Core JavaScript 1.5 Guide:Regular Expressions

From MDC

Contents

[hide]

Creating a Regular Expression [edit]

You construct a regular expression in one of two ways:

- Using a regular expression literal, as follows:

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

Regular expression literals provide compilation of the regular expression when the script is evaluated. When the regular expression will remain constant, use this for better performance.

- Calling the constructor function of the RegExp object, as follows:

```
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 [edit]

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 [edit]

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 it does not contain the substring 'abc'.

Using Special Characters [edit]

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.

Character	Meaning
\	Either of the following: ■ 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.

	■ 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 item 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*'.
^	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".
*	Matches the preceding character 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".
+	Matches the preceding character 1 or more times. Equivalent to {1,}. For example, /a+/ matches the 'a' in "candy" and all the a's in "caaaaaaandy".
?	Matches the preceding character 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). For example, using /\d+/ non-globle match "123abc" return "123", if using /\d+?/, only "1" will be matched. Also used in lookahead assertions, described under x(?=y) and x(?!y) in 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. These are called capturing parentheses. For example, /(foo)/ matches and remembers 'foo' in "foo bar." The matched substring can be recalled from the resulting array's elements [1], ..., [n].
(?:x)	Matches 'x' but does not remember the match. These are called non-capturing parentheses. The matched substring can not be recalled from the resulting array's elements [1], ..., [n].
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. The regular expression /\d+(?!\.)/.exec("3.141") matches '141' but not '3.141'.
x|y	Matches either 'x' or 'y'. For example, /green|red/ matches 'green' in "green apple" and 'red' in "red apple."
{n}	Where n is a positive integer. Matches exactly n occurrences of the preceding character. 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."
{n,}	Where n is a positive integer. Matches at least n occurrences of the preceding character. For example, /a{2,}/ doesn't match the 'a' in "candy", but matches all of the a's in "caandy" and in "caaaaaaandy."
{n,m}	Where n and m are positive integers. Matches at least n and at most m occurrences of the preceding character. 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]	A character set. Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen. For example, [abcd] is the same as [a-d]. They match the 'b' in "brisket" and the 'c' in "ache".
[^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. For example, [^abc] is the same as [^a-c]. They initially match 'r' in "brisket" and 'h' in "chop."
[\b]	Matches a backspace. (Not to be confused with \b.)
\b	Matches a word boundary, such as a space or a newline character. (Not to be confused with [\b].) For example, /\bn\w/ matches the 'no' in "noonday";/\wy\b/ matches the 'ly' in "possibly yesterday."
\B	Matches a non-word boundary. For example, /\w\Bn/ matches 'on' in "noonday", and /y\B\w/ matches 'ye' in "possibly yesterday."
\cX	Where X is a control character. Matches a control character in a string. For example, /\cM/ matches control-M in a string.
\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 any 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.
\n	Matches a linefeed.
\r	Matches a carriage return.

\s	Matches a single white space character, including space, tab, form feed, line feed. Equivalent to [\f\n\r\t\v\u00A0\u2028\u2029]. 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\u2028\u2029]. For example, /\S\w*/ matches 'foo' in "foo bar."
\t	Matches a tab.
\v	Matches a vertical tab.
\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%."
\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 NUL character. Do not follow this with another digit.
\xhh	Matches the character with the code hh (two hexadecimal digits)
\uhhhh	Matches the character with the code hhhh (four hexadecimal digits).

Table 4.1: Special characters in regular expressions.

Using Parentheses [edit]

Parentheses around any part of the regular expression pattern cause 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.

- Executing a Global Search, Ignoring Case, and Considering Multiline Input
- Using Parenthesized Substring Matches
- Examples of Regular Expressions

Working With Regular Expressions [edit]

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 Core JavaScript Reference.

Method	Description
exec	A RegExp method that executes a search for a match in a string. It returns an array of information.
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.

Table 4.2: Methods that use regular expressions

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 converts to false).

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

```
<SCRIPT type="text/javascript">
  myRe = /d(b+)d/g;
  myArray = myRe.exec("cdbbdbsbz");
</SCRIPT>
```

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

```
<SCRIPT type="text/javascript">
  myArray = /d(b+)d/g.exec("cdbbdbsbz");
</SCRIPT>
```

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

```
<SCRIPT type="text/javascript">
  myRe = new RegExp ("d(b+)d", "g");
  myArray = myRe.exec("cdbbdbsbz");
</SCRIPT>
```

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

Object	Property or index	Description	In this example
myArray		The matched string and all remembered substrings.	["dbbd", "bb"]
	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 Executing a Global Search, Ignoring Case, and Considering Multiline Input.)	5
	source	The text of the pattern. Updated at the time that the regular expression is created, not executed.	"d(b+)d"

Table 4.3: Results of regular expression execution.

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:

