Using regular expressions – Part 1: Understanding the basic building blocks

by David Powers

http://foundationphp.com/

Content

Created
21 June 2010

Requirements

Prerequisite knowledge	Required products	Sample files
No prior knowledge of regular expressions is needed, but an understanding of a programming language such as JavaScript or PHP would be useful.	Dreamweaver (Download trial)	regex_pt1.zip (1 KB)

User level
Beginning

This is Part 1 of a two-part tutorial series on using regular expressions with Dreamweaver.

The ability to find and replace text is expected in even the simplest of editing programs. But what if you're looking for values likely to be different every time, for example, phone numbers displayed in the standard North American format? Every number is different. Some numbers might have parentheses around the area code; others might not have parentheses. Wouldn't it be great if you could perform a find-and-replace operation that strips or adds the parentheses while preserving the actual phone numbers? Well, you can.

The answer is to use a *regular expression* (often shortened to *regex*). Regular expressions are patterns that describe character combinations in text. As long as what you're looking for fits a regular pattern, a regex can be created to find it. Many people call this a *wildcard search*, but regular expressions are much more powerful than searches using wildcard characters.

The first part of this tutorial introduces you to the basic building blocks using the Find And Replace dialog box in Dreamweaver. In Part 2, I'll show you some practical uses of regexes in Dreamweaver, and how to adapt them for use in ColdFusion, JavaScript, and PHP.

Regular expressions 101

Nearly every modern programming language offers support for regular expressions. That's the good news. The bad news is that regexes can be difficult to create; some symbols change meaning according to context; and to top it all, they look like Klingon poetry to the untrained eye. Don't despair, you can achieve a great deal with a little knowledge and patience. There are also useful resources where you can find ready-made regexes.

Regexes come in many flavors

There are two main flavors of regex—POSIX (Portable Operating System Interface) and Perl-style (or Perl-compatible). POSIX was an attempt to create a unified standard, but Perl-style regular expressions are more efficient and much more widely used.

Even among Perl-style regexes, there are minor variations in the features supported by each language, but the basic principles are common to all of them.

This tutorial covers only Perl-style regular expressions.

The basic building blocks of a regex

Regular expressions are used to search for patterns of text using one or more of the following devices:

- **Literal text:** With the exception of a small number of characters that have a special meaning in a regex, text matches itself.
- **Special wildcard characters:** Known as *metacharacters* or *metasequences*, these match or exclude specific types of text, such as *any number*.
- **Character classes:** When a suitable metacharacter or metasequence doesn't exist, you can create your own definition to match or exclude specified characters.
- **Quantifiers:** These specify how many times you want the preceding expression to match or if it's optional.
- **Capturing groups and backreferences:** These specify parts of the regex that you want remembered, either to find a similar match later on, or to preserve the value in a find and replace operation.
- **Boundaries and anchors:** These specify where the match should be made, for example at the beginning of a line or word.
- **Alternation:** This specifies alternatives.

As you can see from this list, regular expressions have a lot of features. There is a lot to absorb in the following pages.

Bookmark this article as a reference, and take things slowly.

Matching literal text

In the land of regular expressions, most characters match themselves. The only exceptions are the following 12 characters:

```
$()*+.?[\^{|
```

These characters have special meanings in regular expressions. If you want your regex to match any of them, precede them with a backslash. For example, to look for a literal dollar sign, your regex needs to use this:

```
\$
```

This is known as *escaping* the special character.

By default, regular expressions are case-sensitive, so *A* and *a* are treated as different.

Setting up for the exercises

A good way to learn about regular expressions is to try them out in the Dreamweaver Find And Replace dialog box.

1. Download the sample files for this tutorial, and unzip them to any convenient location. They do not need to be in a Dreamweaver defined site.
2. Open the pages for each exercise in Code view, and choose Edit > Find And Replace to open the Find And Replace dialog box (you can also press Ctrl+F on Windows or Cmd+F on Mac OS X).
3. Select Current Document for the Find In option and select Source Code for the Search option.
4. Select Use Regular Expression (see Figure 1).

