JavaScript Regular Expressions: Beyond the Basics

by Monty Shokeen 5 Jun 2018 Difficulty: Intermediate Length: Medium Languages: English

JavaScript Regular Expressions Programming Fundamentals

In our previous tutorial about regular expressions in JavaScript, you learned about the usefulness of regular expressions and how to write some of your own to match simple patterns.

After reading the previous tutorial, you should now have a good understanding of special characters like a backslash and character sequences like `\w` or `\W`. Here is a really quick summary of those character sequences:

1. You can use `\d` or `\D` to match a digit or non-digit character respectively in any given string. Digit characters include 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. All other characters will be matched by `\D`.
2. You can use `\w` or `\W` to match a word or non-word character in any given string. Word characters include alphabets, digits, and underscore. Everything else, like ₹, %, etc., is considered a non-word character.

3. You can use `\s` or `\S` to match space characters or non-space characters in a string. Space characters include space, tab, form feed, and line feed.

Instead of matching one character at a time, you can use the `*` symbol to match the preceding expression zero or more times. The `+` character will similarly match the preceding expression 1 or more times.

You can match a pattern any specific number of times by appending `{n, m}` to it. Here, `n` is the minimum number of times you want to match it, and `m` is the maximum limit. If you don't specify a value for `m`, the preceding expression will be matched as many times as possible.

You should check out my previous tutorial if anything that we just covered is not clear. I explained everything in more detail there.

Now, let's move on to some more sophisticated character sequences in regular expressions, so that you can get the most out of them and figure out how to write expressions that match complicated patterns.

Non-Greedy Matches Using the `?` Character

The `?` character means different things in different situations.

When used alone, this character matches the expression that came before it 0 or 1 times. In this sense, it is the same as `{0,1}`.

You can also use `?` immediately after other quantifiers like `*`, `+` and `{}` to match the minimum possible number of characters. In other words, it will turn those greedy quantifiers into non-greedy. This can be a bit hard to understand without looking at live examples, so let's see an example first.

Consider the following sentence:

> *I have been assigned 17321HDGE as user id while my friend was assigned FHES193EK1.*

Now, let's see all the matches that would have been returned by different quantifiers and their non-greedy counterpart.

If we use the expression `/\d+/g` in the example, it will match one or more consecutive digit characters. Due to the global flag, there will be three matches: **17321**, **193**, and **1**.

You should note that **193** and **1** are considered different matches because they are separated by **EK**.

The following example shows the matches without the use of any quantifiers.

```
var re = /\d+/g;
var count = 0;
var textString = "I have been assigned 17321HDGE as user id while my friend

var match = re.exec(textString);
while(match !== null) {
  console.log(match[0]);

```

```
  match = re.exec(textString);
  count++;
}

console.log("Total Matches: " + count);

/* Output
17321
193
1

Total Matches: 3
*/
```

Now, adding a `?` character after `\d+` will return nine different matches. Basically, `/\d+?/` will turn each digit character into a separate match. Why is that?

It is because `\d+` is by definition supposed to match one or more digits. Since the `?` character is supposed to match the minimum possible number of characters, it just matches a single digit at a time.

The non-greedy `?` quantifier will return 9 smaller single digit matches this time. For brevity, I have commented out the line that logs the matches to console.

```
var re = /\d+?/g;
var count = 0;
var textString = "I have been assigned 17321HDGE as user id while my friend

var match = re.exec(textString);
while(match !== null) {
  // console.log(match[0]);
  match = re.exec(textString);
  count++;
}

console.log("Total Matches: " + count);


/* Output
Total Matches: 9
*/
```

Let's take another example. The regular expression `/\w+/` will keep matching word characters as long as they are not interrupted by a non-word character like space. In our case, it will match whole space-separated words like **assigned** and **17321HDGE** one a time.

If we replace our original regular expression with `/\w+/`, we will get 14 different matches. Basically, each word will be its own match. You can see the output yourself by commenting out the line.

```
var re = /\w+/g;
var count = 0;
var textString = "I have been assigned 17321HDGE as user id while my friend

var match = re.exec(textString);
while(match !== null) {
  // console.log(match[0]);
  match = re.exec(textString);
  count++;
}

console.log("Total Matches: " + count);

/* Output
Total Matches: 14
*/
```

Now, changing the expression to `/\w+?/` will return each word character as a separate match, and you will get 68 matches.

Let's take a look at one last example before we proceed further. The regular expression `/\w{4,}/` will return all words in our sentence that are four characters or longer. So it matches **have**, **been**, **assigned**, and **17321HDGE**, among others. Now turning it to `/\w{4,}?/` would return multiple matches from words with more than four characters. In our example, the returned matches would be **have**, **been**, **assi**, **gned**, **1732**, and **1HGD**. The character **E** at the end of **17321HDGE** is not part of any match because it could not be in the group of any four consecutive word characters.

```
var re = /\w{4,}/g;
var count = 0;
var textString = "I have been assigned 17321HDGE as user id while my friend

var match = re.exec(textString);
while(match !== null) {
  console.log(match[0]);
  match = re.exec(textString);
  count++;
}

console.log("Total Matches: " + count);

/* Output
have
been
assigned
17321HDGE
user
while
friend
assigned
FHES193EK1

Total Matches: 9
*/
```

Using Parentheses With the ? Character

In my previous regex tutorial, I briefly covered how parentheses can be used to remember part of a match. When used with a `?` character, they can serve other purposes as well.

