LEARNING REGULAR EXPRESSIONS

BEN FORTA

Learning Regular Expressions

Note

The example here worked properly. But is it correct? Think about it. Why were the U.S. ZIP codes not matched? Is it because they are made up of just digits, or is there some other reason?

I'm not going to give you the answer to this question because, well, the pattern worked. The key here is that there is rarely a right or wrong regular expression (as long as it works, of course). More often than not, there are varying degrees of complexity that correspond to varying degrees of pattern-matching strictness.

Matching Whitespace (and Nonwhitespace)

The final class you should look at is the whitespace class. Earlier in this lesson, you learned the metacharacters for specific whitespace characters. Table 4.4 lists the class shortcuts for all whitespace characters.

Table 4.4 **Whitespace Metacharacters**

Metacharacter	Description
`\s`	Any whitespace character (same as `[\f\n\r\t\v]`)
`\S`	Any nonwhitespace character (same as `[^\f\n\r\t\v]`)

Note

`[\b]`, the backspace metacharacter, is not included in `\s` or excluded by `\S`.

Specifying Hexadecimal or Octal Values

Although you'll not find yourself needing to refer to specific characters by their octal or hexadecimal value, it is worth noting that this is doable.

Using Hexadecimal Values

Hexadecimal (base 16) values may be specified by preceding them with `\x`. Therefore, `\x0A` (ASCII character `10`, the linefeed character) is functionally equivalent to `\n`.

Using Octal Values

Octal (base 8) values may be specified as two- or three-digit numbers proceeded by `\0`. Therefore, `\011` (ASCII character `9`, the tab character) is functionally equivalent to `\t`.

Note

Many regular expression implementations also allow the specification of control characters using `\c`. For example, `\cZ` would match `Ctrl-Z`. In practice, you'll find very little use for this syntax.

Using POSIX Character Classes

A lesson on metacharacters and shortcuts for various character sets would not be complete without a mention of the POSIX character classes. POSIX is a special set of standard character classes, and these are yet another form of shortcut that is supported by many (but not all) regular expression implementations.

Note

JavaScript does not support the use of POSIX character classes in regular expressions.

Table 4.5 **POSIX Character Classes**

Class	Description
`[:alnum:]`	Any letter or digit (same as `[a-zA-Z0-9]`)
`[:alpha:]`	Any letter (same as `[a-zA-Z]`)
`[:blank:]`	Space or tab (same as `[\t]`)
`[:cntrl:]`	ASCII control characters (ASCII `0` through `31` and `127`)
`[:digit:]`	Any digit (same as `[0-9]`)
`[:graph:]`	Same as `[:print:]` but excludes space
`[:lower:]`	Any lowercase letter (same as `[a-z]`)
`[:print:]`	Any printable character
`[:punct:]`	Any character that is in neither `[:alnum:]` nor `[:cntrl:]`
`[:space:]`	Any whitespace character including spaces (same as `[\f\n\r\t\v]`)
`[:upper:]`	Any uppercase letter (same as `[A-Z]`)
`[:xdigit:]`	Any hexadecimal digit (same as `[a-fA-F0-9]`)

The POSIX syntax is quite different from the metacharacters seen thus far. To demonstrate the use of POSIX classes, let's revisit an example from Lesson 3. The example used a regular expression to locate RGB values in a block of HTML code:

Text

```
body {
   background-color: #fefbd8;
}
h1 {
   background-color: #0000ff;
}
div {
   background-color: #d0f4e6;
}
span {
   background-color: #f08970;
}
```

RegEx

```
#[[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]][[:xdigit:]]
```

Result

```
body {
   background-color: #fefbd8;
}
h1 {
   background-color: #0000ff;
}
div {
   background-color: #d0f4e6;
}
span {
   background-color: #f08970;
}
```

Analysis

The pattern used in the previous lesson repeated the character set `[0-9A-Fa-f]` six times. Here each `[0-9A-Fa-f]` has been replaced by `[[:xdigit:]]`. The result is the same.

Note

Notice that the regular expression used here starts with `[[` and ends with `]]` (two sets of brackets). This is important and required when using POSIX classes. POSIX classes are enclosed within `[:` and `:]`; the POSIX we used is `[:xdigit:]` (not `:xdigit:`). The outer `[` and `]` are defining the set; the inner `[` and `]` are part of the POSIX class itself.

Caution

All 12 POSIX classes enumerated here are generally supported in any implementation that supports POSIX. However, there may be subtle variances from the preceding descriptions.

Summary

Building on the basics of character and set matching shown in Lessons 2 and 3, this lesson introduced metacharacters that match specific characters (such as tab or linefeed) or entire sets or classes of characters (such as digits or alphanumeric characters). These shortcut metacharacters and POSIX classes may be used to simplify regular expression patterns.

This page intentionally left blank

Lesson 5

Repeating Matches

In the previous lessons, you learned how to match individual characters using a variety of metacharacters and special class sets. In this lesson, you'll learn how to match multiple repeating characters or sets of characters.

How Many Matches?

You've learned all the basics of regular expression pattern matching, but all the examples have had one very serious limitation. Consider what it would take to write a regular expression to match an email address. The basic format of an email address looks something like the following:

```
text@text.text
```

Using the metacharacters discussed in the previous lesson, you could create a regular expression like the following:

```
\w@\w\.\w
```

The `\w` would match all alphanumeric characters (plus an underscore, which is valid in an email address); `@` does not need to be escaped, but `.` does.

This is a perfectly legal regular expression, albeit a rather useless one. It would match an email address that looked like `a@b.c` (which, although syntactically legal, is obviously not a valid address). The problem with it is that `\w` matches a single character and you can't know how many characters to test for. After all, the following are all valid email addresses, but they all have a different number of characters before the `@`:

```
b@forta.com
ben@forta.com
bforta@forta.com
```

What you need is a way to match multiple characters, and this is doable using one of several special metacharacters.