RegExp

In This Article

The `RegExp` constructor creates a regular expression object for matching text with a pattern.

For an introduction to regular expressions, read the Regular Expressions chapter in the JavaScript Guide.

Syntax

Literal, constructor, and factory notations are possible:

```
/pattern/flags
new RegExp(pattern[, flags])
RegExp(pattern[, flags])
```

Parameters

`pattern`
The text of the regular expression.

`flags`
If specified, flags can have any combination of the following values:

`g`
global match; find all matches rather than stopping after the first match

`i`

ignore case

`m`

multiline; treat beginning and end characters (^ and $) as working over multiple lines (i.e., match the beginning or end of *each* line (delimited by \n or \r), not only the very beginning or end of the whole input string)

`u`

unicode; treat pattern as a sequence of unicode code points

`y`

sticky; matches only from the index indicated by the `lastIndex` property of this regular expression in the target string (and does not attempt to match from any later indexes).

Description

There are 2 ways to create a `RegExp` object: a literal notation and a constructor. To indicate strings, the parameters to the literal notation do not use quotation marks while the parameters to the constructor function do use quotation marks. So the following expressions create the same regular expression:

```
/ab+c/i;
new RegExp('ab+c', 'i');
new RegExp(/ab+c/, 'i');
```

The literal notation provides compilation of the regular expression when the expression is evaluated. Use literal notation when the regular expression will remain constant. For example, if you use literal notation to construct a regular expression used in a loop, the regular expression won't be recompiled on each iteration.

The constructor of the regular expression object, for example, `new RegExp('ab+c')`, 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.

Starting with ECMAScript 6, `new RegExp(/ab+c/, 'i')` no longer throws a `TypeError` ("can't supply flags when constructing one RegExp from another") when the first argument is a `RegExp` and the second `flags` argument is present. A new `RegExp` from the arguments is created instead.

When using the constructor function, the normal string escape rules (preceding special characters with \ when included in a string) are necessary. For example, the following are equivalent:

```
var re = /\w+/;
var re = new RegExp('\\w+');
```

Special characters meaning in regular expressions

- Character Classes
- Character Sets
- Boundaries
- Alternation
- Grouping and back references
- Quantifiers
- Assertions

Character Classes	
Character	Meaning
`.`	(The dot, the decimal point) matches any single character *except* line terminators: `\n`, `\r`, `\u2028` or `\u2029`. Inside a character set, the dot loses its special meaning and matches a literal dot. Note that the `m` multiline flag doesn't change the dot behavior. So to match a pattern across multiple lines, the character set `[^]` can be used (if you don't mean an old version of IE, of course), it will match any character including newlines. For example, `/.y/` matches "my" and "ay", but not "yes", in "yes make my day".
`\d`	Matches any digit (Arabic numeral). Equivalent to `[0-9]`. For example, `/\d/` or `/[0-9]/` matches "2" in "B2 is the suite number".
`\D`	Matches any character that is not a digit (Arabic numeral). Equivalent to `[^0-9]`. For example, `/\D/` or `/[^0-9]/` matches "B" in "B2 is the suite number".
`\w`	Matches any alphanumeric character from the basic Latin alphabet, including the underscore. Equivalent to `[A-Za-z0-9_]`. For example, `/\w/` matches "a" in "apple", "5" in "$5.28", and "3" in "3D".
`\W`	Matches any character that is not a word character from the basic Latin alphabet. Equivalent to `[^A-Za-z0-9_]`.

