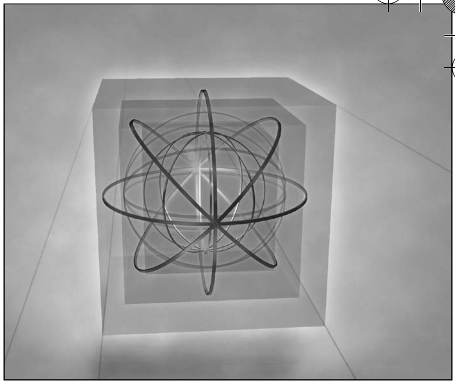


12 Regular Expressions

Regular expressions are a well-established and standard way to parse text files as well as to search and optionally replace occurrences of substrings and text patterns. If you aren't familiar with regular expressions, just think about the wildcard characters you use in MS-DOS to indicate a group of files (as in **.txt*) or the special characters you can use with the LIKE statement in SQL queries:

```
SELECT name, city FROM customers WHERE name LIKE "A%"
```

Many computer scientists have thoroughly researched regular expressions, and a few programming languages—most notably Perl and Awk—are heavily based on regular expressions. In spite of their usefulness in virtually every text-oriented task (including parsing log files and extracting information from HTML files), regular expressions are relatively rarely used among Windows programmers for at least a couple of reasons:

- Many developers believe that regular expressions are associated closely with Perl and Awk and can't be accessed from other languages.
- Regular expressions are based on a rather obscure syntax, which has gained them a reputation for complexity—this reputation keeps many developers from learning more about them.

The first point is only partially true. Starting with version 5.0, the VBScript language comes with a good implementation of a regular expression engine, and you can access it even from Visual Basic 6 by adding a reference to the Microsoft VBScript Regular Expressions type library. (Incidentally, the VBScript regular expression engine supports a large subset of the features of the .NET engine, so you can apply many of the techniques I describe in this chapter to your VBScript and Visual Basic 6 applications.)

The second point has some merit: you can regard regular expressions as a highly specific programming language, and you know that all languages take time to learn and have their idiosyncrasies. But when you see how much time regular expressions can save you—and I am talking about both coding time and CPU time—you'll probably agree that the effort you spend learning their contorted syntax is well spent.

Regular Expression Overview

The .NET Framework comes with a very powerful regular expression engine that's accessible from any .NET language, so you can leverage the parsing power of languages such as Perl without having to switch from your favorite language.

The powerful .NET Framework regular expression engine allowed Microsoft to clean up the Visual Basic language by getting rid of regular expression–like (but far less versatile) features, such as the LIKE operator or the Replace string function. (The Replace function in the Microsoft.VisualBasic namespace is for backward compatibility only, and the String.Replace method supports only the replacement of single characters.)

Note All the classes you need to work with regular expressions belong to the System.Text.RegularExpressions namespace, so all the code in this section assumes that you added the following Imports statement at the top of your module:

```
Imports System.Text.RegularExpressions
```

The Fundamentals

Regex is the most important class in this group, and any regular expression code instantiates at least an object of this class (or uses the class's shared methods). This object represents an immutable, compiled regular expression: you instantiate this object by passing to it the search pattern, written using the special regular expression language, which I'll describe later:

```
' This regular expression defines any group of 2 characters
' consisting of a vowel followed by a digit (\d).
Dim re As New Regex("[aeiou]\d")
```

The Matches method of the Regex object applies the regular expression to the string passed as an argument; it returns a MatchCollection object, a read-only collection that represents all the nonoverlapping matches:

```
Dim re As New Regex("[aeiou]\d")
' This source string contains 3 groups that match the Regex.
Dim source As String = "a1 = a1 & e2"
' Get the collection of matches.
Dim mc As MatchCollection = re.Matches(source)
' How many occurrences did we find?
Console.WriteLine("Found {0} occurrences", mc.Count)
    ' => Found 3 occurrences
```

You can also pass to the Matches method a second argument, which is interpreted as the index where the search begins.

The MatchCollection object contains individual Match objects, which expose properties such as Index (the position in the source string at which the matching string was found) and Length (the length of the matching string, which is useful when the regular expression can match strings of different lengths):

```
' ...(Continuing the previous example)...
Dim m As Match
For Each m In mc
    ' Display text and position of this match.
    Console.WriteLine("'{0}' at position {1}" , m.ToString, m.Index)
Next
```

The preceding code displays these lines in the console window:

```
'a1' at position 0
'a1' at position 5
'e2' at position 10
```

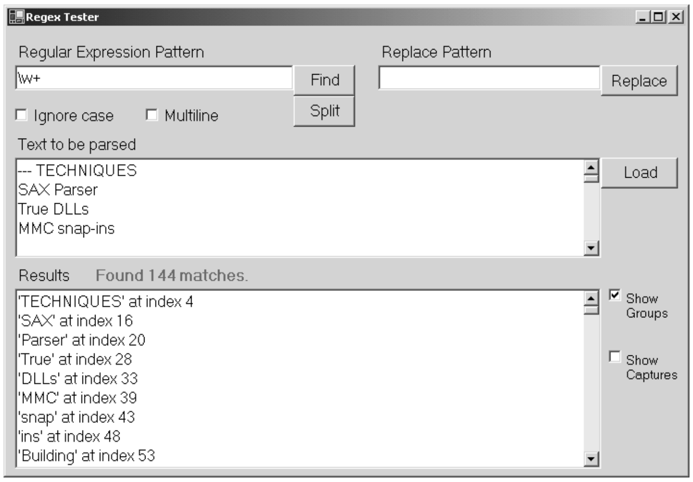
The Regex object is also capable of modifying a source string by searching for a given regular expression and replacing it with something else:

```
Dim source As String = "a1 = a1 & e2"
' Search for the "a" character followed by a digit.
Dim re As New Regex("a\d")
' Drop the digit that follows the "a" character.
Console.WriteLine(re.Replace(source, "a"))   ' => a = a & e2
```

The Regex class also exposes shared versions of the Match, Matches, and Replace methods. You can use these shared methods when you don't want to explicitly instantiate a Regex object:

```
' This code snippet is equivalent to the previous one, but it doesn't
' instantiate a Regex object.
Console.WriteLine(Regex.Replace("a1 = a1 & e2", "a\d", "a"))
```

As you would find for any programming language new to you, the best way to learn regular expressions is, not surprisingly, through practice. To help you in this process, I have created a RegexTester application, which lets you test any regular expression against any source string or text file. (See Figure 12-1.) This application has been a precious tool for me in exploring regular expression intricacies, and I routinely use it whenever I have a doubt about how a construct works.



Regular Expression Pattern: `\w+`

Replace Pattern:

[ ] Ignore case [ ] Multiline

Text to be parsed

```
--- TECHNIQUES
SAX Parser
True DLLs
MMC snap-ins
```

Results Found 144 matches.

