree of objects called nodes. The nodes can belong to a few categories, but we went over the most important ones: text, comments, and node elements. At the beginning of this chapter, we also covered some basic **HTML** and **CSS** terminology and syntax because we will use them to access the **DOM** elements. So far, we have seen how we can use the **HTML ID** with the **getElementByID**() method. I will like to point out that there are a few more methods we can use for DOM manipulation purposes:

Attribute	Selector	Method
ID	#id	getElementById()
Class	.className	getElementByClassName()
Tag	<p>	getElementByTagName()
Single Selector		querySelector()
Select All		querySelectorAll()

In order to practice these new methods, I suggest opening your computer and playing with the downloaded files, but if you are interested in reading only, I will provide the code together with the screenshots, so you can sit back, read, and enjoy the content. The entire **HTML** and **CSS** code will be located in the chapter5 folder and in the ‘index1.html and style.css’ files. Here is the entire HTML5 markup:

```
<!DOCTYPE html>
<html lang="en">

<head>
 <meta charset="utf-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
 <title>DOM - access to its elements</title>
 <link rel="preconnect" href="https://fonts.googleapis.com">
 <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
 <link href="https://fonts.googleapis.com/css2?family=Lato:wght@300&display=swap" rel="stylesheet">
 <link rel="stylesheet" href="style.css">

</head>

<body>

 <h1>Methods we use to access the DOM elements</h1>

 <h2>ID</h2>
 <div id="divID" class="redBg">Access me by id using the getElementById()</div>

 <h2>Class</h2>
 <div class="firstClass paleBg">Access me by class name firstClass (1)</div>
 <div class="firstClass paleBg">Access me by class name firstClass (2)</div>

 <h2>Tag</h2>
 <p class="spanClass tealBg"><span >Access me by the tag name-span (1)</span></p>
 <p class="spanClass tealBg"><span>Access me by the tag name-span (1)</span></p>

 <h2>Query Selector</h2>
 <div id="singleID" class="slateBg">Access me by single query selector</div>

 <h2>Query Selector All</h2>
 <div class="queryAll salmonBg">Access me by query all class name (1)</div>
 <div class="queryAll salmonBg">Access me by query all class name (2)</div>

</body>

</html>
```

If we open the same file on our web browser, it will look like this:

Methods we use to access the DOM elements

ID

Access me by id using the getElementById()

Class

Access me by class name firstClass (1)

Access me by class name firstClass (2)

Tag

Access me by the tag name-span (1)

Access me by the tag name-span (1)

Query Selector

Access me by single query selector

Query Selector All

Access me by query all class name (1)

Access me by query all class name (2)

Let us now start doing all the methods we listed inside the table, starting with the **getElementById**() method that

we already know about.

Getting Element By ID

We have already used this method, and it is one of the easiest ways to grab an element from the **DOM** by its unique **ID**. In this example. we are going to use the **ID** with value '**divID'** to select this div tag':

```
<div id="divID" class="redBg">Access me by id using the getElementById()</div>
```

In our console, we can get this element and save it into a variable called 'first':

```
> let first = document.getElementById('divID');
  console.log(first);
                                                    VM4501:2
    <div id="divID" class="redBg">Access me by id
    using the getElementById()</div>
< undefined
```

The screenshot above shows that we used the **getElementById** method that takes only one parameter: the actual element **ID** we have in the **HTML** tag. We also save whatever the method returns into a variable called first, and we console log that variable at the end. As we can see, we have successfully selected the right div element; therefore, the method we have chosen works as perfectly as intended. If you look in the **HTML** and **CSS** file, this div tag also has a class name as an attribute:

```
<div id="divID" class="redBg">Access me by id using the getElementById()</div>
```

I have used this class to add the red background to the entire div. Here is the CSS code:

```
.redBg{
   background-color: red;
}
```

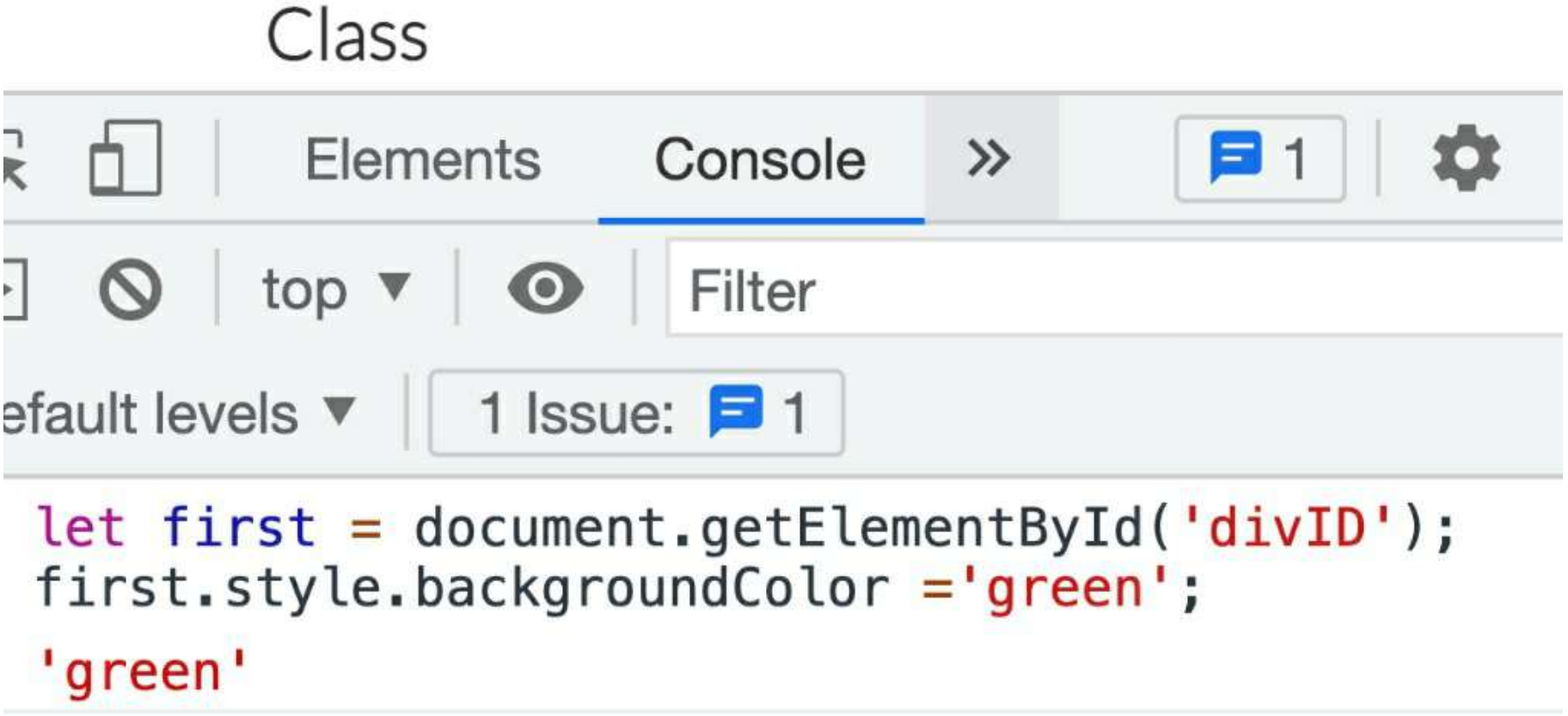
How we can change the background from our JavaScript code?
Here is the code we need to write together with the output:

Methods we use to access the DOM elements

ID

Access me by id using the getElementById()

Class



Great, we have successfully changed the background of the div tag to green, but we can also set other CSS properties to the same div tag, for example let us set its border to be yellow:

Methods we use to access the DOM elements

ID

Access me by id using the getElementById()

Class


```
let first = document.getElementById('divID');
console.log(first);
first.style.backgroundColor ='green';
first.style.border = '3px solid yellow';
```

This method is very easy to use, but it also has disadvantages. One of the disadvantages is that the element has to have a unique **ID** in the document. Another disadvantage is that with this method we can select only one element from the **DOM**. In short, the **getElementById**() is a method that will select only one element from the document, and that element needs to have a unique ID throughout the entire page. If the document does not find the element with that **ID**, it will return 'null'. Let us look at selecting one or multiple elements from our **DOM**.

Getting Elements by Class name

As you know, each element in our **HTML** file can have different attributes, and one of those attributes can be the class name. We use this class name to style that element inside the **CSS** file, but we can also use this attribute to access one or more elements from the DOM. The method we need to use is called **getElementsByClassName**(). The key here is the word 'Elements' in the method definition. It tells us that we can get multiple elements from the same page. And this is obvious because we can have many elements in our HTML file with the same class name as their attribute. In our document, I have two div tags with the same class name:

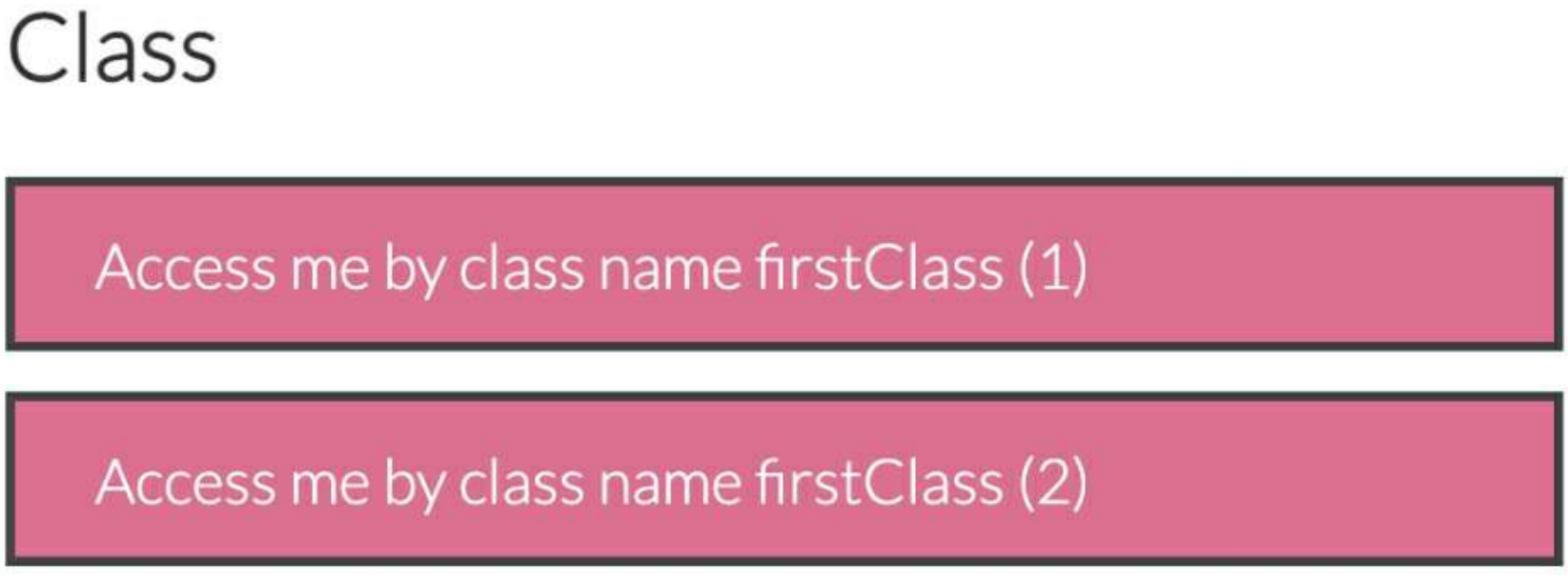
```
<h2>Class</h2>
<div class="firstClass paleBg">Access me by class name firstClass (1)
</div>


<div class="firstClass paleBg">Access me by class name firstClass (2)
</div>
```

And the CSS styles for these two div tags:

```
.paleBg{
```

```
  background-color: palevioletred;
}
```

The output before we do some manipulation



Now let us access these two elements and set a yellow border:

```
> let second =
  document.getElementsByClassName('firstClass');
  second.style.border = '3px solid yellow';
```

```
▶Uncaught TypeError: Cannot set properties VM4851:2
of undefined (setting 'border')
    at <anonymous>:2:21
```

Well, this didn't work at all. Why do we have a **TypeError** saying that we cannot set properties on undefined elements? How has this happened? Well, we can console log the variable called 'second' and see what we have selected using the **getElementsByClassName**() method:

```
> let second =
  document.getElementsByClassName('firstClass');
  second.style.border = '3px solid yellow';
```

```
▶Uncaught TypeError: Cannot set properties VM4851:2
of undefined (setting 'border')
    at <anonymous>:2:21
```

```
> console.log(second);
```

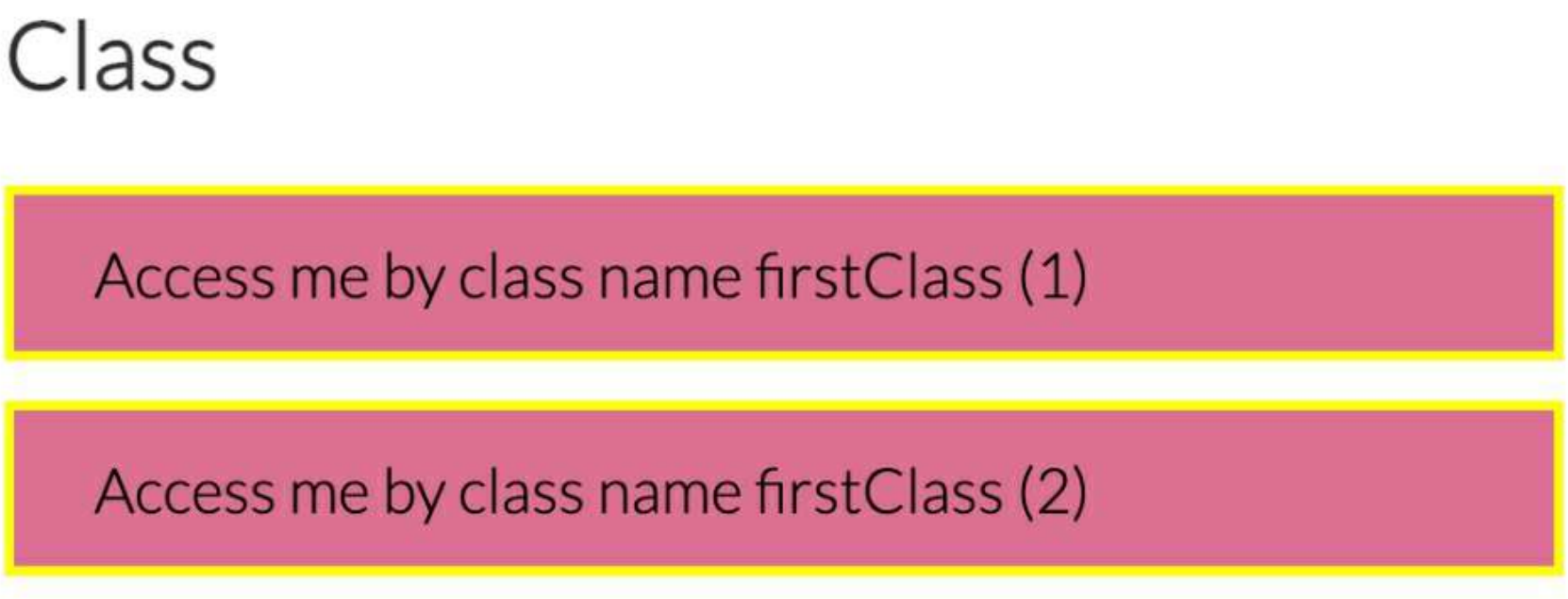
```
                                                VM4900:1
HTMLCollection(2) [div.firstClass.paleBg,
div.firstClass.paleBg] i
 ▶0: div.firstClass.paleBg
 ▶1: div.firstClass.paleBg
   length: 2
 ▶[[Prototype]]: HTMLCollection
```

Now we have the answer. This is happening because we selected multiple elements with the same class name instead of selecting only one element. So, this will return an *HTML Collection*, an array-like object of elements. So

this method returns a collection, not a single element, and that is why when we try to style the border, the method does not know which element we are targeting from the *HTML Collection*. So we can use a for loop, for example, to loop through every item in this collection and change the border to all of the elements in the array or collection.

```
let second = document.getElementsByClassName('firstClass');
//second.style.border = '3px solid yellow';//this will not work
for(item of second){
    //console.log(item);
   item.style.border = '3px solid yellow';
   item.style.color = 'black';
}
```

From the example above, we used the **for/of** loop, but any for loop will work here, and we have set the color of the text to be black, and we added the border to be 3px thick and yellow:



Getting Elements by Tag name

Same as the class method we can use the tag method to get multiple elements from same **DOM** document. This is less used method but it is still worth mentioning. In the **HTML** document we have two paragraph tags with span tag nested inside:

```
<h2>Tag</h2>
<p class="spanClass tealBg">
<span >Access me by the tag name-span (1)</span>
</p>


<p class="spanClass tealBg">
<span>Access me by the tag name-span (1)</span>
</p>
```

The **CSS** code for styling these two paragraphs is:

```
.tealBg{
   background-color: teal;
}
```

Here is the output before any DOM manipulation:

It is perfectly normal to have multiple class names inside the class attribute. This is just an example of the class attribute **'spanClass'** is not being used in the **CSS**, but I would like to show you that each tag can have multiple class names.