	For example, `/\W/` or `/[^A-Za-z0-9_]/` matches "%" in "50%".
`\s`	Matches a single white space character, including space, tab, form feed, line feed and other Unicode spaces. Equivalent to `[ \f\n\r\t\v\u00a0\u1680\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]`. For example, `/\s\w*/` matches " bar" in "foo bar".
`\S`	Matches a single character other than white space. Equivalent to `[^ \f\n\r\t\v\u00a0\u1680\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]`. For example, `/\S\w*/` matches "foo" in "foo bar".
`\t`	Matches a horizontal tab.
`\r`	Matches a carriage return.
`\n`	Matches a linefeed.
`\v`	Matches a vertical tab.
`\f`	Matches a form-feed.
`[\b]`	Matches a backspace. (Not to be confused with `\b`)
`\0`	Matches a NUL character. Do not follow this with another digit.
`\cX`	Where `X` is a letter from A - Z. Matches a control character in a string. For example, `/\cM/` matches control-M in a string.
`\xhh`	Matches the character with the code `hh` (two hexadecimal digits).
`\uhhhh`	Matches a UTF-16 code-unit with the value `hhhh` (four hexadecimal digits).
`\u{hhhh}or\u{hhhhh}`	(only when u flag is set) Matches the character with the Unicode value U+`hhhh or` U+`hhhhh` (hexadecimal digits).
`\`	For characters that are usually treated literally, indicates that the next character is special and not to be interpreted literally. For example, `/b/` matches the character "b". By placing a backslash in front of "b", that is by using `/\b/`, the character becomes special to mean match a word boundary. *or*

	For characters that are usually treated specially, indicates that the next character is not special and should be interpreted literally. For example, "*" is a special character that means 0 or more occurrences of the preceding character should be matched; for example, `/a*/` means match 0 or more "a"s. To match `*` literally, precede it with a backslash; for example, `/a\*/` matches "a*".
Character Sets	
Character	Meaning

`[xyz]`
`[a-c]`
	A character set. Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first or last character enclosed in the square brackets it is taken as a literal hyphen to be included in the character set as a normal character. It is also possible to include a character class in a character set.
	
	For example, `[abcd]` is the same as `[a-d]`. They match the "b" in "brisket" and the "c" in "chop".
	
	For example, [abcd-] and [-abcd] match the "b" in "brisket", the "c" in "chop" and the "-" (hyphen) in "non-profit".
	
	For example, [\w-] is the same as [A-Za-z0-9_-]. They match the "b" in "brisket", the "c" in "chop" and the "n" in "non-profit".

`[^xyz]`
`[^a-c]`
	A negated or complemented character set. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first or last character enclosed in the square brackets it is taken as a literal hyphen to be included in the character set as a normal character.
	
	For example, `[^abc]` is the same as `[^a-c]`. They initially match "o" in "bacon" and "h" in "chop".

Alternation	
Character	Meaning
`x|y`	Matches either `x` or `y`. For example, `/green|red/` matches "green" in "green apple" and "red" in "red apple".
Boundaries	
Character	Meaning

`^`	Matches beginning of input. If the multiline flag is set to true, also matches immediately after a line break character. For example, `/^A/` does not match the "A" in "an A", but does match the first "A" in "An A".
`$`	Matches end of input. If the multiline flag is set to true, also matches immediately before a line break character. For example, `/t$/` does not match the "t" in "eater", but does match it in "eat".
`\b`	Matches a word boundary. This is the position where a word character is not followed or preceded by another word-character, such as between a letter and a space. Note that a matched word boundary is not included in the match. In other words, the length of a matched word boundary is zero. Examples: `/\bm/` matches the 'm' in "moon" ; `/oo\b/` does not match the 'oo' in "moon", because 'oo' is followed by 'n' which is a word character; `/oon\b/` matches the 'oon' in "moon", because 'oon' is the end of the string, thus not followed by a word character; `/\w\b\w/` will never match anything, because a word character can never be followed by both a non-word and a word character.
`\B`	Matches a non-word boundary. This is a position where the previous and next character are of the same type: Either both must be words, or both must be non-words. Such as between two letters or between two spaces. The beginning and end of a string are considered non-words. Same as the matched word boundary, the matched non-word bondary is also not included in the match. For example, `/\Bon/` matches "on" in "at noon", and `/ye\B/`matches "ye" in "possibly yesterday".
Grouping and back references	
Character	Meaning
`(x)`	Matches `x` and remembers the match. These are called capturing groups. For example, `/(foo)/` matches and remembers "foo" in "foo bar". The capturing groups are numbered according to the order of left parentheses of capturing groups, starting from 1. The matched substring can be recalled from the resulting