[✓] Show Groups [ ] Show Captures

```
'TECHNIQUES' at index 4
'SAX' at index 16
'Parser' at index 20
'True' at index 28
'DLLs' at index 33
'MMC' at index 39
'snap' at index 43
'ins' at index 48
'Building' at index 53
```

Figure 12-1. The RegexTester application lets you experiment with all the most important methods and options of the Regex object.

The Regular Expression Language

Table 12-1 lists all the constructs that are legal as regular expression patterns, grouped in the following categories:

- **Character escapes** are used to match single characters. You need them to deal with nonprintable characters (such as the newline and the tab character) and to provide escaped versions for the characters *.$^{[(|)*+?* which have a special meaning inside regular expression patterns.
- **Character classes** offer a means to match one character from a group that you specify between square brackets as in *[aeiou]*. Note that you don't need to escape special characters when they appear in

square brackets except in the cases of the dash and the closing square bracket because they're the only characters that have special meaning inside square brackets. For example, *[()[\]{}]* matches opening and closing parentheses, square brackets, and curly brackets.

- **Atomic zero-width assertions** don't cause the engine to advance through the source string and don't consume characters. For example, the *abc$* regular expression matches any *abc* word immediately before the end of a line without also matching the end of the line.
- **Quantifiers** add optional quantity data to regular expressions. A particular quantifier applies to the character, character class, or group that immediately precedes it. For example, *\w+* matches all the words with one or more characters, whereas *\w{3,}* matches all the words with at least three characters.
- **Grouping constructors** can capture and name groups of subexpressions as well as increase the efficiency of regular expressions with noncapturing look-ahead and look-behind modifiers. For example, *(abc)+* matches repeated sequences of the "abc" string; *(?<total>\d+)* matches a group of one or more consecutive digits and assigns it the name *total*, which can be used later inside the same regular expression pattern or for substitution purposes.
- **Substitutions** can be used only inside a replacement pattern and, together with character escapes, are the only constructs that can be used inside replacement patterns. For example, when the sequence *({total})* appears in a replacement pattern, it inserts the value of the group named *total*, after enclosing it in parentheses. Note that parentheses have no special meanings in replacement patterns, so you don't need to escape them.
- **Backreference constructs** let you reference a previous group of characters in the regular expression pattern via its group number or name. You can use these constructs as a simple way to say "match the same thing again." For example, *(?<value>\d+)=\k<value>* matches identical numbers separated by an = symbol, as in the "123=123" sequence.
- **Alternating constructs** provide a way to specify alternatives; for example, the sequence *I (am|have)* can match both the "I am" and "I have" strings.

- **Miscellaneous constructs** include constructs that allow you to modify one or more regular expression options in the middle of the pattern. For example, *A(?i)BC* matches all the variants of the "ABC" word that begin with uppercase "A" (such as "Abc", "ABc", "AbC", and "ABC"). See Table 12-2 for a description of all the regular expression options.

Table 12-1 The Regular Expression Language*

Category	Sequence	Description
Character escapes	any character	Characters other than .$^{[(|)*+?\ are matched to themselves.
	\a	The bell alarm character (same as *\x07*).
	\b	The backspace (same as *\x08*), but only when used between square brackets or in a replacement pattern. Otherwise, it matches a word boundary.
	\t	The tab character (same as *\x09*).
	\r	The carriage return (same as *\x0A*).
	\v	The vertical tab character (same as *\x0B*).
	\f	The form-feed character (same as *\x0C*).
	\n	The newline character (same as *\x0D*).
	\e	The escape character (same as *\x1B*).
	\040	An ASCII character expressed in octal notation (must have exactly three octal digits). For example, *\040* is a space.
	\x20	An ASCII character expressed in hexadecimal notation (must have exactly two digits). For example, *\x20* is a space.
	\cC	An ASCII control character. For example, *\cC* is control+C.
	\u0020	A Unicode character in hexadecimal notation (must have exactly four digits). For example, *\u0020* is a space.
	*	When the backslash is followed by a character in a way that doesn't form an escape sequence, it matches the character. For example, * matches the * character.
Character classes	.	The dot character matches any character except the newline character. It matches any character, including newline, if you're using the Singleline option.
	[aeiou]	Any character in the list between the square brackets; *[aeiou]* matches any vowel.
	[^aeiou]	Any character except those in the list between the square brackets; *[^aeiou]* matches any nonvowel.

Table 12-1 The Regular Expression Language* *(continued)*

Category	Sequence	Description
	[a-zA-Z]	The - (dash) character lets you specify ranges of characters: *[a-zA-Z]* matches any lowercase or upper-case character; *[^0-9]* matches any nondigit character.
	\w	A word character, which is an alphanumeric character or the underscore character; same as *[a-zA-Z_0-9]*.
	\W	A nonword character; same as *[^a-zA-Z_0-9]*.
	\s	A white-space character, which is a space, a tab, a form-feed, a newline, a carriage return, or a vertical-feed character; same as *[\f\n\r\t\v]*.
	\S	A character other than a white-space character; same as *[^ \ f\n\r\t\v]*.
	\d	A decimal digit; same as *[0-9]*.
	\D	A nondigit character; same as *[^0-9]*.
Atomic zero-width assertions	^	The beginning of the string (or the beginning of the line if you're using the Multiline option).
	$	The end of the string (or the end of the line if you're using the Multiline option).
	\A	The beginning of a string (like ^ but ignores the Multiline option).
	\Z	The end of the string or before the newline character at the end of the line (like *$* but ignores the Multiline option).
	\z	Exactly the end of the string, whether or not there's a newline character (ignores the Multiline option).
	\G	The position at which the current search started—usually one character after the point at which the previous search ended.
	\b	The word boundary between *\w* (alphanumeric) and *\W* (nonalphanumeric) characters. It indicates the first and last characters of a word delimited by spaces or other punctuation symbols.
	\B	Not on a word boundary.
Quantifiers	***	Zero or more matches; for example, *\bA\w** matches a word that begins with "A" and is followed by zero or more alphanumeric characters; same as *{0,}*.
	+	One or more matches; for example, *\b[aeiou]+\b* matches a word composed only of vowels; same as *{1,}*.

