Regular Expressions

« Previous Next »

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`, `replace`, `search`, and `split` methods of `String`. This chapter describes JavaScript regular expressions.

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:

```
var re = /ab+c/;
```

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:

```
var 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 parenthesized substring matches.

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 exactly the characters 'abc' occur 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, the pattern includes special characters. For example, the pattern `/ab*c/` matches any character combination in which a single 'a' is followed by zero or more 'b's (`*` means 0 or more occurrences of the preceding item) and then immediately followed by 'c'. In the string "cbbabbbbcdebc," the pattern matches the substring 'abbbbc'.

The following table provides a complete list and description of the special characters that can be used in regular expressions.

Special characters in regular expressions.

Character	Meaning

Character	Meaning
`\`	Matches according to the following rules: A backslash that precedes a non-special character indicates that the next character is special and is not to be interpreted literally. For example, a '`b`' without a preceding '\' generally matches lowercase 'b's wherever they occur. But a '`\b`' by itself doesn't match any character; it forms the special word boundary character. A backslash that precedes a special character indicates that the next character is not special and should be interpreted literally. For example, the pattern `/a*/` relies on the special character '`*`' to match 0 or more a's. By contrast, the pattern `/a\*/` removes the specialness of the '`*`' to enable matches with strings like 'a*'. Do not forget to escape \ itself while using the RegExp("pattern") notation because \ is also an escape character in strings.
`^`	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 'A' in "An E". The '`^`' has a different meaning when it appears as the first character in a character set pattern. See complemented character sets for details and an example.
`$`	Matches end of input. If the multiline flag is set to true, also matches immediately before a line break character. For example, `/t$/` does not match the 't' in "eater", but does match it in "eat".
`*`	Matches the preceding expression 0 or more times. Equivalent to {0,}. For example, `/bo*/` matches 'boooo' in "A ghost booooed" and 'b' in "A bird warbled" but nothing in "A goat grunted".
`+`	Matches the preceding expression 1 or more times. Equivalent to `{1,}`. For example, `/a+/` matches the 'a' in "candy" and all the a's in "caaaaaaandy", but nothing in "cndy".

Character	Meaning
`?`	Matches the preceding expression 0 or 1 time. Equivalent to `{0,1}`. For example, `/e?le?/` matches the 'el' in "angel" and the 'le' in "angle" and also the 'l' in "oslo". If used immediately after any of the quantifiers *, +, ?, or {}, makes the quantifier non-greedy (matching the fewest possible characters), as opposed to the default, which is greedy (matching as many characters as possible). For example, applying `/\d+/` to "123abc" matches "123". But applying `/\d+?/` to that same string matches only the "1". Also used in lookahead assertions, as described in the `x(?=y)` and `x(?!y)` entries of this table.
`.`	(The decimal point) matches any single character except the newline character. For example, `/.n/` matches 'an' and 'on' in "nay, an apple is on the tree", but not 'nay'.
`(x)`	Matches 'x' and remembers the match, as the following example shows. The parentheses are called *capturing parentheses*. The '`(foo)`' and '`(bar)`' in the pattern `/(foo) (bar) \1 \2/` match and remember the first two words in the string "foo bar foo bar". The `\1` and `\2` in the pattern match the string's last two words. Note that `\1`, `\2`, ..., `\n` are used in the matching part of the regex. In the replacement part of a regex the syntax `$1`, `$2`, ..., `$n` must be used, e.g.: `'bar foo'.replace(/(...) (...)/, '$2 $1')`. `$&` means the whole matched string.
`(?:x)`	Matches 'x' but does not remember the match. The parentheses are called *non-capturing parentheses*, and let you define subexpressions for regular expression operators to work with. Consider the sample expression `/(?:foo){1,2}/`. If the expression was `/foo{1,2}/`, the `{1,2}` characters would apply only to the last 'o' in 'foo'. With the non-capturing parentheses, the `{1,2}` applies to the entire word 'foo'.
`x(?=y)`	Matches 'x' only if 'x' is followed by 'y'. This is called a lookahead. 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.

Character	Meaning
`x(?!y)`	Matches 'x' only if 'x' is not followed by 'y'. This is called a negated lookahead. For example, `/\d+(?!\.)/` matches a number only if it is not followed by a decimal point. The regular expression `/\d+(?!\.)/.exec("3.141")` matches '141' but not '3.141'.