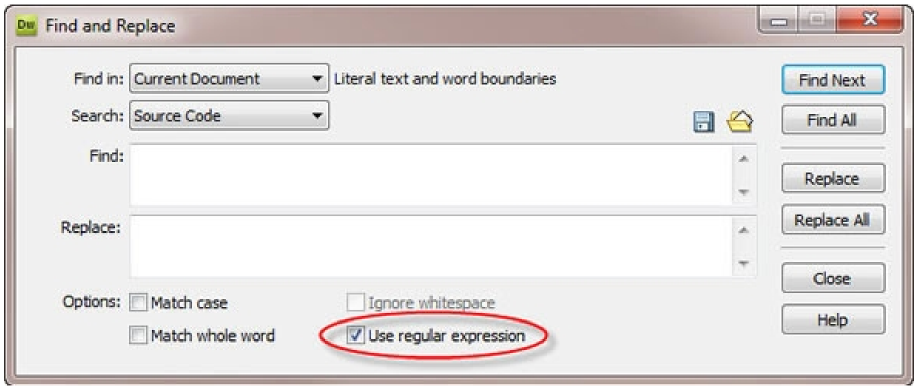


Figure 1. Select the option to use a regular expression in the Find And Replace dialog box.

Note: When you select the option to use a regular expression, Ignore Whitespace is disabled, but the other two options (Match Case and Match Whole Word) remain selectable. I'll explain the purpose of these options in the following exercise.

Matching literal text and special characters

Many patterns contain at least some literal text, so it's useful to see what happens when your text includes one of the dozen special characters. It's also important to know how to specify your text precisely, so you don't get unwanted matches.

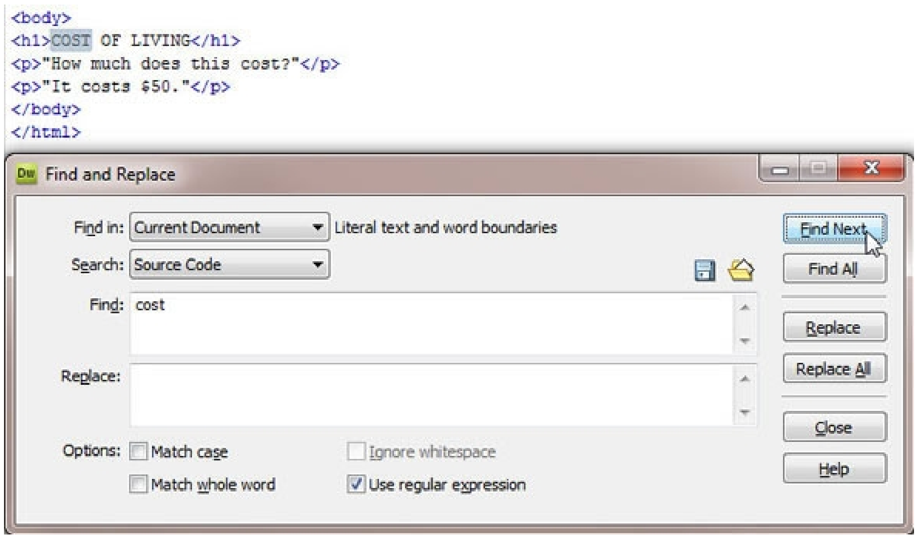
1. Open regex_01.html from the sample files in Code view.

   The `<body>` section of the page contains the following code:

```
<h1>COST OF LIVING</h1>
<p>"How much does this cost?"</p>
<p>"It costs $50."</p>
```

2. Position your cursor above this section of code, and choose Edit > Find And Replace to open the Find And Replace dialog box. Make sure Use Regular Expression is selected.