(continued)

Table 12-1 The Regular Expression Language* *(continued)*

Category	Sequence	Description
	?	Zero or one match; for example, \b[aeiou]\d?\b matches a word that starts with a vowel and is followed by zero or one digits; same as {0,1}.
	{N}	Exactly N matches; for example, [aeiou]{4} matches four consecutive vowels.
	{N,}	At least N matches; for example, \d{3,} matches groups of three or more digits.
	{N,M}	Between N and M matches; for example, \d{3,5} matches groups of three, four, or five digits.
	*?	Lazy *: the first match that consumes as few repeats as possible.
	+?	Lazy +: the first match that consumes as few repeats as possible, but at least one.
	??	Lazy ?: zero repeats if possible, or one.
	{N}?	Lazy {N}: equivalent to {N}.
	{N,}?	Lazy {N,}: as few repeats as possible, but at least N.
	{N,M}?	Lazy {N,M}: as few repeats as possible, but between N and M.
Grouping constructs	(substr)	Captures the matched substring. These captures are numbered automatically, based on the order of the left parenthesis, starting at 1. The zeroth capturing group is the text matched by the whole regular expression pattern.
	(?<name>substr)	Captures the substring and assigns it a name. The name must not contain any punctuation symbols.
	(?'name'substr)	
	(?:substr)	Noncapturing group.
	(imnsx-imnsx: subexpr)	Enables or disables the options specified in the subexpression. For example, (?i-s) uses case-insensitive searches and disables single-line mode. (See Table 12-2 for information about regular expression options.)
	(?=subexpr)	Zero-width positive look-ahead assertion: continues match only if the subexpression matches at this position on the right. For example, \w+(?=,) matches a word followed by a comma, without matching the comma.
	(?!subexpr)	Zero-width negative look-ahead assertion: continues match only if the subexpression doesn't match at this position on the right. For example, \w+\b(?![,:;]) matches a word that isn't followed by a comma, a colon, or a semicolon.

Table 12-1 The Regular Expression Language* *(continued)*

Category	Sequence	Description
	(?<=subexpr)	Zero-width positive look-behind assertion: continues match only if the subexpression matches at this position on the left. For example, *(?<=\d+[EeDd])\d+* matches the exponent part of a number in exponential notation (the *45* in *123E45*). This construct doesn't backtrack.
	(?<!subexpr>	Zero-width negative look-behind assertion: continues match only if the subexpression doesn't match at this position on the left. For example, *(?<!,)\b\w+* matches a word that doesn't follow a comma.
	(?>subexpr)	Nonbacktracking subexpression. The subexpression is fully matched once, and it doesn't participate in backtracking. (The subexpression matches only strings that would be matched by the subexpression alone.)
Substitutions	*$N*	Substitutes the last substring matched by group number *N*.
	${name}	Substitutes the last substring matched by a (*?<name>*) group.
Back reference constructs	\N \NN	Back reference to a previous group. For example, *(\w)\1* finds doubled word characters, such as *ss* in *expression*. A backslash followed by a single digit is always considered a back reference (and throws a parsing exception if such a numbered reference is missing); a backslash followed by two digits is considered a numbered back reference if there's a corresponding numbered reference; otherwise, it's considered an octal code. In case of ambiguity, use the \k<name> construct.
	\k<name> \k'name'	Named back reference. *(?<char>\w)\d\k<char>* matches a word character followed by a digit and then by the same word character, as in the "B2B" string.
Alternating constructs	|	Either/or. For example, *vb|c#|java*. Leftmost successful match wins.
	(?(expr)yes|no)	Matches the *yes* part if the expression matches at this point; otherwise, matches the *no* part. The expression is turned into a zero-width assertion. If the expression is the name of a named group or a capturing group number, the alternation is interpreted as a capture test. (See next case.)
	(?(name)yes|no)	Matches the *yes* part if the named capture string has a match; otherwise, matches the *no* part. The *no* part can be omitted. If the given name doesn't correspond to the name or number of a capturing group used in this expression, the alternation is interpreted as an expression test. (See previous case.)

(continued)

Table 12-1 The Regular Expression Language* *(continued)*

Category	Sequence	Description
Miscellaneous constructs	*(?imnsx-imnsx)*	Enables or disables one or more regular expression options. For example, it allows case sensitivity to be turned on or off in the middle of a pattern. Option changes are effective until the closing parenthesis. (See also the corresponding grouping construct, which is a cleaner form.)
	(?#)	Inline comment inserted within a regular expression. The text that follows the # sign and continues until the first closing) character is ignored.
	#	X-mode comment: the text that follows an unescaped # until the end of line is ignored. This construct requires that the *x* option or the *RegexOptions.IgnorePatternWhiteSpace* enumerated option be activated. (X-mode comments are currently experimental.)

* The regular expression language; only the constructs in the character escapes and substitutions categories can be used in replacement patterns.

Regular Expression Options

The Match, Matches, and Replace shared methods of the Regex object support an optional argument, which lets you specify one or more options to be applied to the regular expression search. (See Table 12-2.) For example, the following code searches for all the occurrences of the “abc” word, regardless of its case:

```
Dim source As String = "ABC Abc abc"
Dim mc As MatchCollection = Regex.Matches(source, "abc")
Console.WriteLine(mc.Count)           ' => 1
mc = Regex.Matches(source, "abc", RegexOptions.IgnoreCase)
Console.WriteLine(mc.Count)           ' => 3
```

By default, the Regex class transforms the regular expression into a sequence of opcodes, which are then interpreted when the pattern is applied to a specific source string. If you specify the RegexOptions.Compiler option, however, the regular expression is compiled into explicit Microsoft Intermediate Language (MSIL) rather than regular expression opcodes. This feature enables the Microsoft .NET Framework Just-in-Time (JIT) compiler to convert the expression to native CPU instructions, which clearly deliver better performance. This extra compilation step adds some overhead, so you should use this option only if you plan to use the regular expression multiple times.

Another factor that you should take into account when using the Regex-Options.Compiler option is that the compiled MSIL code isn’t unloaded when the Regex object is released and garbage collected—it continues to take memory until the application terminates. So you should preferably limit the number of compiled regular expressions, especially on systems with scarce memory.

Also, consider that the Regex class caches all regular expression opcodes in memory, so a regular expression isn't generally reparsed each time it's used. (The caching mechanism also works when you use shared methods and don't create Regex instances.) The *Regex* class also exposes a *CompileToAssembly* shared method for explicitly compiling a regular expression into an assembly. (See the .NET Framework SDK for more details about this method.)

The RegexOptions.Multiline option enables multiline mode, which is especially useful when you're parsing text files instead of plain strings. This option modifies the meaning and the behavior of the ^ and *$* assertions so that they match the start and end of each line of text, respectively, rather than the start or end of the whole string. The following example parses a .vb file looking for all the lines that contain a variable assignment. Notice that you can combine multiple Regex options by using the Or operator:

```
Sub TestRegexOptions()
    ' Read a .vb file into a string.
    Dim source As String = FileText("Module1.vb")
    Dim pattern As String = "^\s*[A-Z]\w* ?=.+(?=\r\n)"
    ' Get the collection of all matches, in multiline mode.
    Dim mc As MatchCollection = Regex.Matches(source, pattern, _
        RegexOptions.IgnoreCase Or RegexOptions.Multiline)
    ' Display all variable assignments and their offset in source file.
    Dim m As Match
    For Each m In mc
        Console.WriteLine("[{0}]  {1}", m.Index, m.ToString)
    Next
End Sub

' Read the contents of a text file.
Function FileText(ByVal path As String) As String
    ' Open a file stream and define a stream reader.
    Dim fs As New System.IO.FileStream(path, System.IO.FileMode.Open)
    Dim sr As New System.IO.StreamReader(fs)
    ' Read the entire contents of this file.
    FileText = sr.ReadToEnd
    ' Clean up code.
    sr.Close()
    fs.Close()
End Function
```

Let's analyze the regular expression pattern ^\s*[A-Z]\w* ?=.+(?=\r\n) used in the preceding example, one subexpression at a time:

1. The ^ character means that the matching string should be at the beginning of each line (and not just at the beginning of the string because of the RegexOptions.Multiline option).
2. The \s* subexpression means there can be zero or more white spaces at the beginning of the line.

