CHAPTER 3

Regular Expressions

Regular expressions (regex) are one of the black arts of practical computer programming. Ask any programmer, and chances are that he or she will, at some point, have had serious problems with them (or, even worse, avoided them altogether).

Yet, regular expressions, although complicated, are not really difficult to understand. Fundamentally, they are a way to describe *patterns* of text using a single set of strings. Unlike a simple search-and-replace operation, such as changing all instances of "Marco" with "Tabini," regex allows for much more flexibility—for example, finding all instances of the letters "Mar" followed by either "co" or "k," and so forth.

Regular expressions were initially described in the 1950s by a mathematician named S. C. Kleene, who formalized models first designed by Warren McCulloch and Walter Pitts to describe the nervous system. Regex, however, were not actually applied to computer science until Ken Thompson (who then went on to become one of the original designers of the UNIX operating system) used them as a means to search and replace text in his *qed* text editor.

Regular expressions eventually made their way into the UNIX operating system (and later into the POSIX standard) and into Perl, where they are considered one of the language's strongest features. PHP actually makes both standards available—the idea being that Perl programmers will feel right at home, and beginners will be able to use the simpler POSIX expressions.

The Basics of Regular Expressions

Regex is, essentially, a whole new language, with its rules, its structures, and its quirks. You'll also find that your knowledge of most other programming languages will have practically

IN THIS CHAPTER

no bearing on learning regex, for the simple reason that regular expressions are highly specialized and follow their own rules.

As defined by Kleene, the basic regex axioms are the following:

- A single character is a regular expression denoting itself.
- A sequence of regular expressions is a regular expression.
- Any regular expression followed by a * character (also known as "Kleene's Star") is a regular expression composed of zero or more instances of that regular expression.
- Any pair of regular expressions separated by a pipe character (|) is a regular expression composed of either the left or the right regular expression.
- Parentheses can be used to group regular expressions.

This may sound complicated to you, and I'm pretty positive that it scared me the first time I read through it. However, the basics are easy to understand. First, the simplest regular expression is a single character. For example, the regex `a` will match the character "a" of the word Marco. Notice that, under normal circumstances, regex are binary operations, so that "a" is *not* equivalent to "A". Therefore, the regex `a` will not match the "A" in MARCO.

Next, single-character regular expressions can be grouped by placing them next to each other. Thus, the regex `wonderful` will match the word "wonderful" in "Today is a wonderful day."

So far, regular expressions are not very different from normal search operations. However, this is where the similarities end. As I mentioned earlier, you can use Kleene's Star to create a regular expression that can be repeated any number of times (including none). For example, consider the following string:

```
seeking the treasures of the sea
```

The regex `se*` will be interpreted as "the letter s followed by zero or more instances of the letter e" and match the following:

- The letters "see" of the word "seeking," where the regex `e` is repeated twice.
- Both instances of the letter s in "treasures," where `s` is followed by zero instances of `e`.
- The letters "se" of the word "sea," where the `e` is present once.

It's important to understand that, in the preceding expression, only the expression `e` is considered when dealing with the star. Although it's possible to use parentheses to group regular expressions, you should not be tempted to think that using `(se)*` is a good idea, because the regex compiler will interpret it as meaning "zero or more occurrences" of "se."

If you apply this regex to the preceding string, you will encounter a total of 30 matches, because *every* character in the string would match the expression. (Remember? *Zero* or more occurrences!)

You will find that parentheses are often useful in conjunction with the pipe operator to specify alternative regex specifications. For example, use the expression `gr(u|a)b` with the following string:

```
grab the grub and pull
```

to match both "grub" and "grab."

Limitations of the Basic Syntax

Even though regular expressions are quite powerful because of the original rules, inherent limitations make their use impractical. For example, there is no regular expression that can be used to specify the concept of "any character." In addition, if you happen to have to specify a parenthesis or star as a regular expression—rather than as a special character—you're pretty much out of luck.