3. Type **cost** in the Find text box, and click Find Next.
4. Dreamweaver will highlight `COST` in the `<h1>` tag (see Figure 2).



```
<body>
<h1>COST OF LIVING</h1>
<p>"How much does this cost?"</p>
<p>"It costs $50."</p>
</body>
</html>
```

Find and Replace

Find in: Current Document — Literal text and word boundaries
Search: Source Code
Find: cost
Replace:
Options: [ ] Match case [ ] Ignore whitespace [ ] Match whole word [✓] Use regular expression

Figure 2. Dreamweaver treats regular expressions as case-insensitive.

5. Make the search case-sensitive by selecting Match Case in the Find And Replace dialog box.

   Note: In programming languages, it's the other way round. Regexes are case-sensitive by default, and you need to turn on a special option to make them case-insensitive.

6. Click in Code view to move the cursor back above the body of the page.

   The Find And Replace dialog box should remain in front of the Document window as you reposition the cursor.

7. Click Find Next again.

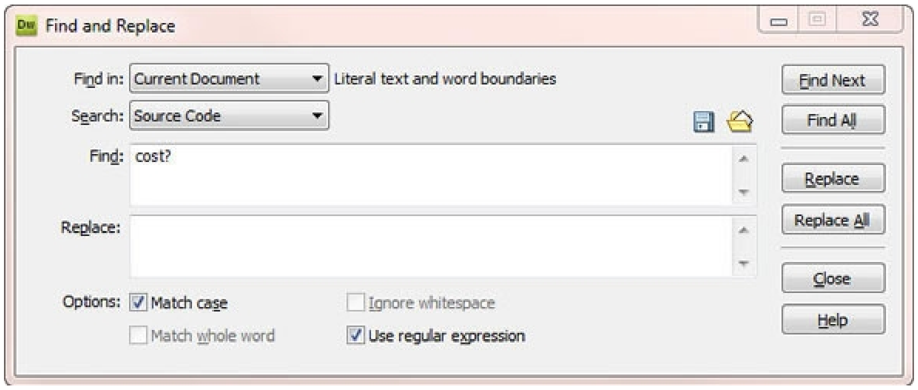
   This time, Dreamweaver skips the uppercase version, and highlights *cost* in the first paragraph.

8. Click Find Next again.

   Now, the first four letters of *costs* in the second paragraph are highlighted.

   No real surprises there, but what if you want to include the question mark at the end of the first paragraph?

9. Change the value in the Find text box by appending a question mark (see Figure 3).



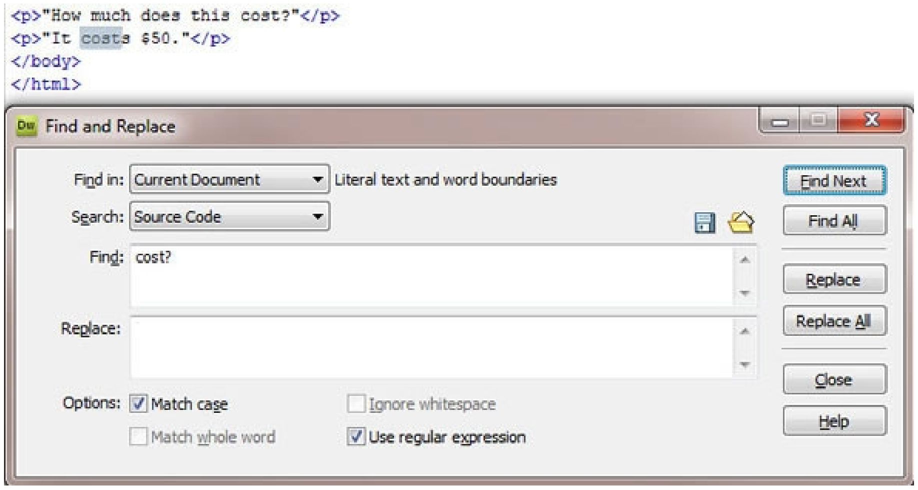
Find and Replace

Find in: Current Document — Literal text and word boundaries
Search: Source Code
Find: cost?
Replace:
Options: [✓] Match case [ ] Ignore whitespace [ ] Match whole word [✓] Use regular expression

Figure 3. A question mark has been added after the literal text.