3. The *[A-Z]\w** subexpression means that there must be an alphabetic character followed by zero or more alphanumeric characters. (This portion of the regular expression represents a variable name in Visual Basic syntax.) The leading character can be uppercase or lowercase because of the RegexOptions.IgnoreCase option.
4. The space followed by a *?* character means there can be zero or one space after the variable name.
5. The = character matches itself, so there must be an equal sign after the space (or directly after the variable name if the optional space is missing).
6. The *.+(?=\r\n)* subexpression means that we're matching any character up to the end of the current line, that is, up to the CR-LF pair of characters, but without including the CR-LF pair in the match. (Note that we might use the simpler *.+$* subexpression, but in that case the CR character would be included in the match.)

Another way to specify a regular expression option is by means of the *(?imnsx-imnsx)* construct, which lets you enable or disable one or more options from the current position to the end of the pattern string. The following code snippet is similar to the preceding one except that it matches all the Dim, Private, and Public variable declarations. Note that the regular expression options are specified inside the pattern string instead of as an argument of the Regex.Matches method:

```
Dim source As String = FileText("Module1.vb")
Dim pattern As String = "(?im)^\s+(dim|public|private) [A-Z]\w* As .+(?=\r\n)"
Dim mc As MatchCollection = Regex.Matches(source, pattern)
```

Table 12-2 Regular Expression Options*

RegexOptions enum Value	Option	Description
None		No option.
IgnoreCase	*i*	Case insensitivity match.
Multiline	*m*	Multiline mode: changes the behavior of ^ and *$* so that they match the beginning and end of any line, respectively, instead of the whole string.
ExplicitCapture	*n*	Captures only explicitly named or numbered groups of the form *(?<name>)* so that naked parentheses act as noncapturing groups without your having to use the *(?:)* construct.

Table 12-2 Regular Expression Options* *(continued)*

RegexOptions enum Value	Option	Description
Compiled	*c*	Compiles the regular expression and generates MSIL code; this option generates faster code at the expense of longer start-up time.
Singleline	*s*	Single-line mode: changes the behavior of the . (dot) character so that it matches any character (instead of any character except newline).
IgnorePattern-Whitespace	*x*	Eliminates unescaped white space from the pattern and enables X-mode comments.
RightToLeft	*r*	Searches from right to left. The regular expression will move to the left of the starting position, so the starting position should be specified at the end of the string instead of its beginning. This option can't be specified in the middle of the pattern. (This restriction avoids endless loops.) The *(?<)* look-behind construct provides something similar, and it can be used as a subexpression.
ECMAScript		Enables ECMAScript-compliant behavior. This option can be used only in conjunction with the Ignore-Case, Multiline, and Compiled flags; in all other cases, the method throws an exception.

* These regular expression options can be specified when you create the Regex object or from inside a *(?)* construct. All these options are turned off by default.

Regular Expression Classes

Now that I have illustrated the fundamentals of regular expressions, it's time to dissect all the classes in the System.Text.RegularExpressions namespace.

The Regex Class

As you've seen in the preceding section, the Regex class provides two overloaded constructors—one that takes only the pattern and another that also takes a bit-coded value that specifies the required regular expression options:

```
' This Regex object can search the word "dim" in a case-insensitive way.
Dim re As New Regex("\bdim\b", RegexOptions.IgnoreCase)
```

The Regex class exposes only two properties, both of which are read-only. The Options property returns the second argument passed to the object constructor, while the RightToLeft property returns True if you specified the RightToLeft option. (The regular expression matches from right to left.) No property returns the regular expression pattern, but you can use the ToString method for this purpose.

Searching for Substrings

The Matches method searches the regular expression inside the string provided as an argument and returns a MatchCollection object that contains zero or more Match objects, one for each nonintersecting match. The Matches method is overloaded to take an optional starting index:

```
' Get the collection that contains all the matches.
Dim mc As MatchCollection = re.Matches(source)

' Print all the matches after the 100th character in the source string.
Dim m As Match
For Each m In re.Matches(source, 100)
    Console.WriteLine(m.ToString)
Next
```

You can change the behavior of the Matches method (as well as the Match method, described later) by using a *\G* assertion to disable scanning. In this case, the match must be found exactly where the scan begins: this point is either at the index specified as an argument (or the first character if this argument is omitted) or immediately after the point where the previous match terminates. In other words, the *\G* assertion finds only *consecutive* matches:

```
' Finds consecutive groups of space-delimited numbers.
Dim re As New Regex("\G\s*\d+")
' Note that search stops at the first non-numeric group.
Console.WriteLine(re.Matches("12 34 56 ab 78").Count)     ' => 3
```

Sometimes, you don't really want to list all the occurrences of the pattern, and determining whether the pattern is contained in the source string would suffice. If that's your interest, the IsMatch method is more efficient than the Matches method because it stops the scan as soon as the first match is found. You pass to this method the input string and an optional start index:

```
' Check whether the input string is a date in the format mm-dd-yy or
' mm-dd-yyyy. (The source string can use slashes as date separators and
' can contain leading or trailing white spaces.)
Dim re As New Regex("^\s*\d{1,2}(/|-)\d{1,2}\1(\d{4}|\d{2})\s*$")
If re.IsMatch(" 12/10/2001  ") Then
    Console.WriteLine("The date is formatted correctly.")
    ' (We don't check whether month and day values are in valid range.)
End If
```

The regular expression pattern in the preceding code requires an explanation:

1. The ^ and *$* characters mean that the source string must contain one date value and nothing else.
2. The *\s** subexpression at the beginning and end of the string means that we accept leading and trailing white spaces.
3. The *\d{1,2}* subexpression means that the month and day numbers can have one or two digits, whereas the *(\d{4}|\d{2})* subexpression means that the year number can have four or two digits. Note that the four-digit case must be tested first; otherwise, only the first two digits are matched.
4. The *(/|-)* subexpression means that we take either the slash or the dash as the date separator between the month and day numbers. (Note that you don't need to escape the slash character inside a pair of parentheses.)
5. The *\1* subexpression means that the separator between day and year numbers must be the same separator used between month and day numbers.