	array's elements `[1], ..., [n]` or from the predefined `RegExp`object's properties `$1, ..., $9`. Capturing groups have a performance penalty. If you don't need the matched substring to be recalled, prefer non-capturing parentheses (see below).
`\n`	Where `n` is a positive integer. A back reference to the last substring matching the n parenthetical in the regular expression (counting left parentheses). For example, `/apple(,)\sorange\1/` matches "apple, orange," in "apple, orange, cherry, peach". A more complete example follows this table.
`(?:x)`	Matches `x` but does not remember the match. These are called non-capturing groups. The matched substring can not be recalled from the resulting array's elements `[1], ..., [n]` or from the predefined `RegExp` object's properties `$1, ..., $9`.
Quantifiers	
Character	Meaning
`x*`	Matches the preceding item *x* 0 or more times. For example, `/bo*/` matches "boooo" in "A ghost booooed" and "b" in "A bird warbled", but nothing in "A goat grunted".
`x+`	Matches the preceding item *x* 1 or more times. Equivalent to `{1,}`. For example, `/a+/` matches the "a" in "candy" and all the "a"'s in "caaaaaaandy".
`x?`	Matches the preceding item *x* 0 or 1 time. For example, `/e?le?/` matches the "el" in "angel" and the "le" in "angle." If used immediately after any of the quantifiers `*`, `+`, `?`, or `{}`, makes the quantifier non-greedy (matching the minimum number of times), as opposed to the default, which is greedy (matching the maximum number of times).
`x{n}`	Where `n` is a positive integer. Matches exactly `n` occurrences of the preceding item *x*. For example, `/a{2}/` doesn't match the "a" in "candy", but it matches all of the "a"'s in "caandy", and the first two

	"a"'s in "caaandy".
`x{n,}`	Where `n` is a positive integer. Matches at least `n` occurrences of the preceding item *x*. For example, `/a{2,}/` doesn't match the "a" in "candy", but matches all of the a's in "caandy" and in "caaaaaaandy".
`x{n,m}`	Where `n` and `m` are positive integers. Matches at least `n` and at most `m` occurrences of the preceding item *x*. For example, `/a{1,3}/` matches nothing in "cndy", the "a" in "candy", the two "a"'s in "caandy", and the first three "a"'s in "caaaaaaandy". Notice that when matching "caaaaaaandy", the match is "aaa", even though the original string had more "a"'s in it.

`x*?`
`x+?`
`x??`
`x{n}?`
`x{n,}?`
`x{n,m}?`
	Matches the preceding item *x* like `*`, `+`, `?`, and `{...}` from above, however the match is the smallest possible match.
	
	For example, `/<.*?>/` matches "<foo>" in "<foo> <bar>", whereas `/<.*>/` matches "<foo> <bar>".
	
	Quantifiers without `?` are said to be greedy. Those with `?` are called "non-greedy".

Assertions	
Character	Meaning
`x(?=y)`	Matches `x` only if `x` is followed by `y`. For example, `/Jack(?=Sprat)/` matches "Jack" only if it is followed by "Sprat". `/Jack(?=Sprat|Frost)/` matches "Jack" only if it is followed by "Sprat" or "Frost". However, neither "Sprat" nor "Frost" is part of the match results.
`x(?!y)`	Matches `x` only if `x` is not followed by `y`. For example, `/\d+(?!\.)/` matches a number only if it is not followed by a decimal point. `/\d+(?!\.)/.exec('3.141')` matches "141" but not "3.141".

Properties

`RegExp.prototype`
  Allows the addition of properties to all objects.

`RegExp.length`
  The value of `RegExp.length` is 2.

`get RegExp[@@species]`
  The constructor function that is used to create derived objects.

`RegExp.lastIndex`
  The index at which to start the next match.

Methods

The global `RegExp` object has no methods of its own, however, it does inherit some methods through the prototype chain.

`RegExp` prototype objects and instances

Properties

See also deprecated `RegExp` properties.

Note that several of the `RegExp` properties have both long and short (Perl-like) names. Both names always refer to the same value. Perl is the programming language from which JavaScript modeled its regular expressions.

`RegExp.prototype.constructor`
  Specifies the function that creates an object's prototype.

`RegExp.prototype.flags`
  A string that contains the flags of the `RegExp` object.

`RegExp.prototype.global`
  Whether to test the regular expression against all possible matches in a string, or only against the first.

`RegExp.prototype.ignoreCase`
  Whether to ignore case while attempting a match in a string.