10. Reposition the cursor above the body of the page, and click Find Next. Dreamweaver will highlight *cost* in the first paragraph, but *not* the question mark.
11. Click Find Next again.

    Dreamweaver selects *cost* in the second paragraph (see Figure 4).



```
<p>"How much does this cost?"</p>
<p>"It costs $50."</p>
</body>
</html>
```

Find and Replace

Find in: Current Document — Literal text and word boundaries
Search: Source Code
Find: cost?
Replace:
Options: [✓] Match case, [ ] Ignore whitespace, [ ] Match whole word, [✓] Use regular expression

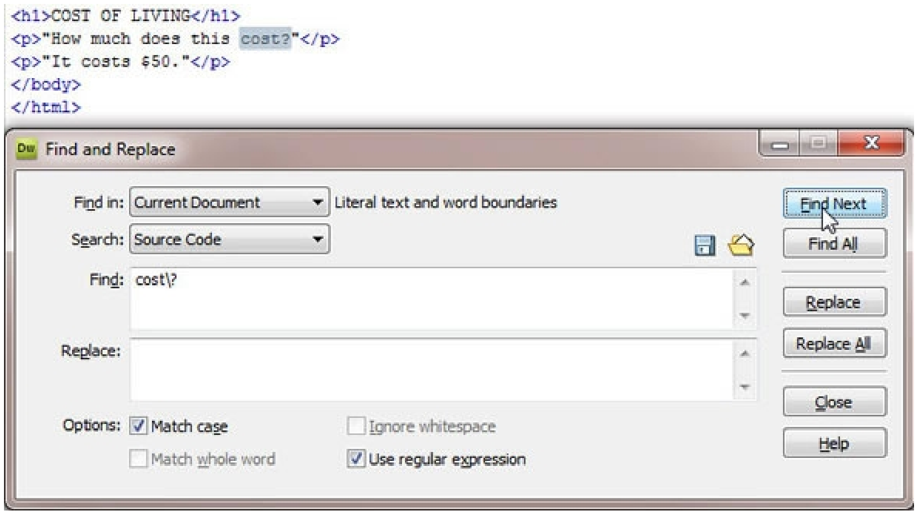
Figure 4. The regex selects cost in the second paragraph, even though it's not followed by a question mark.

Why is this happening? The question mark is one of the dozen characters that have a special meaning in a regex. In this context, it makes the preceding character optional. So, the regex matches *cos* or *cost*. It does not match *cost?*.

To match the question mark (or any of the dozen special characters), you need to *escape it* with a backslash.

12. Insert a backslash in front of the question mark; the Find text box will now contain **cost\?** .
13. Reposition the cursor and click Find Next again.

    This time, Dreamweaver selects both cost and the question mark that follows it in the first paragraph (see Figure 5).



```
<h1>COST OF LIVING</h1>
<p>"How much does this cost?"</p>
<p>"It costs $50."</p>
</body>
</html>
```

Find and Replace

Find in: Current Document — Literal text and word boundaries
Search: Source Code
Find: cost\?
Replace:
Options: [✓] Match case, [ ] Ignore whitespace, [ ] Match whole word, [✓] Use regular expression

Figure 5. Escaping the question mark with a backslash treats it as a literal character.

14. Click Find Next again.
15. Dreamweaver reports that it found only one item.

    The regex no longer matches *cost* in the second paragraph, because it's looking for *cost?*.

16. Delete the regex from the Find text box, and in its place type **$50**.
17. Reposition your cursor, and click Find Next.

    Dreamweaver reports that the value wasn't found.