The Matches method is rarely used for parsing very long strings because it returns the control to the caller only when the entire string has been parsed. When parsing long strings, you should use the Match method instead; this method returns only the first Match object and lets you iterate over the remaining matches using the Match.NextMatch method, as this example demonstrates:

```
' Search all the dates in a source string.
Dim source As String = " 12-2-1999  10/23/2001 4/5/2001 "
Dim re As New Regex("\s*\d{1,2}(/|-)\d{1,2}\1(\d{4}|\d{2})")

' Find the first match.
Dim m As Match = re.Match(source)
' Enter the following loop only if the search was successful.
Do While m.Success
    ' Display the match, but discard leading and trailing spaces.
    Console.WriteLine(m.ToString.Trim)
    ' Find the next match; exit if not successful.
    m = m.NextMatch
Loop
```

The Split method is similar to the String.Split method except that it defines the delimiter by using a regular expression rather than a single character. For example, the following code prints all the elements in a comma-delimited list of numbers, ignoring leading and trailing white-space characters:

```
Dim source As String = "123, 456,,789"
Dim re As New Regex("\s*,\s*")

Dim s As String
For Each s In re.Split(source)
    ' Note that the third element is a null string.
    Console.Write(s & "-")     ' => 123-456--789-
Next
```

(You can modify the pattern to *\s*[,]+\s** to discard empty elements.) The Split method supports several overloaded variations, which let you define the maximum count of elements to be extracted and a starting index. (If there are more elements than the given limit, the last element contains the remainder of the string.)

```
' Split max 5 items.
Dim arr() As String = re.Split(source, 5)
' Split max 5 items, starting at the 100th character.
Dim arr2() As String = re.Split(source, 5, 100)
```

The Replace Method

As I explained earlier in this chapter, the Regex.Replace method lets you selectively replace portions of the source string. The Replace method requires that you create numbered or named groups of characters in the pattern and then use those groups in the replacement pattern. The following code example takes a string that contains one or more dates in the mm-dd-yy format (including their variations with / separator and four-digit year number) and converts them to the dd-mm-yy format, while preserving the original date separator:

```
Dim source As String = "12-2-1999  10/23/2001  4/5/2001 "
Dim pattern As String = _
    "\b(?<mm>\d{1,2})(?<sep>(/|-))(?<dd>\d{1,2})\k<sep>(?<yy>(\d{4}|\d{2}))\b"
Dim re As New Regex(pattern)
Console.WriteLine(re.Replace(source, "${dd}${sep}${mm}${sep}${yy}"))
    ' => 2-12-1999  23/10/2001  5/4/2001
```

The pattern string is similar to the one seen previously, with an important difference: it defines four groups—named *mm*, *dd*, *yy*, and *sep*—that are later rearranged in the replacement string. The *\b* assertion at the beginning and end of the pattern ensures that the date is a word of its own.

The Replace method supports other overloaded variants. For example, you can pass two additional numeric arguments, which are interpreted as the maximum number of substitutions and the starting index:

```
' Expand all "ms" abbreviations to "Microsoft," regardless of their case.
Dim source As String = "Welcome to MS Ms ms MS"
```

C12613753.fm Page 497 Friday, March 15, 2002 4:21 PM

```
Dim re As New Regex("\bMS\b", RegexOptions.IgnoreCase)
' Replace up to three occurrences, starting at the tenth character.
Console.WriteLine(re.Replace(source, "Microsoft", 3, 10))
    ' => Welcome to Microsoft Microsoft Microsoft MS
```

If the replacement operation does something more sophisticated than simply delete or change the order of named groups, you can use another overloaded version of the Replace function, which uses a delegate to call a function that you define in your application. This feature gives you tremendous flexibility, as the following code demonstrates:

```
Sub TestReplaceWithCallback()
    ' This pattern defines two integers separated by a plus sign.
    Dim re As New Regex("\d+\s*\+\s*\d+")
    Dim source As String = "a = 100 + 234: b = 200+345"
    ' Replace all sum operations with their results.
    Console.WriteLine(re.Replace(source, AddressOf DoSum))
        ' => a = 334: b = 545
End Sub

Function DoSum(ByVal m As Match) As String
    ' Find the position of the plus sign.
    Dim i As Integer = m.ToString.IndexOf("+"c)
    ' Parse the two operands.
    Dim n1 As Long = Long.Parse(m.ToString.Substring(0, i))
    Dim n2 As Long = Long.Parse(m.ToString.Substring(i + 1))
    ' Return their sum, as a string.
    Return (n1 + n2).ToString
End Function
```

The delegate must point to a function that takes a Match object and returns a String object. The code inside this function can query the Match object's properties to learn more about the match. For example, you can use the Index property to peek at what immediately precedes or follows in the source string so that you can make a more informed decision.

Shared Methods

All the methods seen so far are also available as shared methods, so most of the time you don't even need to explicitly create a Regex object. You generally pass the regular expression pattern to the shared methods as a second argument, after the source string. For example, you can split a string into individual words as follows:

```
' \W means "any nonalphanumeric character."
Dim words() As String = Regex.Split("Split these words", "\W+")
```

The Regex class exposes two shared methods that have no instance method counterpart. The Escape method takes a string and converts the special characters *.$^{[(|)*+?* to their equivalent escaped sequence. This method is especially useful when you let the end user enter the search pattern:

```
Console.Write(Regex.Escape("(x)"))     ' => \(x\)

' Check whether the character sequence the end user entered in
' the txtChars TextBox control is contained in the source string.
If Regex.IsMatch(source, Regex.Escape(txtChars.Text))
```

The Unescape shared method converts a string that contains escaped sequences back into its unescaped equivalent.

Working with Groups

The Regex class exposes four instance methods that let you quickly access the groups (both numbered or named) defined in the pattern string. For example, let's start by assuming that you have a Regex object defined as follows:

```
' The Regex object used for all the examples that follow.
' (Note that there are two named and two unnamed groups.)
Dim re As New Regex("(\s*)(?<name>\w+)\s*=\s*(?<value>\d+)(.*)")
```

The GetGroupNames method returns an array of strings that contains the names of all the groups in the pattern string. Groups without an explicit name are assigned names such as 1, 2, and so on. Group 0 always exists and corresponds to the entire pattern. The array returned by GetGroupNames contains all the numbered groups followed by all the named groups, so they aren't necessarily in the same order in which they appear in the pattern string:

```
Dim s As String
For Each s In re.GetGroupNames
    Console.Write(s & " ")     ' => 0 1 2 name value
Next
```

The GetGroupNumbers method works in a similar way but returns an Integer array, whose elements correspond to all the named or unnamed groups in the pattern string (which is not really useful information, admittedly):

```
Dim n As Integer
For Each n In re.GetGroupNumbers
    Console.Write(n.ToString & " ")     ' => 0 1 2 3 4
Next
```

The GroupNameFromNumber method returns the group name corresponding to the given group number. If the group has no name, its number is returned; if the number is higher than the number of existing groups, an IndexOutOfRange exception is thrown:

```
Console.WriteLine(re.GroupNameFromNumber(2))    ' => 2
Console.WriteLine(re.GroupNameFromNumber(3))    ' => name
```

The GroupNumberFromName method returns the group number if a group with that name exists or –1 if no group with that name exists:

```
Console.WriteLine(re.GroupNumberFromName("name"))  ' => 3
Console.WriteLine(re.GroupNumberFromName("foo"))   ' => -1
```

The MatchCollection and Match Classes

The MatchCollection class represents a set of matches. It has no constructor because you can create a MatchCollection object only by using the Regex.Matches method. This class supports the ICollection and IEnumerable interfaces, and therefore it exposes all the properties and methods that collections make available, including Count, Item, CopyTo, and GetEnumerator.