As a result of these limitations, the practical implementations of regular expressions have grown to include a number of other rules:

- The special character "^" is used to identify the beginning of the string.
- The special character "$" is used to identify the end of the string.
- The special character "." is used to identify the expression "any character."
- Any nonnumeric character following the character "\" is interpreted literally (instead of being interpreted according to its regex meaning). Note that this escaping technique is relative to the regex compiler, and *not* to PHP itself. This means that you must ensure that an actual backslash character reaches the regex functions by escaping it as needed (that is, if you're using double quotes, you will need to input `\\`). Any regular expression followed by a "+" character is a regular expression composed of one or more instances of that regular expression.
- Any regular expression followed by a "?" character is a regular expression composed of either zero or one instances of that regular expression.
- Any regular expression followed by an expression of the type {*min*[,|,*max*]} is a regular expression composed of a variable number of instances of that regular expression. The *min* parameter indicates the minimum acceptable number of instances, whereas the *max* parameter, if present, indicates the maximum acceptable number of instances. If only the comma is available, no upper limit exists to the number of instances that can be found in the string. Finally, if only *min* is defined, it indicates the *only* acceptable number of instances.

- Square brackets can be used to identify groups of characters acceptable for a given character position.

Let's start from the beginning. It's sometimes useful to be able to recognize whether a portion of a regular expression should appear at the beginning or at the end of a string. For example, suppose you're trying to determine whether a string represents a valid HTTP URL. The regex `http://` would match both `http://www.phparch.com`, which is a valid URL, and `nhttp://www.phparch.com`, which is not (and could easily represent a typo on the user's part).

By using the "^" special character, you can indicate that the following regular expression should be matched only at the beginning of the string. Thus, the regex `^http://` will create a match only with the first of the two strings.

The same concept—although in reverse—applies to the end-of-string marker "$", which indicates that the regular expression preceding it must end exactly at the end of the string. For example, `com$` will match "sams.com" but not "communication."

The special characters "+" and "?" work similarly to the Kleene Star, with the exception that they represent "at least one instance" and "either zero or one instances" of the regex they are attached to, respectively.

As I briefly mentioned earlier, having a "wildcard" that can be used to match any character is extremely useful in a wide range of scenarios, particularly considering that the "." character is considered a regular expression in its own right, so that it can be combined with the Kleene Star and any of the other modifiers. For example, the expression

```
.+@.+\..+
```

can be used to indicate:

> At least one instance of any character, followed by
>
> The "@" character, followed by
>
> At least one instance of any character, followed by
>
> The "." character, followed by
>
> At least one instance of any character.

As you might have guessed, this expression is a very rough form of email address validation. Note how I have used the backslash character (\) to force the regex compiler to interpret the penultimate "." as a literal character, rather than as another instance of the "any character" regular expression.

However, that is a rather primitive way of checking for the validity of an email address. After all, only letters of the alphabet, the underscore character (_), the minus character

(–), and digits are allowed in the name, domain, and extension portion of an email. This is where the range denominators come into play.

As mentioned previously, anything within nonescaped square brackets represents a set of alternatives for a particular character position. For example, `[abc]` indicates either an "a", a "b", or a "c". However, representing something like "any character" by including every possible symbol in the square brackets would give birth to some ridiculously long regular expressions—and regex are complex enough as it is.

Luckily, it's possible to specify a "range" of characters by separating them with a dash. For example, `[a-z]` means "any lowercase character." You can also specify more than one range and combine them with individual characters by placing them side-by-side. For example, our email validation requirements can be satisfied by the expression `[A-Za-z0-9_]`, which turns the overall regex into

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]+
```

The range specifications that we have seen so far are all *inclusive*—that is, they tell the regex compiler which characters *can* be in the string. Sometimes, it's more convenient to use *exclusive* specifications, dictating that any character *except* the characters you specify are valid. This can be done by prepending a caret character (^) to the character specifications inside the square bracket. For example, `[^A-Z]` means "any character except any uppercase letter of the alphabet."

Going back to the email validation regex, it's still not as good as it could be. For example, we know for sure that a domain extension (for example, .ca or .com) must have a minimum of two characters (as in .ca) and a maximum of four (as in .info). We can therefore use the minimum-maximum length specifier that I introduced earlier to specify this additional requirement:

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{2,4}
```

Naturally, you may want to allow only email addresses that have a three-letter domain (such as .com). This can be accomplished by omitting the comma and *max* parameters from the length specifiers:

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{3}
```

If, on the other hand, you would like to leave the maximum number of characters open in anticipation of the fact that longer domain extensions may be introduced in the future, you could use the following regex:

```
[A-Za-z0-9_]+@[A-Za-z0-9_]+\.[A-Za-z0-9_]{3,}
```

This indicates that the last regex in the expression should be repeated at least a minimum of three times, with no fixed upper limit.

POSIX Regular Expressions

The regular expression standard that made its way through the POSIX standard is perhaps the simplest form of regex available to PHP programmers. As such, it makes a great learning tool because the functions that implement it do not provide any particular "advanced" features.

In addition to the standard rules that we have already discussed, the POSIX regex standard defines the concept of *character classes* as a way to make it even easier to specify character ranges. Character classes are always enclosed in a set of colon characters (:) and must be enclosed in square brackets. There are 12 character classes:

- **alpha** represents a letter of the alphabet (either upper- or lowercase). This is equivalent to `[A-Za-z]`.
- **digit** represents a digit between 0–9 (equivalent to `[0-9]`).
- **alnum** represents an alphanumeric character, just like `[0-9A-Za-z]`.
- **blank** represents "blank" characters, normally space and Tab.
- **cntrl** represents "control" characters, such as DEL, INS, and so forth.
- **graph** represents all the printable characters except the space.
- **lower** represents lowercase letters of the alphabet only.
- **upper** represents uppercase letters of the alphabet only.
- **print** represents all printable characters.
- **punct** represents punctuation characters such as "." or ",".
- **space** is the whitespace.
- **xdigit** represents hexadecimal digits.

This makes it possible, for example, to rewrite our email validation regex as follows:

```
[[:alnum:]_]+@[[:alnum:]_]+\.[[:alnum:]_]{2,4}
```

This notation is much simpler, and it makes mistakes a *little* less obvious.

Another important concept introduced by the POSIX extension is the *reference*. Earlier in the chapter, we have already had a chance to see how parentheses can be used to group regular expressions. When you do so in a POSIX regex, when the expression is executed the interpreter assigns a numeric identifier to each grouped expression that is matched. This identifier can later be used in various operations—such as finding and replacing.

For example, consider the following string and regular expression:

```
marcot@tabini.ca

([[:alpha:]]+)@([[:alpha:]]+)\.([[:alpha:]]{2,4})
```

The regex should match the preceding email address. However, because we have grouped the username, the domain name and the domain extensions will each become a reference, as shown in Table 3.1.

TABLE 3.1 Regex References

Reference Number	Value
0	marcot@tabini.ca (the string matches by the entire regex)
1	marcot
2	tabini
3	ca

PHP provides support for POSIX through functions of the `ereg*` class. The simplest form of regex matching is performed through the `ereg()` function:

```
ereg (pattern, string[, matches)
```

The `ereg` function works by compiling the regular expression stored in `pattern` and then comparing it against *`string`*. If the regex is matched against *`string`*, the result value of the function is TRUE—otherwise, it is FALSE. If the `matches` parameter is specified, it is filled with an array containing all the references specified by `pattern` that were found in *`string`* (see Listing 3.1).

LISTING 3.1 Filling Patterns with `ereg`

```
<?php

    $s = 'marcot@tabini.ca';

    if (ereg ('([[:alpha:]]+)@([[:alpha:]]+)\.([[:alpha:]]{2,4})', $s, $matches))
    {
      echo "Regular expression successful. Dumping matches\n";
      var_dump ($matches);
    }
    else
    {
      echo "Regular expression unsuccessful.\n";
    }

?>
```

If you execute the preceding script, you should see this result:

```
Regular expression successful. Dumping matches
array(4) {
  [0]=>
  string(16) "marcot@tabini.ca"
  [1]=>
  string(6) "marcot"
  [2]=>
  string(6) "tabini"
  [3]=>
  string(2) "ca"
}
```

This indicates that the regular expression was successfully matched against the string stored in `$s` and returned the various references in the `$matches` array.

If you're not interested in case-sensitive matching (and you don't want to have to specify all characters twice when creating a regular expression), you can use the `eregi` function instead. It accepts the same parameters and behaves the same way as `ereg()`, with the exception that it ignores the case when matching a regular expression against a string (see Listing 3.2):

LISTING 3.2 Case-insensitive Pattern Matching

```
<?php

    $a = "UPPERCASE";

    echo (int) ereg ('uppercase', $a);
    echo "\n";
    echo (int) eregi ('uppercase', $a);
    echo "\n";

?>
```

The first regex will fail because `ereg()` performs a case-sensitive match against the contents of `$a`. The second regex, however, will be successful, because the `eregi` function performs its matches using an algorithm that is not case sensitive.

References make regular expressions an even more effective tool for handling search-and-replace operations. For this purpose, PHP provides the `ereg_replace` function, and its cousin `eregi_replace()`, which is not case sensitive:

```
ereg_replace (pattern, replacement, string);
```

The `ereg_replace()` function first matches the regular expression *`pattern`* against *`string`*. Then, it applies the references created by the regular expression in *`replacement`* and returns the resulting string. Here's an example (see Listing 3.3):

LISTING 3.3 Using `ereg_replace`