18. Escape the dollar sign by preceding it with a backslash (**\$50**), and try again.

This time, the regex matches the *$50* in the second paragraph.

Finding different types of characters

The real power of regular expressions comes from the ability to find different types of characters. For example, `\d` matches any single-digit number. So, the following regex matches any 10-digit phone number formatted like *958 555-0123*:

```
\d\d\d \d\d\d-\d\d\d\d
```

This regex matches three digits and a literal space, followed by another three digits, a literal hyphen, and four more digits.

Area codes are often enclosed in parentheses. As you learned in the previous section, opening and closing parentheses are among the 12 special characters that need to be escaped with a backslash, so the following regex matches a number formatted like *(958) 555-1234*:

```
\(\d\d\d\) \d\d\d-\d\d\d\d
```

As a quick exercise, test this regex in Dreamweaver.

1. Open regex_02.html in Code view, and choose Edit > Find And Replace.
2. In the Find And Replace dialog box, make sure Use Regular Expression is selected.
3. Type **\(\d\d\d\) \d\d\d-\d\d\d\d** in the Find text box and click Find Next.

   Dreamweaver will highlight the first phone number.
4. Click Find Next again. Dreamweaver skips the next two phone numbers, and highlights the fourth.

   The second and third phone numbers are skipped because the parentheses around the area code are missing from the second one, and the third one doesn't have a hyphen. The pattern must match *exactly*, or the regex will fail.

   In the previous exercise, a question mark without a backslash made the preceding character optional. You can use the question mark to make the parentheses optional.
5. Delete the current regex and type **\(?\d\d\d\)? \d\d\d-\d\d\d\d** in the Find text box.
6. Test the regex again.

   This time, it should highlight the first, second, and fourth phone numbers.
7. Now try adding a question mark after the hyphen, so the regex looks like this:
   \(?\d\d\d\)? \d\d\d-?\d\d\d\d

```
\(?\d\d\d\)? \d\d\d-?\d\d\d\d
```

8. Test it again. It still doesn't get the third phone number. This is because the question mark means *match the preceding character or nothing at all*. The last four digits of the third phone number are preceded by a space, which is not a hyphen and not *nothing*, so the regex does not match.

You'll see later how to include the third phone number in the pattern.

Special wildcard characters (metacharacters)
The previous exercise introduced you to `\d`, one of the special wildcard characters used in regular expressions, but there are several others.

Table 1 describes the most important wildcard characters used in regular expressions. Strictly speaking, most of them are metasequences because they are made up of two characters. However, metasequences are commonly referred to as *metacharacters*.

Table 1. Commonly used metacharacters

Metacharacter	**Matches**
.	Any single character, except a newline
\d	Any digit character (0-9)
\D	Any non-digit character—in other words, anything except 0-9
\w	Any alphanumeric character or the underscore
\W	Any character, except an alphanumeric character or the underscore
\s	Any white-space character, including space, tab, form feed, or line feed
\S	Any character, except a white-space character
\f	A form feed character
\n	A line feed character

\r	A carriage return character
\t	A tab character

With the exception of the dot (or period), all the wildcard character sequences begin with a backslash. The dot metacharacter matches anything, including a space, punctuation mark, or even itself. The only thing it doesn't match is a newline. This is a common source of mistakes when composing regular expressions.

Note: ColdFusion is an exception to this rule. In ColdFusion, the dot metacharacter also matches a newline.

The other thing to note about Table 1 is that `\d`, `\w`, and `\s` each have opposites. The uppercase version matches anything not matched by the lowercase version: `\d` matches a number, `\D` matches anything except a number. If you get the case of your metacharacter wrong, the regex does the exact opposite of what you intended.

Defining what to match with character classes

Although metacharacters are very useful, you often want to be more selective. Character classes let you do just that. A character class allows you to specify a range of permitted characters. To create a custom character class, list the characters inside a pair of square brackets like this:

```
[aeiou]
```

This matches any vowel. So, `c[aeiou]t` matches *cat*, *cet*, *cit*, *cot*, and *cut*.

