CODE > REGULAR EXPRESSIONS

A Beginner's Guide to Regular Expressions in JavaScript

by Monty Shokeen 25 May 2018 Difficulty: Beginner Length: Medium Languages: English

Regular Expressions JavaScript Programming Fundamentals Web Development

Everyone working with JavaScript will have to deal with strings at one point or other. Sometimes, you will just have to store a string inside another variable and then pass it over. Other times, you will have to inspect it and see if it contains a particular substring.

However, things are not always this easy. There will be times when you will not be looking for a particular substring but a set of substrings which follow a certain pattern.

Let's say you have to replace all occurrences of "Apples" in a string with "apples". You could simply use `theMainString.replace("Apples", "apples")`. Nice and easy.

Now let's say you have to replace "appLes" with "apples" as well. Similarly, "appLES" should become "apples" too. Basically, all case variations of "Apple" need to be changed to "apple". Passing simple strings as an argument will no longer be practical or efficient in such cases.

This is where regular expressions come in—you could simply use the case-insensitive flag `i` and be done with it. With the flag in place, it doesn't matter if the original string contained "Apples", "APPles", "ApPlEs", or "Apples". Every instance of the word will be replaced with "apples".

Just like the case-insensitive flag, regular expressions offer a lot of other features which will be covered in this tutorial.

Using Regular Expressions in JavaScript

You have to use a slightly different syntax to indicate a regular expression inside different `String` methods. Unlike a simple string, which is enclosed in quotes, a regular expression consists of a pattern enclosed between slashes. Any flags that you use in a regular expression will be appended after the second slash.

Going back to the previous example, here is what the `replace()` method would look like with a regular expression and a simple string.

```
"I ate Apples".replace("Apples", "apples");
// I ate apples

"I ate Apples".replace(/Apples/i, "apples");
// I ate apples

"I ate aPPles".replace("Apples", "apples");
// I ate aPPles

"I ate aPPles".replace(/Apples/i, "apples");
// I ate apples
```

As you can see, the regular expression worked in both cases. We will now learn more about flags and special characters that make up the pattern inside a regular expression.

Backslash in Regular Expressions

You can turn normal characters into special characters by adding a backslash before them. Similarly, you can turn special characters into normal characters by adding a backslash before them.

For example, `d` is not a special character. However, `\d` is used to match a digit character in a string. Similarly, `D` is not a special character either, but `\D` is used to match non-digit characters in a string.

Digit characters include 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. When you use `\d` inside a regular expression, it will match any of these nine characters. When you use `\D` inside a regular expression, it will match all the non-digit characters.

The following example should make things clear.

```
"L8".replace(/\d/i, "E");
// LE

"L8".replace(/\D/i, "E");
// E8

"LLLLL8".replace(/\D/i, "E");
// ELLLL8
```

You should note that only the first matched character is replaced in the third case. You can also use flags to replace all the matches. We will learn about such flags later.

Just like `\d` and `\D`, there are other special character sequences as well.

1. You can use `\w` to match any "word" character in a string. Here, word character refers to A-Z, a-z, 0-9, and _. So, basically, it will match all digits, all lowercase and uppercase alphabets, and the underscore.
2. You can use `\W` to match any non-word character in a string. It will match characters like %, $, #, ₹, etc.
3. You can use `\s` to match a single white space character, which includes space, tab, form feed, and line feed. Similarly, you can use `\S` to match all other characters besides white space.
4. You can also look for a specific white space character using `\f`, `\n`, `\r`, `\t`, and `\v`, which stand for form feed, line feed, carriage return, horizontal tab, and vertical tab.

Sometimes, you will face situations where you need to replace a word with its substitute, but only if it is *not* part of a larger word. For example, consider the following sentence:

"A lot of pineapple images were posted on the app".

In this case, we want to replace the word "app" with "board". However, using a simple regular expression pattern will turn "apple" into "boardle", and the final sentence would become:

"A lot of pineboardle images were posted on the app".

In such cases, you can use another special character sequence: `\b`. This checks for word boundaries. A word boundary is formed by use of any non-word characters like space, "$", "%", "#", etc. Watch out, though—it also includes accented characters like "ü".

```
"A lot of pineapple images were posted on the app".replace(/app/, "board");
// A lot of pineboardle images were posted on the app

"A lot of pineapple images were posted on the app".replace(/\bapp/, "board")
// A lot of pineapple images were posted on the board
```

Similarly, you can use `\B` to match a non-word boundary. For example, you could use `\B` to only match "app" when it is within another word, like "pineapple".

Matching a Pattern "n" Number of Times

You can use `^` to tell JavaScript to only look at the beginning of the string for a match. Similarly, you can use `$` to only look at the end of the string for a match.

