



JS

Regular Expressions in JavaScript Article

Kevin Yank

November 21, 2000

If you've ever programmed in Perl, or have had to work as a Unix system administrator, then you probably already have more than a passing familiarity with regular expressions. But even if Perl looks like spaghetti to you and you feel that no mere mortality is equipped to manage a Unix system, the fact that JavaScript includes support for Perl-style regular expressions is good news... trust me!

In this article, I'll start out by explaining what, exactly, regular expressions are and what they can do for you. I'll then present an overview of the most common features of regular expressions (which the Perl aficionados in the audience can safely skip over). Finally, I'll finish off by explaining how regular expressions are used in JavaScript, with a practical example or two to help the concepts gel. By the end of this article, you'll probably still be a mere mortal, but you'll certainly be able to impress at parties with your newly acquired text juggling skills!

What are they?

Regular expressions use special (and, at first, somewhat confusing) codes to detect **patterns** in strings of text. For example, if you're presenting your visitors with an HTML form to enter their details, you might have one field for their phone number. Now let's face it: some site visitors are better at following instructions than others. Even if you put a little hint next to the text field indicating the required format of the phone number (e.g., "(XXX) XXX-XXXX" for North American numbers), some people are going to get it wrong. Writing a script to check every character of the entered string to ensure that all the numbers are where they belong, with parentheses and a dash in just the right spots, would be a pretty tedious bit of code to write. And a telephone number is a relatively simple case! What if you had to check that a user had indeed entered an email address or, worse yet, a URL?

Regular expressions provide a quick and easy way of matching a string to a pattern. In our phone number example, we could write a simple regular expression and use it to check â€œ in one quick step â€œ whether or not any given string is a properly formatted phone number. We'll explore this example a little further, once we've taken care of a few technical details.

What do they look like?

Regular expressions can look fairly complex at times, but when it comes right down to it they're actually just text strings themselves. The following, for example, is a regular expression that searches for the text "JavaScript" (without the quotes):

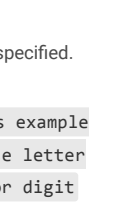
```
JavaScript
```

Not much to it, is there? Any string containing the text "JavaScript" is said to **match** this regular expression. Thus, this regular expression allows us to detect strings containing this particular string of text.

Well, the bad news is that it's not always so simple. As I mentioned above, there are special codes that may be used in regular expressions. Some of these can be downright confusing and difficult to remember, so if you intend to make extensive use of them you may wish to find a good reference for yourself. Fortunately, JavaScript supports the very same regular expression syntax as Perl, and there are lots of Websites out there documenting Perl regular expressions. One of the better ones is <http://web.archive.org/web/200012121853/http://www.delorie.com/gnu/docs/rx/rx-toc.html>. There's also the documentation provided in the official Perl manual at <http://www.perl.com/pub/doc/manual/html/pod/perlre.html>. Both of these are cryptic in places, so you may want to refer to both of them at once. Let's work our way through a few examples to learn the basic regular expression syntax.

Basic Syntax

First of all, a caret (^) may be used to indicate the beginning of the string, while a dollar sign (\$) is used to mark the end.



```
JavaScript // Matches "Isn't JavaScript great?"
^JavaScript // Matches "JavaScript rules!",
// not "What is JavaScript?"
JavaScript$ // Matches "I love JavaScript",
// not "JavaScript is great!"
^JavaScript$ // Matches "JavaScript", and nothing else
```

Obviously, you may sometimes want to use ^, \$, or other special characters to represent the corresponding character in the search string rather than the special meaning implied by regular expression syntax. To remove the special meaning of a character, prefix it with a backslash:

```
$$$ // Matches "Show me the $$$!"
```

Square brackets may be used to define a set of characters that may match. For example, the following regular expression will match any digit from 1 to 5 inclusive.

```
[12345] // Matches "1" and "3", but not "a" or "12"
```

Ranges of numbers and letters may also be specified.

```
[1-5] // Same as the previous example
[a-z] // Matches any lowercase letter
[0-9-a-Z-_-] // Matches any letter or digit
```

By putting a ~ immediately following the opening square bracket, you can **invert** the set of characters, meaning the set will match any character not listed:

```
[^a-zA-Z] // Matches anything except a letter
```

The characters ~, *, and + also have special meanings. Specifically, ~ means "the preceding character is optional", + means "one or more of the previous character", and * means "zero or more of the previous character".

```
banana // Matches "banana" and "banna",
// but not "banaana".
banana+ // Matches "banana" and "banaana",
// but not "banna".
banana* // Matches "banna", "banana", and "banaana",
// but not "banna".
^[a-zA-Z]*$ // Matches any string of one or more
// letters and nothing else.
```

Parentheses may be used to group strings together to apply ~, +, or * to them as a whole.

```
ba(na)+na // Matches "banana" and "banananana",
// but not "bana" or "banaana".
```