Let us finally select them and change the color, border, background color, and font size:

```
let third = document.getElementsByTagName('p');
  for(item of third){
     //console.log(item);
     item.style.backgroundColor = 'blue';
     item.style.border = '3px solid teal';
     item.style.color = 'white';
     item.style.fontSize = '20px';
  }
```

As you can see, we used the for loop again because, same as the class method, this one will also return an **HTMLCollection**. The tag method as a parameter accepts the name of the HTML tag we are trying to access. Here is the output:

Query Selectors

We can use two query selectors in JavaScript:

```
document.querySelector();
document.querySelectorAll();
```

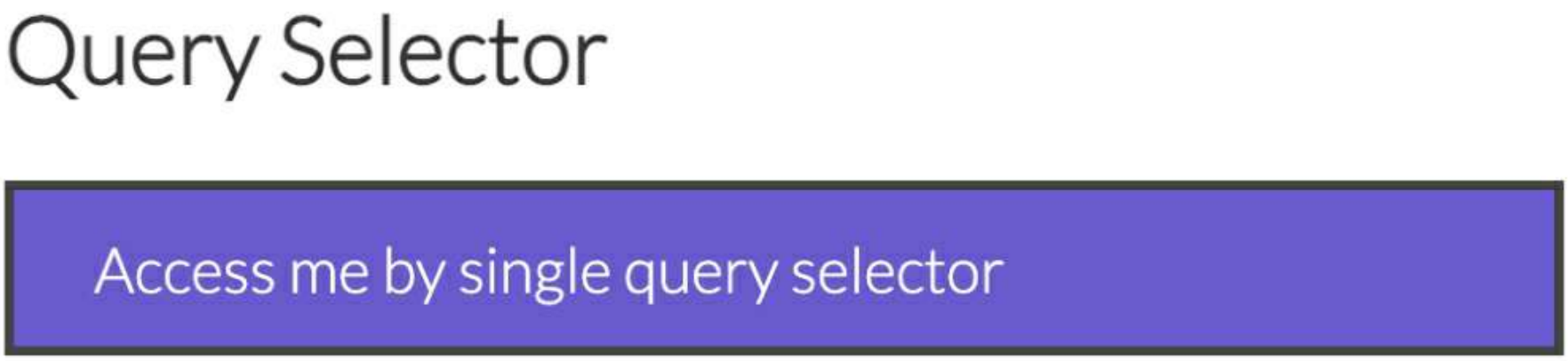
If we want to access a single element, we can use the first method. The **querySelector**() method can take **ID** attribute or class name attribute, or even a tag name. Here is the **HTML** code that we will work on:

```
<h2>Query Selector</h2>
<div id="singleID" class="slateBg">
            Access me by single query selector
</div>
```

As we can see from the example above, we have the div with an id called '**singleID'**. We also have a class **slateBg** class with the corresponding CSS code:

```
.slateBg{
   background-color: slateblue;
}
```

The current output is:



Now let us use the single querySelector where we can pass the 'singleID' as parameter:

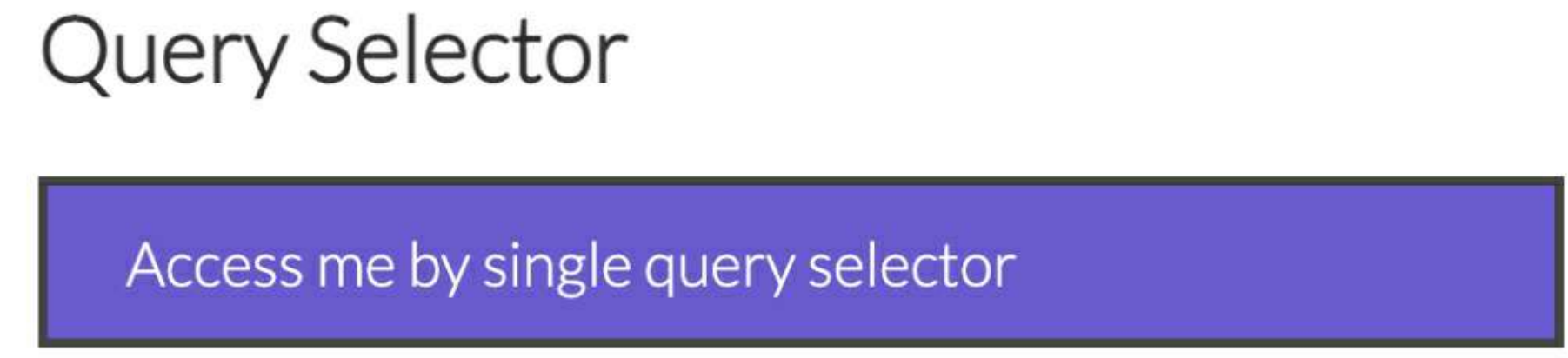
```
let fourth = document.querySelector('#singleID');
fourth.style.border = 'none';
fourth.style.color = 'white';
fourth.style.backgroundColor = '#4d0000';
```

The selector for an id attribute must have the hash '#' symbol and then the name of the id we are trying to target. We can use hexadecimal values to set different colors if we want. So we removed the border from the current div with 'singleID' attribute, changed the font color to white, and added a darker background color. Here is how it will look after the changes:

Let us do something more interesting, let us now use the class name from the same div and put back the original CSS values:

```
//Single query selector with class name
let fifth = document.querySelector('.slateBg');
fifth.style.border = '3px solid black';
fifth.style.color = 'white';
fifth.style.backgroundColor = 'slateblue';
```

From the code above, we can see that we have used the element class name, but instead of '#', we need to use dot '.' or full stop before the class name, just like this: '.slateBg'. Now this will change the element to its old **CSS** values.



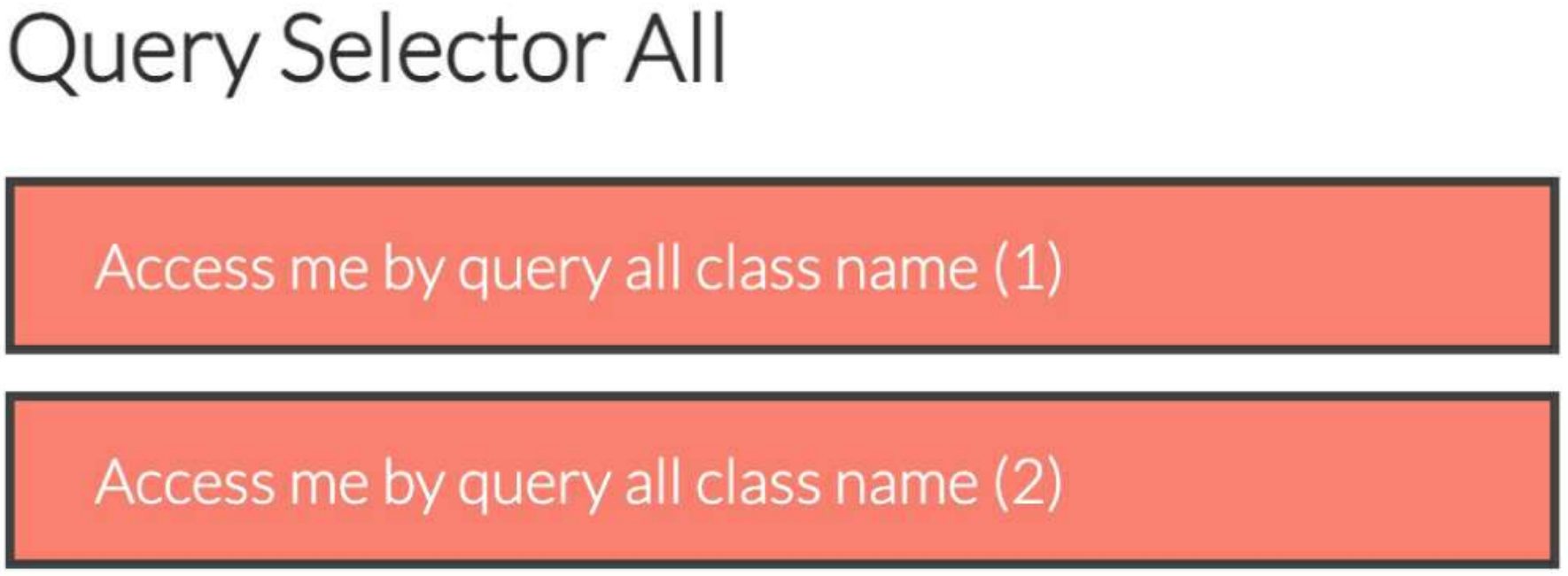
Important! The query selector is not a good choice when multiple elements have the same class name, for example. This method will look at the current **DOM** and find the first element with that class name, and it will return only that single element ignoring the rest of the elements. Therefore, if we have multiple elements in our **DOM** with the same class name and want them all, we can use the second method **querySelectorAll**(). This method will get all of

the elements matched by the query.
Ok let us select these two div tags:

```
<h2>Query Selector All</h2>
 <div class="queryAll salmonBg">
            Access me by query all class name (1)
 </div>
  <div class="queryAll salmonBg">
            Access me by query all class name (2)
</div>
```

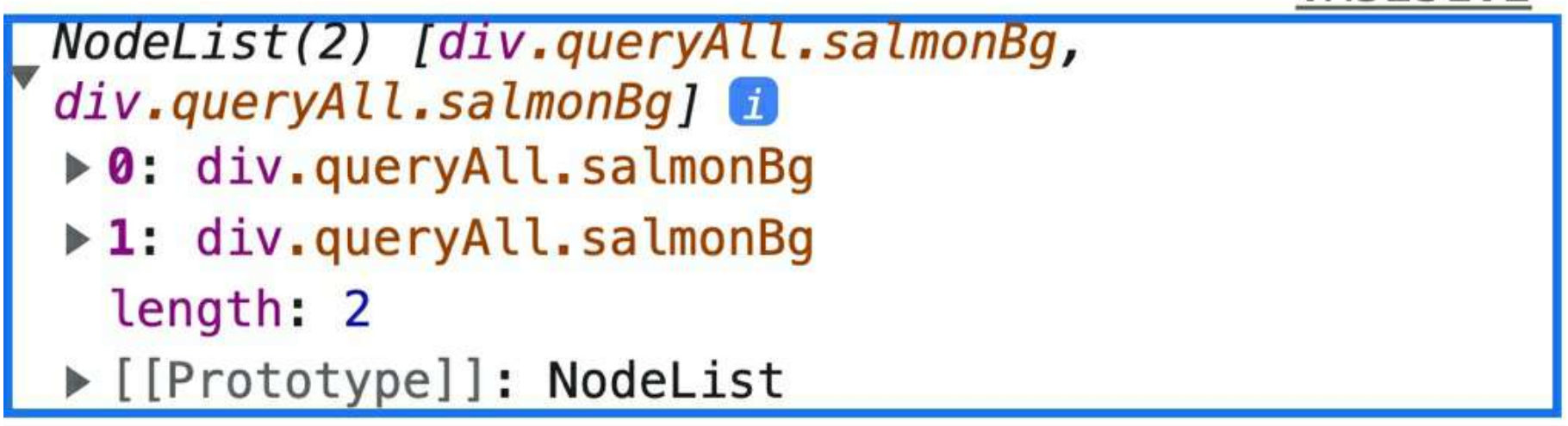
We will use one of their class names, '**queryAll'**, and as you can see, the div tags have two class names separated by a space. Here is the **CSS** style and output for these two div tags:

```
.salmonBg{
   background-color: salmon;
}
```



Let us write this code:

```
let sixth = document.querySelectorAll('.queryAll');
console.log(sixth)
```

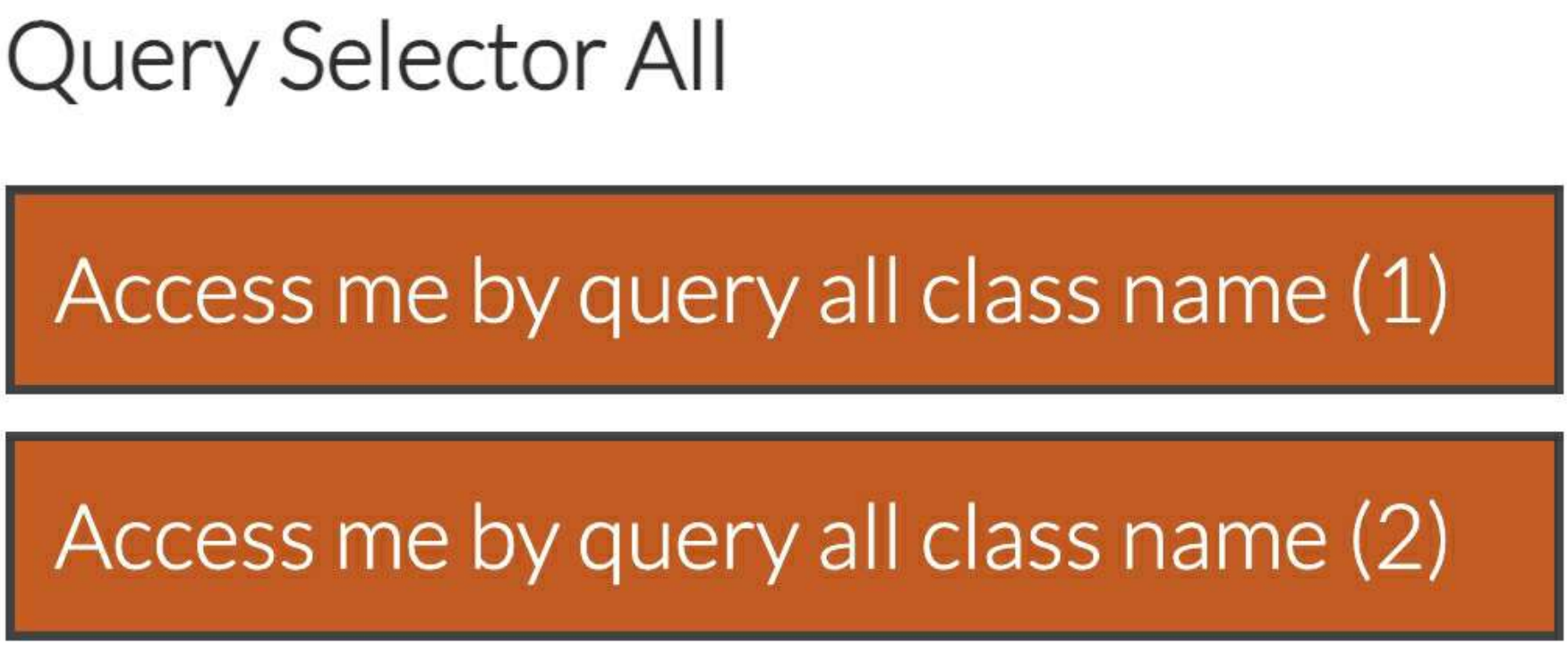


From the screenshot above, we can see that we have a **NodeList** with two-node elements inside? In the previous examples, we had **HTMLCollection**, but are they any different than the **NodeList**? Yes, before, we had **HTMLCollection**, which was a collection of **HTML** elements, and the **NodeList** is a collection of document nodes. These are both an array-like collections of objects. The difference is that the **NodeList** object can have attribute and text nodes, and these items can be accessed by their index. The **HTMLCollection** items can be accessed by their id, index number, or name, and the collection does not include text nodes. We can loop **NodeList** with the **forEach** method because it is predefined, and we can use other methods like items, entries, keys, and values.
Here is the JavaScript code:

```
let sixth = document.querySelectorAll('.queryAll');
//console.log(sixth);
sixth.forEach(item =>{
   item.style.color = 'white';
   item.style.padding = '10px';
```

```
  item.style.backgroundColor = '#D35400';
  item.style.fontSize = '23px';
})
```

Here is the output:



Another important point is that in the **querySelector**() we can have comma separated values. For example, we want to target two div tags with different classes we can do this:

```
let seventh = document.querySelector('.queryAll, .slateBg');
console.log(seventh);
```

Here the comma-separated parameter values work as the OR operator. The output is something that you might not expect:

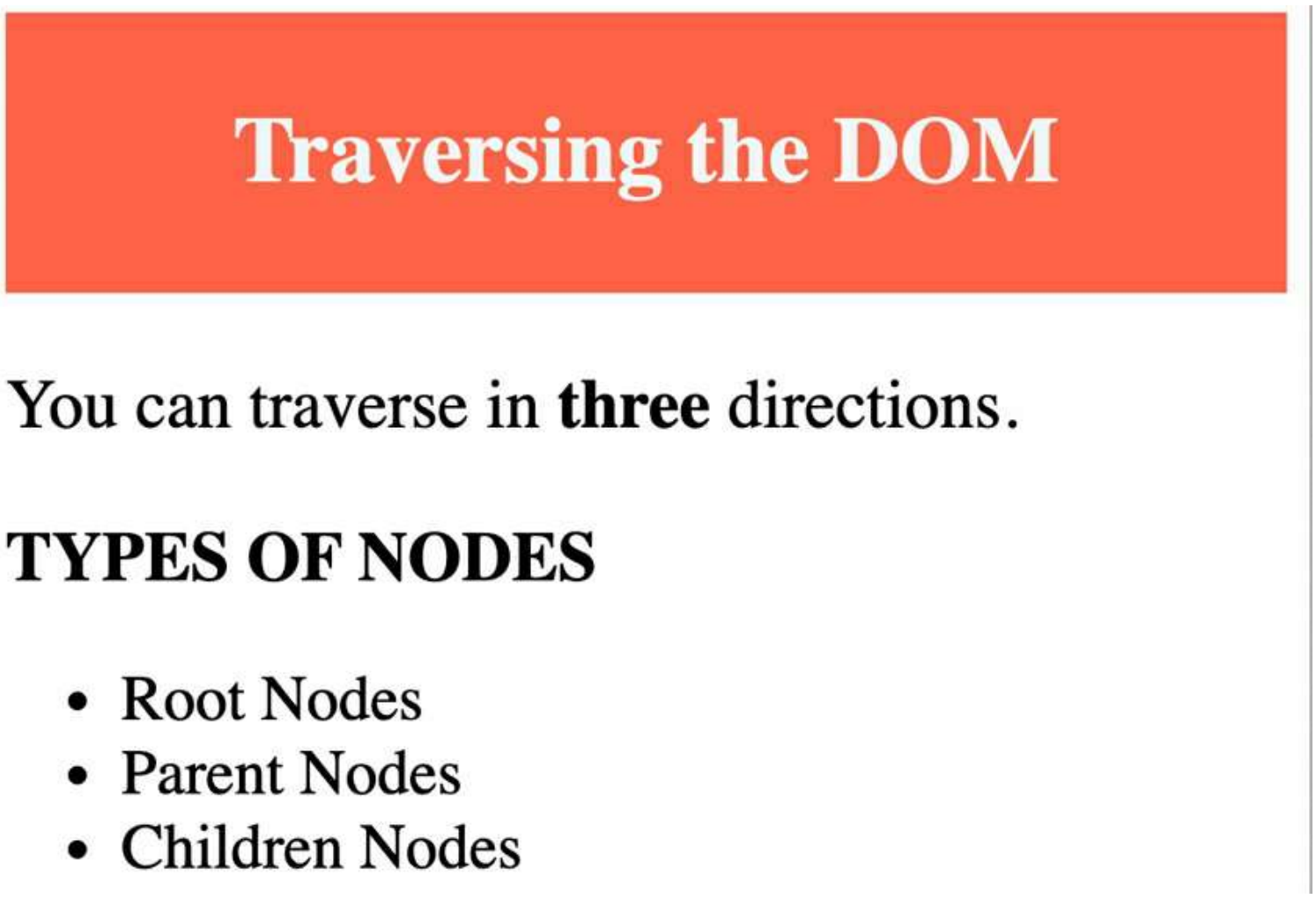
It will return the tag that appears first in the document, and in our case, that was the tag that had the class name '.slateBg'.
If we use the **querySelectorAll** with multiple commas separated attributes, then the values separated with a comma are considered as **AND** operators.

```
let eight = document.querySelectorAll('.queryAll, .slateBg');
console.log(eight);
```

The example above will match all the tags with '.slateBg' and '**queryAll'** class names and put them in a **NodeList**. And that is about these five mostly used methods to access the elements in the **DOM**. The code from the examples above will be located in the **index1** file. All you need to do is uncomment if you want to test all of them.

Traversing the DOM

So far, we have learned about the most useful in-built methods we can use to access elements within our **DOM** document. In this section, we will learn how to navigate the **DOM** tree, a tree of nodes. The code for this section will be located in the **index2** file. When we load the **index2** file in our browser, we will see:



The HTML markup:

```
<!DOCTYPE html>
<html>
<head>
  <title>Traversing the DOM</title>
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <link href="https://fonts.googleapis.com/css2?family=Lato:wght@300&display=swap" rel="stylesheet">
  <link rel="stylesheet" href="style1.css">


</head>


<body>
  <h1 class='headingDom'>Traversing the DOM</h1>
  <p class="p1">You can traverse in <strong>three</strong> directions.</p>
  <h2 class='headingNode'>Types of Nodes</h2>
  <ul class='uList'>
    <li>Root Nodes</li>
    <li>Parent Nodes</li>
    <li>Children Nodes</li>
  </ul>
</body>
</html>
```

In this example, we have an external **CSS** file called **style1**, and we are linking this file in our document head. There are some very basic **CSS** rules that I created just for this example.

Root Nodes

When we load an **HTML** file in our browser, even if it does not contain any code, the browser automatically will create three nodes. These nodes will be accessible and parsed into the **DOM** document. The document object is the most important one, which is the root of every other node we have in our **DOM**. The document object is a property of the window object, and this window object is referred to as a global object. With the window object, we can access the window's height and width and have access to its methods like alert and prompt methods.