You can use `*` to match the preceding expression 0 or more times. For example, `/Ap*/` will match **A**, **Ap**, **App**, **Appp**, and so on.

In a similar manner, you can use `+` to match the preceding expression 1 or more times. For example, `/Ap+/` will match **Ap**, **App**, **Appp**, and so on. The expression will not match the single **A** this time.

Sometimes, you only want to match a specific number of occurrences of a given pattern. In such cases, you should use the `{n}` character sequence, where n is a number. For instance, `/Ap{2}/` will match **App** but not **Ap**. It will also match the first two 'p's in **Appp** and leave the third one untouched.

You can use `{n,}` to match at least 'n' occurrences of a given expression. This means that `/Ap{2,}/` will match **App** but not **Ap**. It will also match all the 'p's in **Apppp** and replace them with your replacement string.

You can also use `{n,m}` to specify a minimum and maximum number and limit the number of times the given expression should be matched. For example, `/Ap{2,4}/` will match **App**, **Appp**, and **Apppp**. It will also match the first four 'p's in **Appppppp** and leave the rest of them untouched.

```
"Appppppples".replace(/Ap*/, "App");
// Apples

"Ales".replace(/Ap*/, "App");
// Apples

"Apppppples".replace(/Ap{2}/, "Add");
// Addppples

"Apppppples".replace(/Ap{2,}/, "Add");
// Addles

"Apppppples".replace(/Ap{2,4}/, "Add");
// Addples
```

Using Parentheses to Remember Matches

So far, we have only replaced patterns with a constant string. For example, in the previous section, the replacement we used was always "Add". Sometimes, you will have to look for a pattern match inside the given string and then replace it with a part of the pattern.

Let's say you have to find a word with five or more letters in a string and then add an "s" at the end of the word. In such cases, you will not be able to use a constant string value as a replacement as the final value depends on the matching pattern itself.

```
"I like Apple".replace(/(\w{5,})/, '$1s');
// I like Apples

"I like Banana".replace(/(\w{5,})/, '$1s');
// I like Bananas
```

This was a simple example, but you can use the same technique to keep more than one matching pattern in memory. The number of sub-patterns in the full match will be determined by the number of parentheses used.

Inside the replacement string, the first sub-match will be identified using `$1`, the second sub-match will be identified using `$2`, and so on. Here is another example to further clarify the usage of parentheses.

```
"I am looking for John and Jason".replace(/(\w+)\sand\s(\w+)/, '$2 and $1');
// I am looking for Jason and John
```

Using Flags With Regular Expressions

As I mentioned in the introduction, one more important feature of regular expressions is the use of special flags to modify how a search is performed. The flags are optional, but you can use them to do things like making a search global or case-insensitive.

These are the four commonly used flags to change how JavaScript searches or replaces a string.

- `g`: This flag will perform a global search instead of stopping after the first match.
- `i`: This flag will perform a search without checking for an exact case match. For instance, Apple, aPPLe, and apPLE are all treated the same during case-insensitive searches.
- `m`: This flag will perform a multi-line search.
- `y`: This flag will look for a match in the index indicated by the `lastIndex` property.

Here are some examples of regular expressions used with flags:

```
"I ate apples, you ate apples".replace(/apples/, "mangoes");
// "I ate mangoes, you ate apples"

"I ate apples, you ate apples".replace(/apples/g, "mangoes");
// "I ate mangoes, you ate mangoes"

"I ate apples, you ate APPLES".replace(/apples/, "mangoes");
// "I ate mangoes, you ate APPLES"

"I ate apples, you ate APPLES".replace(/apples/gi, "mangoes");
// "I ate mangoes, you ate mangoes"


var stickyRegex = /apples/y;
stickyRegex.lastIndex = 3;
"I ate apples, you ate apples".replace(stickyRegex, "mangoes");
// "I ate apples, you ate apples"

var stickyRegex = /apples/y;
stickyRegex.lastIndex = 6;
"I ate apples, you ate apples".replace(stickyRegex, "mangoes");
// "I ate mangoes, you ate apples"

var stickyRegex = /apples/y;
stickyRegex.lastIndex = 8;
"I ate apples, you ate apples".replace(stickyRegex, "mangoes");
// "I ate apples, you ate apples"
```

Final Thoughts

The purpose of this tutorial was to introduce you to regular expressions in JavaScript and their importance. We began with the basics and then covered backslash and other special characters. We also learned how to check for a repeating pattern in a string and how to remember partial matches in a pattern in order to use them later.

Finally, we learned about commonly used flags which make regular expressions even more powerful. You can learn more about regular expressions in this article on MDN.

If there is anything that you would like me to clarify in this tutorial, feel free to let me know in the comments.

Monty Shokeen

I am a full-stack developer who also loves to write tutorials in his free time. Other than that, I love learning about new and interesting JavaScript libraries.