Parentheses also let you define several strings that may match, using the pipe (|) character to separate them.

```
^(ba|na)$ // Matches "banana", "nababa", "baba",
// "nana", "ba", "na", and others.
```

Here are a few special codes that can be used for matching characters in regular expressions:

```
n // A newline character
. // Any character except a newline
r // A carriage return character
t // A tab character
b // A word boundary (the start or end of a word)
B // Anything but a word boundary
d // Any digit (same as [0-9])
D // Anything but a digit (same as [^0-9])
s // Single whitespace (space, tab, newline, etc.)
S // Single nonwhitespace
w // A "word character" (same as [A-Za-z0-9_])
W // A "nonword character" (same as [^A-Za-z0-9_])
```

There are more special codes and syntax tricks for regular expressions, all of which should be covered in any complete reference (such as those mentioned above). For now, we have more than enough for our purposes.

Using Regular Expressions in JavaScript

Using regular expressions in JavaScript is so easy that it's a wonder more people don't know that it can be done. You can create a regular expression in JavaScript as follows:

```
var myRE = /regex/;
```

Where regex is the regular expression code, as described above. For example, the following creates the first example regular expression I presented in the previous section, the one that detects the string "JavaScript":

```
var myRE = /JavaScript/;
```

Similarly, here's how to create the last example:

```
var myRE = /^[ba]na+$/;
```

By default, JavaScript regular expressions are case sensitive and only search for the first match in any given string. By adding the g (for **global**) and i (for **ignore case**) modifiers after the second /, you can make a regular expression search for all matches in the string and ignore case, respectively. Here are a few example regular expressions. For each, I've indicated what portion(s) of the string "test1 Test2 TEST3" they would match:

```
// RegExp // Match(es):
/Test[0-9]+/ // "Test2" only
/Test[0-9]+/i // "test2" only
/Test[0-9]+/gi // "test1", "Test2", and "TEST3"
```

Using a regular expression is easy. Every JavaScript variable containing a text string supports three methods (or functions, if you aren't used to object-oriented terminology) for working with regular expressions: `match()`, `replace()`, and `search()`.

match()

`match()` takes a regular expression as a parameter and returns an array of all the matching strings found in the string under consideration. If no matches are discovered, then `match()` returns false. Returning to our original example, let's say that we wanted a function that can check that a string entered by the user as his or her phone number is of the form (XXX) XXX-XXXX. The following code would do the trick:

```
function checkPhoneNumber(phoneNo) {
  var phoneRE = /^(ddd) ddd-dddd$/;
  if (phoneNo.match(phoneRE)) {
    return true;
  } else {
    alert( "The phone number entered is invalid!" );
    return false;
  }
}
```

As its first order of business, this function defines a regular expression. Let's break it down to understand how it works. The regular expression begins with ^, to indicate that any match must begin at the start of the string. Next is (, which will just match the opening parenthesis. We prefixed the character with a backslash to remove its special meaning in regular expression syntax (to mark the start of a set of alternatives for matching). As mentioned previously, d is a special code that matches any digit; thus, ddd matches any three digits. We could have written [0-9][0-9][0-9] to achieve the same effect, but this is shorter. The rest of the pattern should be pretty self-explanatory.) matches the closing parenthesis, the space matches the space that must be left in the phone number, then ddd-dddd matches three digits, followed by a dash, followed by four more digits. Finally, the \$ indicates that any match must end at the end of the string.

Incidentally, we could shorten this regular expression to the following, by using another shortcut that we did not mention above. If you can see how this works, you're a natural!

```
var phoneRE = /^(d{3}) d{3}-d{4}$/;
```

Our function then checks if `phoneNo.match(phoneRE)` evaluates to `true` or `false`. In other words, it checks whether or not the string contained in `phoneNo` matches our regular expression (thus returning an array, which in JavaScript will evaluate to `true`). If a match is detected, our function returns `true` to certify that the string is indeed a phone number. If not, a message is displayed warning of the problem and the function returns `false`.

The most common use for this type of function is in validating user input to a form before allowing it to be submitted. By calling our function in the `onSubmit` event handler for the form, we can prevent the form from being submitted if the information entered is not properly formatted. Here's a simple example demonstrating the use of our `checkPhoneNumber()` function:

```
<form action="..."
onSubmit="return checkPhoneNumber(this.phone.value);">
  <p>Enter phone number (e.g. (123) 456-7890):
  <input type="text" name="phone">
  <p><input type="submit" value="Submit">
</form>
```

The user will be unable to submit this form unless a phone number has been entered. Any attempt to do so will produce the error message generated by our `checkPhoneNumber()` function.

replace()

As its name would suggest, `replace()` lets you replace matches to a given regular expression with some new string. Let's say you were a spelling nut and wanted to enforce the old adage "I before E, except after C" to correct such misspellings as "acheive" and "cieling". What we'd need is a function that takes a string and performs two search-and-replace operations. The first would replace "cie" with "cei". Here's the code:



```
theString = theString.replace(/cie/gi,"cei");
```

Simple enough, right? The first parameter is the regular expression that we're searching for (notice that we've set it to "ignore case" and to be "global" so that it finds all occurrences, not just the first), and the second parameter is the string that we want to replace any matches with.