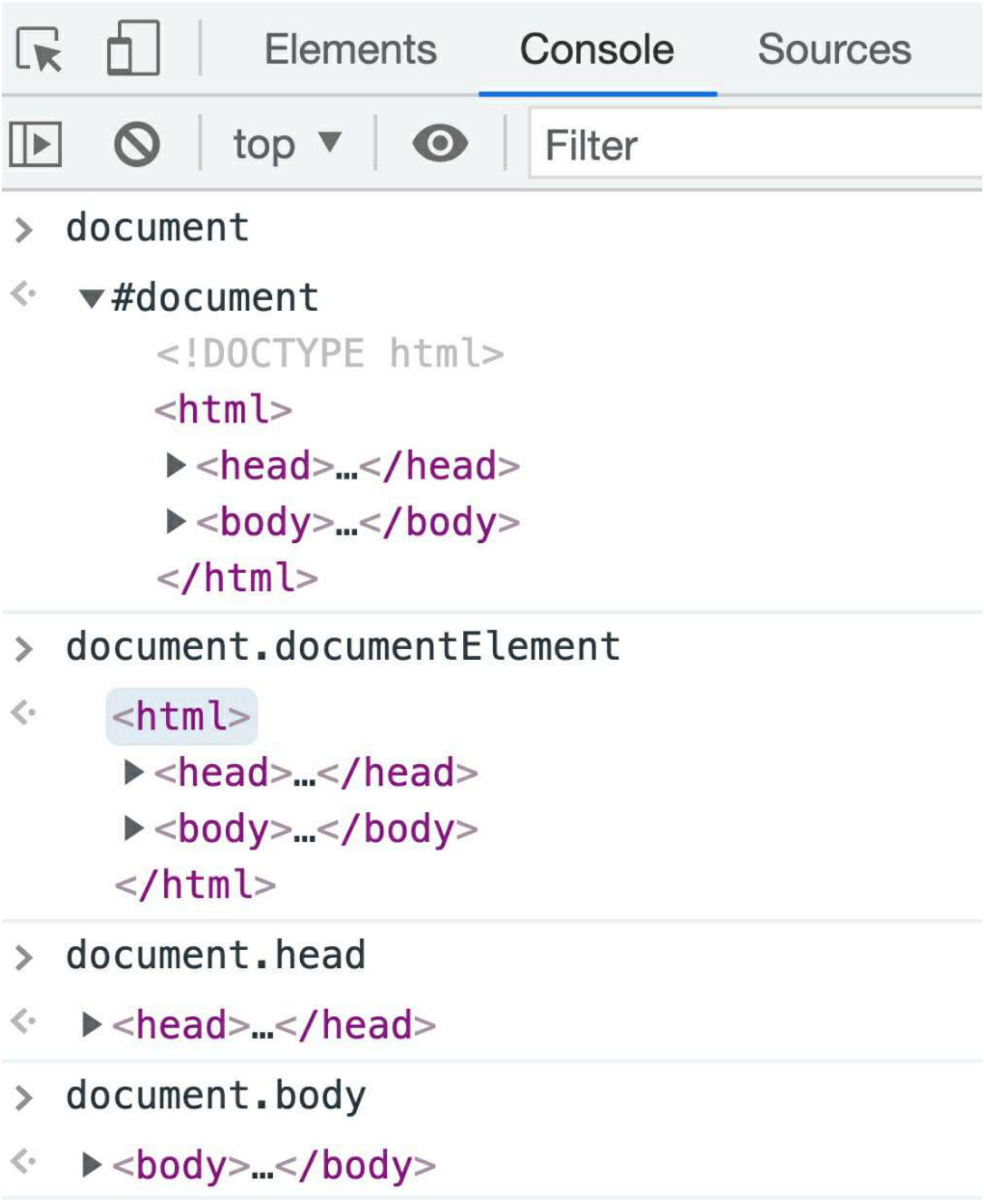
This is the list of the most important nodes that every document will have:

Property	Node	Node Type
document	#document	DOCUMENT_NODE
document.documentElement	html	ELEMENT_NODE

document.head	head	ELEMENT_NODE
document.body	body	ELEMENT_NODE

Let us type this in the Developer tools:

1. document
2. document.documentElement
3. document.head
4. document.body



We can still get the same result if we type in the console:

```
1. window.document
2. window.document.body
3. window.document.head
```

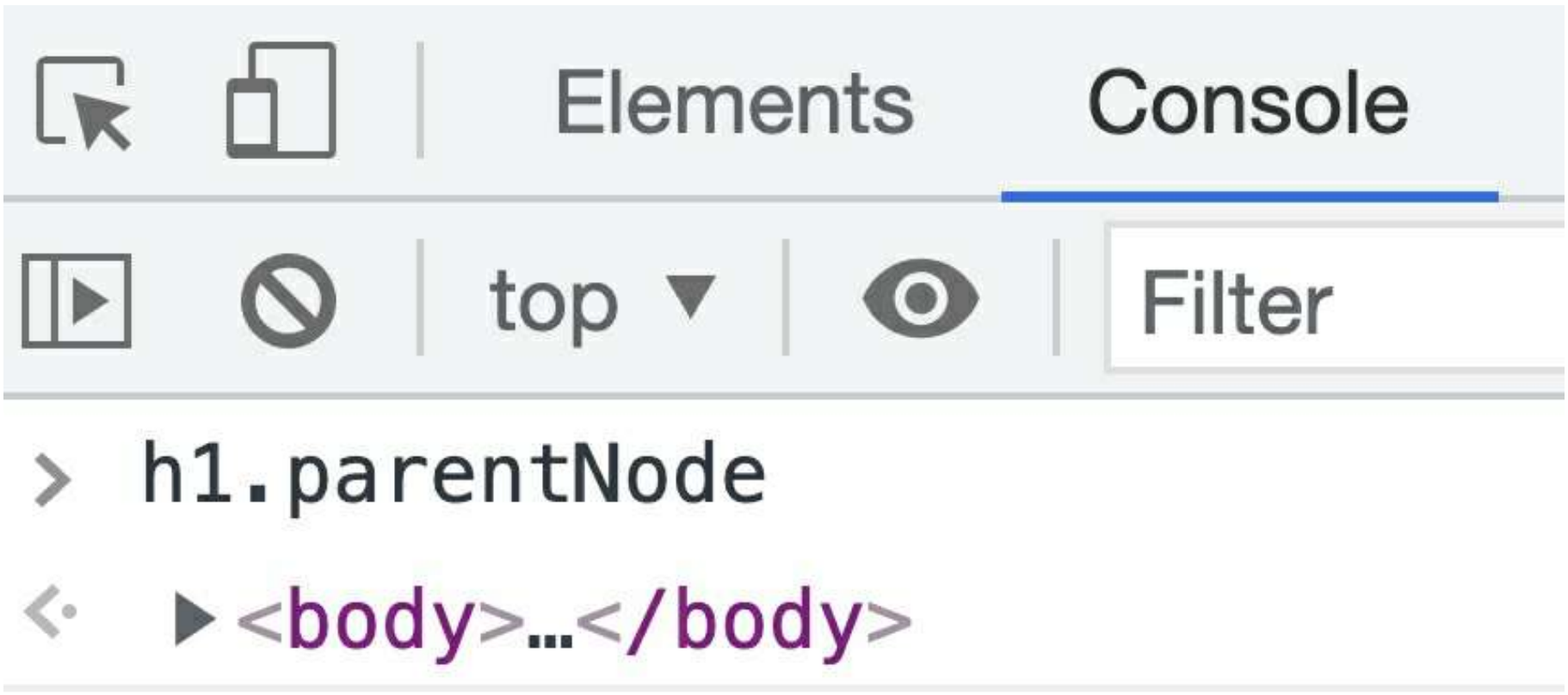
Parent Nodes

Depending on the relation to other nodes, the **DOM** nodes are referred to as:

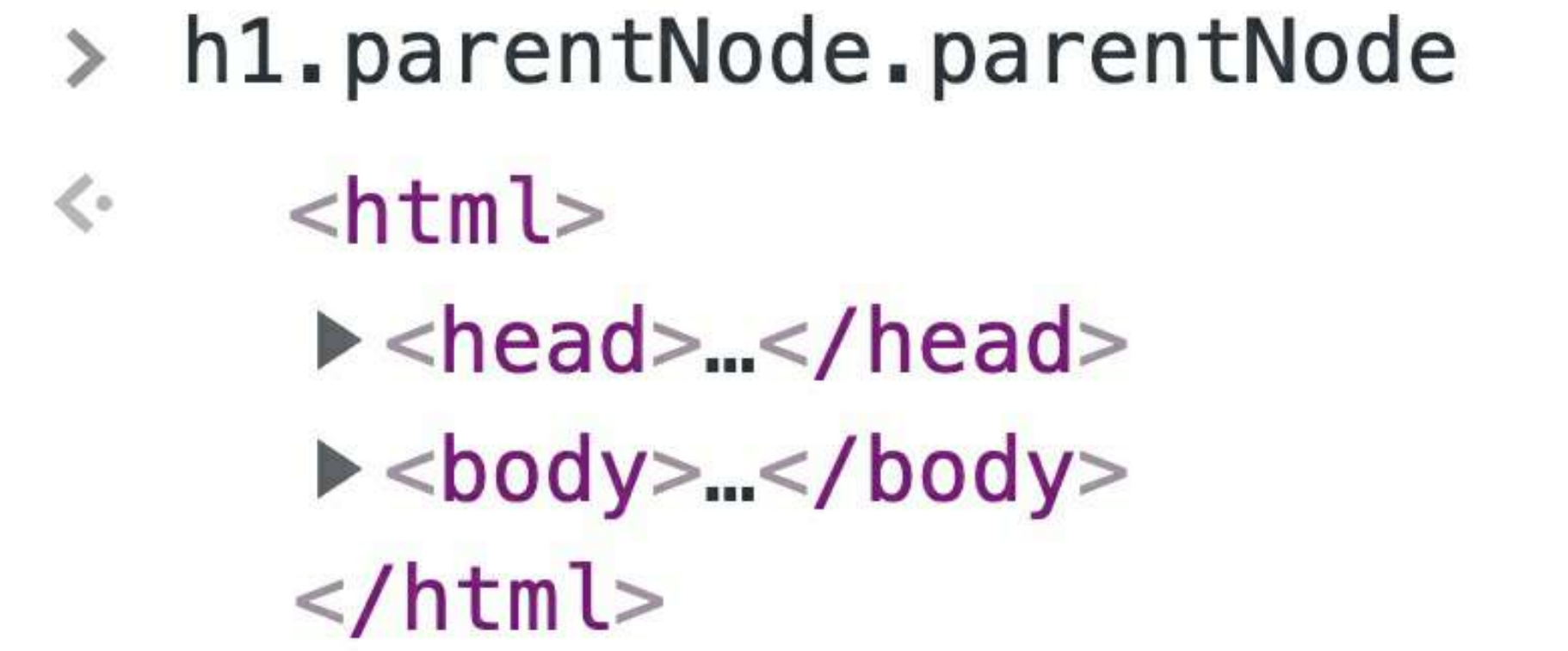
- Parent

- Children
- Siblings

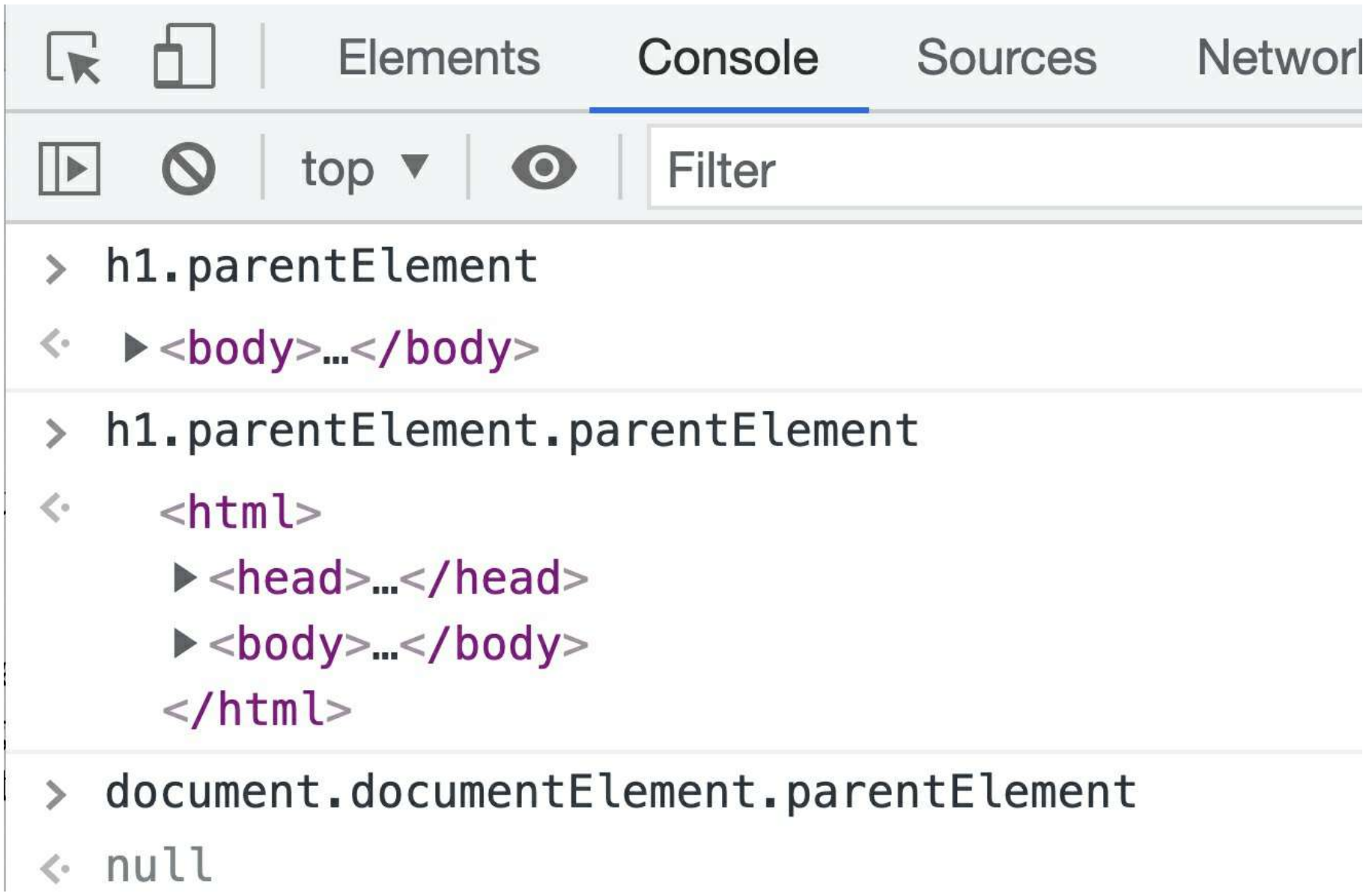
How can we recognize if some of the nodes in our **DOM** document are the parent? The parent node must be one level above the rest of the nodes, and it is always closer to the ‘document’ node in the **DOM** tree of nodes. In our example, the **HTML** node is the parent of the head and body nodes. On the other hand, the body is the parent of **h1,h2,p** and **ul**. See, here I did not mention the **li** or the list element, but why? Well, the **li** element is the child of the **ul** element, and that is two levels down from the body, so it is not an actual child. Consider that the li element is the grandchild of the body element. There is no grandchild terminology, just to be clear. So how can we test the parent of, for example, the h1 element? We can use the **parentNode** property, which will get us to the parent node.



As you can see, we got the ‘body’ as Parent Node, but how can we get the grandparent node or the parent of the ‘body’ element? Well, we can do something called property chaining:



We can use another property to get the parent nodes, and that one is called **parentElement**. So there is a very small difference between them, but I suggest using the **parentNode**, which is more commonly used in the developer community. Why can we use **parentElement** property? The parent of any node is an Element node except the **HTML**. The parent of the HTML node will be the ‘document node’, and if we do **html.parentElement** it will return null.
Here is how we can use the **parentElement** property, just remember it will not work for the last line of code:



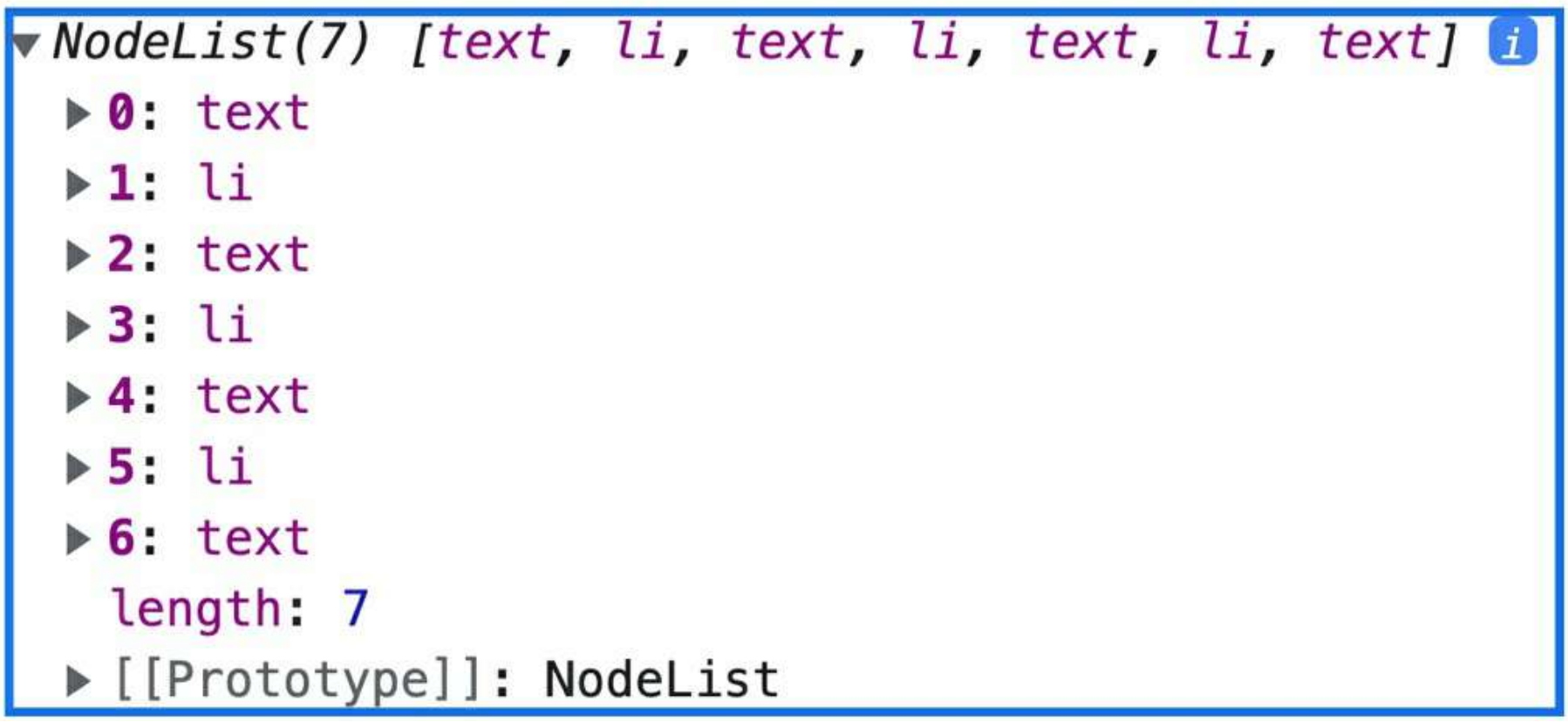
Children Nodes

As we saw in the previous example, the body is the parent of h1,h2,p, and ul nodes. These nodes are children of the body node, right? If we have nodes that are deeper than one level, then they are not referred to as children but as descendants:

Property	Returns
childNodes	Children Nodes
firstChild	The First Child Node
lastChild	Last Child Node
firstElementChild	First Child Element Node
lastElementChild	Last Child Element Node
children	All Element Child Nodes

So let us test these properties above. The first one I would like to test is the **childNodes** property. We have an unordered list in our HTML file with three list elements. This is output together with the code we need to type:

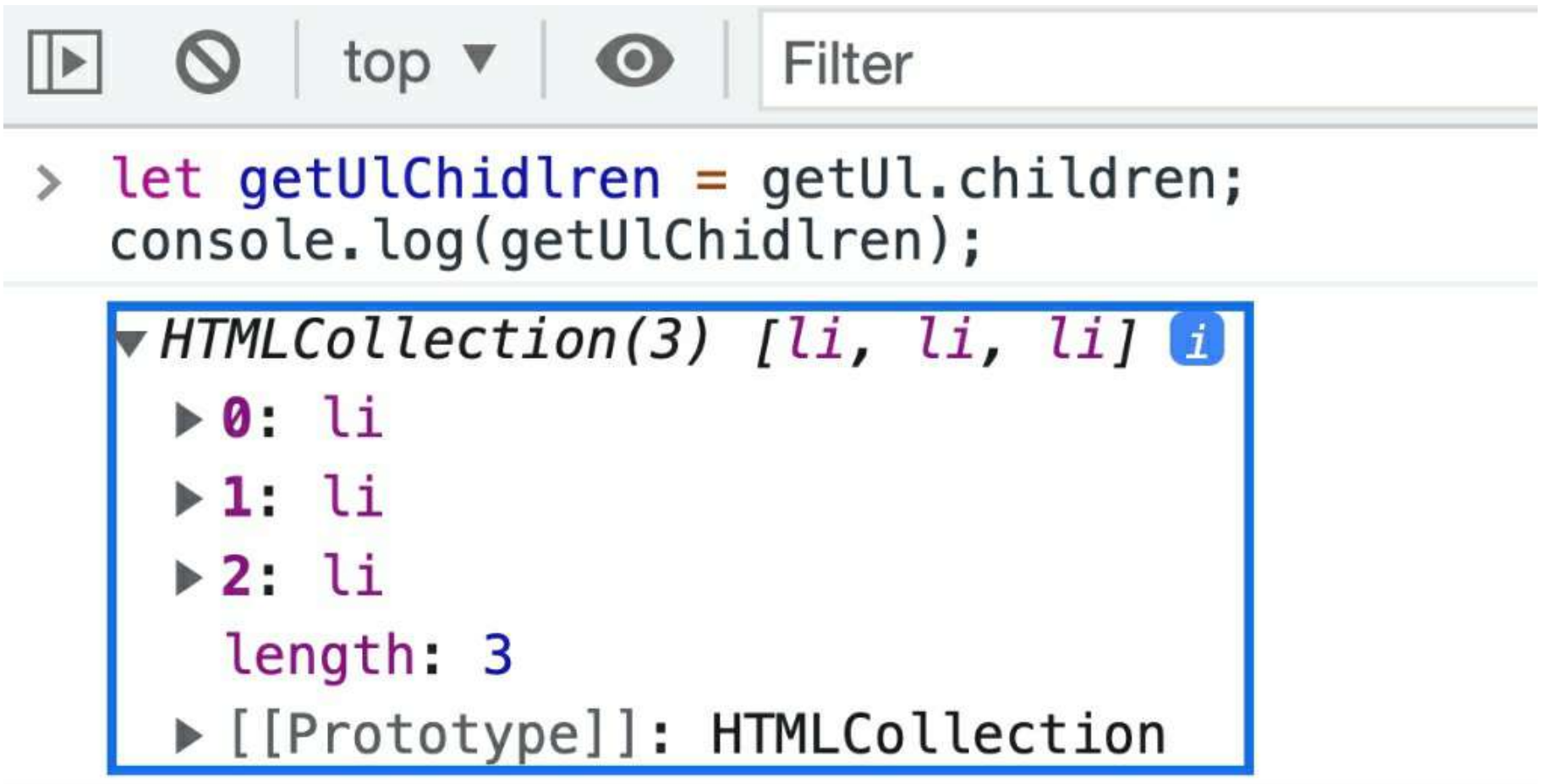
```
let getUl = document.querySelector('ul');
let getChildrenNodes = getUl.childNodes;
console.log(getChildrenNodes);
```



You have probably expected to get only **NodeList** of three children's nodes, but we got seven nodes back. How come? This is happening because the indentation between elements is also counted, and these are considered as a text node. For example if we to set styles on the first and last element then we can use the **firstChild** or **lastChild** property to set those styles, but that will not be successful because we will not get the right node to set those styles. As you can see from the example above, the firstChild and lastChild are text nodes, so styling them simply will not work:

```
getUl.firstChild.style.backgroundColor = 'green';
```

Obviously, this will throw **TypeError**. So how can we fix this? Well, we can use the children properly because it will retrieve the element nodes only, not the text nodes:



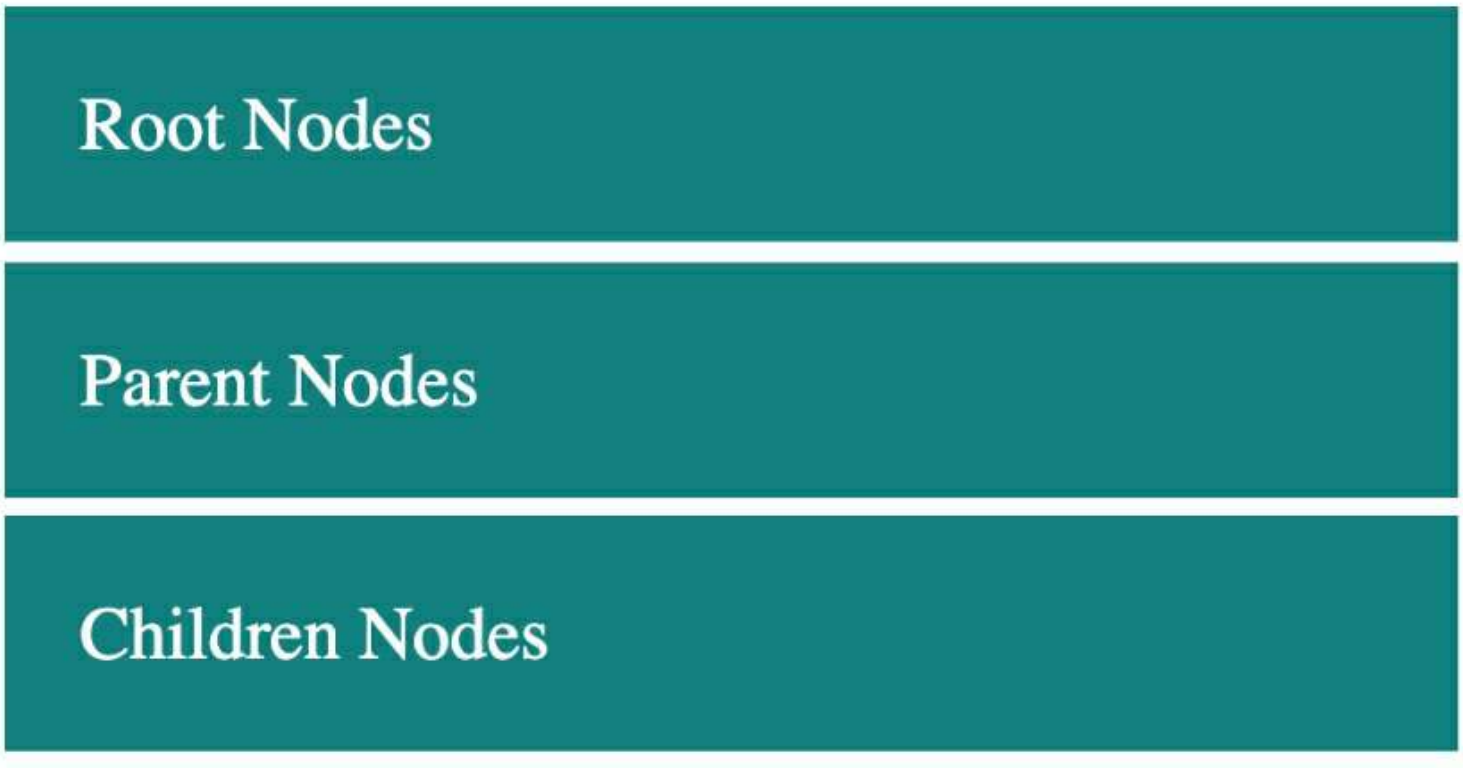
If we use **firstElementChild** we can change the background color for the first list element from the unordered list. Here is a screenshot with the background set as green:

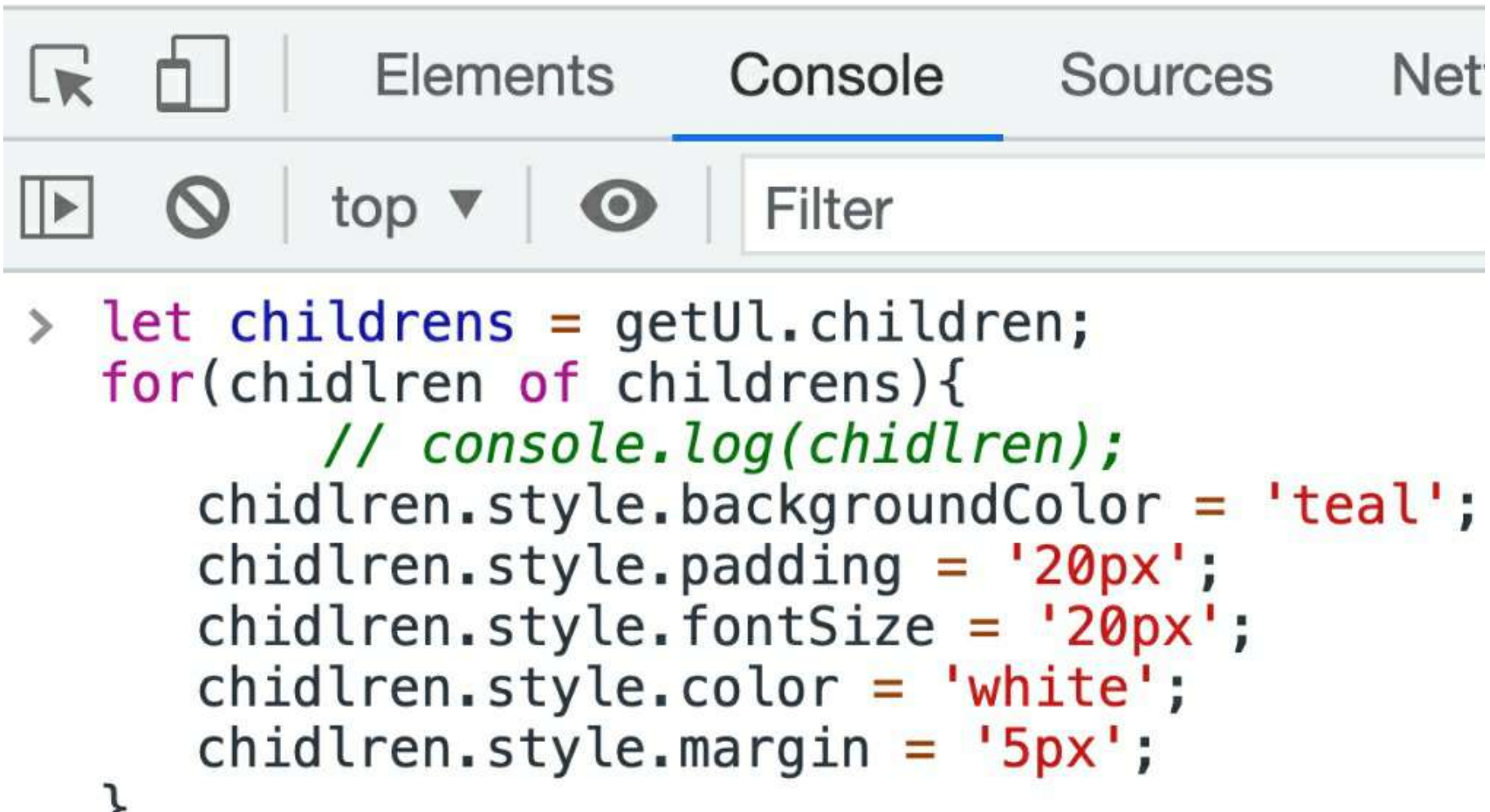
You can traverse in **three** directions.

TYPES OF NODES

- Root Nodes
- Parent Nodes
- Children Nodes

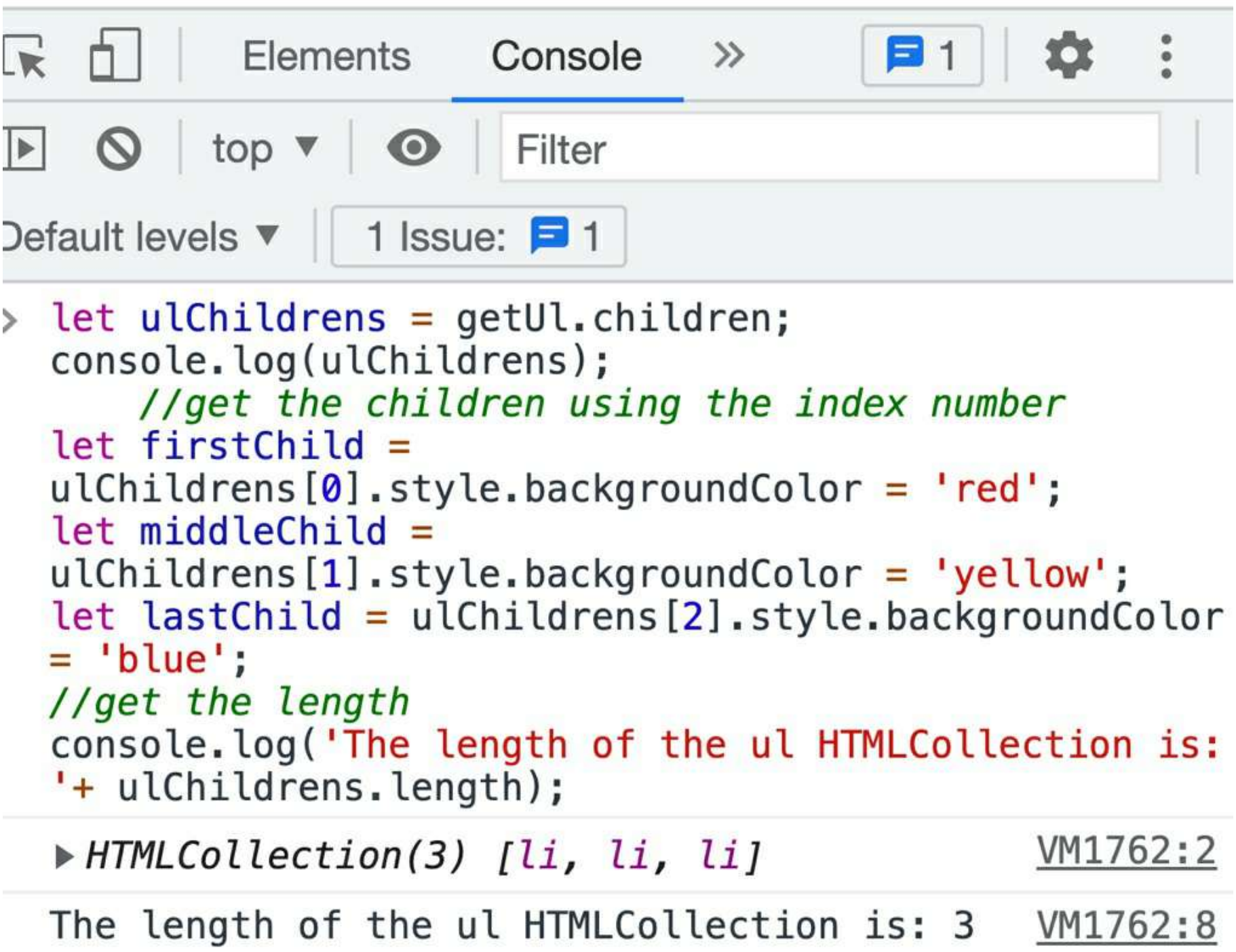
So getUl.chidlren will return an **HTMLCollection**, and we know that we can visit each element in the list with a simple for loop. For example, let us loop through this collection and set some new **CSS** styles to our list elements. Here is the whole example with the corresponding output:





So, the **childNodes** and children behave like arrays in JavaScript, and this means that we can get the collection length or use their indexes to perform further **DOM** manipulation. Please check the following example where I have used the element indexes:

- Root Nodes
- Parent Nodes
- Children Nodes



If we look closely in our HTML file, we will see that the **p** element has text and other nested elements inside. Here is the HTML markup:

```
<p class="p1">
You can traverse in <strong>three</strong> directions.
</p>
```

If we want to get everything inside the p tag including the text and the rest of the elements then we can do this:

```
> let pTag = document.querySelector('p');
  console.log(pTag);

  for(elements of pTag.childNodes){
      console.log(elements);
  }
```

```
▶ <p class="p1">…</p>
  "You can traverse in "
  <strong>three</strong>
  " directions."
```

Sibling Properties

Like the children’s properties, we have sibling properties as well, and they work similarly. When it comes to siblings of a node, this can be any node in the same level in the DOM tree. The sibling node does not have to be from the same node type. It can be a comment, text, or element as long as it is the same level in the DOM hierarchy. Here is the table for sibling nodes:

Property	Returns
nextSibling	Next Sibling Node
previousSibling	Previous Sibling Node
nextElementSibling	Next Sibling Element Node
previousElementSibling	Previous Sibling Element Node

The first two properties from the table above are to traverse the nodes only, and the last two properties are for traversing the element nodes. The first two properties will get you the nodes preceding or following the specified node. As an example, I would like to get the nodes that are before and after the heading two from our document, but before that, I will print all of the nodes that we have so we can get a clear picture:

```
let allNodes = document.body.childNodes;
  console.log(allNodes);
```

VM3422:2

```
NodeList(15) [text, h1.headingDom, text, p.p1, text
, h2.headingNode, text, ul.uList, text, comment,
text, script, text, script, text]
▶ 0: text
▶ 1: h1.headingDom
▶ 2: text
▶ 3: p.p1
▶ 4: text
▶ 5: h2.headingNode
▶ 6: text
▶ 7: ul.uList
▶ 8: text
▶ 9: comment
▶ 10: text
▶ 11: script
▶ 12: text
▶ 13: script
▶ 14: text
  length: 15
▶ [[Prototype]]: NodeList
```

Great, we have the full node list, and from the screenshot above, we can see that heading two is located as the node with index number five. Before and after, we have two text nodes so let us use the **previousSibling** and **nextSibling** and check what we will get:

```
let headingEl = document.querySelector('h2');
console.log(headingEl);
const previousSiblingH2 = headingEl.previousSibling;
console.log(previousSiblingH2);
```

VM3515:2

```
<h2 class="headingNode">Types of Nodes</h2>
```

```
▶ #text
```

VM3515:4

So, with the **querySelector**, we can select the heading two, and after that, we can check the previous sibling. In our case, the previous sibling is the text node. If we look at the complete node list screenshot, we will see that this is the case. But I do not want to get a text node; instead, I need to get the paragraph node. How can we do this? Well, we

can chain multiple **previousSibling** properties so we can get the desired node we want:

```
let headingEl = document.querySelector('h2');
const previousSiblingH2 =
headingEl.previousSibling.previousSibling;
console.log(previousSiblingH2);
```

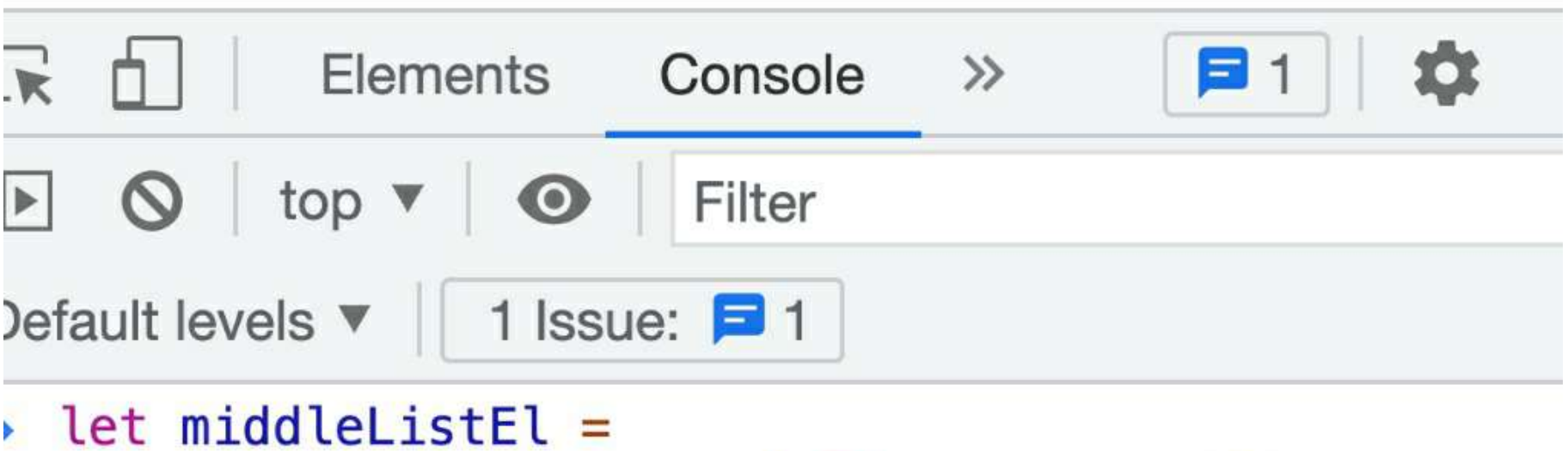
```
▶<p class="p1">…</p>
```

The process of getting the next sibling node of heading two is same as in the previous example, just we are going in a different direction, and here is the code:

```
const nextSiblingH2 = headingEl.nextSibling;
console.log(nextSiblingH2);
```

As you can see, with the next and previous siblings, we most likely will get a text node because there are white spaces in our DOM document and as you know, they are considered text nodes. So to avoid this and to avoid the confusion of using the chaining method of multiple properties, we can simply use the next and previous Element sibling property to get the desired node. As an example, let us get the middle element of the unordered list and style the previous and next element node with different background colors:

- Root Nodes
- Parent Nodes
- Children Nodes




```
let middleListEl =
document.querySelector('ul').children[1];
let middleListElRed =
middleListEl.style.backgroundColor = 'red';
let previousListEl =
middleListEl.previousElementSibling;
let previousListElYellow =
previousListEl.style.backgroundColor = 'yellow';
let nextListEl = middleListEl.nextElementSibling;
let nextListElGreen =
nextListEl.style.backgroundColor = 'green';
```

Directions of Traversing

We have covered how we can traverse the DOM, but we did not mention anything about the directions of traversing. This section will not give you new content, but because we already know how to traverse the DOM, it will be very easy for me to explain the directions of traversing as well. So, sit back and enjoy this section because it will cover the properties that we have already discussed. We can traverse the DOM in these three directions:

- Downward
- Sideway
- Upward

When traversing the document downward from a specific element, we can use one of the two **CSS** selectors, the **querySelector** or **querySelectorAll**. We can also use the children, which will give us the option to select the direct descendants. As you know, the children will produce an HTML Collection in which we can use the element indexes to select the element we want. For this section, I have added a new file called **index3,** but I will provide enough screenshots and code that you can simply just continue reading.
Here is the entire HTML code:

```
<!DOCTYPE html>
<html>

<head>
  <title>Traversing the DOM1</title>
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <link href="https://fonts.googleapis.com/css2?family=Lato:wght@300&display=swap" rel="stylesheet">
</head>

<body>
  <h1 class='headingDom'>Ways to Traverse the DOM</h1>
  <p class="p1">You can traverse in <strong>three</strong> directions.</p>
  <h2 class='headingNode'>Types of Nodes</h2>
  <ul class='menu'>
    <li>Downwards</li>
    <li>Sideways</li>
    <li>Upwards</li>
  </ul>
  <div class = 'gallery'>
      <h2 class="gallery__title">Title 1</h2>
      <h2 class="gallery__description">Description goes here!</h2>
  </div>
  <ul class="menu1">
    <li class='link'>
                <a href="index.html">Home</a>
    </li>
    <li class='link'>
       <a href="about.html">About Us</a>
    </li>
    <li class='link'>
       <a href="contact.html">Contact Us</a>
    </li>
    <li class='link'>
       <a href="jobs.html">Work for Us</a>
    </li>
    <li class='link'>
       <a href="portfolio.html">Our Projects</a>
    </li>
  </ul>
</body>
<script>


</script>
</html>
```

Here is how it looks if we open this file:

Ways to Traverse the DOM

You can traverse in **three** directions.

Types of Nodes

- Downwards
- Sideways
- Upwards

Title 1

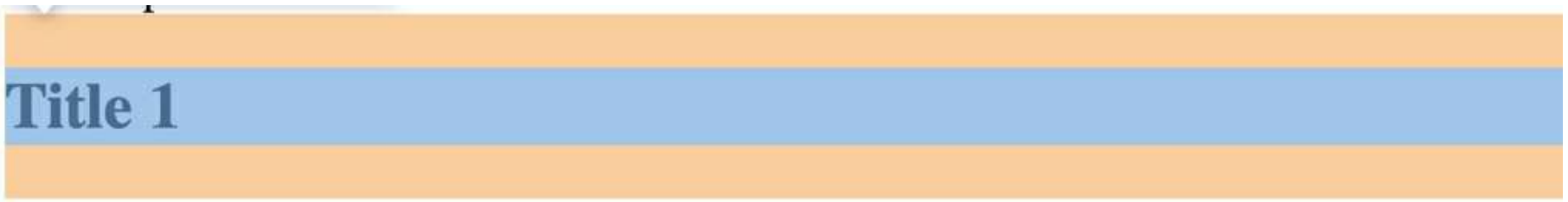
Description goes here!

- Home
- About Us
- Contact Us
- Work for Us
- Our Projects

So to traverse downward from a specific elements in our document, we can use the **querySelector** or **querySelectorAll** methods. Let us focus on this markup here from the example above:

```
<div class = 'gallery'>
  <h2 class="gallery__title">Title 1</h2>
  <h2 class="gallery__description">Description goes here!</h2>
</div>
```

The main div tag with class ‘gallery’ has two elements inside. Both of them are headings. The div tag acts as a container, and inside we do have two separate compartments, and we need to select only one compartment, for example, the heading with class **‘gallery__title’**. Instead of wasting time and performing a search on the entire document top to bottom, we can simply select the container we want and search what is inside this container. This will give us the quicker and more efficient result:



Description goes here!

- Home
- About Us
- Contact Us
- Work for Us
- Our Projects


```
> const galleryContainer =
  document.querySelector('.gallery');
  const galleryContainerTitle =
  galleryContainer.querySelector('.gallery__title')
  console.log(galleryContainerTitle);
```

```
<h2 class="gallery__title">Title 1</h2>   VM5759:3
```

The children property is also very efficient when we traverse the DOM downwards because, as we mentioned, it will give us the children or the direct descendants. It will return an HMTL Collection, an array-like object with indexes so we can loop over using the famous **forEach** loop.
Here is one example where we will get the menu list elements from our document:

```
const menuList = document.querySelector('.menu');
const menuItems = menuList.children;
co  length le.log(menuItems);
```

```
▼HTMLCollection(3) [li, li, li] i   VM5793:3
  ▶0: li
  ▶1: li
  ▶2: li
    length: 3
  ▶[[Prototype]]: HTMLCollection
```

We can loop over this collection easily but first we need to convert it to an array using the **Array.from** method, and then we can use the **forEach** loop on the array:

```
const menuList = document.querySelector('.menu');
const menuItems = menuList.children;
const items = Array.from(menuItems);
items.forEach(element => {
    console.log(element);
});
```

```
▶ <li>…</li>                                    VM585
▶ <li>…</li>                                    VM585
▶ <li>…</li>                                    VM585
```

Select a specific child

From the children property, we are getting the HTMLCollection, and from **querySelectorAll**, we are getting **NodeList**. From both of these collections, we can select the **nth-item** using the index of the element, this is exactly the same when we select an elements from the arrays. Here is one more example for traversing the DOM downwards, selecting the children from HTML Collections and Node Lists:

```
//nth-child from HTML Collections
const menuList = document.querySelector('.menu');
const menuItems = menuList.children;
const firstHTMLCollectionItem = menuItems[0];
const secondHTMLCollectionItem = menuItems[1];
const thirdHTMLCollectionItem = menuItems[2];
console.log('HTML Collection nth-items')
console.log(firstHTMLCollectionItem);
console.log(secondHTMLCollectionItem);
console.log(thirdHTMLCollectionItem);
//nth-child from NodeList
const links = document.querySelectorAll('.link');
const firstNodeListItem = links[0];
const secondNodeListItem = links[1];
const thirdNodeListItem = links[2];
const fourthNodeListItem = links[3];
const fifthNodeListItem = links[4];
console.log('NodeList nth-items')
console.log(firstNodeListItem);
console.log(secondNodeListItem);
console.log(thirdNodeListItem);
console.log(fourthNodeListItem);
console.log(fifthNodeListItem);
```

Output:

Traversing DOM Upwards

We can traverse the DOM upwards using the two methods:

- parentElement
- closest

The parent element will let us select the parent element. For example, the parent for gallery title will be the div container with class ‘gallery’.
Here is the markup again:

```
<div class = 'gallery'>
  <h2 class="gallery__title">Title 1</h2>
  <h2 class="gallery__description">Description    goes here!
  </h2>
</div>
```

And the code:

```
const galleryTitle = document.querySelector('.gallery__description')
const parentEl = galleryTitle.parentElement
console.log(parentEl);//<div class="gallery"> ... </div>
```

We have used the parent element to select elements that are one level upward, right? But what if we need to find an element that is multiple levels above from the current element. For this we can use the closest method:

```
const aHref = document.querySelector('a');
const theClosest = aHref.closest('.menu1');
console.log(theClosest); // <ul class="menu1"> ... </ul>
```

So we can select the ‘.menu1’ from the a href tag with the closest method. The closest method will start searching from the current element, and then it will go upwards until it reaches the document object, then it will stop and return the first element that will find. The closest method is still not supported by all of the browsers, but by the time you are reading this it might be.

Traversing the DOM Sideways

We can traverse the **DOM** sideways using one of these methods:

- previosElementSibling
- nextElementSibling
- Combination of (parentElement, children, and index)

I’m not going to write an example for previous and next sibling nodes because we have already covered one, but let us combine the parent, children, and index.
As an example, let us select the third link from our second unordered. Here is the markup:

```
<ul class="menu1">
   <li class='link'>
<a href="index.html">Home</a>
</li>
   <li class='link'>
<a href="about.html">About Us</a>
</li>
   <li class='link'>
<a href="contact.html">Contact Us</a>
</li>
   <li class='link'>
<a href="jobs.html">Work for Us</a>
</li>
   <li class='link'>
<a href="portfolio.html">Our Projects</a>
</li>
  </ul>
```

The third link will be the list element that contains the text ‘**Work for Us**’. So, the first thing we need to do is to grab the first element from the unordered list using the query selector:

```
const firstURLLink = document.querySelector('.link');
```

The output will be:

```
▶<li class="link">…</li>
```

Then the next step will be to get the parent element for the <li> elements:

```
const parent = firstURLLink.parentElement;
```

```
The Parent Element:
▶<ul class="menu1">…</ul>
```

This will return the parent of <li> which is the <ul> element. Now we can need to get the HTML Collection of all of the list <li> elements:

```
const allLinks = parent.children;
```

```
                                                    index3.htm
▶ HTMLCollection(5) [li.link, li.link, li.link,
  li.link, li.link]
```

And finally, we can use the index to get the third list element:

```
const thirdLink = allLinks[3];
console.log(thirdLink);
```

Output:

Great now we know how we can use combination of parent element, children and indexes to get the specific element. The steps above can be shortened in just two lines of code:

```
const firstURLLink = document.querySelector('.link');
const theThirdLink = firstURLLink.parentElement.children[3];
```

Creating, Inserting, and Removing Nodes from DOM

So far, we know how to get specific elements from our DOM document. We have also learned how we can traverse the DOM using different properties. We now know how important the ids, tags, selectors, and classes are because we can easily locate any node we want. We can also use different properties to find the previous or next nodes in the same document. But there are more things we can do with our DOM document. We can add, remove, replace or edit nodes as well. Same as in my online courses, I will use a simple to-do list application to learn how to create, add, replace and remove elements from the DOM. The code will be in the file called **index4** located in the **chapter5** folder.

Creating New DOM Nodes

If we open the index4 file in our VsCode editor, this is the markup we have:

```
<!DOCTYPE html>
<html>

<head>
  <title>Traversing the DOM1</title>
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" cro
  <link href="https://fonts.googleapis.com/css2?family=Lato:w
</head>
<body>
        <ul >
            <li>Call Mom</li>
            <li>Finish Assignment </li>
            <li>Visit my Grandparents</li>
            <li>Write New Article</li>
            <li>Read a book</li>
            <li>Check Comments</li>
        </ul>
</body>
<script>
</script>
</html>
```

This is a static webpage because we wrote the code directly into our **HTML** file, but if we want to make our page dynamic, we can use JavaScript and its methods to create new nodes in the **DOM**. As you can see, we do not have any headings in our current file, so let us create one element just for the heading. We can use the **createElement()** method on the document object and create a new H1 element.
In between the script tags we can add this code:

```
const heading1 = document.createElement('h1');
```

If you open this file in your browser, nothing will show because firstly, the heading element is not included in our DOM yet, and secondly, it is an empty heading element without content or text. So let us add the content using the next method called **textContent()**.

```
heading1.textContent = 'Create New Elements and Add New Text Content';
```

As you can see, we have combined **createElement** and **textContent** to create a new element node and give the same element content. I want to point out that we can add new content to an element even if we use the **innerHTML** property, which will allow us to add text to an element and add **HTML** code. It is important to note that **innerHTML** can have cross-site scription **XSS** issues because we are adding new **HTML** directly from our JavaScript script. Therefore, it is recommended to use the **textContent** property instead of **innerHTML**.
Here is one example for the **innerHTML**:

```
const heading2 = document.createElement('h2');
heading2.innerHTML = 'New Heading 2 <i> Element </i>';
```

As you can see, we can add text to the new element and add **HTML** code like the italic tag. There is another way to create a text node, which is by using the **createTextNode**() method.

```
const text = document.createTextNode("I'm a text node.");
```

Insert Created Nodes Into the DOM

If you open the same file in your browser, you will not see any changes because we created new nodes with some content but never inserted it into our DOM. To see these newly created nodes, we need to use one of these three methods:

- node.appendChild()
- node.insertBefore()
- node.replaceChild()

The appendChild() method will add the node as the last child of the parent element. Insert before method inserts the created node before the specified sibling node. The last method, 'replaceChild', will replace the existing node with a new one.
So let us add the two new headings we created, but first I want to add the heading1 as a last child of the document body and after that to add the heading number two before the heading1 using the **insertBefore**() method:

- Call Mom
- Finish Assignment
- Visit my Grandparents
- Write New Article
- Read a book
- Check Comments

New Heading 2 *Element*

Create New Elements and Add New Text Content


```
> const heading1 = document.createElement('h1');
  heading1.textContent = 'Create New Elements and Add New Text
  const heading2 = document.createElement('h2');
  heading2.innerHTML = 'New Heading 2 <i> Element </i>';
  const text = document.createTextNode("I'm a text node.");
  document.body.appendChild(heading1);
  document.body.insertBefore(heading2,heading1);
```

As you can see, we have heading two before heading one in the screenshot above. The only method we need to test is the replace method, so as an exercise, please try this on your own, but if you can, I will include the code you need. Look at the unordered list that we have in the document, from there find the list element that has this content: 'Read a book', and finally, using the replace method, replace the same element with a new list element with this new content: 'Read a Sci-Fi book'. If you cannot do it, then please look at the following code:

```
//get the unordered list
const ul = document.querySelector('ul');
//get all the li tags
const listElements = document.querySelectorAll('li');
//we need the 'Read a book' list element the index is 4
const fifthElement = listElements[4];
//create new element
const newFifthElement = document.createElement('li');
//add new content to the new element
newFifthElement.textContent = 'Read a Sci-Fi Book';
//replace the fifth child with the new one and append it to the ul tag
```

```
ul.replaceChild(newFifthElement,fifthElement);
```

Modify DOM Classes, Styles, and Attributes

This section will cover how we can do further document manipulations using the classes, attributes, and styles. The classes are the **CSS** classes we have used so far, and for these classes, we are using the **CSS** selectors to select them. Please note we are not talking about **ES6** Classes. They are completely different. As you know, at the beginning of this chapter, we have used **IDs** and classes to style the **HTML** elements, but we also learned that we could use the same class names for multiple elements. The **IDs** are a bit different, and we need to have unique **IDs** in the document. In JavaScript, we have two properties that can work with the class attributes. These are called the **className** and **classList** properties. Here is a table of these properties:

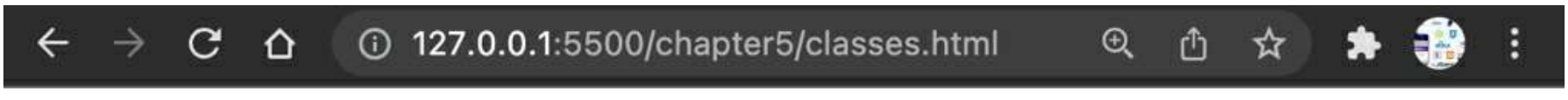
Property	Description
className	Gets the class name/value
classList.add()	Adds one or more class values/names
classList.remove()	Remove class name/value
classList.toggle()	Toggles a class name on or off
classList.contains()	Check if a class name/value exists
classList.replace()	Replace class name with new class name

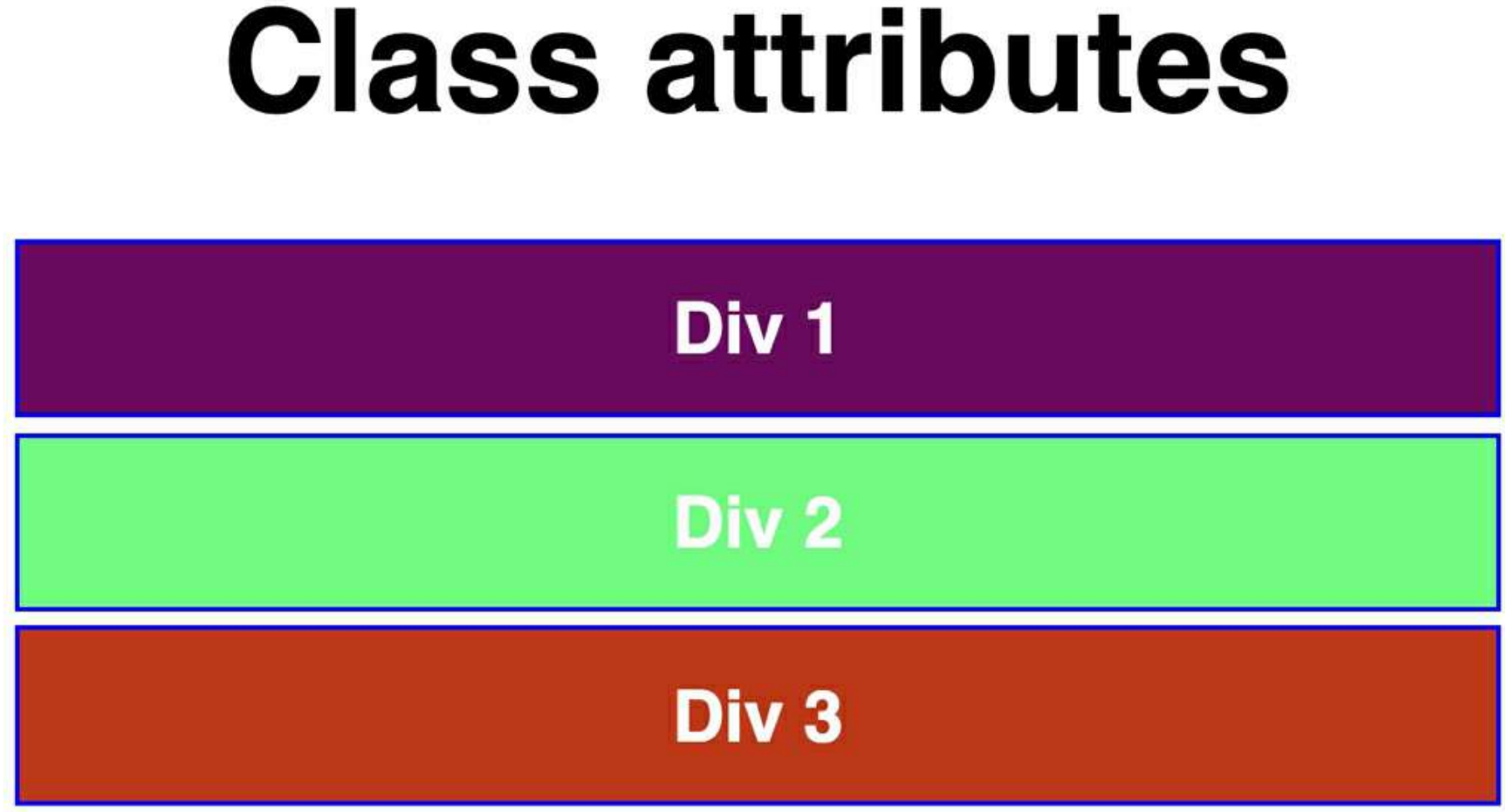
In order to test these methods, we will use the ‘classes.html’ file. Here is the markup:

```
<!DOCTYPE html>
<html lang="en">

<style>
   body {
      max-width: 600px;
      margin: 0 auto;
      font-family: sans-serif;
      text-align: center;
      font-size: 25px;
      color: #fff;
      font-weight: bolder;
   }
   .active {
      background-color: rgb(114, 250, 130);
   }

   .selected {
      background-color: rgb(189, 55, 25);
   }
   .styled{
      background-color: rgb(106, 11, 95);
   }

   .hidden {
      display: none;
   }

   div {
      border: 2px solid blue;
      padding: 15px;
      margin: 5px;
   }
   div:hover{
      background-color: sandybrown;
      cursor: pointer;
   }
   .heading1{
      color:#000;
   }
</style>

<body>
   <h1 class="heading1">Class attributes</h1>
```

```
  <div class="styled">Div 1</div>
  <div class="active">Div 2</div>
  <div class="selected">Div 3</div>

</body>

</html>
```

This is how it looks when it is open in the browser:



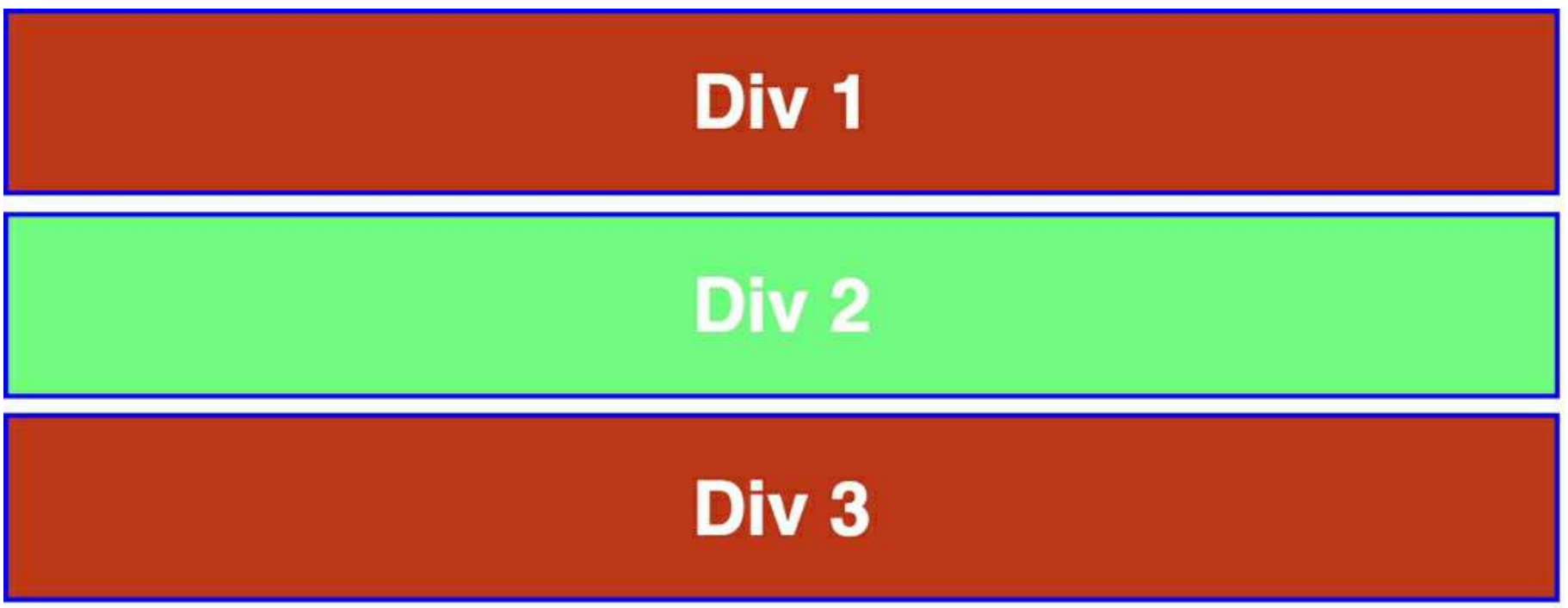


In this file we have three div tags with different class names. For example, let us get the first div tag and see what is the class name:

```
let div1 = document.querySelector('div');
console.log(div1.className);// Output:styled
```

This will give us the class name of the first div tag, but what if we want to set a new class name for the same element? Well, again, we can use the same method just like this:

```
div1.className = 'selected';
```



Now the div1 have the same background as the last div because both of them have the same class names. We can

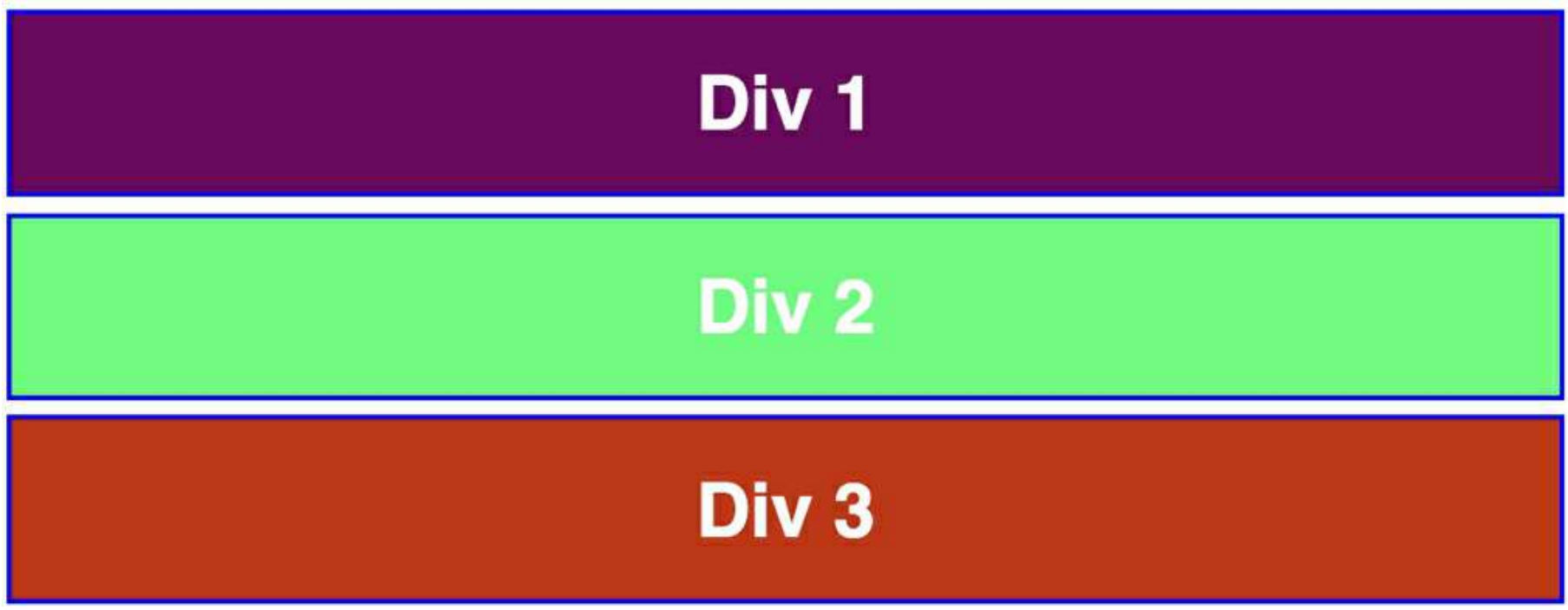
also modify the classes if we are using the **classList** property. Here are few examples of these methods:

```
//select the last div3
let div3 = document.querySelector('.selected');
//add class hidden - it will hide the current div-
div3.classList.add('hidden');
```

The output now does not include the div3 tag because it is hidden, (the class hidden we added sets the div to be display:none):

We can remove the same class hidden and make our div visible again:

```
//remove the hidden classs
div3.classList.remove('hidden');
```

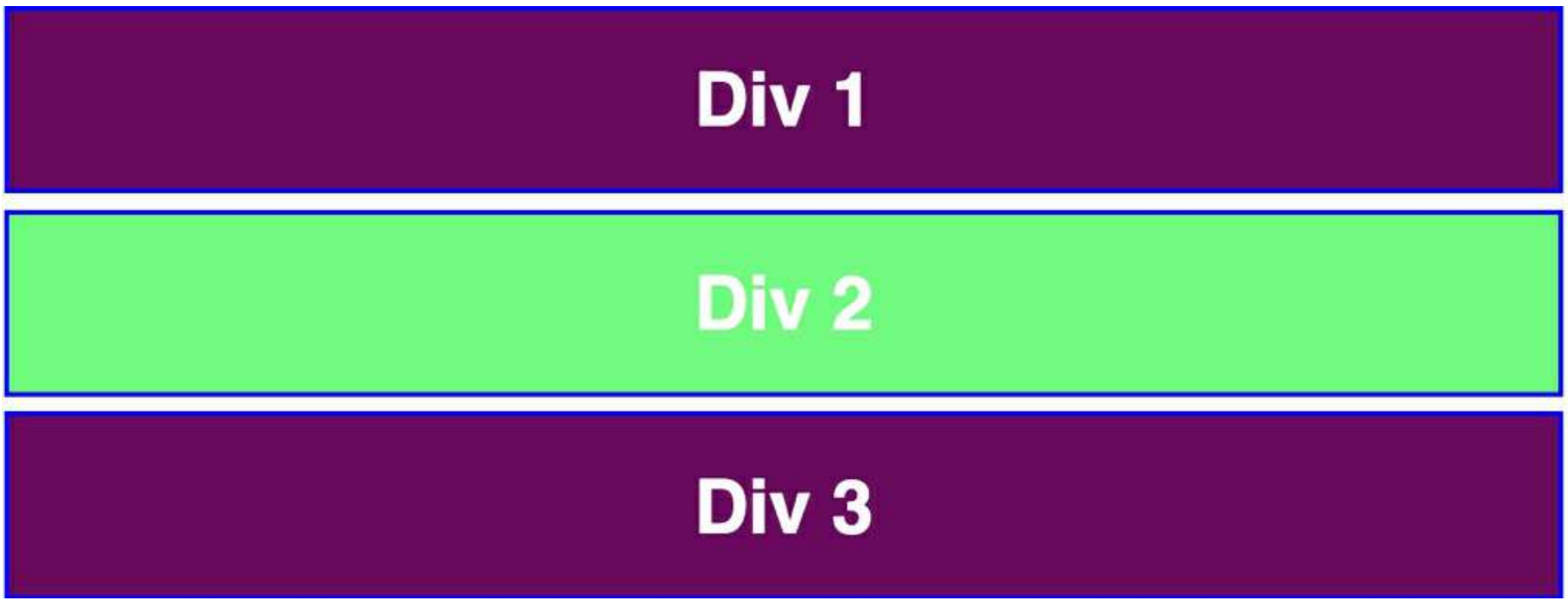


We can toggle the same class:

```
//toggle the hidden class
div3.classList.toggle('hidden');
```

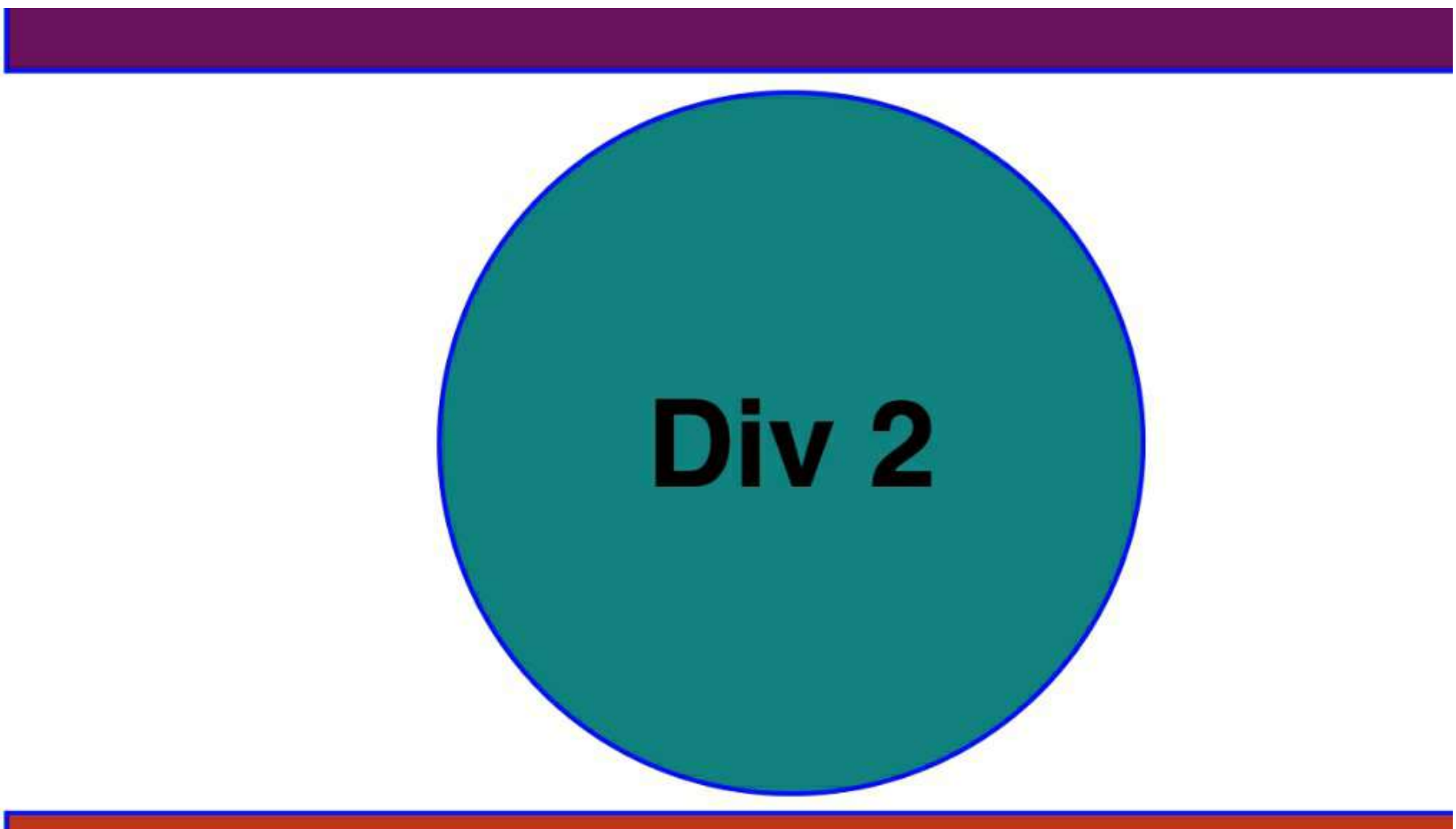
And finally, we can replace the ‘selected’ class with styled class name:

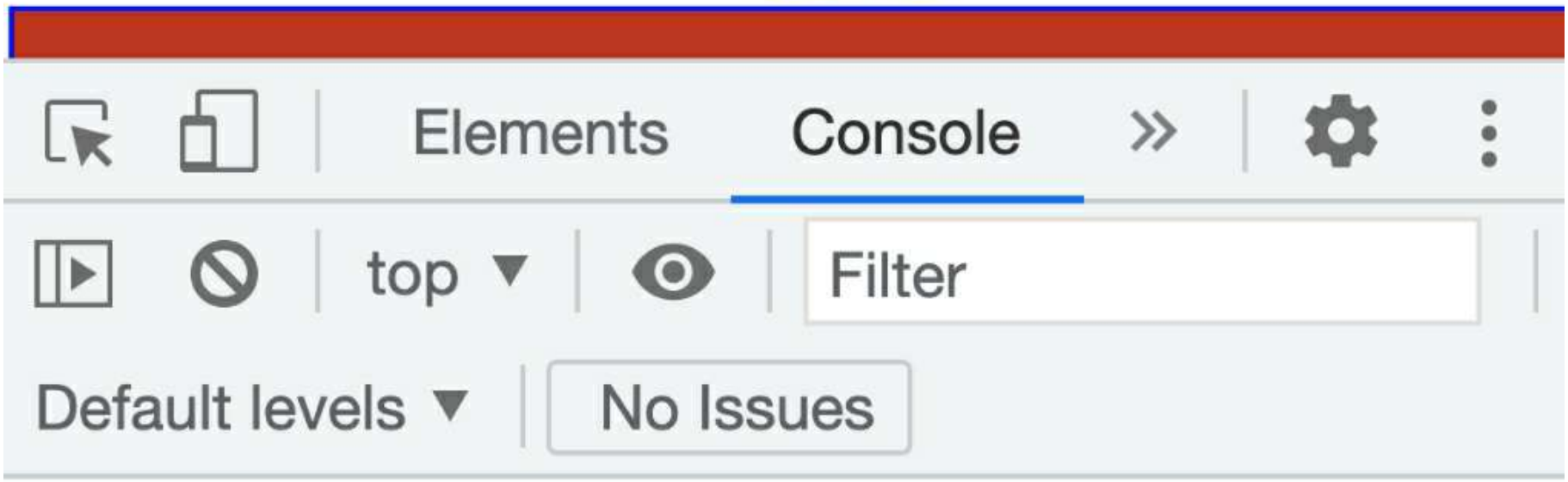
```
div3.classList.replace('selected','styled');
```



Modify the CSS styles

As you know, we can style the document elements from our **CSS** stylesheet file. They are most likely external files, but there can be inline **CSS** styles. In our case we can do CSS element styling directly from our JavaScript file, but what we will modify represents the inline styles of an **HTML** element. We will use the same ‘classes.html’ again and make sure if you are trying this on your computer to comment out the previous code. As an example, let us style the middle div tag:






```
> let div2 =
  document.querySelector('.active');
  div2.style.margin = '0 auto';
  div2.style.width = '200px';
  div2.style.height = '200px';
  div2.style.borderRadius = '50%';
  div2.style.fontSize = '40px';
  div2.style.color = 'black';
  div2.style.backgroundColor = 'teal';

  //make our new div item to be flex
  div2.style.display = 'flex';
  div2.style.justifyContent = 'center';
  div2.style.alignItems = 'center';
```

Instead of using the style attribute directly, we can use another option to set new styles: the **setAttribute**() method. It is important to note is that **setAttribute** will remove all of the existing inline styles from the particular element,

and that is not what we want to happen.
Here is one example:

```
div2.setAttribute('style', 'font-size: 10px');
```

Modify the Attributes

The attributes are the name/value pairs containing additional information about our **HTML** elements. For example, we have seen common **HTML** element attributes like classes, ids, styles, src, href, etc. I would like you to visit the **MDN** website to get the full list of the HTML attributes:

https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes

The question is, how can we modify the attribute values in JavaScript? Well, we have four important methods how we can achieve this:

Property	Description
hasAttribute()	Returns true or false
getAttribute()	Returns the value of the attribute or null
setAttribute()	Add or updates new attribute value
removeAttribute()	Remove an attribute from the element

Let us test these new methods, and in order to do this I have created another sample file called ‘attributes.html’. In this file you can find the entire code. This is markup in the attributes file:

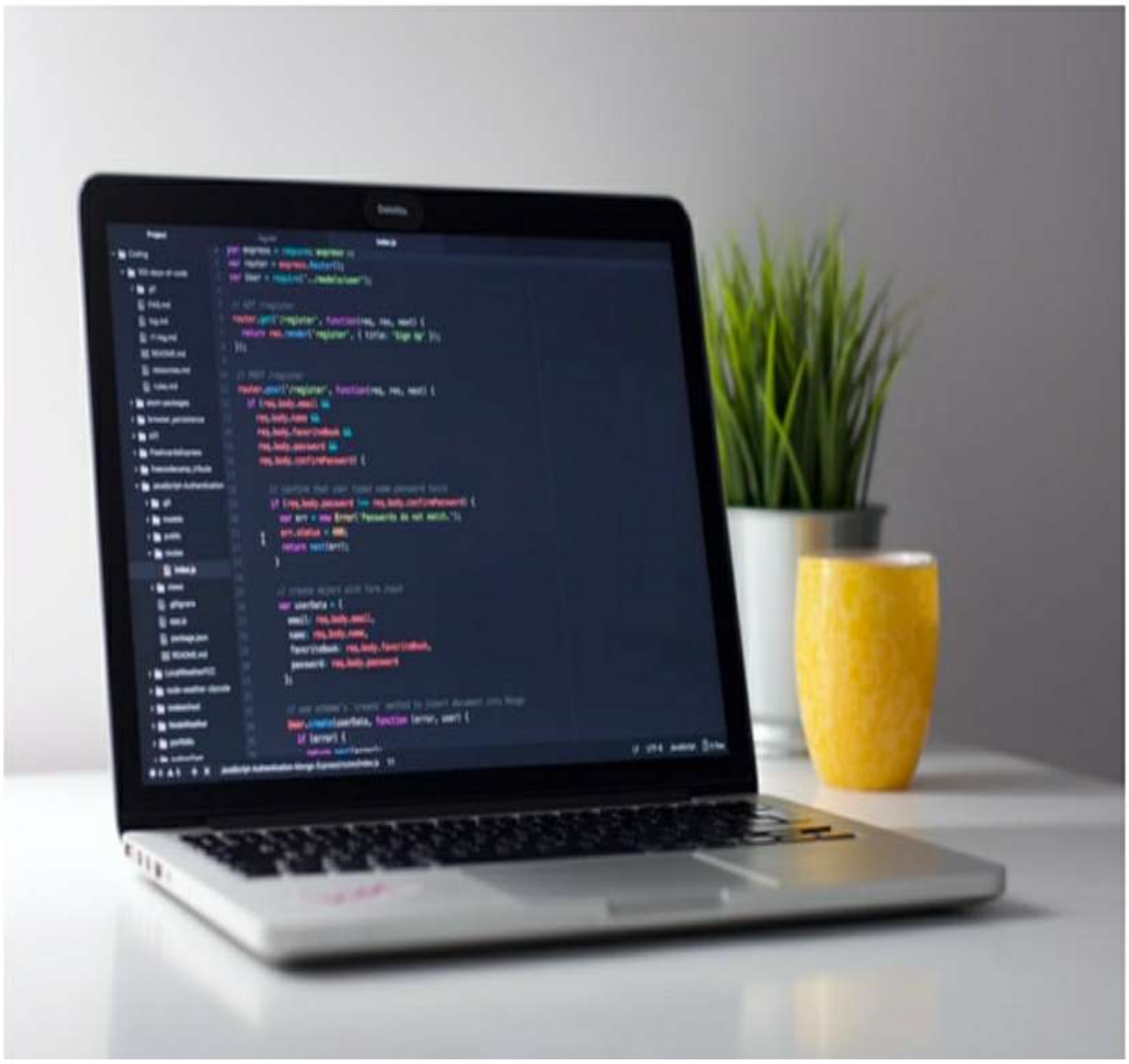
```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Modify Attributes</title>
   <style>
      img{
         height: 400px;
         width: 400px;
      }
   </style>
</head>
<body>
   <h1>This are images from unsplash.com</h1>
   <h2>These are not mine big thanks to:</h2>
   <ul>
      <li><h3>1) DESIGNECOLOGIST</h3></li>
      <li><h3>2) Clement Helardot</h3></li>
   </ul>
   <img src="./img/clement-helardot-95YRwf6CNw8-unsplash.jpg">
</body>
</html>
```

And this is how it looks when it is opened in the browser:

This are images from unsplash.com

These are not mine big thanks to:

- **1) DESIGNECOLOGIST**
- **2) Clement Helardot**



Inside the HTML file, we have **<img>** tag, and this tag has only one attribute called **src,** which stands for source. In this case, the source is a local image stored in the img folder. I have another image in the same img folder that I will also use later in the following examples. Let us test some of the attribute methods.
Here is the test code:

```
const theImg =
document.querySelector('img');
const imgSource =
theImg.hasAttribute('src');
console.log('1) hasAttribute method: ' +
imgSource);
const imgAttribute =
theImg.getAttribute('src');
console.log('2) getAttribute method: ' +
imgAttribute);
const imgRemove =
theImg.removeAttribute('src');
console.log('3) removeAttribute method: ' +
theImg);
```

```
1) hasAttribute method: true                    VM601:3
2) getAttribute method:                         VM601:5
./img/clement-helardot-95YRwf6CNw8-
unsplash.jpg
3) removeAttribute method: [object              VM601:7
HTMLImageElement]
```

The first method will return true because the img tag has the src attribute. The second method, **getAttribute**, will return the entire path of the local image. And finally, we used the remove the attribute which removes the path to the local image. This is the output now:

```
▼<body>
    <h1>This are images from unsplash.com</h1>
    <h2>These are not mine big thanks to:</h2>
  ▶<ul>…</ul>
    <img> == $0
    <script> </script>
```

But here we have a problem, the entire source or src attribute is gone, and we cannot show the image in our browser. Luckily, we can add the removed src attribute back if we use the **setAttribute** method and then add the second image that we have stored in the **img** folder.

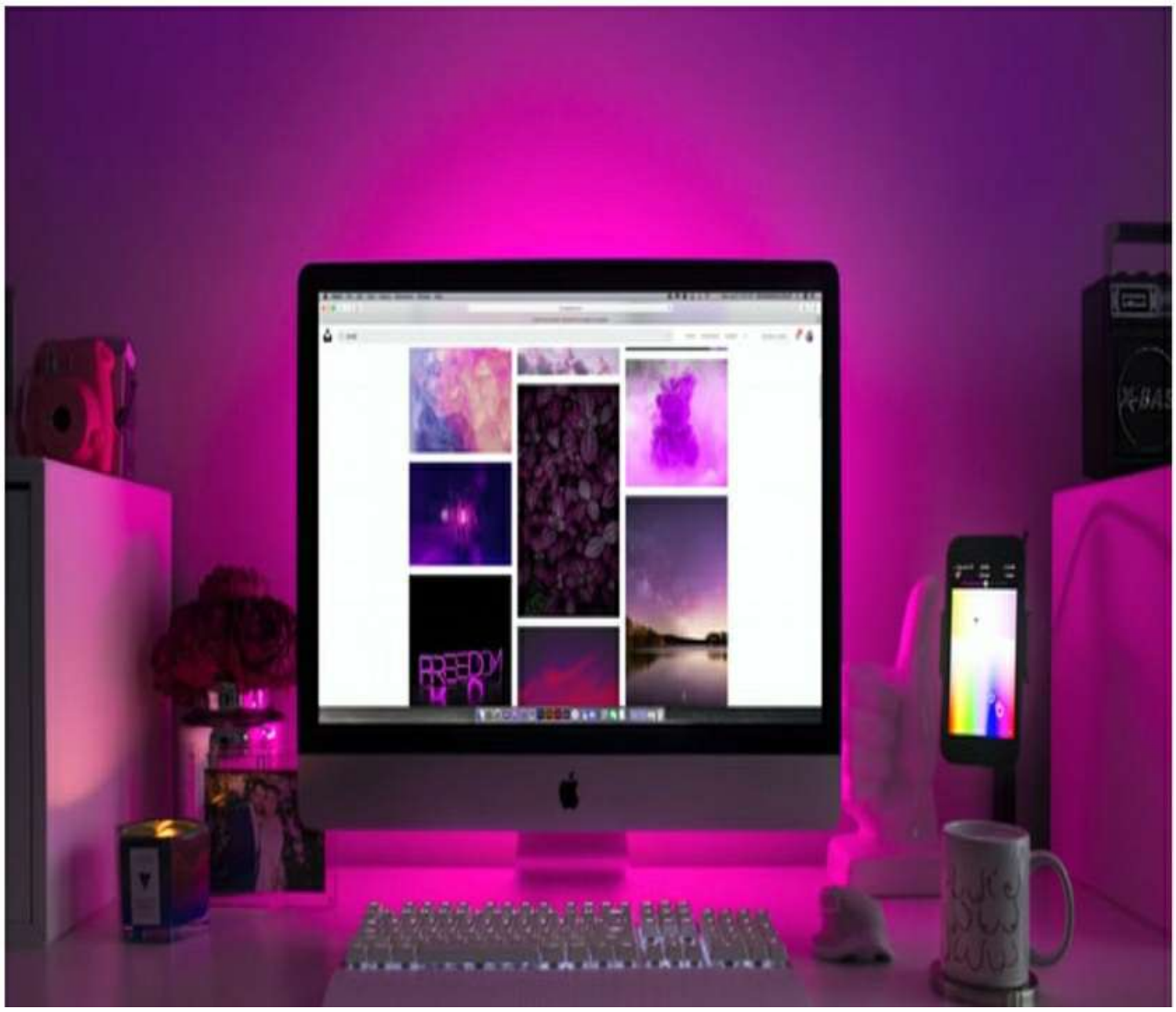
```
theImg.setAttribute('src', './img/designecologist-Pmh0UoG1vlE-unsplash.jpg');
```

If we load the browser, this is what we have:

This are images from unsplash.com

These are not mine big thanks to:

- **1) DESIGNECOLOGIST**
- **2) Clement Helardot**



Let us add another very common attribute in the img tag called alt attribute, which stands for alternative text. This text is visible only when we have a problem loading the image. If the image is loaded successfully, this alternative description will not be seen on the page.

```
theImg.setAttribute('alt','Designecologist');
```

Our DOM img element will look like this now:

And that is all you need to know when it comes to modifying classes, attributes, and styles in **DOM**.

JavaScript Events

So far, we have covered a lot of **DOM** features, and we learned that the **DOM** is a tree of nodes. We also learned that we could traverse that tree in different ways. Later on, we mastered the properties and methods that are most popular for removing, modifying, and replacing nodes and elements. At this point, you should be confident that you can make almost any changes and modifications to the DOM document, but again we are still missing something. The picture is not complete. The thing we are missing now is called events, and they are so important in JavaScript because they will tie everything, we have learned so far, and everything will start making sense.
What are events? Well, JavaScript evens are simply actions that can be performed by the user or even the browser itself. When we open a website, there are many events taking place, but we are not aware of them. For example, you open a web page, and it takes time until that page finishes loading its content. Another event example is when a user clicks on the submit button to submit the registration form. Another event example is when a user clicks on the website menu and gets the dropdown list of submenu items. The website search bar is another common event. Every time a user presses the key on their keyboards, the autocomplete box will appear with handy suggestions. Events are a crucial feature of JavaScript. Without them, our page will be boring and not interactive.

Event Handler & Event Listener

Two very important concepts when it comes to events in JavaScript. The event handler is simply a function. This JavaScript function will run as soon as the event is triggered.
On the other hand, the event listener is attached to a particular element, and it will wait and listen for any changes that might happen on that particular element. There are three ways how we can assign events to the DOM elements:

- Inline event handlers
- Event handler properties
- Event listeners

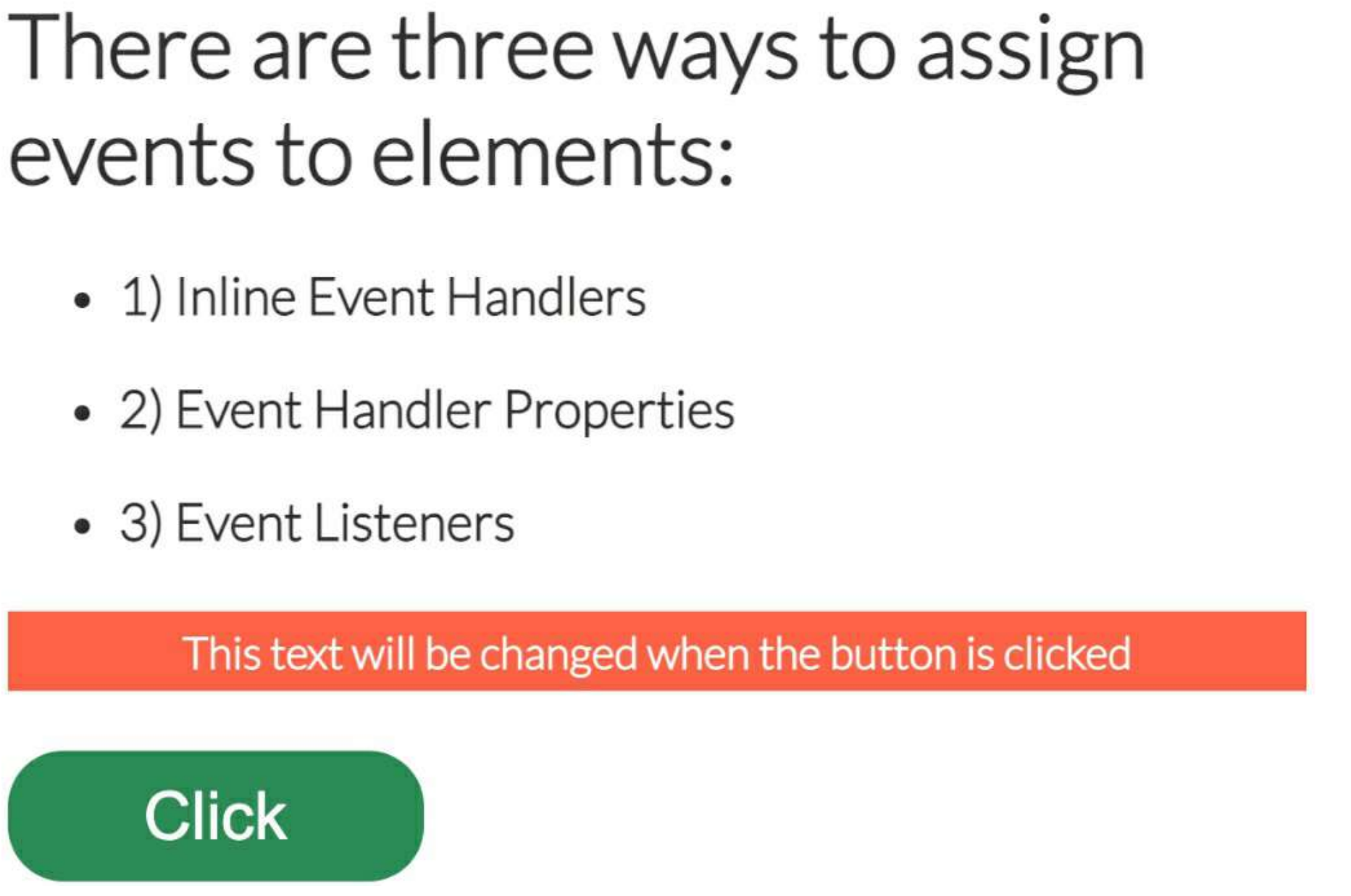
Inline Event Handlers

This is the most basic when it comes to event handlers. The example I will use here is located in the file called ‘events.html’. Here is the entire markup:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>JavaScript Events</title>
   <link rel="preconnect" href="https://fonts.googleapis.com">
   <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
   <link href="https://fonts.googleapis.com/css2?family=Lato:wght@300&display=swap" rel="stylesheet">
   <style>
   html{
       font-family: 'Lato', sans-serif;
       color: #333;
       }
      button{
         background-color: seagreen;
```

```
        color: white;
        border: none;
        border-radius: 20px;
        font-size: 24px;
        padding: 10px 20px;
        width: 150px;
        cursor: pointer;
      }
      button:hover{
        background-color: rgb(24, 89, 52);
      }
      .heading4{
        background-color: tomato;
        text-align: center;
        padding: 5px 10px;
        color: white;
      }
    </style>
</head>
<body>
    <h1>There are three ways to assign events to elements:</h1>
    <ul>
      <li><h3>1) Inline Event Handlers</h3></li>
      <li><h3>2) Event Handler Properties</h3></li>
      <li><h3>3) Event Listeners</h3></li>
    </ul>
    <h4 class='heading4'>This text will be changed when the button is clicked</h4>
    <button>Click</button>
    <script>
    </script>
</body>
</html>
```

As you can see from the markup above, we have a very simple **HTML** file with one heading and one unordered list with a button. The idea is when the user clicks on the button, it will trigger an event that will change the text content we have in our <**h4**> tag. Here is how it looks if we load the file in our browser:



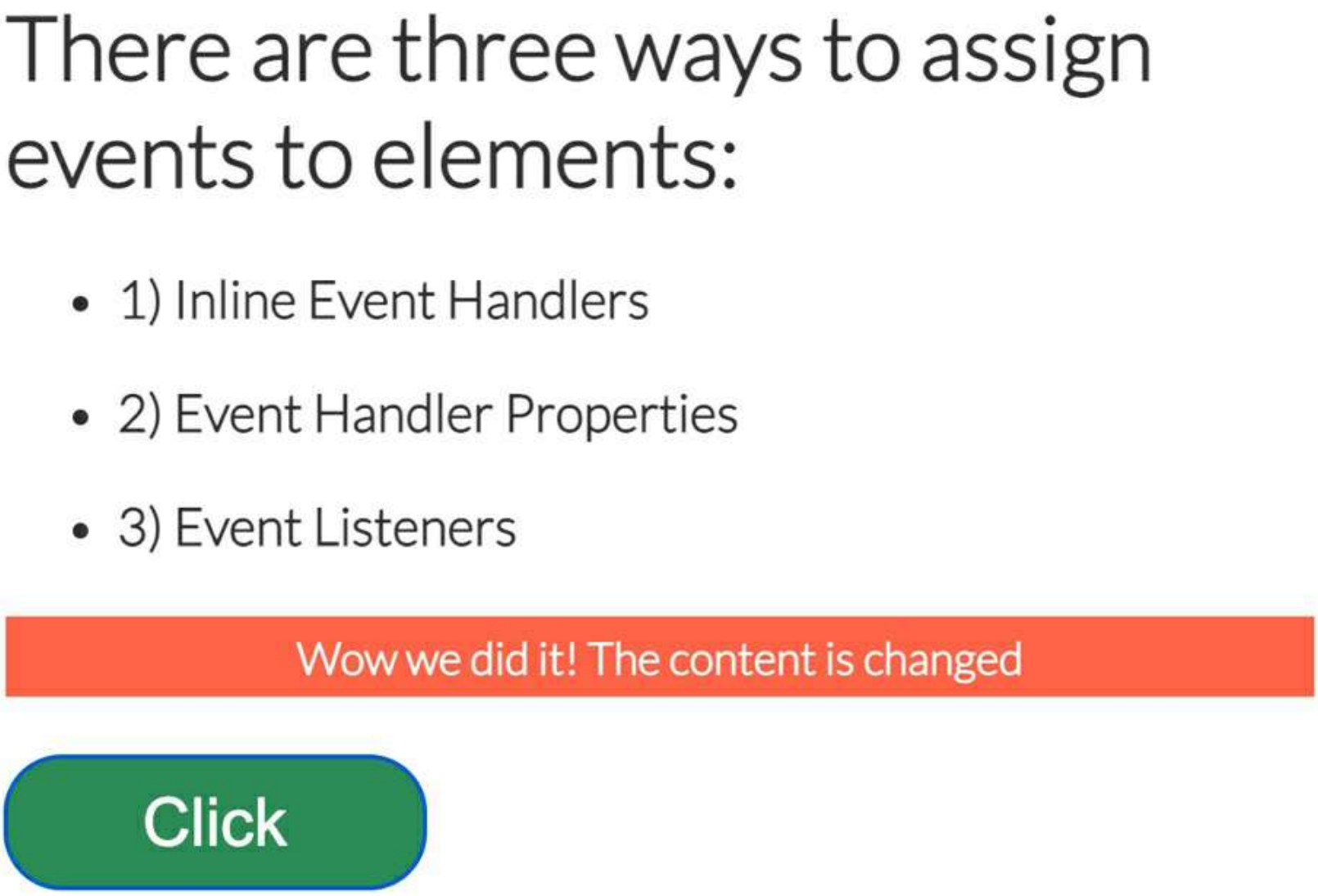
We can add attributes directly to our **HTML** elements with inline event handlers. As soon as the user clicks on the button, we need something to change. We can achieve this by adding an attribute called **onclick** directly to our **HTML** button element. This onclick attribute will need a value which will be a function called **changeContent**(). Therefore, we need to modify our <button> tag in the HTML file, and after the changes, it will look like this:

```
<button onclick="changeContent()">Click</button>
```

We need to write the code between the script tags, so we will create a function that will modify the content text of our <h4> tag in the document. I have chosen to write the JavaScript code here because it is easier for me to explain and include the screenshots. The best way is to create an external JavaScript file and provide the link of that file inside the HTML file. Okay, here is the JavaScript code we need to write:

```
<script>
   //This function will modify the text of the h4 tag
   const changeContent = () => {
       const h4 = document.querySelector('.heading4');
       h4.textContent = "Wow we did it! The content is changed";
   }
</script>
```

When the file is loaded in our browser and when we click on the button, then we should see a different output compared to the first one (‘only the text will be changed inside the h4 tag’):



As we can see, after we clicking on the button, the text of the h4 tag will be replaced with a new text that is coming directly from the JavaScript function.
Congratulations, now you know how the inline event handlers can be created and how they work, they are very easy and straightforward, but generally, they are not much used nowadays. Let’s talk more about the event handler properties.

Event Handler Properties

If the inline handlers were easy for you to understand, I assure you that the event handler properties will also be easy. The event handler property is very similar to the inline events, with only one exception. We do not need to directly associate any attribute and values to our HTML element. We can achieve the same result as in the previous example with writing the few lines of code in our JavaScript file; therefore, this inline property will no longer be needed inside the button:

```
onclick="changeContent()"
```

The idea here is to access the button element in our JavaScript file and assign the event.

```
<button>Click</button>
<script>
//get the button
const btn = document.querySelector('button');
//This function will modify the text of the h4 tag
const changeContent = () => {
```

```
            const h4 =    document.querySelector('.heading4');
            h4.textContent = "Wow we did it! The content is changed";
}
//Add event hander as a new property to the selected html element
btn.onclick = changeContent;
//Event handlers, we do not use camel case notation


</script>
```

If you need to play with the code, then please check the file ‘event1.html’. The result will be exactly the same as the previous. Therefore, no screenshot is needed. Please make sure that the event handler is set on the button after the function **changeContent(),** not before the function declaration; otherwise, it will throw this error:

Uncaught ReferenceError: Cannot access 'changeContent' before initialization

It is important to note that the ‘btn.onclick’ is a function reference, and because of this, we do not include the parentheses () as we do with a normal function call or invocation. So we are not invoking the function with the **btn.onclick**, but simply we are saying that this is only a reference.

```
//ok
btn.onclick = changeContent;


//this is wrong!!
btn.onclick() = changeContent;
```

Event Listeners

So far, we have learned how to handle events in two different ways, and the last one is very important and is all about event listeners. The event listeners listen for a particular event to happen on an element. So we are not going to assign an event directly to an element, but we will use the **addEventListener**() method that will listen for the event to happen. The **addEventListener**() method can take two parameters, the first is the event we are listening for, and the second parameter will be the callback function. To explain this better, I will use the same HTML file as before, but I will slightly change the JavaScript code. The function **changeContext** will stay the same, so here is the entire code:

```
<h4 class='heading4'>This text will be changed when the button is clicked</h4>
   <button>Click</button>
   <script>
      //get the button
 const btn = document.querySelector('button');


      //This function will modify the text of the h4 tag
const changeContent = () => {
     const h4 =    document.querySelector('.heading4');
 h4.textContent = "Wow we did it! The content is changed";
      }
//add event listener on the button       btn.addEventListener('click',changeContent);


</script>
```

As you can see, we selected the button with the help of the CSS selector, and on that button, we are adding the **addEventListener** method that takes two parameters. In the **addEventListener,** the first method will be the actual event, and that is the mouse click. If you go back in the previous examples, you will see that the same event was referred to as ‘onclick’, so now, in the event listeners, we have slightly modified syntax where we are dropping the ‘on’ prefix from the actual event name.
If you open the file (event2.html) and click on the button, you will see the same output as in the previous examples, so nothing has changed, but we have changed the way we handle the events. The callback function, in this case, is **changeContent** which we have as a second parameter and can be written directly as an anonymous function. Anonymous functions are functions that do not have a name:

```
//with anonymous callback function
btn.addEventListener('click',()=>{
  console.log('cliked');
  const h4 = document.querySelector('.heading4');
  h4.textContent = "Wow we did it again! The content is changed";
});
```

Using anonymous does not affect how the function works because the output will be exactly the same again. Event

listeners are the most common way to handle events in JavaScript. There is a way to remove the event that we created to a specific element using the **removeEventListener**() method where the first parameter will be the name of the event like ‘click’ and the second parameter will again be a callback function where we put our logic.

Most Common JavaScript Events

So far, we have covered the three different ways to handle events in JavaScript, but many more events in JavaScript are worth mentioning. I will not create separate examples for them, but I will create tables to read the event type and what they do.
As a professional developer, you will need to know about the mouse events because they are the ones we use mostly. Every time we move a mouse or click on the mouse, then we are performing some action, and all of the actions we do as users are possible events in JavaScript, so here are the mouse events I want you to know:

Event	Description
click	This event fires only when the mouse button is pressed and released
dbClick	This event fires only when an element is clicked twice
mousemove	This event fires only when a mouse pointer movies inside a particular element
mouseenter	This event fires only when a mouse pointer enters a particular element
mouseleave	This event fires only when a mouse pointer leaves the particular element

The click event is a combination of two separate events like **mousedown** and **mouseup** events, and this happens when a user is pressing down a mouse button or lifting the mouse button up. When we use the **mouseenter** and **mouseleve**, we are basically recreating the hover effect when we have a mouse over some element.

Keyboard Events

Same as the mouse events, we have keyboard events, and these events are fired every time the user press, lift or hold down a particular key.

Event	Description
keypress	This event fires continuously as long as the key is pressed
keydown	This event fires only once when a key is pressed down
keyup	This event fires only once when a key is released

Form Events

When a user interacts with a form, then there are certain events we should be aware of:

Event	Description
focus	This event fires when an element like input field receives focus
submit	This event fires when an html form is submitted
blur	This event fires only when an element loses its focus

Summary

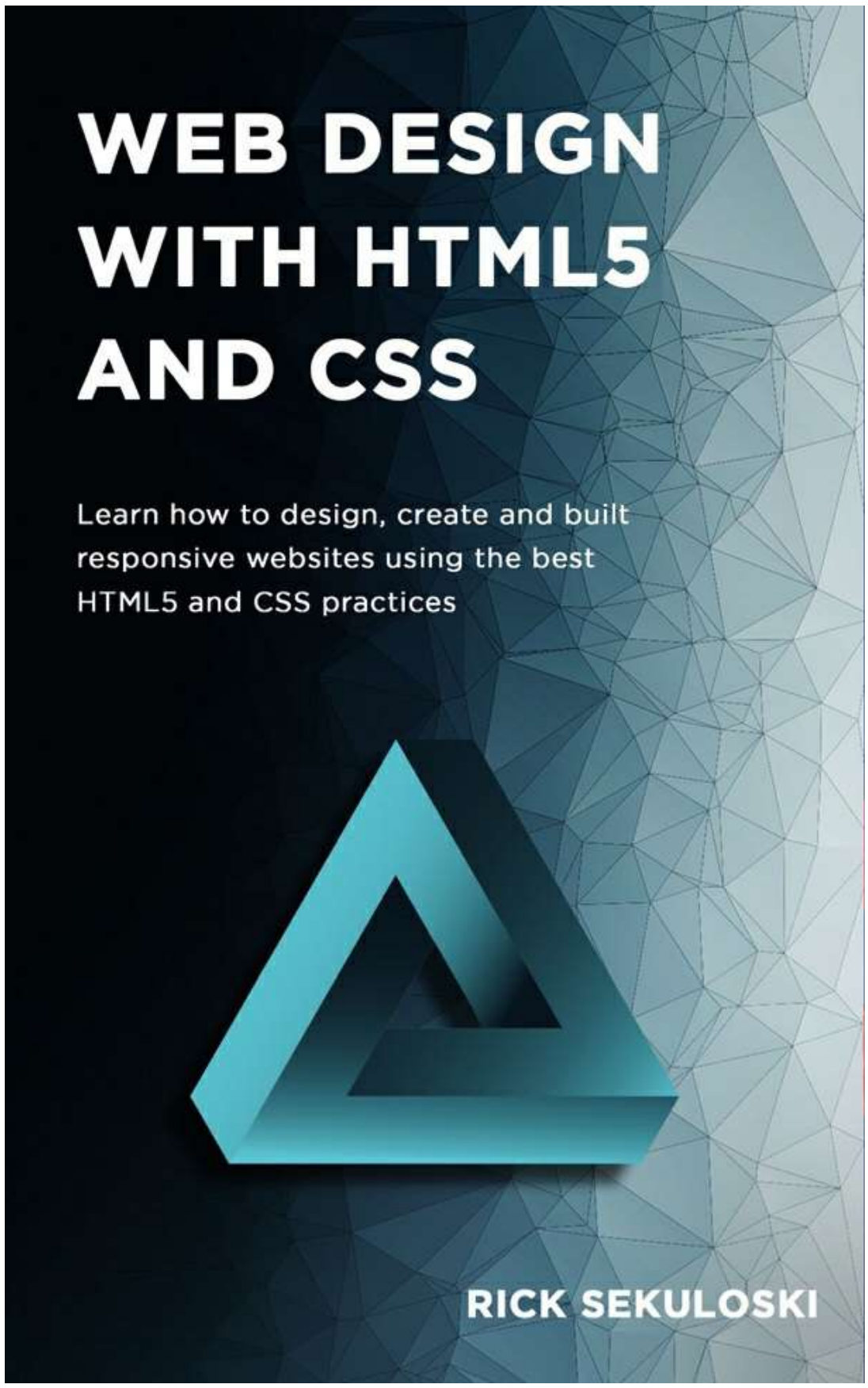
Congratulations! This was an amazing chapter full of practical examples and theory. I believe, and I hope that by now you can manipulate the DOM without any problems and start working on serious projects. This chapter started with the most basic things for web development: the **HTML** and **CSS** syntax. Later on, we covered the **DOM** and the most common methods developers use to select and manipulate elements from the document. We also have learned about the **DOM** tree, what it is, how it is generated, and how it can be traversed. Finally, we have learned about JavaScript events. In the last two chapters, we laid the foundation of modern JavaScript covering the essential core JavaScript features. Without these features, we as developers will not be able to progress and learn more advanced topics discussed in the first three chapters. After reading this guide, I believe that you are knowledgeable enough to apply for any JavaScript job out there. Please check out my social media accounts and subscribe to some of them so that you never miss out on any of my new books, courses, or special discounts because I know you will acquire new lifetime skills and you can start making huge profits. I wish you good luck and hope to see you again sometime.

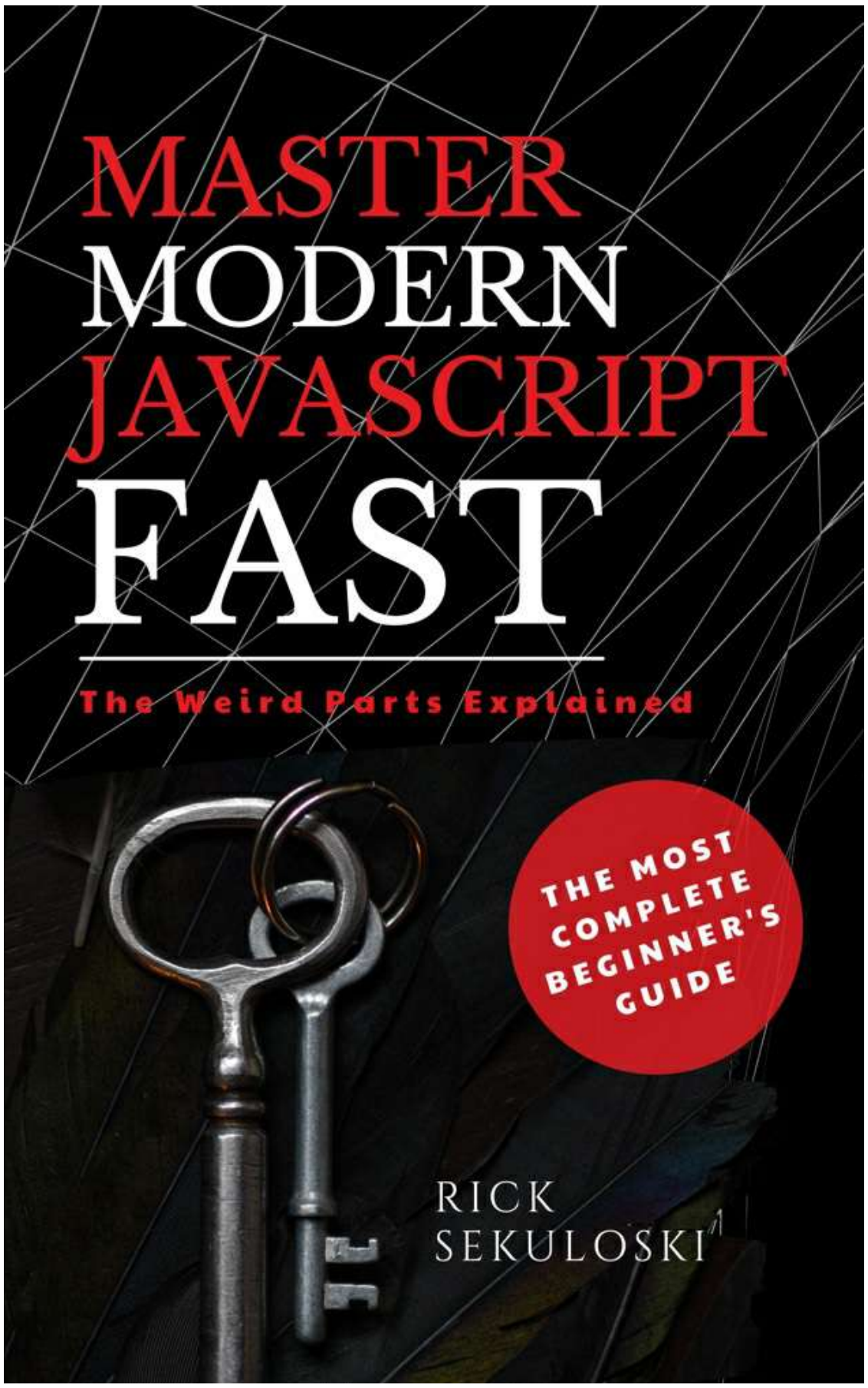
About the author

Rick Sekuloski is a web designer/developer since 2010. He is currently creating courses and content for his students on multiple platforms like Udemy. He is a full-stack developer working on front and backend projects. Before his web development career, he migrated to Australia to pursue his career as a professional IT person. He graduated from Federation University with a degree in Software web development. Outside of work, he enjoys his time with his family, especially his newborn son.

Appendix A: Basic to Intermediate JavaScript book

If you still want to learn more about the basic to intermediate JavaScript features then I suggest taking my two best seller eBooks:




MASTER
MODERN
JAVASCRIPT
FAST
The Weird Parts Explained
THE MOST COMPLETE BEGINNER'S GUIDE
RICK SEKULOSKI

Appendix B: Exercises and learn more about JavaScript, HTML, and PHP

Please check out my courses if you want to learn more about JavaScript or other Web Development languages. I promise you will gain huge experience working on real-life examples.

You can find my courses on Udemy:

https://www.udemy.com/user/riste3/

Ultimate PHP, Laravel, CSS & Sass! Learn PHP, Laravel & Sass

Rick Sekuloski

4.4 ★★★★½ (128)

95.5 total hours • 444 lectures • All Levels

A$13.99 ~~A$109.99~~

Master JavaScript - The Most Complete JavaScript Course 2021

Rick Sekuloski

4.2 ★★★★☆ (27)

45.5 total hours • 181 lectures • All Levels

A$16.99 ~~A$129.99~~

Appendix C: Resources

MDN Web Docs. (2021). *JavaScript:* https://developer.mozilla.org/en-US/docs/Web/JavaScript