`RegExp.prototype.multiline`
  Whether or not to search in strings across multiple lines.

`RegExp.prototype.source`
  The text of the pattern.

`RegExp.prototype.sticky`
  Whether or not the search is sticky.

`RegExp.prototype.unicode`
  Whether or not Unicode features are enabled.

Methods

`RegExp.prototype.compile()`
(Re-)compiles a regular expression during execution of a script.

`RegExp.prototype.exec()`
Executes a search for a match in its string parameter.

`RegExp.prototype.test()`
Tests for a match in its string parameter.

`RegExp.prototype[@@match]()`
Performs match to given string and returns match result.

`RegExp.prototype[@@replace]()`
Replaces matches in given string with new substring.

`RegExp.prototype[@@search]()`
Searches the match in given string and returns the index the pattern found in the string.

`RegExp.prototype[@@split]()`
Splits given string into an array by separating the string into substring.

`RegExp.prototype.toSource()`
Returns an object literal representing the specified object; you can use this value to create a new object. Overrides the `Object.prototype.toSource()` method.

`RegExp.prototype.toString()`
Returns a string representing the specified object. Overrides the `Object.prototype.toString()` method.

Examples

Using a regular expression to change data format

The following script uses the `replace()` method of the `String` instance to match a name in the format *first last* and output it in the format *last, first*. In the replacement text, the script uses `$1` and `$2` to indicate the results of the corresponding matching parentheses in the regular expression pattern.

```
var re = /(\w+)\s(\w+)/;
var str = 'John Smith';
var newstr = str.replace(re, '$2, $1');
console.log(newstr);
```

This displays "Smith, John".

Using regular expression to split lines with different line endings/ends of line/line breaks

The default line ending varies depending on the platform (Unix, Windows, etc.). The line splitting provided in this example works on all platforms.

```
var text = 'Some text\nAnd some more\r\nAnd yet\rThis
var lines = text.split(/\r\n|\r|\n/);
console.log(lines); // logs [ 'Some text', 'And some m
```

Note that the order of the patterns in the regular expression matters.

Using regular expression on multiple lines

```
var s = 'Please yes\nmake my day!';
s.match(/yes.*day/);
// Returns null
s.match(/yes[^]*day/);
// Returns ["yes\nmake my day"]
```

Using a regular expression with the sticky flag

The sticky flag indicates that the regular expression performs sticky matching in the target string by attempting to match starting at `RegExp.prototype.lastIndex`.

```
var str = '#foo#';
var regex = /foo/y;

regex.lastIndex = 1;
regex.test(str); // true
regex.lastIndex = 5;
regex.test(str); // false (lastIndex is taken into acc
regex.lastIndex; // 0 (reset after match failure)
```

Regular expression and Unicode characters

As mentioned above, `\w` or `\W` only matches ASCII based characters; for example, "a" to "z", "A" to "Z", "0" to "9" and "_". To match characters from other languages such as Cyrillic or Hebrew, use `\uhhhh`, where "hhhh" is the character's Unicode value in hexadecimal. This example demonstrates how one can separate out Unicode characters from a word.

```
var text = 'Образец text на русском языке';
var regex = /[\u0400-\u04FF]+/g;

var match = regex.exec(text);
console.log(match[0]);        // logs 'Образец'
console.log(regex.lastIndex); // logs '7'

var match2 = regex.exec(text);
console.log(match2[0]);       // logs 'на' [did not lo
console.log(regex.lastIndex); // logs '15'

// and so on
```

Here's an external resource for getting the complete Unicode block range for different scripts: Regexp-unicode-block.

Extracting sub-domain name from URL

```
var url = 'http://xxx.domain.com';
console.log(/[^.]+/.exec(url)[0].substr(7)); // logs '
```

Specifications

Specification	Status	Comment
ECMAScript 3rd Edition (ECMA-262)	Standard	Initial definition. Implemented in JavaScript 1.1.
ECMAScript 5.1 (ECMA-262) The definition of 'RegExp' in that specification.	Standard	
ECMAScript 2015 (6th Edition, ECMA-262) The definition of 'RegExp' in that specification.	Standard	The `RegExp` constructor no longer throws when the first argument is a `RegExp` and the second argument is present. Introduces Unicode and sticky flags.
ECMAScript Latest Draft (ECMA-262) The definition of 'RegExp' in that specification.	Living Standard	