```
<SCRIPT type="text/javascript">
  myRe = /d(b+)d/g;
  myArray = myRe.exec("cdbbdbsbz");
  document.writeln("The value of lastIndex is " + myRe.lastIndex);
</SCRIPT>
```

This script displays:

```
The value of lastIndex is 5
```

However, if you have this script:

```
<SCRIPT type="text/javascript">
  myArray = /d(b+)d/g.exec("cdbbdbsbz");
  document.writeln("The value of lastIndex is " + /d(b+)d/g.lastIndex);
</SCRIPT>
```

It displays:

```
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 [edit]

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.

Example 1.
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.

```
<script type="text/javascript">
 re = /(\w+)\s(\w+)/;
 str = "John Smith";
 newstr = str.replace(re, "$2, $1");
 document.write(newstr);
</script>
```

This prints "Smith, John".

Example 2.
Note: in the `getInfo` function, the `exec` method is called using the () shortcut notation that works in Firefox but not in most other browsers.

```
<html>

<script type="text/javascript">
 function getInfo(field){
   var a = /(\w+)\s(\d+)/(field.value);
   window.alert(a[1] + ", your age is " + a[2]);
 }
</script>

Enter your first name and your age, and then press Enter.

<form>
  <input type="text" name="NameAge" onchange="getInfo(this);">
</form>

</html>
```

Advanced Searching With Flags [edit]

Regular expressions have four optional flags that allow for global and case insensitive searching. To indicate a global search, use the `g` flag. To indicate a case-insensitive search, use the `i` flag. To indicate a multi-line search, use the `m` flag. To perform a "sticky" search, that matches starting at the current position in the target string, use the `y` flag. These flags can be used separately or together in any order, and are included as part of the regular expression.

Firefox 3 note

Support for the `y` flag was added in Firefox 3. The `y` flag fails if the match doesn't succeed at the current position in the target string.

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

```
re = /pattern/flags
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.

```
<script type="text/javascript">
 re = /\w+\s/g;
 str = "fee fi fo fum";
 myArray = str.match(re);
 document.write(myArray);
</script>
```

This displays ["fee ", "fi ", "fo "]. In this example, you could replace the line:

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

with:

```
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.

Examples [edit]

The following examples show some uses of regular expressions.

Changing the Order in an Input String [edit]

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 first) separated by blanks, tabs and exactly one semicolon. Finally, it reverses the name order (last name first) and sorts the list.

```
<script type="text/javascript">

// The name string contains multiple spaces and tabs,
// and may have multiple spaces between first and last names.
var names = "Harry Trump ;Fred Barney; Helen Rigby ; Bill Abel ; Chris Hand ";

var output = new Array(
  "---------- Original String<br><br>",
  names + "<br><br>");

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

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

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

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

// New array for holding names being processed.
var bySurnameList = new Array();

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

output.push("---------- After Split by Regular Expression<br>");

var i, len;
for (i = 0, len = nameList.length; i < len; i++)
{
  output.push(nameList[i] + "<br>");
  bySurnameList[i] = nameList[i].replace(pattern, "$2, $1")
}

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

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

output.push("---------- End<br>");

document.write(output.join("\n"));

</script>
```

Using Special Characters to Verify Input [edit]

In the following example, a user enters a phone number. When the user presses Enter, the script checks the validity of the number. If the number is valid (matches the character sequence specified by the regular expression), the script posts a window thanking the user and confirming the number. If the number is invalid, the script posts a window informing the user that the phone number is not valid.

The regular expression looks for zero or one open parenthesis `\(?`, followed by three digits `\d{3}`, followed by zero or one close parenthesis `\)?`, 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 PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <meta http-equiv="Content-Script-Type" content="text/javascript">
    <script type="text/javascript">
      var re = /\(?\d{3}\)?([-\/\.])\d{3}\1\d{4}/;

      function testInfo(phoneInput)
      {
        var OK = re.exec(phoneInput.value);

        if (!OK)
        {
          window.alert(RegExp.input + " isn't a phone number with area code!");
        }
        else
        {
          window.alert("Thanks, your phone number is " + OK[0]);
        }
      }
    </script>
  </head>

  <body>
    <p>Enter your phone number (with area code) and then press Enter.</p>
    <form action="">
      <input name="phone" onchange="testInfo(this);">
    </form>
  </body>
</html>
```

« Previous Next »