Code > Regular Expressions

JavaScript Regex Cheat Sheet

Monty Shokeen Last updated Mar 23, 2022

Read Time: 9 min | 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.

Creating a Regular Expression in JavaScript

Method	Example	Benefits
Regular Expression Literal	`/^(\d+)/`	better performance
Constructor Function	`new RegExp('^(\d+)')`	can use variables

Matching a Specific Set of Characters

Operator	Matches	Operator	Matches
`.`	any character		
`[abc]`	any character listed	`[^abc]`	any character not listed
`\w`	word characters	`\W`	non-word characters
`\d`	digits	`\D`	non-digits
`\s`	whitespace	`\S`	non-whitespace
`\b`	word boundaries	`\B`	non-word boundaries

Specifying the Number of Times to Match

Symbol	Meaning	Symbol	Meaning
`^`	beginning of the line or string	`$`	end of the line or string
`*`	zero or more repetitions	`+`	one or more repetitions
`?`	possibly exists		
`{n}`	specific number of repetitions	`{n,m}`	range of repetitions

Using Flags With Regular Expressions

Flag	Meaning	Flag	Meaning
`g`	global	`m`	multiline mode
`i`	case-insensitive	`y`	sticky search

Parenthesis in Regular Expressions

Regular Expression Methods in JavaScript

Function	Purpose
`test()`	test if a pattern matches
`search()`	find a matching pattern
`match()` and `exec()`	get matches
`replace()`	replace matched text
`split()`	split on matches

Creating a Regular Expression in JavaScript

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

Regular Expression Literal

`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.

Constructor Function

`var rgx = new RegExp('^(\d+)')` : The constructor function is useful when the regular expression may change programmatically. These are compiled during runtime.

[back]

Matching a Specific Set of Characters

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

Any Character `.`

The period matches any character except the newline `'\n'`.

Expression	String	Result
`..`	`a`	no matches
	`ab`	1 match: `ab`
	`abc`	1 match: `ab`
	`abcde`	2 matches: `ab` and `cd`

Custom Character Groups `[...]`

The square brackets are used to specify a collection of characters. `[abc]` would match the letter `a`, `b`, or `c`. `[a-z]` would match any letter between `a` and `z`.

With a `^` symbol, the square brackets match any character not listed. For example, `[^0-9]` matches any non-digit character.

Expression	String	Result
`[abc]`	`a`	match: `a`

`[abc]`	`ac`	matches: `a` and `c`
`[abc]`	`Hello!`	no matches
`[abc]`	`The boy is happy.`	matches: `b` and `a`
`[^0-9]`	`user1234`	matches: `u`, `s`, `e`, and `r`

Word Characters `\w` and `\W`

`\w` matches word characters. Word characters are alphanumeric (a-z, A-Z characters, and underscore). `\W` matches non-word characters: everything except alphanumeric characters and underscore.

Expression	String	Result
`\w`	`12"&; :c`	3 matches: `1`, `2`, and `c`
`\w`	`%>"!`	no matches
`\W`	`12"&; :c`	4 matches: `"`, `&`, `:`, and `;`
`\W`	`tutorial`	no matches

Digit Characters `\d` and `\D`

`\d` matches digit characters—any digit from `0` to `9`. It's equivalent to `[0-9]` in bracket notation. `\D` matches non-digit characters, i.e. everything except `0` to `9`. It's equivalent to `[^0-9]` in bracket notation.

Expression	String	Result
`\d\d\d`	123tea4	1 match: `123`
\d	tutorial	no match

\D	12tea3"4	4 matches: `t`, `e`, `a` ,and `"`
\D	123	no match

Whitespace Characters `\s` and `\S`

`\s` matches whitespace characters. This includes spaces, tabs, and line breaks. Whitespace is equivalent to `[\t\n\r\f\v]` with bracket notation. `\S` matches all other characters except whitespace. It's equivalent to `[^ \t\n\r\f\v]`.

Expression	String	Result
`\s`	`a b c`	2 matches
`\s`	`tutorial`	no match
`\S`	`a b c`	3 matches: `a`, `b`, and `c`

Beginning or End of Word `\b` or Not `\B`

`\b` matches the beginning or end of a word.

Expression	String	Result
`\bcup`	`a cupboard`	match: `cup`
`cup\b`	`the cupboard`	no match
`cup\b`	`the teacup is yellow`	match: `cup`
`\bcup`	`englishcupoftea`	no match

`\B` matches only within a word.

Expression	String	Result

Expression	String	Result
`\Bcup`	`cupboard`	no match
`cup\B`	`cupboard`	match: `cup`
`cup\B`	`teacup`	no match

Caret `^`

The caret symbol `^` matches the beginning of the string.

Expression	String	Result
`^ab`	`a`	no match
`^ab`	`ab`	match: `ab`
`^ab`	`abc`	match: `ab`
`^ab`	`acb`	no match

Dollar `$`

The dollar symbol `$` matches the end of the string.

Expression	String	Result
`ab$`	`a`	no match
`ab$`	`ab`	match: `ab`
`ab$`	`abc`	no match

[back]

[Back]

Specifying the Number of Times to Match

All the meta-characters given below can be used to evaluate patterns based on the number of times a character is repeated. You can add quantifiers to specify how many characters should be included in the match at once.

Star `*`

The `*` symbol is used to check if there are zero or more occurrences of the pattern to the left of the symbol.

Expression	String	Result
`ab*c`	`ac`	match: `ac`
`ab*c`	`abbc`	match: `abbc`
`ab*c`	`ad`	no match
`ab*c`	`abd`	no match

Plus `+`

The `+` symbol is used to check if there is at least one occurrence of the pattern to the left of the symbol.

Expression	String	Result
`ab+c`	`ac`	no match

`ab+c`	`abc`	match: `abc`
`ab+c`	`dabbcd`	match: `abbc`

Question `?`

The `?` symbol is used to check if there is zero or one occurrence of the pattern to the left of the symbol.

Expression	String	Result
`ab?c`	`ac`	match: `ac`
`ab+c`	`abbc`	no match
`ab+c`	`abc`	match: `abc`

Number of Repetitions `{n}` or `{n,m}`

Braces with a single number denote a specific number of repetitions. If the brace has two numbers, in the format `{n,m}`, `n` denotes the minimum, and `m` denotes the maximum number of repetitions of the pattern to the left.

Expression	String	Result
`a{2,3}`	`abc def`	no match
`a{2,3}`	`abc daad`	1 match: `aa`
`a{2,3}`	`aabc daaad`	2 matches: `aa` and `aaa`
`[0-9]{2,4}`	`4`	no match
`[0-9]{2,4}`	`12 here 345678 there`	3 matches: `12`, `3456`, and `78`

[back]

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.

Global `g`

`g`: search the string for all matches of a given expression instead of returning just the first one.

Case-Insensitive `i`

`i`: make the search case-insensitive so that words like apple, Apple, and APPLE can all be matched at once.

Multiline Mode `m`

`m`: make the `^` and `$` tokens look for a match at the beginning or end of each line instead of the whole string.

Unicode Mode `u`

`u`: enable Unicode code point escapes in your regular expression. Not all browsers support this option.

```
var regex = new regexp('\u{61}', 'u');
console.log(regex.unicode); // true
```

Sticky Search `y`

`y`: only look for a match at the current position in the target string. `lastindex` indicates where the search will begin.

[back]

Parenthesis in Regular Expressions

The parenthesis is used for grouping. Sub patterns can be formed using `()`.

Expression	String	Result
`(ab)+`	`ab`	match: `ab`
	`ababcd`	match: `abab`
	`cabaacd`	match: `ab`

You can also combine the or `|` operator with `()` to combine pick from a number of possible patterns.

Expression	String	Result
`([0-9]|[x])+`	`a123x3b`	match: `123x3`
	`axd`	match: `x`
	`c9x23d3`	matches: `9x23` and `3`

Finally, the parts of a matching string that match some pattern in brackets will be broken out and returned separately. This means you can use regular expressions to extract or modify data from a string.

It's with nested patterns that the power of regular expressions really starts to become apparent.

[back]

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 JavaScript methods to search, replace, or split a string. Here are some methods related to regular expressions.

Test If a Pattern Matches `test()`

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

`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.

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));
```

Get Matches `match()` and `exec()`

`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 Matched Text `replace()`

`replace()` search for a match in a string and then replace the matched pattern with it