Introduction to Regular Expressions

Text is a crucial part of many people's work with computers. From writing documents to editing code, text is almost everywhere. Web pages commonly consist largely of text, some of which is Hypertext Markup Language (HTML) or Extensible Hypertext Markup Language (XHTML) markup and some of which is regular text, but all of it consists of sequences of characters that can be matched using regular expressions. Forms on the Web accept text as input, which can be matched against allowable input. Business documents consist of text, and searches for specific sequences of characters can be made using regular expressions. E-mail messages consist of text. Developers' code consists of text. And regular expressions can be beneficially used in many situations where text is used.

Not only is text everywhere, but there also is lots of it, and increasingly, text must be updated or aggregated. As the volume of text created or to which you have access increases, you need efficient and effective ways to find text of particular interest or to change specific pieces of text.

Finding and changing individual pieces of text can be straightforward if you are dealing with a single document only a page or two in length. It becomes a more daunting task, potentially prone to human error, if you are dealing with dozens of documents, each hundreds of pages in length, or with thousands of relatively short documents. It is for tasks such as this that regular expressions are used, because regular expressions allow automation of many useful types of text processing.

For example, in a Web form you will want to check that a credit card number is correctly structured or that a postal code is correctly formed. In a lengthy document, you might want to find a hazily recalled URL for an important source of information. You might want to convert HTML code so that it conforms to the rules of Extensible Markup Language (XML) syntax and complies with company policy to use XHTML code. You might want to check that user input into a Windows application satisfies necessary criteria to allow correct processing.

In this chapter, you will learn the following:

- What regular expressions are
- What regular expressions can be used for
- Why regular expressions can seem daunting

The list of possible uses for a tool that allows the manipulation of text is almost endless, with text being so widespread. Sadly, many computer users and developers have little or no knowledge of regular expressions and how they can help in working with text. This book aims to change that.

What Are Regular Expressions?

Regular expressions are patterns of characters that match, or fail to match, sequences of characters in text. To allow developers to create regular expression patterns, certain characters and combinations of characters have special meanings and uses, and this book spends considerable time looking at those. But first, here are some more basic ideas.

Regular expressions, at the most basic level, allow computer users and developers to find desired pieces of text and, often, to replace those pieces of text with something that is preferred. At other times, regular expressions are used to test whether a sequence of characters that might be intended to be a credit card number or a Social Security number has an allowed pattern of characters. Whether it's finding existing sequences of characters or testing sequences of characters for their suitability (or not) for storage, the key aspect of regular expressions is matching a pattern against a sequence of characters.

It is reasonable, in a broad sense, to refer to a regular expression language, but strictly speaking, there is no regular expression language. Like scripting languages such as JavaScript and VBScript, which can be used only in the context of another application or language, regular expressions can be used only in the context of a "proper" programming language, including scripting languages, or as part of an application such as Microsoft Word and OpenOffice.org Writer or a command-line utility such as the `findstr` utility. Regular expressions can be discussed in an abstract way, but they are *used* together with another language or application.

This chapter focuses on very simple examples of regular expressions. Chapter 2 takes a closer look at regular expression tools. For the moment, either download the latest version of OpenOffice.org from `www.openoffice.org` to try out the simple examples in this chapter or simply read the text and examine the figures carefully. I encourage you to download OpenOffice.org, because it is a very simple tool to allow you to explore many aspects of regular expression syntax.

Try It Out Matching Literal Characters

The simplest type of regular expression pattern is a sequence of characters. For example, if you want to find the sequence of three characters `car`, you can use a regular expression pattern `car` to find those characters.

First, try to express the problem in plain English:

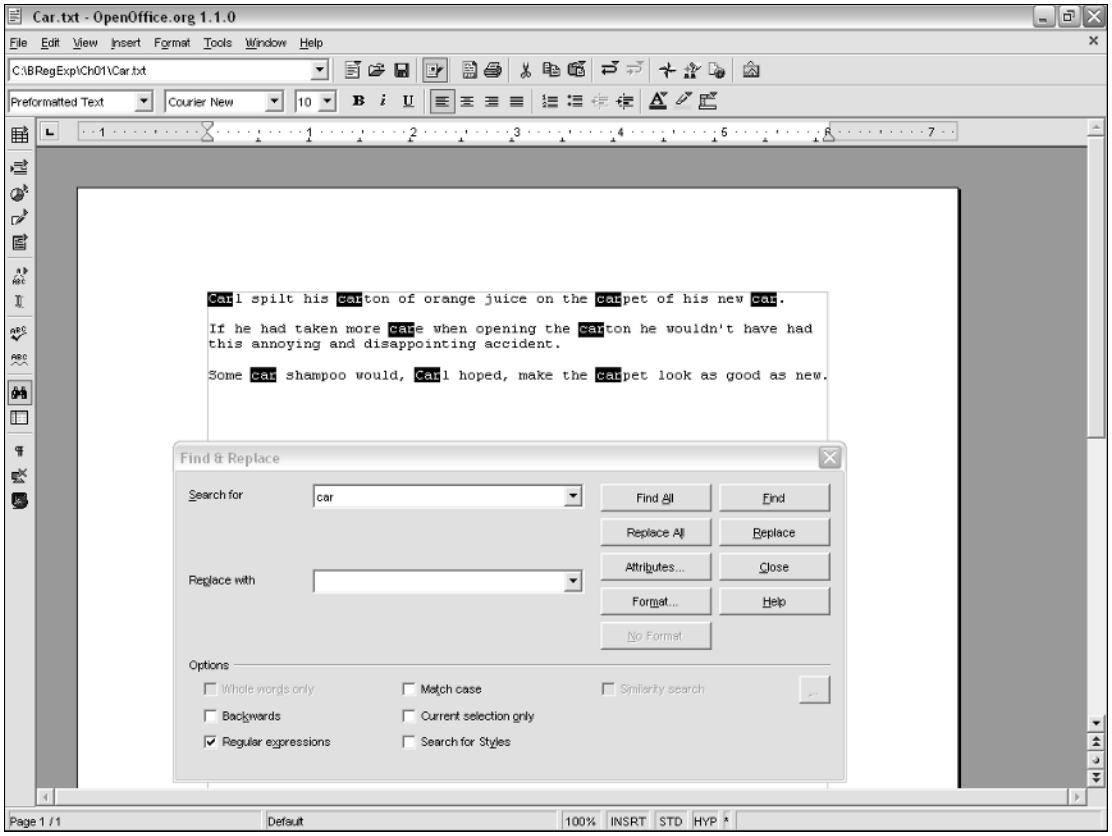
Match a sequence of characters; first match the letter `c`, followed by the letter `a`, followed by the letter `r`.

Suppose that you had the following text in a document, `Car.txt`:

```
Carl spilt his carton of orange juice on the carpet of his new car.
If he had taken more care when opening the carton he wouldn't have had this
annoying and disappointing accident.
Some car shampoo would, Carl hoped, make the carpet look as good as new.
```

There are many occurrences of the sequence of characters `car`, as shown in a simple regular expressions search in OpenOffice.Org Writer in Figure 1-1. To try it out for yourself, follow these steps:

1. Open `Car.txt` in OpenOffice.org Writer (regular expressions are supported in version 1.1 and above).
2. Use Ctrl+F to open the Find and Replace dialog box.
3. Check the Regular Expressions check box.
4. Enter **car** in the Search For text box.



```
Carl spilt his carton of orange juice on the carpet of his new car.

If he had taken more care when opening the carton he wouldn't have had
this annoying and disappointing accident.

Some car shampoo would, Carl hoped, make the carpet look as good as new.
```

Search for: car

[✓] Regular expressions

Figure 1-1

As you can see in Figure 1-1, the sequences of characters `car` have been selected whether or not they formed a word and whether the initial character of the sequence was uppercase or lowercase.

The following table shows a more formal breakdown for the very simple regular expression pattern `car`.

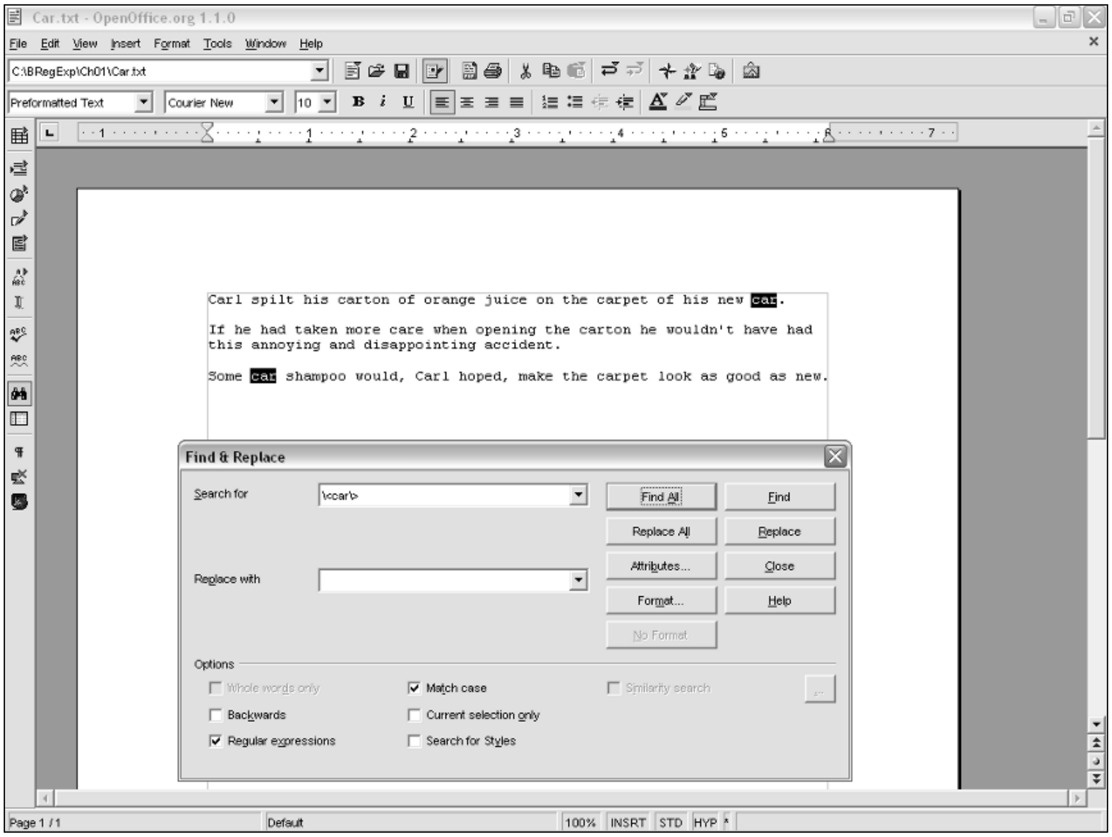
When you have simple literal patterns such as `car`, a formal layout of the meaning of a regular expression seems like overkill, but when you begin to create significantly more complex regular expression patterns later in the book, laying out each part of a regular expression helps you keep track of the pieces.

Letter	Instruction
c	Match the letter `c`
a	Match the letter `a`
r	Match the letter `r`

In short documents, as in this example, whether matches that are whole words or simply character sequences, if you use the Find All option in a search in OpenOffice.org Writer, you can quickly scan all the matches by eye, because they are shown in reversed highlight.

However, regular expressions can be used to tighten up the match. For example, you might want to find occurrences of the sequence of characters `car` that make up a word, but you don't want to find it when it is part of another word. And you might also want to make the search case sensitive.

You can do both in OpenOffice.org Writer by using the regular expression `\<car\>` and checking the Match Case check box, as shown in Figure 1-2. The `\<` and `\>` metacharacters simply match the boundary of a word.



```
Carl spilt his carton of orange juice on the carpet of his new car.

If he had taken more care when opening the carton he wouldn't have had
this annoying and disappointing accident.

Some car shampoo would, Carl hoped, make the carpet look as good as new.
```

Search for: \<car\>

Replace with:

Options
- [ ] Whole words only
- [✓] Match case
- [ ] Similarity search
- [ ] Backwards
- [ ] Current selection only
- [✓] Regular expressions
- [ ] Search for Styles

Figure 1-2

Don't worry too much about the syntax for matching the positions at the beginning and end of words for the moment. You'll see the various forms of syntax for that in Chapter 6.

What Can Regular Expressions Be Used For?

The potential number of uses for regular expressions is enormous. This section briefly describes some examples of what regular expressions can be used for.

Finding Doubled Words

Regular expressions can be used to find text where words have been doubled. In some text, such as the following sentence, a doubled word may be intentional:

It is for tasks such as that that regular expressions are used.

The doubling of the word *that* is how I wanted to express an idea. In other situations, the doubling of a word is inappropriate and undesired, as in the following:

Paris in the the Spring

The techniques to find doubled words are described in detail in Chapter 7.

In some settings, you don't need regular expressions to identify doubled words. In the preceding phrase, the second `the` is often underlined with a wavy red line in Microsoft Word, for example (depending on settings).

Checking Input from Web Forms

Another common use of regular expressions is to check that data entered in Web forms conforms to a structure that will be acceptable to the server-side process to which the form data will be submitted. For example, suppose someone attempts to enter the following as a supposed U.S. Social Security number (SSN):

```
Fred-123-4567
```

You can be confident that it isn't a valid SSN, and you want to be sure that it isn't entered into a back-end database. If you want to check the supposed SSN client side, you could use JavaScript and its `RegExp` object to check whether a string entered by a user conforms to a pattern that is a valid SSN.

One of the reasons for checking input from Web forms is that the data collected will be sent to a database, likely a relational database. If the wrong datatype is entered in a field in a form, you might end up with an attempt being made to enter a name into a date column in the database, which will likely cause an error when the attempt is made to write the data to the database. You also need to be able to make checks to ensure that you don't allow dates such as 2005-02-31, because there are never 31 days in February. The data you collect from a form is simply a sequence of characters; therefore, regular expressions are ideal to ensure that inappropriate data is detected on the client side and that the user is asked to enter appropriate data in place of the erroneous data.

Changing Date Formats

Imagine that you are "translating" a business document from U.S. English to British English. One of the components of the document's text may represent dates, and you will need to locate and, very possibly, change those dates, because the conventional representation of dates in the United States and in the U.K. differs. For example, in the United States, the date for Christmas Day 2001 would be written as `12/25/2001`. In the U.K., this might be written as `25/12/2001`. If you also had to represent dates for Japanese customers, you might express the same date as `2001-12-25`.

Assuming that you had a document with dates using the U.S. English conventions, you could create a regular expression to detect those sequences of characters wherever they occurred in the document. Depending on what the desired output format is, you could also replace a U.S. English date on the input side with a British English or Japanese date on the output side.

Finding Incorrect Case

Because there are a lot of jargon and acronyms associated with computing, it is very easy for incorrect case to creep into documents. This can happen either because of a word processor attempting to autocorrect what it imagines (wrongly) are incorrect doubling of uppercase (capital) letters. The sample document shown here, `XPath.txt`, is designed to illustrate one of the problems that can creep into technical documents.

```
Xpath is an abbreviation for the XML Path Language.

XPath is used to navigate around a tree model of an XML document.

There are significant differences in the data model of XpatH 1.0 and Xpath 2.0.

XSLT is one of the technologies with which xpath is often used.
```

The correct way to write `XPath` is with two uppercase initial letters. As you can see, the sample document has several incorrect forms of the word due to errors in the case of one or more characters.

The sensible approach to a problem like this depends, in part, on whether the word at issue can also be used in normal English. In the case of XPath there is only one correct form, and it doesn't occur as a word in normal English. That allows you to simply find all occurrences of the characters `xpath`, whether upper- or lowercase, and replace them with the correct sequence of characters, `XPath`.

There are several possible approaches to problems of this type, and you will see many examples of them later in the book.

Adding Links to URLs

Suppose that you have URLs located in a document that you want to convert for displaying on the Web. If the URL is stored in a separate column in a relational database, it may be straightforward just to place the URL in the column as the value of the `href` attribute of an HTML/XHTML `a` element. However, if the URL is included in a piece of text such as the following, the problem of recognizing a URL becomes a little tougher.

```
The World Wide Web Consortium, the W3C, has developed many specifications for
XML and associated languages. The W3C's home page is located at
http://www.w3.org and its technical reports are located at
http://www.w3.org/tr/.
```

Finding URLs depends on being able to recognize a URL as it occurs anywhere in what might potentially be a very long document. In addition, you don't know what the actual titles of the Web pages are to which the URLs point, so you can't supply the page title as text but have to use the URL both as the value of the `href` attribute of an XHTML `a` element and also as the text contained between the start tag and end tag of the `a` element. What you want to do is locate each URL and then replace it with a new piece of text constructed like this (the italicized *theURL* stands for the actual URL that you find inside the text):

```
<a href="theURL">theURL</a>
```

Regular Expressions You Already Use

If you have used a computer for any length of time, you very likely are familiar with at least some uses of regular expressions, although you may not use that term to describe the text patterns that you use in word processors or directory listings from the command line, for example.

Search and Replace in Word Processors

Most modern word processors have some sort of regular expression support, although for some word processors the term *regular expressions* is not used. In Microsoft Word, for example, the limited regular expression support available in the word processor itself uses the term *wildcards* to describe the supported regular expression patterns.

The simplest *pattern* is literal text. So if you want to find a text pattern of `Star`, you can enter those four characters in the Find What box in the Find and Replace dialog box in Microsoft Word. As you will see a little later in this chapter, an approach like that can have its problems when you're handling substantial quantities of text.

Directory Listings

If you have done any work at the command line, you have probably used simple regular expressions when doing directory listings. Two metacharacters are available: `*` (the asterisk) and `?` (the question mark).

For example, on the Windows platform, if you want to find the executable files in the current directory you can use the following command from the command line:

```
dir *.exe
```

The `dir` command is an instruction to list all files in the current directory and is equivalent to entering the following at the command line:

```
dir *.*
```

The `*.exe` pattern matches any sequence of zero or more characters followed by a period followed by the literal sequence of characters `exe` in a filename. Similarly, the `*.*` pattern indicates zero or more characters followed by a period followed by zero or more characters.

The syntax of wildcards in directory paths differs significantly from that of regular expressions, so the intention of this section is not to focus on the details of path wildcards, but to remind you that you likely already apply patterns to find suitable pieces of text (in this case, filenames).

On other occasions, when searching a directory you will know the exact number of characters that you want to match. Suppose that you have a directory containing multiple Excel workbooks, each of which contains monthly sales. If you know that the filename consists of the word *Sales* followed by two digits for the year followed by three alphabetic characters for the month, you could search for all sales workbooks from 2004 by using this command:

```
Dir Sales04???.xls
```

This would display the Excel workbooks whose filenames are constructed as just described. But if you had some other workbooks named, for example, Sales04123.xls and Sales04234.xls, the command would also cause those to be displayed, although you don't necessarily want to see those.

The ways in which regular expressions can be used together with the `dir` command are, as you can see, very limited due to the provision of only two metacharacters in path wildcards.

Online Searching

Another scenario where regular expressions, although admittedly simple ones, are widely used is in online searching. The search box on eBay.com, for example, will accept the asterisk wildcard so that `photo*` matches words such as `photo`, `photos`, `photograph`, and `photographs`.

Why Regular Expressions Seem Intimidating

There are several reasons why many developers find regular expressions intimidating. Among the reasons are the compact, cryptic syntax of regular expressions. Another is the absence of a standards body for regular expressions, so that the regular expression patterns for a particular meaning vary among languages or tools that support regular expressions.

Compact, Cryptic Syntax

Regular expressions syntax is very compact and can seem totally cryptic to those unfamiliar with regular expressions. At times it seems as if there are backslash characters, parentheses, and square brackets everywhere. Once you understand what each character and metacharacter does in a regular expression pattern, you will be able to build your own regular expressions or analyze those created by other developers.

*A **metacharacter** is a character, or combination of characters, that have a special meaning when they are contained in a regular expression pattern.*

Whitespace Can Significantly Alter the Meaning

Placing unintended whitespace in a regular expression can radically alter the meaning of the regular expression and can turn what ought to be matches into nonmatches, and vice versa. When creating regular expression patterns, you need to be meticulous about handling whitespace.

For example, suppose that you want to match the content of a document that stores information about people. A sample document, `People.txt`, is shown here:

```
Cardoza, Fred
Catto, Philipa
Duncan, Jean
Edwards, Neil
England, Elizabeth
Main, Robert
Martin, Jane
Meens, Carol
Patrick, Harry
Paul, Jeanine
Roberts, Clementine
Schmidt, Paul
Sells, Simon
Smith, Peter
Stephens, Sheila
Wales, Gareth
Zinni, Hamish
```

Assuming that each name is laid out in the preceding format, you can use a regular expression to locate all names where the surname begins with an uppercase *S* by using the following regular expression:

```
^S.*
```

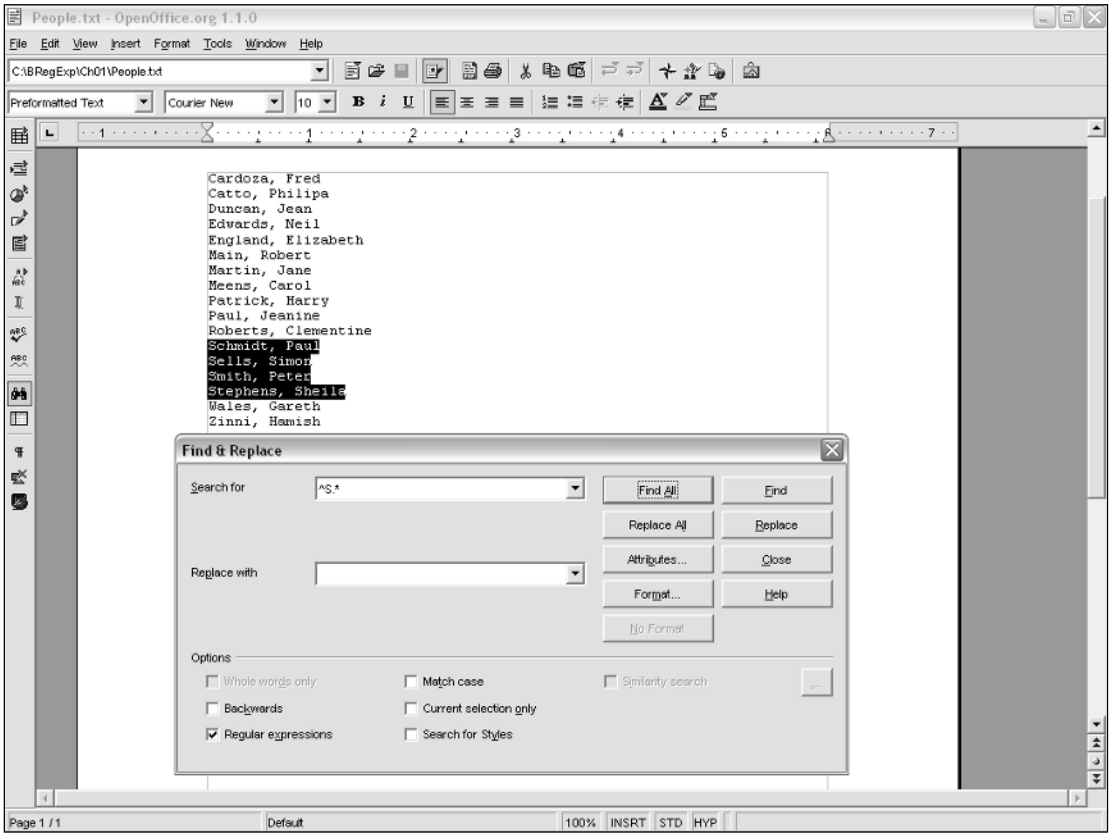
Figure 1-3 shows the result of using that regular expression in OpenOffice.org Writer in `People.txt`. Notice that all the names where the surname begins with *S* are selected.

However, if you insert a single space character between the `^` and the `S` in the regular expression pattern, as follows, there is no match at all, as illustrated in Figure 1-4.

```
^ S.*
```

This occurs because the `^` metacharacter is a marker for the beginning of a line (or paragraph in OpenOffice.org Writer). So inserting a space character immediately after the caret, `^`, means that the regular expression is searching for lines that begin with a space character. Because each of the lines in `People.txt` has an alphabetic character at the beginning of the line, there is no line that begins with a space character, and therefore, there is no match for the regular expression pattern.

Spotting an error like the existence of the inadvertent space character can be tough. I recommend that you always look carefully at the data you are going to manipulate so that you understand its characteristics and know that at least some matches for the regular expression should be present. If you follow that advice (which I discuss in more detail in Chapter 2), you will know that there are some surnames beginning with *S*, and therefore, you should be aware that something is wrong with the regular expression pattern if the regular expression is returning zero matches.



People.txt - OpenOffice.org 1.1.0

C:\BRegExp\Ch01\People.txt

```
Cardoza, Fred
Catto, Philipa
Duncan, Jean
Edwards, Neil
England, Elizabeth
Main, Robert
Martin, Jane
Meens, Carol
Patrick, Harry
Paul, Jeanine
Roberts, Clementine
Schmidt, Paul
Sells, Simon
Smith, Peter
Stephens, Sheila
Wales, Gareth
Zinni, Hamish
```

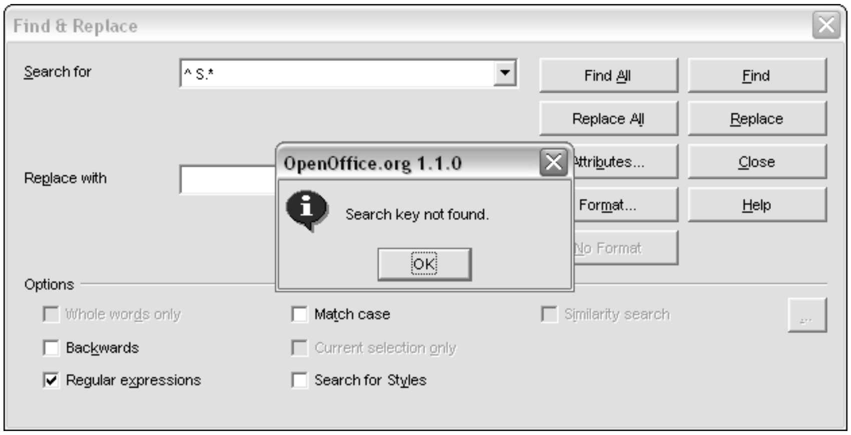
Find & Replace

Search for: ^S.*

Replace with:

Options: [✓ Regular expressions]

Figure 1-3



Find & Replace

Search for: ^ S.*

Replace with:

Options: [✓ Regular expressions]

OpenOffice.org 1.1.0

Search key not found.

Figure 1-4

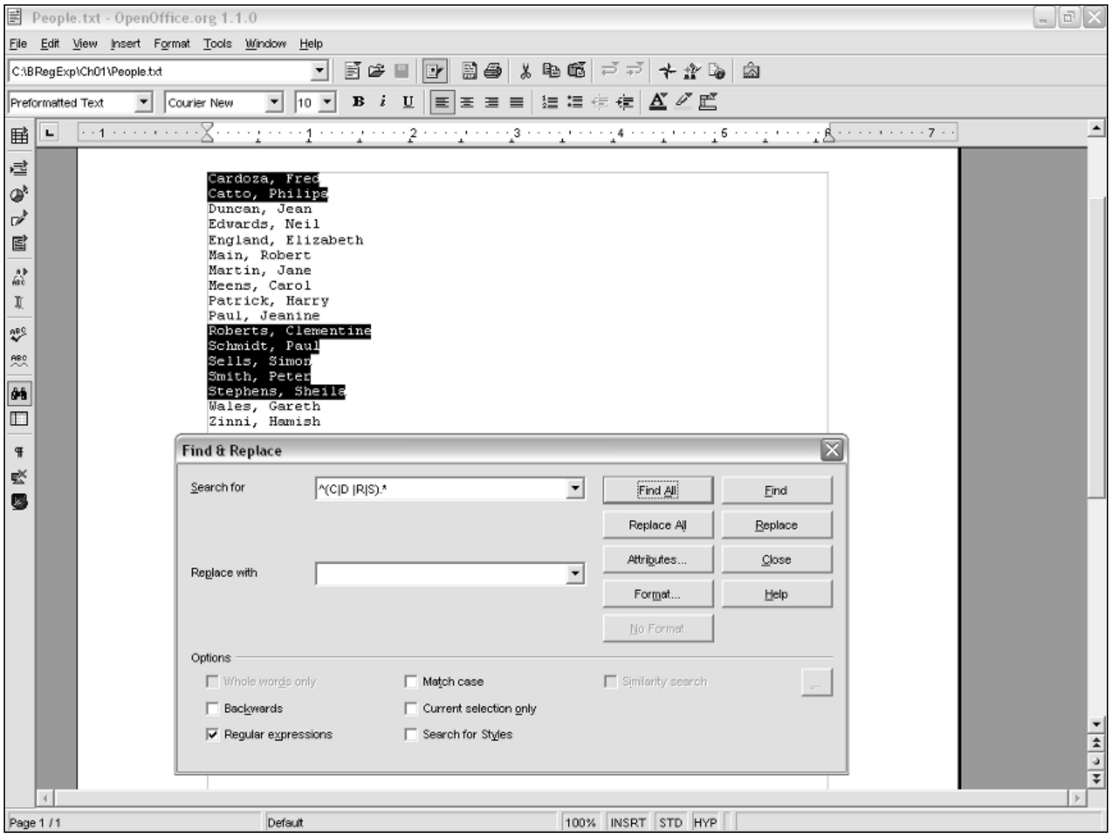
However, when a regular expression returns some matches, you might overlook the undesired effects of inadvertently introduced whitespace. Suppose that you want to find names where the surname begins with *C, D, R,* or *S.* You could do that with the following regular expression pattern:

```
^(C|D|R|S).*
```

What if you accidentally introduce a space character after the *D,* giving the following regular expression?

```
^(C|D |R|S).*
```

You might not see the space character, or you might not realize the change in meaning introduced by a space character, and you might easily be fooled because there are a good number of names returned, although there is no name returned where the surname begins with *D,* as you can see in Figure 1-5.


People.txt - OpenOffice.org 1.1.0
File Edit View Insert Format Tools Window Help
C:\BRegExp\Ch01\People.txt
Preformatted Text
Courier New
10

```
Cardoza, Fred
Catto, Philipa
Duncan, Jean
Edwards, Neil
England, Elizabeth
Main, Robert
Martin, Jane
Meens, Carol
Patrick, Harry
Paul, Jeanine
Roberts, Clementine
Schmidt, Paul
Sells, Simon
Smith, Peter
Stephens, Sheila
Wales, Gareth
Zinni, Hamish
```

Find & Replace

Search for: ^(C|D |R|S).*

Replace with:

Options:
- [ ] Whole words only
- [ ] Match case
- [ ] Similarity search
- [ ] Backwards
- [ ] Current selection only
- [x] Regular expressions
- [ ] Search for Styles

Figure 1-5

With the small amount of ordered data in `People.txt` you might easily notice the absence of expected names with surnames starting with *D.* In larger documents where the data is not ordered, however, it might be a different story.

The use of whitespace inside regular expression patterns is discussed in Chapter 4.

No Standards Body

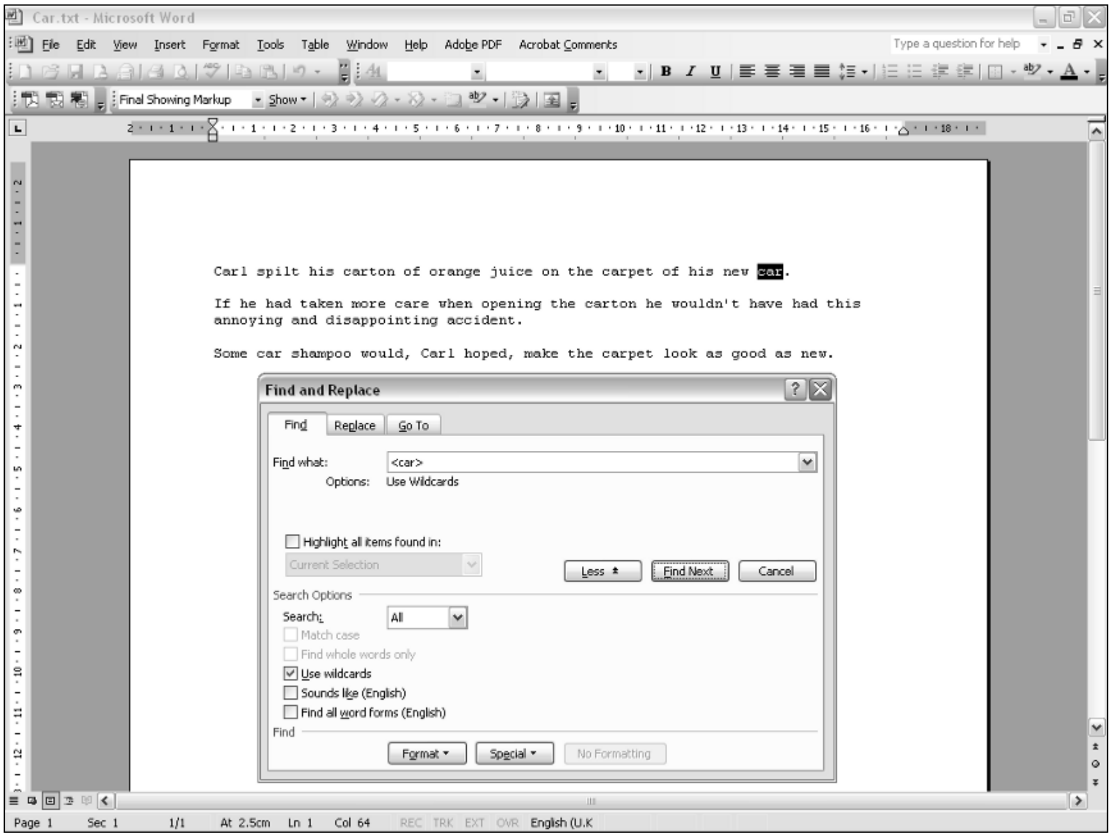
One reason for the variation in regular expressions is that there is no standards body to define the syntax for regular expressions. Regular expressions first came to prominence in the Perl language and were adopted, with varying degrees of exactness, in various other languages and applications over an extended period.

Without a formal standards body for regular expressions, variation in implementations is substantial.

Differences between Implementations

Regular expressions and wildcards have a lot in common among many of the various implementations, but many implementations are quite visibly nonstandard.

You saw in Figure 1-2 that you could use the regular expression pattern `\<car\>` to find occurrences of the sequence of characters `car` only when they made up a whole word. However, if you want to do the same functional search — that is, find occurrences of `car` only as a whole word in Microsoft Word — you would use the regular expression pattern `<car>`, as shown in Figure 1-6.



```
Carl spilt his carton of orange juice on the carpet of his new car.

If he had taken more care when opening the carton he wouldn't have had this
annoying and disappointing accident.

Some car shampoo would, Carl hoped, make the carpet look as good as new.
```

Find what: <car>
Options: Use Wildcards

- [ ] Highlight all items found in: Current Selection

Search Options
Search: All
- [ ] Match case
- [ ] Find whole words only
- [x] Use wildcards
- [ ] Sounds like (English)
- [ ] Find all word forms (English)

Figure 1-6

Similarly, if you want to match any single alphabetic character, you use the `.` (period) metacharacter in OpenOffice.Org Writer and several scripting languages such as Perl. But we must use the `?` (question mark) character in Word for exactly the same function. In that aspect Word is distinctly nonstandard, resembling file path usage rather than true regular expression patterns.

Differences in the implementation of regular expressions will be discussed as you progress through the fundamental techniques in each of the next several chapters. In addition, later chapters of this book are dedicated to specific languages and applications, and characteristics of the implementation of regular expressions syntax are discussed in detail.

Characters Change Meaning in Different Contexts

Another reason why people find regular expressions confusing is that individual characters, or metacharacters, can have significantly different meanings depending on where you use them.

For example, the `^` metacharacter can signify the beginning of a line in regular expressions in some languages. However, in those same languages the same `^` metacharacter can, when used inside a character class, signify negation. So the regular expression pattern `^and` matches the sequence of characters `and` at the beginning of a word, but the regular expression `[^and]` signifies a character class that must not include any of the characters `a`, `n`, and `d`.

Character classes are introduced and described in detail in Chapter 5.

The test document, `And.txt`, is shown here:

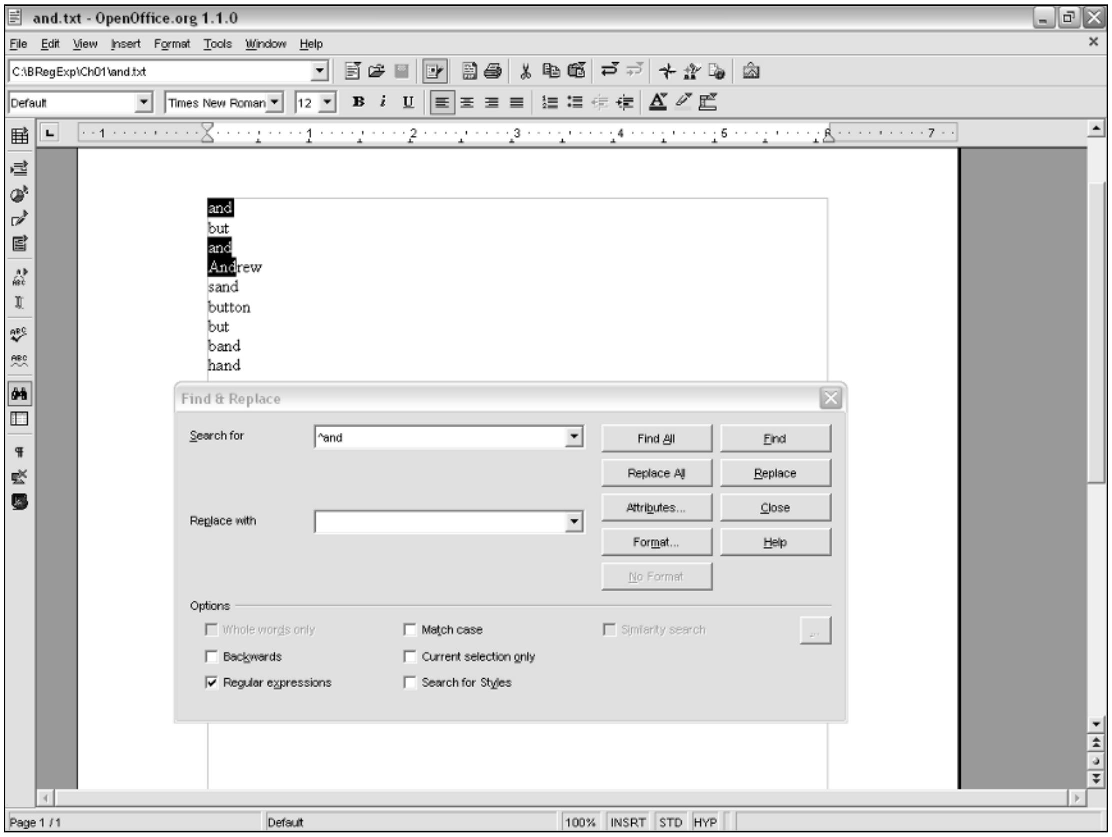
```
and
but
and
Andrew
sand
button
but
band
hand
```

If you use the regular expression pattern `^and` in OpenOffice.org Writer, you expect to select all the occurrences when the beginning of a line, indicated by the `^` metacharacter, is followed by the exact sequence of characters `a`, `n`, and `d` (regardless of case of those characters). Figure 1-7 shows the application of this regular expression in OpenOffice.Org Writer.

The regular expression selects the word `and` when it occurs at the beginning of a line (twice) and the first three characters of the word `Andrew`. The first three characters of `Andrew` are selected because the search is case insensitive.

However, if you use the `^` metacharacter as the first character inside a character class in the regular expression pattern `.[^and].*`, the sequences of characters shown in Figure 1-8 are selected.

What the regular expression pattern `.[^and].*` means is a single character followed by a character that is *not* `a` or `n`, or `d`, followed by zero or more characters.



and
but
and
Andrew
sand
button
but
band
hand

Search for: ^and

Replace with:

Whole words only [ ]
Backwards [ ]
Regular expressions [✓]
Match case [ ]
Current selection only [ ]
Search for Styles [ ]
Similarity search [ ]

Figure 1-7

If the ^ is used inside a character class in any position other than the first character, it has the normal meaning that it has outside a character class. The caret functions as a negation metacharacter only when it is the first character inside the opening square bracket of a character class. Don't worry about the detail of the interpretation at this stage. The relevant issues are discussed in detail in Chapter 5.

Another character that can have different meanings inside and outside character classes is the hyphen. Outside a character class, a dash simply represents itself, for example, in a date value:

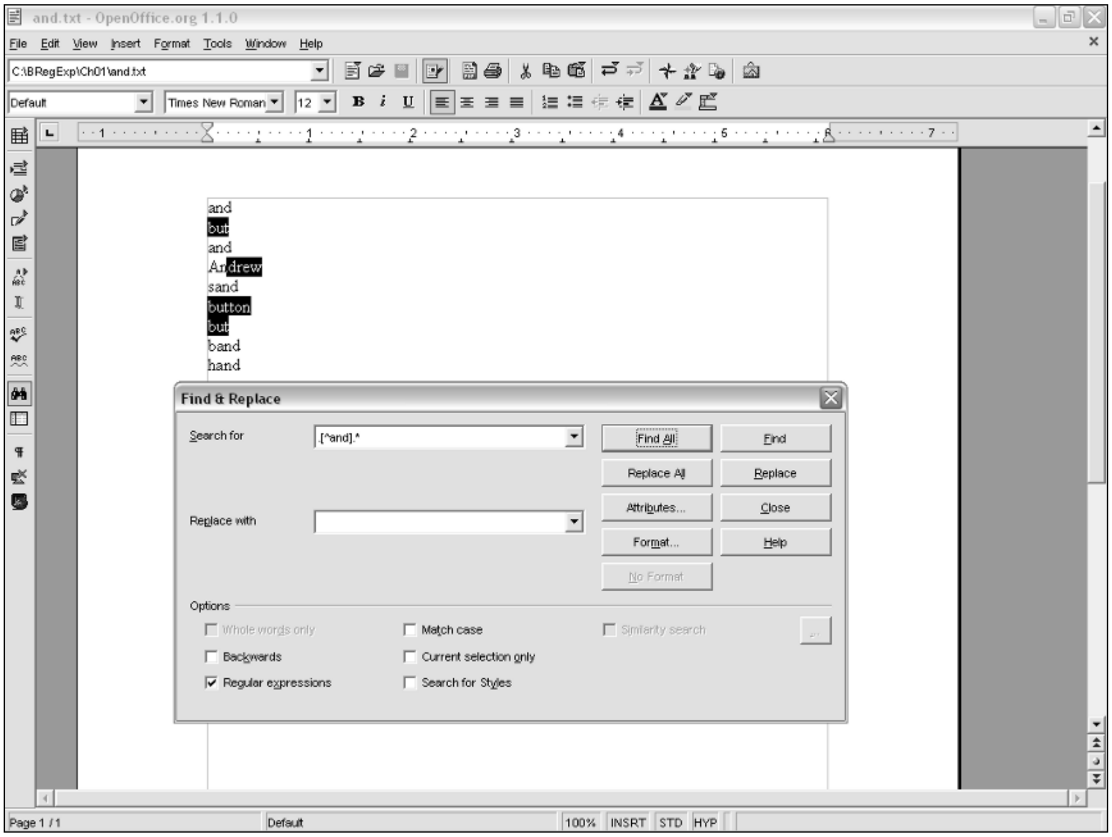
```
2004-12-25
```

However, inside a character class, when it isn't the first character the dash indicates a range. For example, to specify a character class that has all lower- and uppercase alphabetic characters, you could use the following pattern:

```
[a-zA-Z]
```

The character class is short for the following:

```
[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]
```



```
and
but
and
Andrew
sand
button
but
band
hand
```

Search for: [^and].*

Replace with:

Options

- [ ] Whole words only
- [ ] Match case
- [ ] Similarity search
- [ ] Backwards
- [ ] Current selection only
- [✓] Regular expressions
- [ ] Search for Styles

Figure 1-8

Regular Expressions Can Be Case Sensitive

A further complicating aspect of using regular expressions is that in some circumstances regular expressions are case sensitive. Case sensitivity has two aspects, one related to whether matching is carried out in a case-sensitive or case-insensitive fashion, and the other being the different, indeed opposite, meaning of metacharacters of different case.

Case-Sensitive and Case-Insensitive Matching

Individual regular expression implementations handle case differently. For example, OpenOffice.org Writer provides a Match Case check box in the Find and Replace dialog box that you saw in examples earlier in this chapter. The default behavior is case-insensitive matching.

By contrast, the default matching in programming languages is often case sensitive. Many programming languages provide switches that indicate whether use of a regular expression pattern is in case-sensitive mode or case-insensitive mode. These issues are discussed in Chapter 4. In addition, use of case-sensitive and case-insensitive searches is demonstrated in many of the chapters dedicated to individual programming languages.

Case and Metacharacters

Case is crucial in how some metacharacters are interpreted. For example, if you want to match numeric digits you can use the pattern

```
\d
```

and numeric digits from `0` through `9` will be matched because `\d` is equivalent to the fairly lengthy pattern

```
(0|1|2|3|4|5|6|7|8|9)
```

which also selects numeric digits. By adding the relevant quantifiers to either of the preceding patterns, any integer value can be matched.

However, if you simply change the case of the pattern to the following, the meaning is changed:

```
\D
```

In fact, it is reversed. The pattern `\D` matches any character other than the numeric digits `0` through `9`.

Continual Evolution in Techniques Supported

Different versions of a programming language or product may provide different types of regular expression functionality. For example, Perl, which was one of the first languages to support regular expressions, has added new functionality to its regular expression support over time. Some of those differences are discussed in the chapters specific to individual programming languages later in this book.

Multiple Solutions for a Single Problem

Frequently, there are multiple regular expression solutions for any particular problem. For example, if you want to specify that a part number in a warehouse parts inventory must begin with an uppercase alphabetic character followed by two numbers, you could use any of the following patterns. The first is pretty lengthy due to using multiple possible options inside parentheses, the options being mutually exclusive:

```
(A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z)\d\d
```

The numeric part of the pattern could use a quantifier for the `\d` metacharacters:

```
(A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z)\d{2}
```

Or you could use a much more succinct pattern that makes use of a character class, which is much shorter and easier to read:

```
[A-Z]\d{2}
```

The flexibility of regular expression syntax can be useful to experienced users, but to beginners, the fact that such different regular expression patterns will match the same sequences of characters is potentially confusing.

What You Want to Do with a Regular Expression

The right regular expression to use depends on what it is you are trying to do and the source data you are attempting to do it on. It also depends on how well you know the data and how precisely you can define what you want to find.

If, for example, you had a document that you knew contained several URLs, you might not need to use a regular expression at all, but simply a plain-text search, using the characters `http` as a proxy for all URLs. For example, that approach works well if the World Wide Web Consortium's home page is expressed as `http://www.w3.org/` but will fail completely if it happens to be expressed as `www.w3.org/`. The data determines the matching behavior of a pattern.

If you think that the W3C's home page URL is expressed as either `www.w3.org` or `www.w3c.org` but can't remember which it is in a particular document, you could do a single search with this regular expression pattern:

```
\.w3c?\.org
```

This would match the literal period character in a URL using the metacharacter `\.` followed by the literal characters `w3`, followed optionally by the literal character `c`, followed by the literal period identified by the `\.` Metacharacter, followed by the literal characters `org`.

Metacharacters are discussed in detail in Chapter 4, where there are many examples of how they are used in regular expressions.

The Languages That Support Regular Expressions

There is a huge range of tools and languages that support regular expressions, ranging from text editors through word processors to scripting languages and full-featured programming languages. Several of those applications and languages are described in detail later in this book.

To conclude this chapter, take a look at a problem that you might be faced with and for which regular expressions, correctly used, could be very useful.

Replacing Text in Quantity

One use of regular expressions is to replace quantities of text, possibly across many thousands of documents. Things can very easily go wrong if you don't understand what you are doing, as the following simple example shows.

Imagine that you have just joined the fictional Star Training Company as a summer intern. Just before you start with the company, someone decides that Star Training Company should become the equally fictional Moon Training Company, possibly as a result of a recent takeover or change in corporate focus. One result of that name change is that the company's Web site needs to reflect the new naming scheme, and a large number of internal and public documents must be updated, too. Because you are at the bottom of the pecking order, that job is given to you.

On the first day of your internship, you are asked to update Moon Training Company's documents and Web site to show the new name consistently throughout the hundreds of existing company documents.

The first document you open, StarOriginal.doc, looks like this:

```
Star Training Company

Starting from May 1st  Star Training Company is offering a startling special offer
to our regular customers - a 20% discount when 4 or more staff attend a single Star
Training Company course.

In addition, each quarter our star customer will receive a voucher for a free
holiday away from the pressures of the office. Staring at a computer screen all day
might be replaced by starfish and swimming in the Seychelles.

Once this offer has started and you hear about other Star Training customers
enjoying their free  holiday you might feel left out. Don't be left on the outside
staring in. Start right now building your points to allow you to start out on your
very own Star Training holiday.

Reach for the star. Training is valuable in its own right but the possibility of a
free holiday adds a startling new dimension to the benefits of Star Training
training.

Don't stare at that computer screen any longer. Start now with Star. Training is
crucial to your company's well-being. Think Star.
```

It's your first day, and you haven't done anything like this before. So you open the document, open up Search and Replace in your favorite word processor, and elect to replace `Star` with `Moon`. You choose Replace All.

And here is the result: `StarSimpleReplace.doc`. Read the following carefully to see the effect of the chosen find-and-replace strategy.

```
Moon Training Company

Moonting from May 1st Moon Training Company is offering a Moontling special offer
to our regular customers - a 20% discount when 4 or more staff attend a single Moon
Training Company course.

In addition, each quarter our Moon customer will receive a voucher for a free
holiday away from the pressures of the office. Mooning at a computer screen all day
might be replaced by Moonfish and swimming in the Seychelles.

Once this offer has Moonted and you hear about other Moon Training customers
enjoying their free  holiday you might feel left out. Don't be left on the outside
Mooning in. Moont right now building your points to allow you to Moont out on your
very own Moon Training holiday.

Reach for the Moon. Training is valuable in its own right but the possibility of a
free holiday adds a Moontling new dimension to the benefits of Moon Training
training.

Don't Moone at that computer screen any longer. Moont now with Moon. Training is
crucial to your company's well-being. Think Moon.
```

As you can see from the resulting text, quite a few things have gone wrong. While you have replaced every occurrence of the word `Star` in the original document, lots of undesired changes have also occurred. The first sentence shouldn't start `Moonting from May 1st`. There are several other problems in the document, too, including the creation of words that don't exist in English and the inappropriate introduction of uppercase initial letters in the middle of several sentences.

As you progress through the next several chapters, you will look at approaches that will yield better results when applied to the test document `Star.txt`.