JavaScript Regex Cheat Sheet

by Monty Shokeen 21 Jun 2018 Length: Medium Languages: English

Regular Expressions JavaScript Language Fundamentals Programming Fundamentals

Successfully working with regular expressions requires you to know what each special character, flag and method does. This is a regular expressions cheat sheet which you can refer to when trying to remember how a method, special character or flag works.

Defining a Regular Expression in JavaScript

There are two ways of defining a regular expression in JavaScript.

- `var rgx = /^(\d+)/` — You can use a regular expression literal and enclose the pattern between slashes. This is evaluated at compile time and provides better performance if the regular expression stays constant.
- `var rgx = new RegExp('^(\d+)')` — The constructor function is useful when the regular expression may change programmatically. These are compiled during runtime.

Matching a Specific Set of Characters

The following sequences can be used to match a specific set of characters.

- `\w` — Matches all the words characters. Word characters are alphanumeric (a-z, A-Z characters, and underscore).

- `\W` — Matches non-word characters. Everything except alphanumeric characters and underscore.
- `\d` — Matches digit characters. Any digit from 0 to 9.
- `\D` — Matches non-digit characters. Everything except 0 to 9.
- `\s` — Matches whitespace characters. This includes spaces, tabs, and line breaks.
- `\S` — Matches all other characters except whitespace.
- `.` — Matches any character except line breaks.
- `[A-Z]` — Matches characters in a range. For example, `[A-E]` will match A, B, C, D, and E.
- `[ABC]` — Matches a character in the given set. For example, `[AMT]` will only match A, M, and T.
- `[^ABC]` — Matches all the characters *not* present in the given set. For example, `[^A-E]` will match all other characters except A, B, C, D, and E.

Specifying the Number of Characters to Match

All the expressions above will match a single character at a time. You can add quantifiers to specify how many characters should be included in the match at once.

- `+` — Matches one or more occurrences of the preceding token. For example, `\w+` will return **ABD12D** as a single match instead of six different matches.
- `*` — Matches zero or more occurrences of the preceding token. For example, `b\w*` matches the bold parts in **b**, **bat**, **bajhdsfbfjhbe**. Basically, it matches zero or more word characters after 'b'.
- `{m, n}` — Matches at least m and at most n occurrences of the previous token. `{m,}` will match at least m occurrences, and there is no upper limit to the match. `{k}` will match exactly k occurrences of the preceding token.
- `?` — Matches zero or one occurrences of the preceding character. For example, this can be useful when matching two variations of spelling for the same work. For example, `/behaviou?r/` will match both **behavior** and **behaviour**.
- `|` — Matches the expression either before or after the pipe character. For example, `/se(a|e)/` matches both see and sea.

Parenthesis-Related Regular Expressions

- `(ABC)` — This will group multiple tokens together and remember the substring matched by them for later use. This is called a capturing group.
- `(?:ABC)` — This will also group multiple tokens together but won't remember the match. It is a non-capturing group.
- `\d+(?=ABC)` — This will match the token(s) preceding the `(?=ABC)` part only if it is followed by `ABC`. The part `ABC` will not be included in the match. The `\d` part is just an example. It could be any other regular expression string.
- `\d+(?!ABC)` — This will match the token(s) preceding the `(?!ABC)` part only if it is *not* followed by `ABC`. The part `ABC` will not be included in the match. The `\d` part is just an example. It could be any other regular expression string.

Other Regular Expression Characters

There are also some other regular expression characters which have not been covered in previous sections:

- `^` — Look for the regular expression at the beginning of the string or the beginning of a line if the multiline flag is enabled.
- `$` — Look for the regular expression at the end of the string or the end of a line if the multiline flag is enabled.
- `\b` — Match the preceding token only if there is a word boundary.
- `\B` — Match the preceding token only if there is no word boundary.