You can also use a character class to exclude specific characters by adding a caret or circumflex (^) immediately after the opening square bracket like this:

```
[^aeiou]
```

This excludes all vowels from a match. So, `to[^aeiou]` matches *top*, but not *too*.

The caret must come first. If it appears anywhere else in the character class, it's treated as a literal character. For example, `[ae^iou]` does not mean *either a or e, but not i, o, or u*. It means *any vowel or a caret*.

Typing out every character you want to include can be tedious, so character classes accept character ranges indicated by a hyphen. For example, `[a-z]` matches all lowercase letters in the alphabet. To match both uppercase and lowercase letters, use `[A-Za-z]`. Don't be tempted to use `[A-z]` for all uppercase and lowercase letters, because that includes several punctuation marks.

If you want to include a literal hyphen as part of a custom character class, put it first:

```
[-A-Za-z]
```

This matches a hyphen or any uppercase or lowercase letter.

You can also use metacharacters inside a character class. For example, `[\d.]` matches any number or a dot (period). Inside the character class, the dot no longer matches any single character. It matches a literal dot.

Inside a character class, the 12 special characters are reduced to just four:

- The backslash indicates a metacharacter.
- The caret indicates characters to be excluded, but only if it comes immediately after the opening square bracket.
- The hyphen indicates a range of characters unless it's the first character inside the class.
- The closing square bracket indicates the end of the character class, so to include a literal square bracket in the class, it must be escaped with a backslash.

Repeating patterns within a regex

Metacharacters and character classes represent only a single character, so you need a way to indicate how many times the match should be repeated. You specify how often to match something by following it with one of the quantifiers in Table 2.

Table 2. Quantifiers used to repeat a pattern

Quantifier	Meaning
*	Match 0 or more times
+	Match 1 or more times
?	Match no more than once (makes the character or group optional)
{n}	Match exactly *n* times
{n,m}	Match at least *n*, but no more than *m* times
*?	Match 0 or more times, but as few times as possible
+?	Match 1 or more times, but as few times as possible
??	Match 0 or 1 times, but as few times as possible
{n}?	Match at least *n* times, but as few times as possible
{n,m}?	Match at least *n* times, no more than *m* times, and as few times as possible

You can now improve the phone regex to avoid typing `\d` ten times. Plus, by creating a character class, you can also match the third phone number.

1. Open regex_02.html from the sample files, and type the following regex in the Find text box of the Find And Replace dialog box: **\(?\d{3}\)? \d{3}-\d{4}**

   It's not much shorter than the original regex, but the numbers make it easier to understand.
2. Click Find Next three times to select the phone numbers one by one.

   This regex is simply a different way of writing the previous version, so it still skips the third phone number.
3. Convert the hyphen between the last two groups of numbers to a character class that accepts either a hyphen or a space. Use a literal space inside the character class like this:
   \(?\d{3}\)? \d{3}[-]\d{4}

```
\(?\d{3}\)? \d{3}[- ]\d{4}
```

4. Test the regex again.

This time, it should match all four phone numbers.

Note: Instead of a literal space inside the character class, you could use the `\s` metacharacter. This is marginally easier to read, but it is less precise, because `\s` matches any white-space character, including a tab or line feed.

Greedy and lazy quantifiers

The most commonly used quantifiers (`*` and `+`) are *greedy* in the sense that they match as much as they can. As a result, it's easy to end up with a regex that grabs far more text than intended. The equivalent quantifiers that end with a question mark do the opposite. They're *lazy*, and grab as little as they can. Choosing the right type of quantifier is one of the biggest challenges in creating a regex.

Using parentheses to group characters

You can group characters and metacharacters together inside parentheses. Any quantifier placed immediately after the closing parenthesis is applied to the whole group.