`x|y`	Matches 'x', or 'y' (if there is no match for 'x'). For example, `/green|red/` matches 'green' in "green apple" and 'red' in "red apple." The order of 'x' and 'y' matters. For example `a*|b` matches the empty string in "b", but `b|a*`matches "b" in the same string.
`{n}`	Matches exactly n occurrences of the preceding expression. N must be a positive integer. For example, `/a{2}/` doesn't match the 'a' in "candy," but it does match all of the a's in "caandy," and the first two a's in "caaandy."
`{n,}`	Matches at least n occurrences of the preceding expression. N must be a positive integer. For example, /a{2,}/ will match "aa", "aaaa" and "aaaaa" but not "a"
`{n,m}`	Where `n` and `m` are positive integers and `n <= m`. Matches at least `n` and at most `m` occurrences of the preceding expression. When `m` is omitted, it's treated as ∞. For example, `/a{1,3}/` matches nothing in "cndy", the 'a' in "candy," the first 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.
`[xyz]`	Character set. This pattern type matches any one of the characters in the brackets, including escape sequences. Special characters like the dot(`.`) and asterisk (`*`) are not special inside a character set, so they don't need to be escaped. You can specify a range of characters by using a hyphen, as the following examples illustrate. The pattern `[a-d]`, which performs the same match as `[abcd]`, matches the 'b' in "brisket" and the 'c' in "city". The patterns `/[a-z.]+/` and `/[\w.]+/` match the entire string "test.i.ng".

Character	Meaning
`[^xyz]`	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. Everything that works in the normal character set also works here. For example, `[^abc]` is the same as `[^a-c]`. They initially match 'r' in "brisket" and 'h' in "chop."
`[\b]`	Matches a backspace (U+0008). You need to use square brackets if you want to match a literal backspace character. (Not to be confused with `\b`.)
`\b`	Matches a word boundary. A word boundary matches the position where a word character is not followed or preceded by another word-character. Note that a matched word boundary is not included in the match. In other words, the length of a matched word boundary is zero. (Not to be confused with `[\b]`.) 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. **Note:** JavaScript's regular expression engine defines a specific set of characters to be "word" characters. Any character not in that set is considered a word break. This set of characters is fairly limited: it consists solely of the Roman alphabet in both upper- and lower-case, decimal digits, and the underscore character. Accented characters, such as "é" or "ü" are, unfortunately, treated as word breaks.
`\B`	Matches a non-word boundary. This matches a position where the previous and next character are of the same type: Either both must be words, or both must be non-words. The beginning and end of a string are considered non-words. For example, `/\B../` matches 'oo' in "noonday", and `/y\B./` matches 'ye' in "possibly yesterday."
`\cX`	Where *X* is a character ranging from A to Z. Matches a control character in a string. For example, `/\cM/` matches control-M (U+000D) in a string.

Character	Meaning
`\d`	Matches a digit character. Equivalent to `[0-9]`. For example, `/\d/` or `/[0-9]/` matches '2' in "B2 is the suite number."
`\D`	Matches a non-digit character. Equivalent to `[^0-9]`. For example, `/\D/` or `/[^0-9]/` matches 'B' in "B2 is the suite number."
`\f`	Matches a form feed (U+000C).
`\n`	Matches a line feed (U+000A).
`\r`	Matches a carriage return (U+000D).
`\s`	Matches a single white space character, including space, tab, form feed, line feed. Equivalent to `[ \f\n\r\t\v\u00a0\u1680\u180e\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\u180e\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]`. For example, `/\S*/` matches 'foo' in "foo bar."
`\t`	Matches a tab (U+0009).
`\v`	Matches a vertical tab (U+000B).
`\w`	Matches any alphanumeric character 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 non-word character. Equivalent to `[^A-Za-z0-9_]`. For example, `/\W/` or `/[^A-Za-z0-9_]/` matches '%' in "50%."

Character	Meaning
`\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."
`\0`	Matches a NULL (U+0000) character. Do not follow this with another digit, because `\0<digits>` is an octal escape sequence. Instead use `\x00`.
`\xhh`	Matches the character with the code hh (two hexadecimal digits)
`\uhhhh`	Matches the character with the code hhhh (four hexadecimal digits).
`\u{hhhh}`	(only when u flag is set) Matches the character with the Unicode value hhhh (hexadecimal digits).

Escaping user input to be treated as a literal string within a regular expression can be accomplished by simple replacement:

```
function escapeRegExp(string) {
  return string.replace(/[.*+?^${}()|[\]\\]/g, '\\$&')
}
```

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, as described in Using Parenthesized Substring Matches.

For example, the pattern `/Chapter (\d+)\.\d*/` illustrates additional escaped and special characters and indicates that part of the pattern should be remembered. It matches precisely the characters 'Chapter ' followed by one or more numeric characters (`\d` means any numeric character and `+` means 1 or more times), followed by a decimal point (which in itself is a special character; preceding the decimal point with \ means the pattern must look for the literal character '.'), followed by any numeric character 0 or more times (`\d` means numeric character, `*` means 0 or more times). In addition, parentheses are used to remember the first matched numeric characters.

This pattern is found in "Open Chapter 4.3, paragraph 6" and '4' is remembered. The pattern is not found in "Chapter 3 and 4", because that string does not have a period after the '3'.

To match a substring without causing the matched part to be remembered, within the parentheses preface the pattern with `?:`. For example, `(?:\d+)` matches one or more numeric characters but does not remember the matched characters.

Working with regular expressions

Regular expressions are used with the `RegExp` methods `test` and `exec` and with the `String` methods `match`, `replace`, `search`, and `split`. These methods are explained in detail in the JavaScript reference.

Methods that use regular expressions

Method	Description
`exec`	A `RegExp` method that executes a search for a match in a string. It returns an array of information or null on a mismatch.
`test`	A `RegExp` method that tests for a match in a string. It returns true or false.
`match`	A `String` method that executes a search for a match in a string. It returns an array of information or null on a mismatch.
`search`	A `String` method that tests for a match in a string. It returns the index of the match, or -1 if the search fails.
`replace`	A `String` method that executes a search for a match in a string, and replaces the matched substring with a replacement substring.
`split`	A `String` method that 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` method; 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.

```
var myRe = /d(b+)d/g;
var 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:

```
var myArray = /d(b+)d/g.exec('cdbbdbsbz'); // equivale
```

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

```
var myRe = new RegExp('d(b+)d', 'g');
var 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"`
`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:

```
var myRe = /d(b+)d/g;
var myArray = myRe.exec('cdbbdbsbz');
console.log('The value of lastIndex is ' + myRe.lastIn

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

However, if you have this script:

```
var myArray = /d(b+)d/g.exec('cdbbdbsbz');
console.log('The value of lastIndex is ' + /d(b+)d/g.l

// "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.

Using parenthesized substring matches

Including parentheses in a regular expression pattern causes the corresponding submatch to be remembered. For example, `/a(b)c/` matches the characters 'abc' and remembers 'b'. To recall these parenthesized substring matches, use the `Array` elements `[1]`, ..., `[n]`.

The number of possible parenthesized substrings is unlimited. The returned array holds all that were found. The following examples illustrate how to use parenthesized substring matches.

The following script uses the `replace()` method to switch the words in the string. For the replacement text, the script uses the `$1` and `$2` in the replacement to denote the first and second parenthesized substring matches.

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

// "Smith, John"
```

Advanced searching with flags

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

Regular expression flags

Flag	Description
`g`	Global search.
i	Case-insensitive search.
m	Multi-line search.
u	unicode; treat pattern as a sequence of unicode code points

Flag	Description
y	Perform a "sticky" search that matches starting at the current position in the target string. See `sticky`

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

```
var re = /pattern/flags;
```

or

```
var 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.

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

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

You could replace the line:

```
var re = /\w+\s/g;
```

with:

```
var re = new RegExp('\\w+\\s', 'g');
```

and get the same result.

The behavior associated with the '**g**' flag is different when the `.exec()` method is used. (The roles of "class" and "argument" get reversed: In the case of `.match()`, the string class (or data type) owns the method and the regular expression is just an argument, while in the case of `.exec()`, it is the regular expression that owns the method, with the string being the argument.

Contrast *`str.match(re)`* versus *`re.exec(str)`*.) The '**`g`**' flag is used with the **`.exec()`** method to get iterative progression.

```
var xArray; while(xArray = re.exec(str)) console.log(x
// produces:
// ["fee ", index: 0, input: "fee fi fo fum"]
// ["fi ", index: 4, input: "fee fi fo fum"]
// ["fo ", index: 7, input: "fee fi fo fum"]
```

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.

Examples

The following examples show some uses of regular expressions.

Changing the order in an input string

The following example illustrates the formation of regular expressions and the use of `string.split()` and `string.replace()`. It cleans a roughly formatted input string containing names (first name last) separated by blanks, tabs and exactly one semicolon. Finally, it reverses the name order (last name first) and sorts the list.

```
// The name string contains multiple spaces and tabs,
// and may have multiple spaces between first and last
var names = 'Harry Trump ;Fred Barney; Helen Rigby ; B

var output = ['---------- Original String\n', names +

// Prepare two regular expression patterns and array s
// Split the string into array elements.

// pattern: possible white space then semicolon then p
var pattern = /\s*;\s*/;

// Break the string into pieces separated by the patte
// store the pieces in an array called nameList
var nameList = names.split(pattern);

// new pattern: one or more characters then spaces the
// Use parentheses to "memorize" portions of the patte
// The memorized portions are referred to later.
pattern = /(\w+)\s+(\w+)/;
```

```
// New array for holding names being processed.
var bySurnameList = [];

// Display the name array and populate the new array
// with comma-separated names, last first.
//
// The replace method removes anything matching the pa
// and replaces it with the memorized string—second me
// followed by comma space followed by first memorized
//
// The variables $1 and $2 refer to the portions
// memorized while matching the pattern.

output.push('---------- After Split by Regular Express

var i, len;
for (i = 0, len = nameList.length; i < len; i++) {
  output.push(nameList[i]);
  bySurnameList[i] = nameList[i].replace(pattern, '$2,
}

// Display the new array.
output.push('---------- Names Reversed');
for (i = 0, len = bySurnameList.length; i < len; i++)
  output.push(bySurnameList[i]);
}

// Sort by last name, then display the sorted array.
bySurnameList.sort();
output.push('---------- Sorted');
for (i = 0, len = bySurnameList.length; i < len; i++)
  output.push(bySurnameList[i]);
}

output.push('---------- End');

console.log(output.join('\n'));
```

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.

Within non-capturing parentheses `(?:` , the regular expression looks for three numeric characters `\d{3}` OR `|` a left parenthesis `\(` followed by three digits `\d{3}`, followed by a close parenthesis `\)`, (end non-capturing parenthesis `)`), followed by one dash, forward slash, or decimal point and when found, remember the character `([-\/\.])`, followed by three digits `\d{3}`, followed by the remembered match of a dash, forward slash, or decimal point `\1`, followed by four digits `\d{4}`.

The `Change` event activated when the user presses Enter sets the value of `RegExp.input`.

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html
    <meta http-equiv="Content-Script-Type" content="te
    <script type="text/javascript">
      var re = /(?:\d{3}|\(\d{3}\))([-\/\.])\d{3}\1\d{
      function testInfo(phoneInput) {
        var OK = re.exec(phoneInput.value);
        if (!OK)
          window.alert(phoneInput.value + ' isn\'t a p
        else
          window.alert('Thanks, your phone number is '
      }
    </script>
  </head>
  <body>
    <p>Enter your phone number (with area code) and th
        <br>The expected format is like ###-###-####.<
    <form action="#">
      <input id="phone"><button onclick="testInfo(docu
    </form>
  </body>
</html>
```

« Previous

Next »

Was this article helpful?

Tags: Guide Intermediate JavaScript Reference regex RegExp Regular Expressions

Contributors to this page: chalin, pinobatch, SphinxKnight, lyndon811, cardin, cPlayIt,telunc, jawline, forwebtech, fscholz, TalhaAydin, MukulGupX-dev, jakimenko, pbangelly, wkhaw94, KurtHarlandLarson, mrmaka, nikjohn,WilliamAvery, cauburtin, stevemao, Fishrock123, Paj,FernandoBasso, bradleymeck, boobl, TTO, RustyDoorknobs, Revolt, juicehonky,MartinRinehart, kscarfone, schooley, edW, lmorchard,sonujaiswar, mjhm, Kizer, bmenasha, BYK, berkerpeksag, zachleat, user01, BenoitL,Mgjbot, Major Small, JesseW

Last updated by: chalin, Nov 1, 2017, 5:30:17 AM

See also

Learn the best of web development

Get the latest and greatest from MDN delivered straight to your inbox.

you@example.com

Sign up now