The Match class represents a single match. You can obtain an instance of this class either by iterating on a MatchCollection object or directly by means of the Match method of the Regex class. The Match object is immutable and has no public constructor.

The Match class's main properties are Value, Length, and Index, which return the matched string, its length, and the index at which it appears in the source string. (The ToString method returns the same string as the Value property does.) I already showed you how to use the Match class's IsSuccess property and its NextMatch method to iterate over all the matches in a string:

```
' Find the first match.
Dim m As Match = re.Match(source)
' Enter the following loop only if the search was successful.
Do While m.Success
    Console.WriteLine("{0} (found at {1})", m.Value, m.Index)
    ' Find the next match; exit if not successful.
    m = m.NextMatch
Loop
```

You must pay special attention when the search pattern matches an empty string, for example, *\d** (which matches zero or more digits). When you apply such a pattern to a string, you typically get one or more empty matches, as you can see here:

```
Dim re As New Regex("\d*")
Dim m As Match
For Each m In re.Matches("1a23bc456de789")
    ' The output from this loop shows that some matches are empty.
    Console.Write(m.Value & ",")  ' => 1,,23,,456,,,789,,
Next
```

As I explained earlier, a search generally starts where the previous search ends. However, the rule is different when the engine finds an empty match because it advances by one character before repeating the search. You would get trapped in an endless loop if the engine didn't behave this way.

If the pattern contains one or more groups, you can access the corresponding Group object by means of the Match object's Groups collection, which you can index by the group number or group name. I'll discuss the Group object in a moment, but you can already see how you can use the Groups collection to extract the variable names and values in a series of assignments:

```
Dim source As String = "a = 123: b=456"
Dim re As New Regex("(\s*)(?<name>\w+)\s*=\s*(?<value>\d+)")

Dim m As Match
For Each m In re.Matches(source)
    Console.WriteLine("Variable: {0}  Value: {1}", _
       m.Groups("name").Value, m.Groups("value").Value)
Next
```

This is the result displayed in the console window:

```
Variable: a  Value: 123
Variable: b  Value: 456
```

The Result method takes a replace pattern and returns the string that would result if the match were replaced by that pattern:

```
' This code produces exactly the same result as the preceding snippet.
For Each m In re.Matches(source)
    Console.WriteLine(m.Result("Variable: ${name}  Value: ${value}"))
Next
```

The only Match member left to be discussed is the Captures property, which I'll cover in the "CaptureCollection and Capture Classes" section later in this chapter.

The Group Class

The Group class represents a single group in a Match object and exposes a few properties whose meaning should be evident: Value (which produces the same result as the ToString method—the text associated with the group), Index (its position in the source string), Length (the group's length), and Success (True if the group has been matched). This code sample is similar to the preceding example, but it also displays information concerning where each variable appears in the source string:

```
Dim source As String = "a = 123: b=456"
Dim re As New Regex("(\s*)(?<name>\w+)\s*=\s*(?<value>\d+)")

Dim m As Match, g As Group
' Iterate over all the matches.
For Each m In re.Matches(source)
    ' Get information on variable name.
    g = m.Groups("name")
    Console.Write("Variable '{0}' found at index {1}", g.Value, g.Index)
    ' Get information on variable value.
    Console.WriteLine(", value is {0}", m.Groups("value").Value)
Next
```

This is the result displayed in the console window:

```
Variable 'a' found at index 0, value is 123
Variable 'b' found at index 9, value is 456
```

The following example is more complex but also more useful: it shows how you can parse <A> tags in an HTML file and display the anchor text (that is, the text that appears underlined in an HTML page) and the URL it points to. As you see, it's just a matter of a few lines of code:

```
Sub TestSearchHyperlinks()
    Dim re As New Regex("<A\s+HREF\s*=\s*""?([^"" >]+)""?>(.+)</A>", _
        RegexOptions.IgnoreCase)
    ' Load the contents of an HTML file.
    Dim source As String = FileText("test.htm")
    ' Display all occurrences.
    Dim m As Match = re.Match(source)
    Do While m.Success
        Console.WriteLine("{0} => {1}", m.Groups(2).Value, _
            m.Groups(1).Value)
        m = m.NextMatch()
    Loop
End Sub
```

To understand how the preceding code works, you must keep in mind that the <A> tag is followed by one or more spaces and then by an HREF attribute, which is followed by an equal sign and then the URL, optionally enclosed between double quotation marks. All the text that follows the closing angle bracket and up to the ending tag </A> is the anchor text. The regular expression defined in the preceding code defines two unnamed groups—the

URL and the anchor text—so displaying details for all the <A> tags in the HTML file is just a matter of looping over all the matches. The regular expression syntax is complicated by the fact that double quotation mark characters must be doubled when they appear in a string constant. Also, note that the URL group is defined by the repetition of any character other than the double quotation marks, spaces, and closing angle brackets.

The CaptureCollection and Capture Classes

The search pattern can include one or more *capturing groups*, which are named or unnamed subexpressions enclosed in parentheses. Capturing groups can be nested and can capture multiple substrings of the source strings because of quantifiers. For example, when you apply the *(\w)*+ pattern to the "abc" string, you get one match for the entire string and three captured substrings, one for each character.

You can access the collection of capture substrings through the Captures method of either the Match or the Group object. This method returns a CaptureCollection object that in turn contains one or more Capture objects. Individual Capture objects let you determine where individual captured substrings appear in the source string. The following code displays all the captured strings in the "abc def" string:

```
Dim source As String = "abc def"
Dim re As New Regex("(\w)+")
Dim m As Match, s As String, c As Capture

' Get the name or numbers of all the groups.
Dim groups() As String = re.GetGroupNames

' Iterate over all matches.
For Each m In re.Matches(source)
    ' Display information on this match.
    Console.WriteLine("Match '{0}' at index {1}", m.Value, m.Index)
    ' Iterate over the groups in each match.
    For Each s In groups
        ' Get a reference to the corresponding group.
        Dim g As Group = m.Groups(s)
        ' Get the capture collection for this group.
        Dim cc As CaptureCollection = g.Captures
        ' Display the number of captures.
        Console.WriteLine("  Found {0} capture(s) for group {1}", cc.Count, s)
        ' Display information on each capture.
        For Each c In cc
            Console.WriteLine("    '{0}' at index {1}", c.Value, c.Index)
        Next
    Next
Next
```

The code that follows is the result produced in the console window. (Notice that group 0 always refers to the match expression itself.)

```
Match 'abc' at index 0
  Found 1 capture(s) for group 0
    'abc' at index 0
  Found 3 capture(s) for group 1
    'a' at index 0
    'b' at index 1
    'c' at index 2
Match 'def' at index 4
  Found 1 capture(s) for group 0
    'def' at index 4
  Found 3 capture(s) for group 1
    'd' at index 4
    'e' at index 5
    'f' at index 6
```

The following example is somewhat more useful in that it defines a pattern that lets you parse a number in exponential format and shows how to locate individual portions of the mantissa and the exponent:

```
Dim source As String = "11.22E33  4.55E6 "
Dim re As New Regex("((\d+).?(\d*))E(\d+)")
' ... (The remainder of code as in previous snippet)...
```

Note that the previous regular expression defines as many as five groups: the entire number, the entire mantissa, the integer portion of the mantissa, the decimal portion of the mantissa, and the exponent. This is the result displayed in the console window:

```
Match '11.22E33' at index 0
  Found 1 capture(s) for group 0
    '11.22E33' at index 0
  Found 1 capture(s) for group 1
    '11.22' at index 0
  Found 1 capture(s) for group 2
    '11' at index 0
  Found 1 capture(s) for group 3
    '22' at index 3
  Found 1 capture(s) for group 4
    '33' at index 6
Match '4.55E6' at index 10
  Found 1 capture(s) for group 0
    '4.55E6' at index 10
  Found 1 capture(s) for group 1
    '4.55' at index 10
  Found 1 capture(s) for group 2
    '4' at index 10
  Found 1 capture(s) for group 3
```

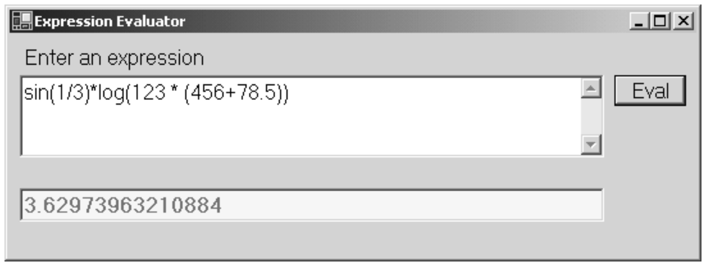
(continued)

```
    '55' at index 12
  Found 1 capture(s) for group 4
    '6' at index 15
```

Regular Expressions at Work

By now it should be evident that regular expressions are a powerful tool in the hands of expert developers. Admittedly, the regular expression sublanguage has a steep learning curve, but the effort pays off nicely. In this section, I show you how you can use regular expressions to implement a full-featured expression evaluator—that is, a routine that takes a string containing a math expression and returns its result. (See Figure 12-2.)

```
Console.Write(Evaluate("10 + (8 - 3) * 4"))    ' => 30
```



sin(1/3)*log(123 * (456+78.5))

3.62973963210884

Figure 12-2. The ExprEvaluator application on the companion CD offers a simple front end for the Evaluate function.

If you have ever tried to build an expression evaluator, you know that it isn't for the faint of heart. I once wrote an expression evaluator in Visual Basic 5 that took me a few days and more than 1000 lines of code, so I decided to see whether the task could have been solved in a simpler way with regular expressions. To my surprise, I was able to come up with a complete solution with a little more than 100 lines of executable code (not counting remarks). The result is even more sophisticated than my original expression evaluator and even supports niceties such as functions with variable numbers of arguments. Here is its complete source code:

```
Function Evaluate(ByVal expr As String) As Double
    ' A number is a sequence of digits optionally followed by a dot and
    ' another sequence of digits. The number is in parentheses in order to
    ' define an unnamed group.
    Const Num As String = "(\-?\d+\.?\d*)"
    ' List of 1-operand functions
```

```
Const Func1 As String = _
    "(exp|log|log10|abs|sqr|sqrt|sin|cos|tan|asin|acos|atan)"
' List of 2-operand functions
Const Func2 As String = "(atan2)"
' List of N-operand functions
Const FuncN As String = "(min|max)"
' List of predefined constants
Const Constants As String = "(e|pi)"

' Define one Regex object for each supported operation. These DIMs
' are outside the loop, so they are compiled only once.
' Binary operations are defined as two numbers with a symbol between
' them and optional spaces in between.
Dim rePower As New Regex(Num & "\s*(\^)\s*" & Num)
Dim reAddSub As New Regex(Num & "\s*([-+])\s*" & Num)
Dim reMulDiv As New Regex(Num & "\s*([*/])\s*" & Num)
' These Regex objects resolve calls to functions (case insensitivity).
Dim reFunc1 As New Regex(Func1 & "\(\s*" & Num & "\s*\)", _
    RegexOptions.IgnoreCase)
Dim reFunc2 As New Regex(Func2 & "\(\s*" & Num & "\s*,\s*" & Num _
    & "\s*\)", RegexOptions.IgnoreCase)
Dim reFuncN As New Regex(FuncN & "\((\s*" & Num & "\s*,)+\s*" & Num _
    & "\s*\)", RegexOptions.IgnoreCase)
' This Regex object drops a + when it follows an operator.
Dim reSign1 As New Regex("([-+/*^])\s*\+")
' This Regex object converts a double minus into a plus.
Dim reSign2 As New Regex("\-\s*\-")
' This Regex object drops parentheses around a number. (It must not
' be preceded by an alphanum char because it might be a function name.)
Dim rePar As New Regex("(?<![A-Za-z0-9])\(\s*([-+]?\d+.?\d*)\s*\)")
' This Regex tells when the expression has been reduced to a number.
Dim reNum As New Regex("^\s*[-+]?\d+\.?\d*\s*$")

' The Regex object deals with constants. (Requires case insensitivity.)
Dim reConst As New Regex("\s*" & Constants & "\s*", _
    RegexOptions.IgnoreCase)
' This statement resolves constants. (Can be kept out of the loop.)
expr = reConst.Replace(expr, AddressOf DoConstants)

' Loop until the entire expression becomes just a number.
Do Until reNum.IsMatch(expr)
    ' Remember current expression.
    Dim saveExpr As String = expr

    ' Perform all the math operations in the source string,
    ' starting with operands with higher operands.
    ' Note that we continue to perform each operation until there are
    ' no matches because we must account for expressions such as
    ' (12*34*56).
```