Using Flags With Regular Expressions

Flags can be used to control how a regular expression should be interpreted. You can use flags either alone or together in any order you want. These are the five flags which are available in JavaScript.

- `g` — Search the string for all matches of given expression instead of returning just the first one.
- `i` — Make the search case-insensitive so that words like Apple, aPPLe and apple can all be matched at once.
- `m` — This flag will make sure that the `^` and `$` tokens look for a match at the beginning or end of each line instead of the whole string.
- `u` — This flag will enable you to use Unicode code point escapes in your regular expression.
- `y` — This will tell JavaScript to only look for a match at the current position in the target string.

You can specify flags for a regular expression in JavaScript either by adding them to the end of a regular expression literal or by passing them to the `RegExp` constructor. For example, `/cat/i` matches all occurrences of **cat** regardless of case, and `RegExp("cat", 'i')` does the same.

Useful Regular Expression Methods in JavaScript

The regular expressions that you create using the flags and character sequences we have discussed so far are meant to be used with different methods to search, replace or split a string. Here are some methods related to regular expressions.

- `test()` — Check if the main string contains a substring which matches the pattern specified by the given regular expression. It returns `true` on successful match and `false` otherwise.

```
var textA = 'I like APPles very much';
var textB = 'I like APPles';
var regexOne = /apples$/i

// Output : false
console.log(regexOne.test(textA));

// Output : true
console.log(regexOne.test(textB));
```

In the above example, the regular expression is supposed to look for the word **apples** only at the end of the string. That's why we got `false` in the first case.

- `search()` — Check if the main string contains a substring which matches the pattern specified by the given regular expression. It returns the index of the match on success and **-1** otherwise.

```
var textA = 'I like APPles very much';
var regexOne = /apples/;
var regexTwo = /apples/i;

// Output : -1
console.log(textA.search(regexOne));

// Output : 7
console.log(textA.search(regexTwo));
```

In this case, the first regular expression returned **-1** because there was no exact case-sensitive match.

- `match()` — Search if the main string contains a substring which matches the pattern specified by the given regular expression. If the `g` flag is enabled, multiple matches will be returned as an array.

```
var textA = 'All I see here are apples, APPles and apPleS';
var regexOne = /apples/gi;

// Output : [ "apples", "APPles", "apPleS" ]
console.log(textA.match(regexOne));
```

- `exec()` — Search if the main string contains a substring which matches the pattern specified by the given regular expression. The returned array will contain information about the match and capturing groups.

```
var textA = 'Do you like apples?';
var regexOne = /apples/;

// Output : apples
console.log(regexOne.exec(textA)[0]);

// Output : Do you like apples?
console.log(regexOne.exec(textA).input);
```

- `replace()` — Search for a substring matching the given pattern and replace it with the provided replacement string.

```
var textA = 'Do you like aPPles?';
var regexOne = /apples/i

// Output : Do you like mangoes?
console.log(textA.replace(regexOne, 'mangoes'));
```

- `split()` — This method will allow you to split the main string into substrings based on the separator specified as a regular expression.

```
var textA = 'This 593 string will be brok294en at places where d1gits are.';
var regexOne = /\d+/g

// Output : [ "This ", " string will be brok", "en at places where d", "gits
console.log(textA.split(regexOne))
```

Conclusion

In previous tutorials, I've covered the basics of regular expressions as well as some more complicated expressions which can prove to be useful every once in a while. These two tutorials explained how different characters or character sequences work in regular expressions.

A Beginner's Guide to Regular Expressions in JavaScript

Regular expressions let you to manipulate strings with your code. They're very

Monty Shokeen
25 May 2018

REGULAR EXPRESSIONS

JavaScript Regular Expressions: Beyond the Basics

This tutorial will teach you how to use some sophisticated regular expressions in

Monty Shokeen
05 Jun 2018

JAVASCRIPT

If regular expressions still confuse you, my advice would be to keep practicing and see how other people come up with regular expressions to make a particular pattern.