Browser compatibility

Desktop Mobile

Feature	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari
Basic Support	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`compile`	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`exec`	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`flags`	(Yes)	(Yes)	37	No	(Yes)	(Yes)
`global`	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`ignoreCase`	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`RegExp.input` (`$_`)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`lastIndex`	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`RegExp.lastMatch` (`$&`)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`RegExp.lastParen` (`$+`)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`RegExp.leftContext`(`` $` ``)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`multiline`	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`RegExp.$1-$9`	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`prototype`	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`RegExp.rightContext`(`$'`)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`source`	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`sticky`	49	13	3	No	36	10
`test`	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`toSource`	No	No	(Yes)	No	No	No
`toString`	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)	(Yes)
`unicode`	50	12[1]	46	No	37	10
`@@match`	(Yes)	(Yes)	49	No	(Yes)	(Yes)
`@@replace`	(Yes)	(Yes)	49	No	(Yes)	(Yes)
`@@search`	(Yes)	(Yes)	49	No	(Yes)	(Yes)
`@@species`	(Yes)	(Yes)	49	No	(Yes)	(Yes)
`@@split`	(Yes)	(Yes)	49	No	(Yes)	(Yes)

1. Case folding is implemented in version 13

Firefox-specific notes

Starting with Firefox 34, in the case of a capturing group with quantifiers preventing its exercise, the matched text for a capturing group is now `undefined` instead of an empty string:

```
// Firefox 33 or older
'x'.replace(/x(.)?/g, function(m, group) {
  console.log("'group:" + group + "'");
}); // 'group:'

// Firefox 34 or newer
'x'.replace(/x(.)?/g, function(m, group) {
  console.log("'group:" + group + "'");
}); // 'group:undefined'
```

Note that due to web compatibility, `RegExp.$N` will still return an empty string instead of `undefined` (bug 1053944).

See also

- Regular Expressions chapter in the JavaScript Guide
- `String.prototype.match()`
- `String.prototype.replace()`

Was this article helpful?

Tags: Constructor JavaScript Reference RegExp Regular Expressions

Contributors to this page: fscholz, DiMateev, pmkary, NoInkling, ReinmarPL,jameshkramer, tregagnon, pietro909, lionel-rowe, vsemozhetbyt, koroliov, madarche,xaka2004, kumar303, Alhadis, torazaburo, albertstill, ysmood, nhahtdh, arai, jnteste rostomyan, RogerPoon, damphat, Mingun, ziyunfei, alexilyaev, rhyne24, alyssa, havarh,AlexChao, jviereck, jmclerck, mvasilk Sarah Hopwood, stepnem,Sspitzer, universian, Jesdisciple, Eino, Tgr, Hamstersoup, Minh Nguyen, Mgjbot, BenoitL,Pbb, Joaopaulo1511, Ptak82, miken32, Linuxmigration1, JesseW, B I R D, Maian, Ben Karel, Dria, Jesse

Last updated by: fscholz, Sep 26, 2017, 4:59:53 AM

See also

Standard built-in objects

`RegExp`

Properties

`RegExp.$1-$9`

`RegExp.input ($_)`

`RegExp.lastMatch ($&)`

`RegExp.lastParen ($+)`

``RegExp.leftContext ($`)``

`RegExp.prototype`

`RegExp.prototype.flags`

`RegExp.prototype.global`

`RegExp.prototype.ignoreCase`

`RegExp.prototype.multiline`

`RegExp.prototype.source`

`RegExp.prototype.sticky`

`RegExp.prototype.unicode`

`RegExp.rightContext ($')`

`get RegExp[@@species]`

`regexp.lastIndex`

Methods

`RegExp.prototype.compile()`

`RegExp.prototype.exec()`

`RegExp.prototype.test()`

`RegExp.prototype.toSource()`

`RegExp.prototype.toString()`

`RegExp.prototype[@@match]()`

`RegExp.prototype[@@replace]()`

`RegExp.prototype[@@search]()`

`RegExp.prototype[@@split]()`

Inheritance:

`Function`

Properties

Methods

`Object`

Properties

Methods