(continued)

```
            ' Perform all power operations.
            Do While rePower.IsMatch(expr)
                expr = rePower.Replace(expr, AddressOf DoPower)
            Loop

            ' Perform all divisions and multiplications.
            Do While reMulDiv.IsMatch(expr)
                expr = reMulDiv.Replace(expr, AddressOf DoMulDiv)
            Loop

            ' Perform functions with variable numbers of arguments.
            Do While reFuncN.IsMatch(expr)
                expr = reFuncN.Replace(expr, AddressOf DoFuncN)
            Loop

            ' Perform functions with 2 arguments.
            Do While reFunc2.IsMatch(expr)
                expr = reFunc2.Replace(expr, AddressOf DoFunc2)
            Loop

            ' One-operand functions must be processed last to deal correctly with
            ' expressions such as SIN(ATAN(1)).
            Do While reFunc1.IsMatch(expr)
                expr = reFunc1.Replace(expr, AddressOf DoFunc1)
            Loop

            ' Discard plus signs (unary pluses) that follow another operator.
            expr = reSign1.Replace(expr, "$1")
            ' Simplify two consecutive minus signs into a plus sign.
            expr = reSign2.Replace(expr, "+")

            ' Perform all additions and subtractions.
            Do While reAddSub.IsMatch(expr)
                expr = reAddSub.Replace(expr, AddressOf DoAddSub)
            Loop

            ' Attempt to discard parentheses around numbers.
            expr = rePar.Replace(expr, "$1")

            ' If the expression didn't change, we have a syntax error.
            ' (This serves to avoid endless loops.)
            If expr = saveExpr Then Throw New SyntaxErrorException()
        Loop

        ' Return the expression, which is now a number.
        Return CDbl(expr)
    End Function

    ' These functions perform the actual math operations.
```

```
' In all cases, the incoming Match object has groups that identify
' the two operands and the operator.

Function DoConstants(ByVal m As Match) As String
    Select Case m.Groups(1).Value.ToUpper
        Case "PI"
            Return Math.PI.ToString
        Case "E"
            Return Math.E.ToString
    End Select
End Function

Function DoPower(ByVal m As Match) As String
    Dim n1 As Double = CDbl(m.Groups(1).Value)
    Dim n2 As Double = CDbl(m.Groups(3).Value)
    ' Group(2) is always the ^ character in this version.
    Return (n1 ^ n2).ToString
End Function

Function DoMulDiv(ByVal m As Match) As String
    Dim n1 As Double = CDbl(m.Groups(1).Value)
    Dim n2 As Double = CDbl(m.Groups(3).Value)
    Select Case m.Groups(2).Value
        Case "/"
            Return (n1 / n2).ToString
        Case "*"
            Return (n1 * n2).ToString
    End Select
End Function

Function DoAddSub(ByVal m As Match) As String
    Dim n1 As Double = CDbl(m.Groups(1).Value)
    Dim n2 As Double = CDbl(m.Groups(3).Value)
    Select Case m.Groups(2).Value
        Case "+"
            Return (n1 + n2).ToString
        Case "-"
            Return (n1 - n2).ToString
    End Select
End Function

Function DoFunc1(ByVal m As Match) As String
    ' Function argument is second group.
    Dim n1 As Double = CDbl(m.Groups(2).Value)
    ' Function name is first group.
    Select Case m.Groups(1).Value.ToUpper
        Case "EXP"
            Return Math.Exp(n1).ToString
        Case "LOG"
```

(continued)

```
                Return Math.Log(n1).ToString
            Case "LOG10"
                Return Math.Log10(n1).ToString
            Case "ABS"
                Return Math.Abs(n1).ToString
            Case "SQR", "SQRT"
                Return Math.Sqrt(n1).ToString
            Case "SIN"
                Return Math.Sin(n1).ToString
            Case "COS"
                Return Math.Cos(n1).ToString
            Case "TAN"
                Return Math.Tan(n1).ToString
            Case "ASIN"
                Return Math.Asin(n1).ToString
            Case "ACOS"
                Return Math.Acos(n1).ToString
            Case "ATAN"
                Return Math.Atan(n1).ToString
        End Select
    End Function

    Function DoFunc2(ByVal m As Match) As String
        ' Function arguments are second and third groups.
        Dim n1 As Double = CDbl(m.Groups(2).Value)
        Dim n2 As Double = CDbl(m.Groups(3).Value)
        ' Function name is first group.
        Select Case m.Groups(1).Value.ToUpper
            Case "ATAN2"
                Return Math.Atan2(n1, n2).ToString
        End Select
    End Function

    Function DoFuncN(ByVal m As Match) As String
        ' Function arguments are from group 2 onward.
        Dim args As New ArrayList()
        Dim i As Integer = 2
        ' Load all the arguments into the array.
        Do While m.Groups(i).Value <> ""
            ' Get the argument, replace any comma with a space,
            ' and convert to double.
            args.Add(CDbl(m.Groups(i).Value.Replace(","c, " "c)))
            i += 1
        Loop

        ' Function name is 1st group.
        Select Case m.Groups(1).Value.ToUpper
            Case "MIN"
                args.Sort()
                Return args(0).ToString
            Case "MAX"
                args.Sort()
                Return args(args.Count - 1).ToString
        End Select
    End Function
```

I won't explain every single line of code in the preceding listing because you can follow the many comments in the code. The code exclusively uses the regular expression features I've described earlier in this chapter, so you shouldn't have problems understanding how it works. But here are just a few hints:

- The feature on which this expression evaluator is built is the ability to provide a delegate function as an argument of the Replace method. The program uses several delegate functions, one for each class of operators and functions. The delegate function receives a Match object whose groups reference the operator or the function name and all the operands.
- The Evaluate function receives a string, which is then passed to the Replace method of the several Regex objects defined internally, where each Regex object is in charge of parsing a given set of operations or of simplifying the expression—for example, by dropping a pair of parentheses around a number.
- A single Regex object processes operators with the same priority—for example, + and –, or * and /. All the Regex objects are instantiated outside the main loop so that they compile to byte codes only once. Inside the main loop, Regex objects related to operators with higher priority are given a chance to process the string before those related to operations with lower priority.
- Thanks to subsequent replacements, the expression string becomes simpler at each iteration of the main loop. The loop is repeated until the string becomes a number in a valid format for the CDbl function.

It's easy to extend the expression evaluator with new functions because you simply need to extend the Func1, Func2, or FuncN constant and add a Case block in the corresponding callback function (DoFunc1, DoFunc2, or DoFuncN). It takes a little more effort to improve the function even further with the ability to define your own variables—but I gladly leave it to you as an exercise.

Multithreading has always been the Achilles' heel of Visual Basic, and in fact you've had to resort to coding in another language to achieve free-threaded applications. Well, this problem will soon become a thing of the past, thanks to the multithreading support offered by the .NET Framework, as you'll learn in the next chapter.