Sometimes, you want a group of characters to match as a unit. For instance, you could be looking for the occurrences of **na** once or twice as a match in the following text.

> *na naa nnaa nana naana*

For clarification, you are looking for the bold text as matches: **na na**a n**na**a **(nana) na**a**na**. The part in the brackets is supposed to be matched as a unit, so it just counts as one match.

Almost everyone who is just starting out with regex will use the expression `/na{1,2}/` with the intention of getting the expected outcome. In their minds, the **{1,2}** part is supposed to match one or two occurrences of **n** and **a** together. However, it actually matches a single occurrence of **n** followed by 1 or 2 occurrences of the character **a**.

I have rendered the matches returned by `/na{1,2}/` in bold for clarification: **na naa** n**naa (na)(na) (naa)(na)**. The parts in the brackets are separate matches. As you can see, we are not getting the result we wanted because `{1,2}` is not considering **na** to be a single unit that has to be matched.

The solution here is to use parentheses to tell JavaScript to match **na** as a unit. However, as we saw in the previous tutorial, JavaScript will start remembering the match because of the parentheses.

If you don't want JavaScript to remember the match, then you will have to add `?:` before the group of characters that you are trying to match. In our case, the final expression would become `/(?:na){1,2}/`. The group **na** will be matched as a unit now, and it won't be remembered. I have highlighted the final matches returned with this expression in bold: **na na**a n**na**a (**nana**) **na**a**na**.

The following example logs all the matches to console. Since there are 6 total matches, the total match count is 6.

```
var re = /(?:na){1,2}/g;
var count = 0;
var textString = "na naa nnaa nana naana";

var match = re.exec(textString);
while(match !== null) {
  console.log(match[0]);
  match = re.exec(textString);
  count++;
}

console.log("Total Matches: " + count);

/* Output
na
na
na
nana
na
na

Total Matches: 6
*/
```


Lookahead and Negated Lookahead

There are many situations where we are looking to match a given set of characters, but only if they are or aren't followed by another set of characters. For example, you could be looking for the word **apples** in a text but only want those matches which are followed by **are**. Consider the following sentence.

> *apples are yummy. We ate apples all day. Everyone who ate apples liked them.*

In the above example, we just want the first word as a match. Every other occurrence of the word should not be in the matches.

One way to achieve this is to use the following regular expression `a(?=b)`. The word we want to match is **a**, and the word which should come after **a** is **b**. In our case, the expression would become `/apples(?=\sare)/`. Remember that the word **are** is not included in this match.

```
var re = /apples(?=\sare)/g;
var count = 0;
var textString = "apples are yummy. We ate apples all day. Everyone who ate

var match = re.exec(textString);
while(match !== null) {
  console.log(match[0]);
  match = re.exec(textString);
  count++;
}

console.log("Total Matches: " + count);

/* Output
apples

Total Matches: 1
*/
```

This regular expression, where we look at what comes next in the string before deciding if the word is a match, is called a lookahead.

A very similar situation would arise if you decided to match **apples** only if it was *not* followed by a specific set of characters. In such cases, you need to replace `?=` with `?!` in your regular expression. If we were looking for all occurrences of **apples** which are *not* followed by **are**, we will use `/apples(?!\sare)/` as our regular expression. There will be two successful matches for our test sentence.

```
var re = /apples(?!\sare)/g;
var count = 0;
var textString = "apples are yummy. We ate apples all day. Everyone who ate

var match = re.exec(textString);
while(match !== null) {
  console.log(match[0]);
  match = re.exec(textString);
  count++;
}

console.log("Total Matches: " + count);

/* Output
apples
apples

Total Matches: 2
*/
```

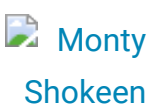
One more thing—you don't need to use two separate regular expressions to find all matches which are followed by either of two given words. All you have to do is add the pipe operator between those words, and you are good to go. For example, if you are looking for all occurrences of apple which are followed by **are** or **were**, you should use `/apples(?!\sare|\swere)/` as your regular expression.

Final Thoughts

In this tutorial, we learned how to write complicated regular expressions to match the patterns we are looking for. We can use the special `?` character to return the minimum required number of the preceding character as a match. Similarly, we can use the `?` inside parentheses to make sure that the group we were matching is not remembered.

Finally, we learned that the `?=` and `?!` character sequences in a regular expression give us the opportunity to return a particular set of characters as a match only if they are or aren't followed by another given set of characters.

If you have any questions related to this tutorial, feel free to let me know and I will do my best to explain them.



Monty Shokeen

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.