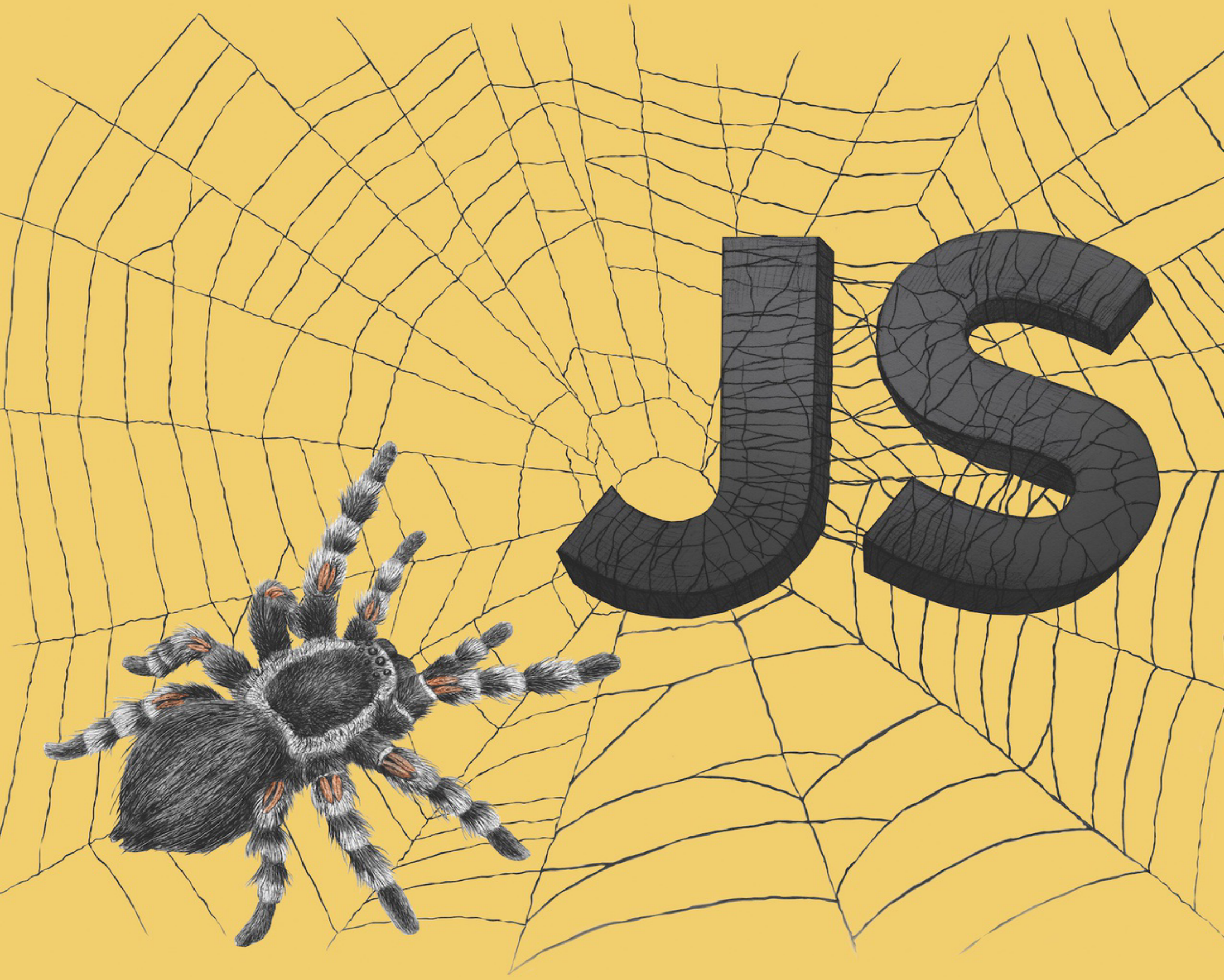


JavaScript From Zero to Hero

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

Copyright © 2022 Rick Sekuloski
All rights reserved.
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:

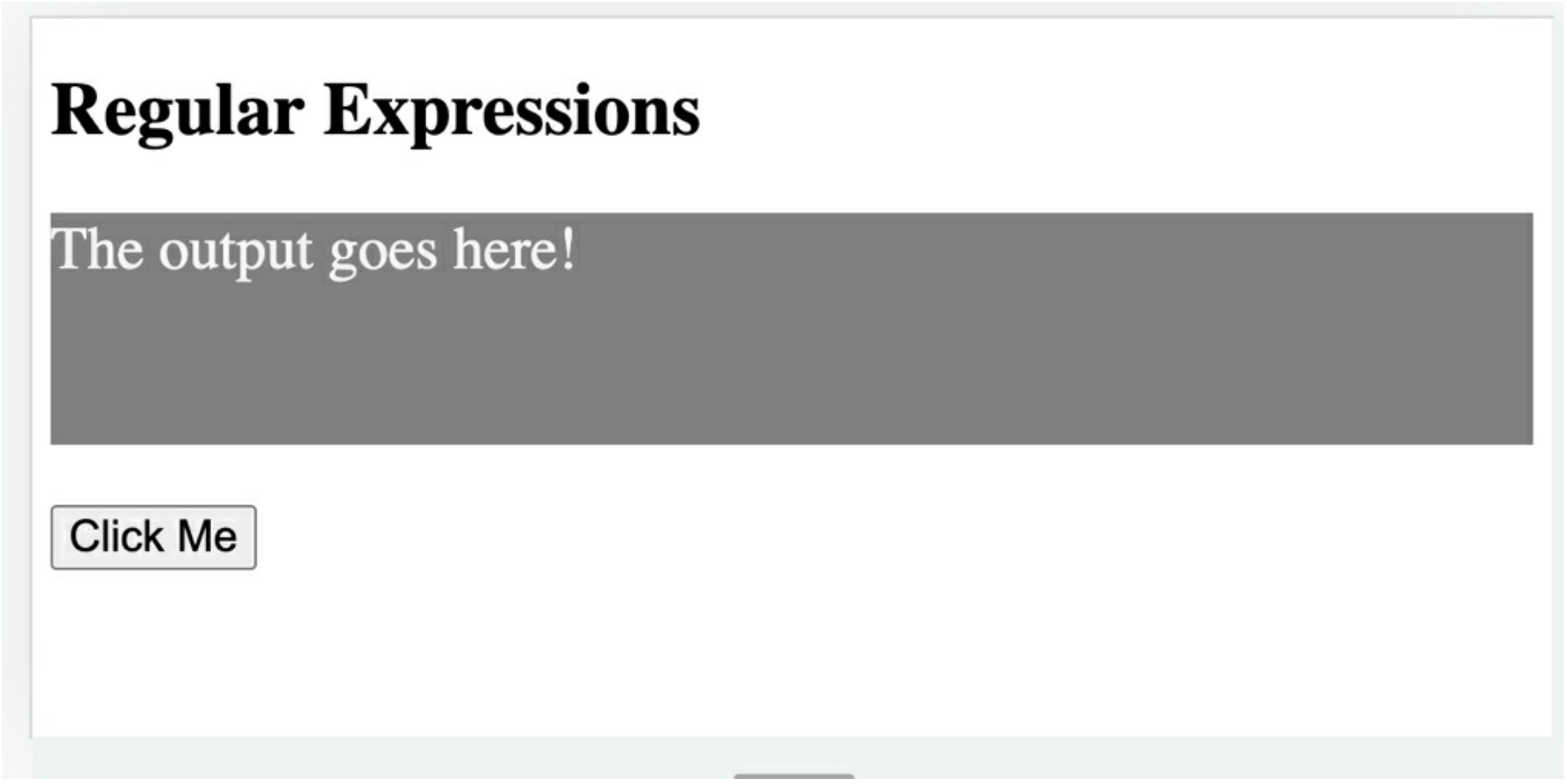
- standard
- g – global
- i – case-insensitive matching
- m – this performs multiline matching
- u – Unicode
- y – sticky

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:

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']
```

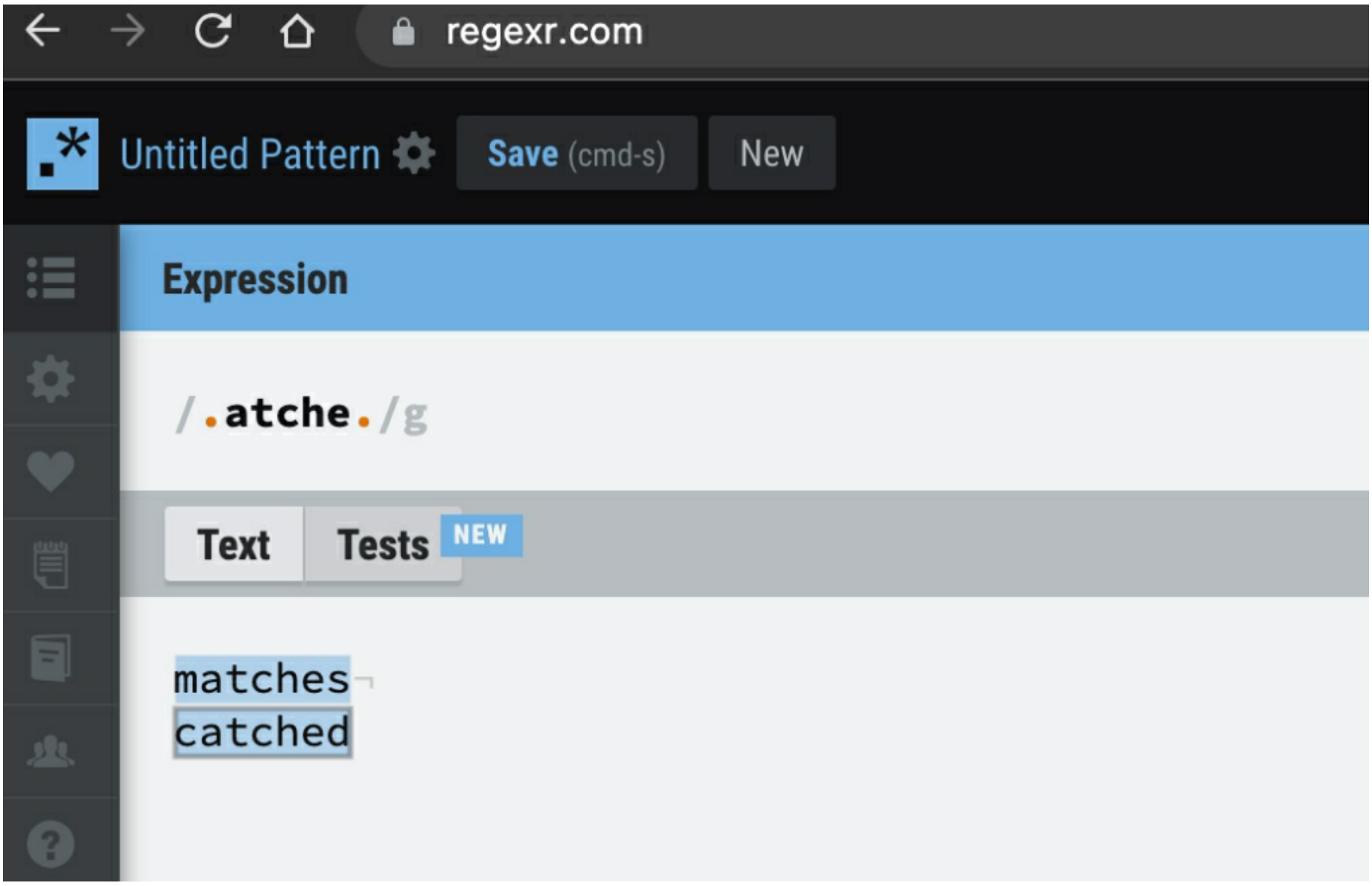
2) The second step is to include the whitespaces before and after. The special character for white spaces was 's' if you look at the previous table, and we need to combine it with the '+' symbol so we can specify that we are saying it has to have at least one whitespace before after the JavaScript.