Capturing groups and backreferences

Parentheses play another important role in addition to grouping characters and metacharacters. They remember what the characters inside them match, and store the value in a *backreference*. The backreference can then be used to find a repeated value or to preserve the original value in a find and replace operation.

The value of the first capturing group in a regex is stored as `\1`, the second as `\2`, and so on. If you go beyond nine capturing groups, the remaining backreferences are stored as `\10` up to a maximum of `\99`.

The final phone number in regex_02.html is (959) 555-0555. The last three digits of the number (555) are the same as the three digits immediately preceding the hyphen. Change the regex from the preceding exercise to this:

```
\(\d{3}\) (\d{3})[- ]\d\1
```

This surrounds the middle three digits with a pair of grouping parentheses, and uses `\1` as a backreference. The backreference tells the regex that the final three digits must match the middle three. If you test the regex, you'll see it matches only the final phone number in the sample page.

In Part 2, you'll see how capturing groups are used in find and replace operations.

Boundaries, anchors, and alternation

Regexes give you the power to determine whether a match should come at the beginning, end, or middle of a word. You can also use anchors to specify whether a match comes at the beginning or end of the subject text or line. Table 3 lists the boundary and anchor symbols, as well as the alternation character.

Table 3. Metacharacters for word boundaries, anchors, and alternation

Character/sequence	Meaning
\b	Match word boundary
\B	Match word non-boundary
^	Match beginning of subject text or line
$	Match end of subject text or line
|	Alternate pattern

Most regex flavors treat word boundaries to mean the boundary between a character that matches `\w` (unaccented characters in the Roman alphabet, numbers, and the underscore) and anything that doesn't match `\w`.

For example, in regex_01.html, the file used in the first exercise, `\bcost\b` matches *cost* in the first paragraph, but not in the second paragraph. The question mark in the first paragraph is not a word character, so it's treated as a word boundary.

For a match in the second paragraph, you need to use `\bcost\B`.

To match *cost* in *accosts*, you need `\Bcost\B`.

The beginning or end of a line is also considered a word boundary. However, the two anchors, `^` and `$`, allow you to specify that the match must be at the beginning or end of the line respectively.

Note: Dreamweaver's Find And Replace dialog box uses JavaScript, which does not support using `^` and `$` to match the beginning and end of lines. To match the beginning or end of a line in Dreamweaver, use the character class `[\r\n]`, which matches either a carriage return or a newline character.

The alternation character matches either of the values to its left or right. For example, `a|b` matches *a* or *b*. To match more than one character, wrap the expression in parentheses. For example, `(red|green)` matches *red* or *green*. As explained in the preceding section, using parentheses also captures the matched value. To create a non-capturing group, insert a question mark and a colon immediately after the opening parenthesis like this: `(?:red|green)`. This matches *red* or *green*, but does not store the result in a backreference.

Where to go from here

That covers most of the basics of building a regular expression. On a first read-through, it probably feels overwhelming, but things should start to pull together once you start putting regexes to practical use in Part 2 of this tutorial.

More Like This

Simple styling with CSS
HTML5 and CSS3 in Dreamweaver CS5.5 – Part 2: Styling the web page
Taking a Fireworks comp to a CSS-based layout in Dreamweaver – Part 2: Markup preparation
Taking a Fireworks comp to a CSS-based layout in Dreamweaver – Part 3: Layout and CSS
Designing with CSS – Part 5: Building site navigation
Small web team uses CSS to develop big-time magazine site
Tableless layouts with Dreamweaver CS4
Creating a simple three-column design
Working with images in Dreamweaver CS4
Understanding specificity

Tutorials and samples

Tutorials
Using the SSI extension in Dreamweaver to create page headers
Creating your first website – Part 5
Creating your first website – Part 1
Creating your first website – Part 2

Samples
Responsive design with jQuery marquee
Customizable starter design for jQuery Mobile
Customizable starter design for HTML5 video
Customizable starter design for multiscreen development