The second replacement is a little more complicated. We want to replace "xer" with "xie" where 'x' is any letter except 'c'. The regular expression to detect instances of "xer" is fairly easy to understand:

```
/[abd-z]ei/gi
```

This just detects any letter except 'c' ('a', 'b', and 'd' to 'z' inclusive), followed by 'ei', and does it in a global, case-insensitive manner.

The complexity comes in defining our replacement string. Obviously, we want to replace the match with "xie", but the difficulty comes in writing the 'x'. Remember, we have to replace 'x' with whatever letter appears in the matching string. To do this, we need to learn a new trick.

Earlier on, I showed you how parentheses could be used to define a set of alternatives in a regular expression (e.g. ^(ba|na)\$). Well as it turns out, parentheses have another meaning, too. They let us "remember" part of a match, so that we can use it in the replacement string. In this case, we want to remember the portion of the match that corresponds to the [abd-z] in the regular expression. Thus, we surround it with parentheses:

```
/([abd-z])ei/gi
```

Now, when specifying the replacement string, we put \$1 where we want to insert the portion of the string corresponding to the parenthesised portion of the regular expression. Thus, the code for performing the required substitutions is as follows:

```
theString = theString.replace(/([abd-z])ei/gi,"$1ie");
```

To sum it up, here's the complete function for performing our auto-correction:

```
function autoCorrect(theString) {
  theString = theString.replace(/cie/gi,"cei");
  theString = theString.replace(/([abd-z])ei/gi,"$1ie");
  return theString;
}
```


Before you go and use this function on your page, realize that there are exceptions to the "I before E except after C" rule. Weird, huh?



search()

The `search()` function is similar to the well-known `indexOf()` function, except it takes a regular expression instead of a string. It then searches the string for the first match to the given regular expression and returns an integer indicating the position in the string (e.g. 0 if the match is at the start of the string, 9 if the match begins with the 10th character in the string). If no match is found, the function returns a value of â€œ".

Summing it up

Regular expressions are an invaluable tool for verifying user input. By taking advantage of support for regular expressions in JavaScript, that verification can be done without having to resort to complex and potentially costly server-side scripting. In fact, you can make your server-side scripts considerably simpler if you verify user input with JavaScript before allowing that data to be submitted, since your server-side scripts can assume that the data it receives is valid. Simpler scripts run faster, and lighten the load on your Web server.

Kevin Yank

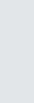


Kevin began developing for the Web in 1995 and is a highly respected technical author. Kev is a world-renowned author, speaker and JavaScript expert. He has a passion for making web technology easy to understand by anyone. Yes, even you!

Recommended for you

- 3 Plugins Every Gruntfile & Gulpfile Needs
- GraphQL Overview: Build a to-Do List API with a React Front-End
- Deploy Your Own REST API in 30 Mins Using mLab and Heroku
- Introduction to JCanvas: jQuery Meets HTML5 Canvas
- 20+ Docs and Guides for Front-end Developers (No. 7)

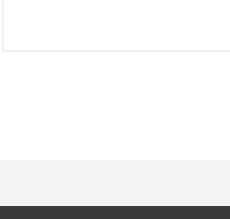
SPECIAL OFFER

Free course!

Get into 11 Bonus course Introduction to Git is yours when you take up a free 14 day SitePoint Premium trial.

Get This Deal

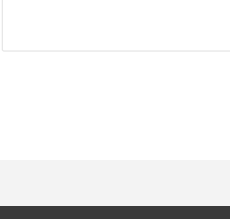
COURSES >



JavaScript: Next Steps

M. David Green

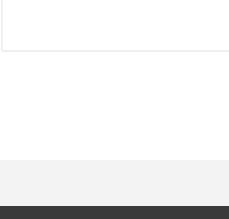
★★★★★



React The ES6 Way

Danm Flaener

★★★★☆



Your First Meteor 1.2 Application

David Turnbull

★★★★☆

BOOKS >



ECMAScript 2015: A SitePoint Anthology

James Hibbard

★★★★☆



Full Stack JavaScript Development with MEAN

Colin Fring

★★★★☆



JavaScript: Novice to Ninja

Darren Jones

★★★★☆

SCREENCASTS >

Building Realtime Web Apps with Firebase

Thomas Greco

Using Helpers and Conditionals in Ember

Thomas Greco

Modifying a Bootstrap Theme with Handlebars

Kaurens

About

Our Story
Advertise
Press Room
Reference
Terms of Use
Privacy Policy
FAQ
Contact Us
Contribute

Visit

SitePoint Home
Forums
Newsletters
Premium
References
Shop
Versioning

Connect

© 2000 – 2016 SitePoint Pty Ltd.