```
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]);
```

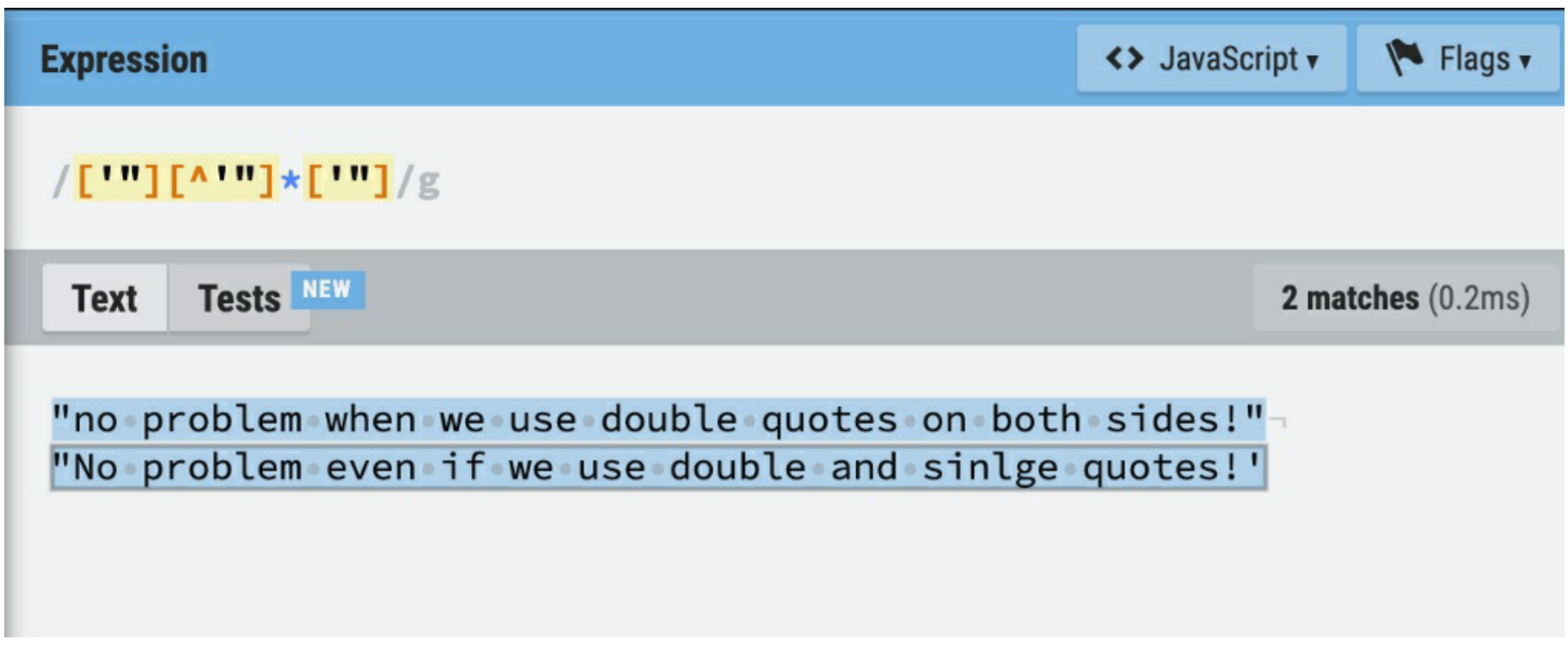
And this is the output:

```
> 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?
```

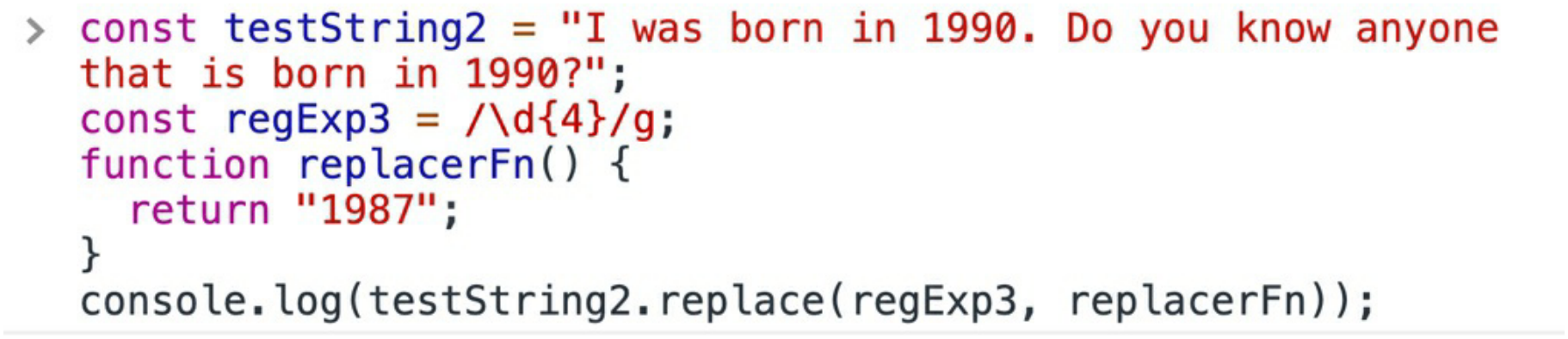
Let us cover something even more interesting and useful. Imagine that you want to replace 2 different substrings within the existing string with a new string. Well, someone will say this is easy. We can use two replace methods on that string, one after the other. But, instead of doing that, we can do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



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));
```

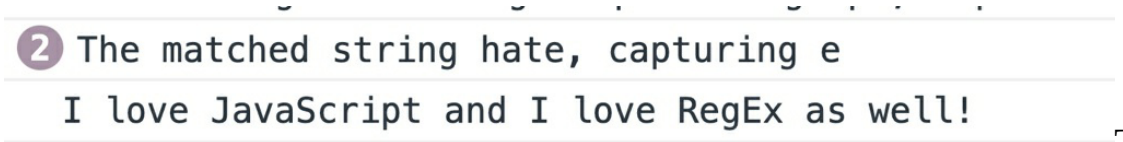
Output:

```
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:



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: