Regular expressions

Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects. These patterns are used with the `exec()` and `test()` methods of `RegExp`, and with the `match()`, `matchAll()`, `replace()`, `replaceAll()`, `search()`, and `split()` methods of `String`. This chapter describes JavaScript regular expressions. It provides a brief overview of each syntax element. For a detailed explanation of each one's semantics, read the regular expressions reference.

Creating a regular expression

You construct a regular expression in one of two ways:

- Using a regular expression literal, which consists of a pattern enclosed between slashes, as follows:

  Regular expression literals provide compilation of the regular expression when the script is loaded. If the regular expression remains constant, using this can improve performance.

- Or calling the constructor function of the `RegExp` object, as follows:

  JS
  ```
  const re = new RegExp("ab+c");
  ```

  Using the constructor function provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input.

Writing a regular expression pattern

A regular expression pattern is composed of simple characters, such as `/abc/`, or a combination of simple and special characters, such as `/ab*c/` or `/Chapter (\d+)\.\d*/`. The last example includes parentheses, which are used as a memory device. The match made with this part of the pattern is remembered for later use, as described in Using groups.

Using simple patterns

Simple patterns are constructed of characters for which you want to find a direct match. For example, the pattern `/abc/` matches character combinations in strings only when the exact sequence `"abc"` occurs (all characters together and in that order). Such a match would succeed in the strings `"Hi, do you know your abc's?"` and `"The latest airplane designs evolved from slabcraft."`. In both cases the match is with the substring `"abc"`. There is no match in the string `"Grab crab"` because while it contains the substring `"ab c"`, it does not contain the exact substring `"abc"`.

Using special characters

When the search for a match requires something more than a direct match, such as finding one or more b's, or finding white space, you can include special characters in the pattern. For example, to match *a single `"a"` followed by zero or more `"b"`s followed by `"c"`*, you'd use the pattern `/ab*c/`: the `*` after `"b"` means "0 or more occurrences of the preceding item." In the string `"cbbabbbbcdebc"`, this pattern will match the substring `"abbbbc"`.

The following pages provide lists of the different special characters that fit into each category, along with descriptions and examples.

Assertions guide

Assertions include boundaries, which indicate the beginnings and endings of lines and words, and other patterns indicating in some way that a match is possible (including look-ahead, look-behind, and conditional expressions).

Character classes guide

Distinguish different types of characters. For example, distinguishing between letters and digits.

Groups and backreferences guide

Groups group multiple patterns as a whole, and capturing groups provide extra submatch information when using a regular expression pattern to match against a string. Backreferences refer to a previously captured group in the same regular expression.

Quantifiers guide

Indicate numbers of characters or expressions to match.

If you want to look at all the special characters that can be used in regular expressions in a single table, see the following:

Special characters in regular expressions.

Characters / constructs	Corresponding article
`[xyz]`, `[^xyz]`, `.`, `\d`, `\D`, `\w`, `\W`, `\s`, `\S`, `\t`, `\r`, `\n`, `\v`, `\f`, `[\b]`, `\0`, `\cX`, `\xhh`, `\uhhhh`, `\u{hhhh}`, `x|y`	Character classes
`^`, `$`, `\b`, `\B`, `x(?=y)`, `x(?!y)`, `(?<=y)x`, `(?<!y)x`	Assertions
`(x)`, `(?<Name>x)`, `(?:x)`, `\n`, `\k<Name>`	Groups and backreferences
`x*`, `x+`, `x?`, `x{n}`, `x{n,}`, `x{n,m}`	Quantifiers

Note: A larger cheat sheet is also available (only aggregating parts of those individual articles).

Escaping

If you need to use any of the special characters literally (actually searching for a `"*"`, for instance), you must escape it by putting a backslash in front of it. For instance, to search for `"a"` followed by `"*"` followed by `"b"`, you'd use `/a\*b/` — the backslash "escapes" the `"*"`, making it literal instead of special.

Similarly, if you're writing a regular expression literal and need to match a slash ("/"), you need to escape that (otherwise, it terminates the pattern). For instance, to search for the string "/example/" followed by one or more alphabetic characters, you'd use `/\/example\/[a-z]+/i`—the backslashes before each slash make them literal.

To match a literal backslash, you need to escape the backslash. For instance, to match the string "C:\" where "C" can be any letter, you'd use `/[A-Z]:\\/` — the first backslash escapes the one after it, so the expression searches for a single literal backslash.

If using the `RegExp` constructor with a string literal, remember that the backslash is an escape in string literals, so to use it in the regular expression, you need to escape it at the string literal level. `/a\*b/` and `new RegExp("a\\*b")` create the same expression, which searches for "a" followed by a literal "*" followed by "b".

The `RegExp.escape()` function returns a new string where all special characters in regex syntax are escaped. This allows you to do `new RegExp(RegExp.escape("a*b"))` to create a regular expression that matches only the string `"a*b"`.

Using parentheses

Parentheses around any part of the regular expression pattern causes that part of the matched substring to be remembered. Once remembered, the substring can be recalled for other use. See Groups and backreferences for more details.

Using regular expressions in JavaScript

Regular expressions are used with the `RegExp` methods `test()` and `exec()` and with the `String` methods `match()`, `matchAll()`, `replace()`, `replaceAll()`, `search()`, and `split()`.

Method	Description
`exec()`	Executes a search for a match in a string. It returns an array of information or `null` on a mismatch.
`test()`	Tests for a match in a string. It returns `true` or `false`.
`match()`	Returns an array containing all of the matches, including capturing groups, or `null` if no match is found.
`matchAll()`	Returns an iterator containing all of the matches, including capturing groups.
`search()`	Tests for a match in a string. It returns the index of the match, or `-1` if the search fails.
`replace()`	Executes a search for a match in a string, and replaces the matched substring with a replacement substring.
`replaceAll()`	Executes a search for all matches in a string, and replaces the matched substrings with a replacement substring.
`split()`	Uses a regular expression or a fixed string to break a string into an array of substrings.

When you want to know whether a pattern is found in a string, use the `test()` or `search()` methods; for more information (but slower execution) use the `exec()` or `match()` methods. If you use `exec()` or `match()` and if the match succeeds, these methods return an array and update properties of the associated regular expression object and also of the predefined regular expression object, `RegExp`. If the match fails, the `exec()` method returns `null` (which coerces to `false`).

In the following example, the script uses the `exec()` method to find a match in a string.

JS

```
const myRe = /d(b+)d/g;
const myArray = myRe.exec("cdbbdbsbz");
```

If you do not need to access the properties of the regular expression, an alternative way of creating `myArray` is with this script:

JS

```js
const myArray = /d(b+)d/g.exec("cdbbdbsbz");
// similar to 'cdbbdbsbz'.match(/d(b+)d/g); however,
// 'cdbbdbsbz'.match(/d(b+)d/g) outputs [ "dbbd" ]
// while /d(b+)d/g.exec('cdbbdbsbz') outputs [ 'dbbd', 'bb', index: 1, input:
'cdbbdbsbz' ]
```

(See Using the global search flag with `exec()` for further info about the different behaviors.)

If you want to construct the regular expression from a string, yet another alternative is this script:

JS

```js
const myRe = new RegExp("d(b+)d", "g");
const myArray = myRe.exec("cdbbdbsbz");
```

With these scripts, the match succeeds and returns the array and updates the properties shown in the following table.

Results of regular expression execution.

Object	Property or index	Description	In this example
`myArray`		The matched string and all remembered substrings.	`['dbbd', 'bb', index: 1, input: 'cdbbdbsbz']`
	`index`	The 0-based index of the match in the input string.	`1`
	`input`	The original string.	`'cdbbdbsbz'`
	`[0]`	The last matched characters.	`'dbbd'`

Object	Property or index	Description	In this example
`myRe`	`lastIndex`	The index at which to start the next match. (This property is set only if the regular expression uses the g option, described in Advanced Searching With Flags.)	`5`
	`source`	The text of the pattern. Updated at the time that the regular expression is created, not executed.	`'d(b+)d'`

As shown in the second form of this example, you can use a regular expression created with an object initializer without assigning it to a variable. If you do, however, every occurrence is a new regular expression. For this reason, if you use this form without assigning it to a variable, you cannot subsequently access the properties of that regular expression. For example, assume you have this script:

JS

```
const myRe = /d(b+)d/g;
const myArray = myRe.exec("cdbbdbsbz");
console.log(`The value of lastIndex is ${myRe.lastIndex}`);

// "The value of lastIndex is 5"
```

However, if you have this script:

JS

```
const myArray = /d(b+)d/g.exec("cdbbdbsbz");
console.log(`The value of lastIndex is ${/d(b+)d/g.lastIndex}`);

// "The value of lastIndex is 0"
```

The occurrences of `/d(b+)d/g` in the two statements are different regular expression objects and hence have different values for their `lastIndex` property. If you need to access the properties of a regular expression created with an object initializer, you should first assign it to a variable.

Advanced searching with flags

Regular expressions have optional flags that allow for functionality like global searching and case-insensitive searching. These flags can be used separately or together in any order, and are included as part of the regular expression.

Flag	Description	Corresponding property
`d`	Generate indices for substring matches.	`hasIndices`
`g`	Global search.	`global`
`i`	Case-insensitive search.	`ignoreCase`
`m`	Makes `^` and `$` match the start and end of each line instead of those of the entire string.	`multiline`
`s`	Allows `.` to match newline characters.	`dotAll`
`u`	"Unicode"; treat a pattern as a sequence of Unicode code points.	`unicode`
`v`	An upgrade to the `u` mode with more Unicode features.	`unicodeSets`
`y`	Perform a "sticky" search that matches starting at the current position in the target string.	`sticky`

To include a flag with the regular expression, use this syntax:

JS

```js
const re = /pattern/flags;
```

or

JS

```js
const re = new RegExp("pattern", "flags");
```

Note that the flags are an integral part of a regular expression. They cannot be added or removed later.

For example, `re = /\w+\s/g` creates a regular expression that looks for one or more characters followed by a space, and it looks for this combination throughout the string.

JS

```js
const re = /\w+\s/g;
const str = "fee fi fo fum";
const myArray = str.match(re);
console.log(myArray);

// ["fee ", "fi ", "fo "]
```

You could replace the line:

JS

```js
const re = /\w+\s/g;
```

with:

JS

```js
const re = new RegExp("\\w+\\s", "g");
```

and get the same result.

The `m` flag is used to specify that a multiline input string should be treated as multiple lines. If the `m` flag is used, `^` and `$` match at the start or end of any line within the input string instead of the start or end of the entire string.

The `i`, `m`, and `s` flags can be enabled or disabled for specific parts of a regex using the modifier syntax.

Using the global search flag with exec()

`RegExp.prototype.exec()` method with the `g` flag returns each match and its position iteratively.

JS

```
const str = "fee fi fo fum";
const re = /\w+\s/g;

console.log(re.exec(str)); // ["fee ", index: 0, input: "fee fi fo fum"]
console.log(re.exec(str)); // ["fi ", index: 4, input: "fee fi fo fum"]
console.log(re.exec(str)); // ["fo ", index: 7, input: "fee fi fo fum"]
console.log(re.exec(str)); // null
```

In contrast, `String.prototype.match()` method returns all matches at once, but without their position.

JS

```
console.log(str.match(re)); // ["fee ", "fi ", "fo "]
```

Using unicode regular expressions

The `u` flag is used to create "unicode" regular expressions; that is, regular expressions which support matching against unicode text. An important feature that's enabled in unicode mode is Unicode property escapes. For example, the following regular expression might be used to match against an arbitrary unicode "word":

JS

```
/\p{L}*/u;
```

Unicode regular expressions have different execution behavior as well. `RegExp.prototype.unicode` contains more explanation about this.

Examples

Note: Several examples are also available in:

- The reference pages for `exec()`, `test()`, `match()`, `matchAll()`, `search()`, `replace()`, `split()`
- The guide articles: character classes, assertions, groups and backreferences, quantifiers

Using special characters to verify input

In the following example, the user is expected to enter a phone number. When the user presses the "Check" button, the script checks the validity of the number. If the number is valid (matches the character sequence specified by the regular expression), the script shows a message thanking the user and confirming the number. If the number is invalid, the script informs the user that the phone number is not valid.

The regular expression looks for:

1. the beginning of the line of data: `^`
2. followed by three numeric characters `\d{3}` OR `|` a left parenthesis `\(`, followed by three digits `\d{3}`, followed by a close parenthesis `\)`, in a non-capturing group `(?:)`
3. followed by one dash, forward slash, or decimal point in a capturing group `()`
4. followed by three digits `\d{3}`
5. followed by the match remembered in the (first) captured group `\1`
6. followed by four digits `\d{4}`
7. followed by the end of the line of data: `$`

HTML

HTML Play

```html
<p>
  Enter your phone number (with area code) and then click "Check".
  <br />
  The expected format is like ###-###-####.
</p>
<form id="form">
  <input id="phone" />
  <button type="submit">Check</button>
</form>
<p id="output"></p>
```

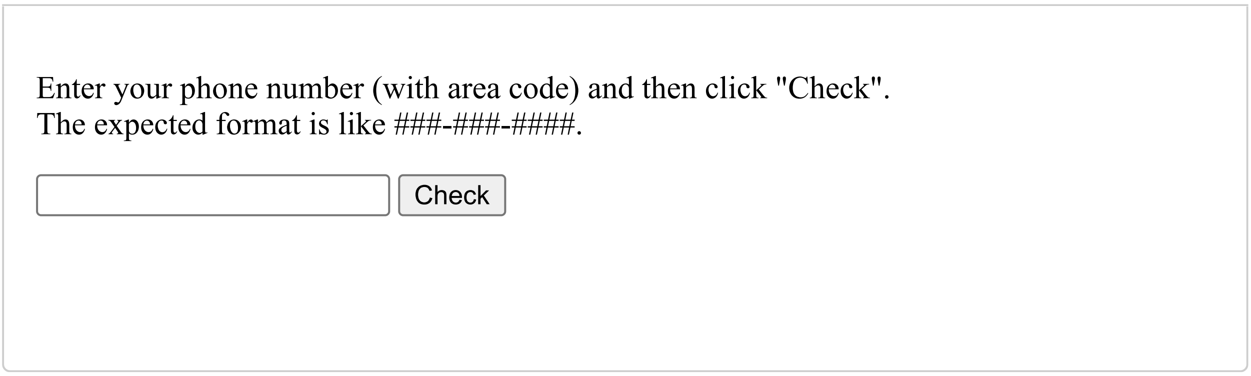
JavaScript

JS Play

```
const form = document.querySelector("#form");
const input = document.querySelector("#phone");
const output = document.querySelector("#output");

const re = /^(?:\d{3}|\(\d{3}\))([-/.])\d{3}\1\d{4}$/;

function testInfo(phoneInput) {
  const ok = re.exec(phoneInput.value);

  output.textContent = ok
    ? `Thanks, your phone number is ${ok[0]}`
    : `${phoneInput.value} isn't a phone number with area code!`;
}

form.addEventListener("submit", (event) => {
  event.preventDefault();
  testInfo(input);
});
```

Result

Play


Enter your phone number (with area code) and then click "Check".
The expected format is like ###-###-####.

Tools

[RegExr](#)

An online tool to learn, build, & test Regular Expressions.

[Regex tester](#)

An online regex builder/debugger

[Regex interactive tutorial](#)

An online interactive tutorials, Cheat sheet, & Playground.

Regex visualizer

An online visual regex tester.