```
<?php

    $s = 'marcot@tabini.ca';

    echo ereg_replace ('([[:alpha:]]+)@([[:alpha:]]+)\.([[:alpha:]]{2,4})',
      '\1 at \2 dot \3', $s)

?>
```


If you execute this script, it will return the following string:

```
marcot at tabini dot ca
```

As you can see, the three references are extracted from the contents of `$s` by the regex compiler and used to substitute the placeholders in the replacement string.

Perl-Compatible Regular Expressions (PCRE)

Perl Compatible Regular Expressions (PCRE) are much more powerful than their POSIX counterparts—and consequently, also more complex and difficult to use.

PCRE adds its own character classes to the extended regular expression rules that we saw earlier:

- `\w` represents a "word" character and is equivalent to the expression `[A-Za-z0-9]`.
- `\W` represents the opposite of \w and is equivalent to `[^A-Za-z0-9]`.
- `\s` represents a whitespace character.
- `\S` represents a nonwhitespace character.
- `\d` represents a digit and is equivalent to `[0-9]`.
- `\D` represents a nondigit character and is equivalent to `[^0-9]`.
- `\n` represents a newline character.
- `\r` represents a return character.
- `\t` represents a tab character.

As you can see, PCRE are significantly more concise than their POSIX counterparts. In fact, our simple email validation regex can now be written as

```
/\w+@\w+\.\w{2,4}/
```

But, wait a minute—what are those slash characters at the beginning and at the end of the regex string? PCRE requires that the actual regular expression be *delimited* by two

characters. By convention, two forward slashes are used, although any character other than the backslash that is not alphanumeric would do just as well.

Naturally, regardless of which character you choose, you will be required to escape the delimiter whenever you use it as part of the regex itself. For example:

```
/face\/off/
```

is the equivalent of the regular expression `face/off`.

PCRE also expands on the concept of references, making them useful not only as a byproduct of the regex operation, but as part of the operation itself.

In PCRE, it is possible to use a reference that was defined previously in a regular expression as part of the expression itself. Let's make an example. Suppose that you find yourself in a situation in which you have to verify that in a string such as the following:

```
Marco is a programmer. Marco's specialty is programming.
John is a programmer. John's specialty is programming.
```

The name of the person to whom the sentence refers is the same in both positions (that is, "Marco" or "John"). Using a normal search-and-replace operation would take a significant effort, and so would using a POSIX regex, because you do not know the name of the person a priori.

With a PCRE, however, this operation is trivial. You start by matching the first portion of the string. The name is the first word:

```
/^(\w+) is a programmer.
```

Next, you specify the name again. As you can see, we included it in parentheses in the preceding expression, which means that we create a reference to it. We can now recall that reference *inside the regex itself* and use it to our advantage:

```
/^(\w+) is a programmer. \1's specialty is programming.$/
```

If you try to match the preceding regex against the following sentence:

```
Marco is a programmer. Marco's specialty is programming.
```

Everything will work fine. However, if you try it against this sentence:

```
Marco is a programmer. John's specialty is programming.
```

The regex compiler will not return a match because the reference won't match.

To give you an idea of how powerful PCREs are and why it's worth trying to learn them, let me give you an alternative to the simple one-line expression using POSIX:

```
<?php

    $s = 'Marco is a programmer. Marco\'s specialty is programming.';

    if (ereg ('^([[:alpha:]]+) is a programmer', $s, $matches)) {
      if (ereg ('([[:alpha:]]+)\'s specialty is programming.$', $s, $matches2)) {
        if ($matches[1] === $matches[1]) {
          echo "MATCH\n";
        } else {
          echo "NO MATCH\n";
       } else {
          echo "NO MATCH\n";
    } else {
      echo "NO MATCH\n";
      }
?>
```


Now, this is a simple example, and the POSIX solution is definitely not as elegant as it could be, but you can see here that it takes three separate operations to approximate the power of just one PCRE.

I should note that the inability to use references within the regex itself is actually a limitation of PHP, rather than of the POSIX standard—which, unfortunately, means that the PHP implementation of regex is not POSIX compliant.

The main PCRE function in PHP is `preg_match()`:

```
preg_match (pattern, string[, matches[, flags]]);
```

As in the case of `ereg()`, this function causes the regular expression stored in *`pattern`* to be matched against *`string`*, and any references matches are stored in *`matches`*. The optional *`flags`* parameter can actually contain only the value `PREG_OFFSET_CAPTURE`. If this parameter is specified, it will cause `preg_match()` to change the format of *`matches`* so that it will contain both the text *and* the position of each reference inside *`string`*. Let's make an example:

```
<?php

    $s = 'Another beautiful day';

    preg_match ('/beautiful/', $s, $matches, PREG_OFFSET_CAPTURE);

    var_dump ($matches);

?>
```

If you execute this script, you should receive the following output:

```
array(1) {
  [0]=>
  array(2) {
    [0]=>
    string(9) "beautiful"
    [1]=>
    int(8)
  }
}
```

As you can see, the `$matches` array now contains another array for each reference. The latter, in turn, contains both the string matched and its position within `$s`.

Another function of the PCRE family is `preg_match_all`, which has the same syntax as `preg_match()`, but searches a string for *all* the occurrences of a regular expression, rather than for a specific one. Here's an example:

```
<?php

$s = 'A beautiful day and a beauty of a lake';

preg_match_all ('/beaut[^ ]+/', $s, $matches);

var_dump ($matches)

?>
```

If you execute this script, it will output the following:

```
array(1) {
  [0]=>
  array(2) {
    [0]=>
    string(9) "beautiful"
    [1]=>
    string(6) "beauty"
  }
}
```

As you can see, the `$matches` array contains an array whose elements are arrays that correspond to the matches found for each of the references. In this case, because no reference was specified, only the 0th element of the array is present, but it contains both the string "beautiful" and "beauty". By contrast, if you had executed this regex using `preg_match()`, only the word "beautiful" would have been returned.

Search-and-replace operations in the world of PCRE are handled by the `preg_replace` function:

```
preg_replace (pattern, replacement, string[, limit]);
```

Much like `ereg_replace()`, this function applies the regex *`pattern`* to *`string`* and then substitutes the placeholders in *replacement* with the references defined in it. The *`limit`* parameter can be used to limit the number of replacements to a maximum number. Here's an example, which will output `marcot at tabini dot ca`:

```
<?php

    $s = 'marcot@tabini.ca';

    echo preg_replace ('/^(\w+)@(\w+)\.(\w{2,4})/', '\1 at \2 dot \3', $s);

?>
```

Keep in mind that this is only one way of using `preg_replace()`, in which the entire input string is substituted by the replacement string. In fact, you can use this function to replace only small portions of text:

```
<?php

    $s = 'The pen is on the table';

    echo preg_replace ('/on/', 'over', $s);

?>
```

If you execute this script, `preg_replace()` will replace the word "on" with the word "over" in `$s`, resulting in the output `The pen is over the table`.

The last function that I want to bring to your attention is `preg_split()`, which is somewhat equivalent to the `explode()` function that we discussed earlier, with the difference that it takes a regular expression as a delimiter, rather than a straight string, and that it includes a few additional features:

```
preg_split (pattern, string[, limit[, flags]]);
```

The `preg_split` function works by breaking *`string`* in substrings delimited by sequences of characters delimited by *`pattern`*. The optional *`limit`* parameter can be used to specify a maximum number of splitting operations. The flags parameter, on the other hand, can be used to modify the behavior of the function as described in Table 3.2.

TABLE 3.2 `preg_split()` Flags

Reference Number	Value
`PREG_SPLIT_NO_EMPTY`	Causes empty substrings to be discarded.
`PREG_SPLIT_DELIM_CAPTURE`	Causes any references inside *`pattern`* to be captured and returned as part of the function's output.
`PREG_SPLIT_OFFSET_CAPTURE`	Causes the position of each substring to be returned as part of the function's output (similar to `PREG_OFFSET_CAPTURE` in `preg_match()`).

Here's an example of how `preg_split()` can be used:

```
<?php

    $s = 'Ten times he called, and ten times nobody answered';

    var_dump (preg_split ('/[ ,]/', $s));

?>
```

This script causes the string `$s` to be split whenever either a space or a comma is found, resulting in the following output:

```
array(10) {
  [0]=>
  string(3) "Ten"
  [1]=>
  string(5) "times"
  [2]=>
  string(2) "he"
  [3]=>
  string(6) "called"
  [4]=>
  string(0) ""
  [5]=>
  string(3) "and"
  [6]=>
  string(3) "ten"
  [7]=>
  string(5) "times"
  [8]=>
  string(6) "nobody"
  [9]=>
  string(8) "answered"
}
```

As you can imagine, the `explode()` function by itself would have been inadequate in this case, because it would have been able to split `$s` based only on a single character.

Named Patterns

An excellent and very useful addition to PCRE is the concept of *named* capturing groups (which everybody always refers to as *named patterns*). A named capturing group lets you refer to a subpattern of your expression by an arbitrary name, rather than by its position inside the regular expression. For example, consider the following regex:

```
/^Name=(.+)$/
```


Now, you would normally address the `(.+)` subpattern as the first item of the match array returned by `preg_match()` (or as `$1` in a substitution performed through a call to `preg_replace()` or `preg_replace_all()`).

That's all well and good—at least as long as you have only a limited number of subpatterns whose position never changes. Heaven forbid, however, that you should ever find yourself in a position to have to add a capturing subpattern at the beginning of a regex that already has six of them!

Luckily, this problem can be solved once and for all by assigning a "name" to each of your subpatterns. Take a look at the following:

```
/^Name=(?P<thename>.+)$/
```

This will create a backreference inside your expression that can be explicitly retrieved by using the name `thename`. If you run this regex through `preg_match()`, the backreference will be inserted in the match array both by number (using the normal numbering rules) and by name. If, on the other hand, you run it through `preg_replace()`, you can backreference it by enclosing it in parentheses and prefixing it with `?P=`. For example:

```
preg_replace ("/^Name=(?P<thename>.+)$/", "My name is (?P=thename)", $value);
you may want to include an example of this functionality.
```

PCRE Modifiers

Remember when I mentioned that you need delimiters to specify a PCRE? If you were wondering why, here's an explanation. PCRE introduces the concept of "modifiers" that can be appended to a regular expression to alter the behavior of the regex compiler and/or interpreter. A modifier is always appended at the end of an expression, right after the delimiter. For example, in the following regex:

```
/test/i
```

the last `i` is a modifier.

There are *many* different modifiers. Perhaps the most commonly used one is `i`, which renders the regular expression non case sensitive. Here's an example of how it works:

```
<?php

    $s = 'Another beautiful day';

    echo (preg_match ('/BEautiFul/i', $s) ? 'MATCH' : 'NO MATCH') . "\n";

?>
```

If you execute the preceding script, it will output the word "MATCH", indicating that the regex succeeded because the `i` modifier made it not case sensitive.

Another commonly used—and *extremely* powerful—modifier is `e`, which, used in conjunction with `i` modifier, causes the regex compiler to interpret the *replacement* parameter not as a simple string but as a PHP expression that is executed and whose result is used as the replacement string.

Here's an example that shows you just how powerful this modifier is:

```
<?php

    $a = array
    (
      'name'  =>  'Toronto',
      'object'=>  'town'
    );

    $s = '{name} is a really cool {object}';

    echo preg_replace ('/{(\w+)}/e', '$a["\1"]', $s);

?>
```

When you execute this script, `preg_replace()` finds all instances of alphanumeric strings delimited by `{` and `}`, replaces the reference they create in `$a["\1"]`, executes the resulting PHP expression, and replaces its value in the original string.

Let's make a step-by-step example. The first match in the regex will be the substring `name`, which is then placed in the replacement string, thus providing the PHP expression `$a["name"]`. The latter, when executed, returns the value `Toronto`, which is then substituted inside the original string. The same process is repeated for the second match `object`, and the final result is then returned:

```
Toronto is a really cool town
```

Imagine how much more complex doing something like this would have been without regular expressions and the `e` modifier!

A Few Final Words

When you understand how they work, regular expressions become the best thing since the invention of the wheel. However, you will find that getting to master regex is a long and difficult process, and it takes a while before the actual reasoning behind how they work starts sinking into your brain.

Generally speaking, the most difficult aspect of regular expressions is debugging them, because PHP doesn't really provide you with any facility to do so, and the language doesn't lend itself well to simple bug-finding techniques (like printing out a result at various stages of the execution). As a result, the best way to debug a regular expression is to get it right the first time. The approach that I recommend is to start small with a simple "core" of your regex and make sure that works without any problem. You can then add to it, one step at a time and checking your work every time, until you've reached the intended result. This way, it's more difficult to let the situation get out of control and lose track of what your expression does.

Another important thing to understand about regex is that they are *not* a panacea. Regular expressions are slower than straight string substitution functions and should therefore be used only when the latter are unable to provide a viable alternative. Finally, Perl regular expressions are often much faster than their POSIX counterparts. As a result, even though they are a bit more complicated and may take a while longer to master, you should consider making the effort and using the former as often as possible.