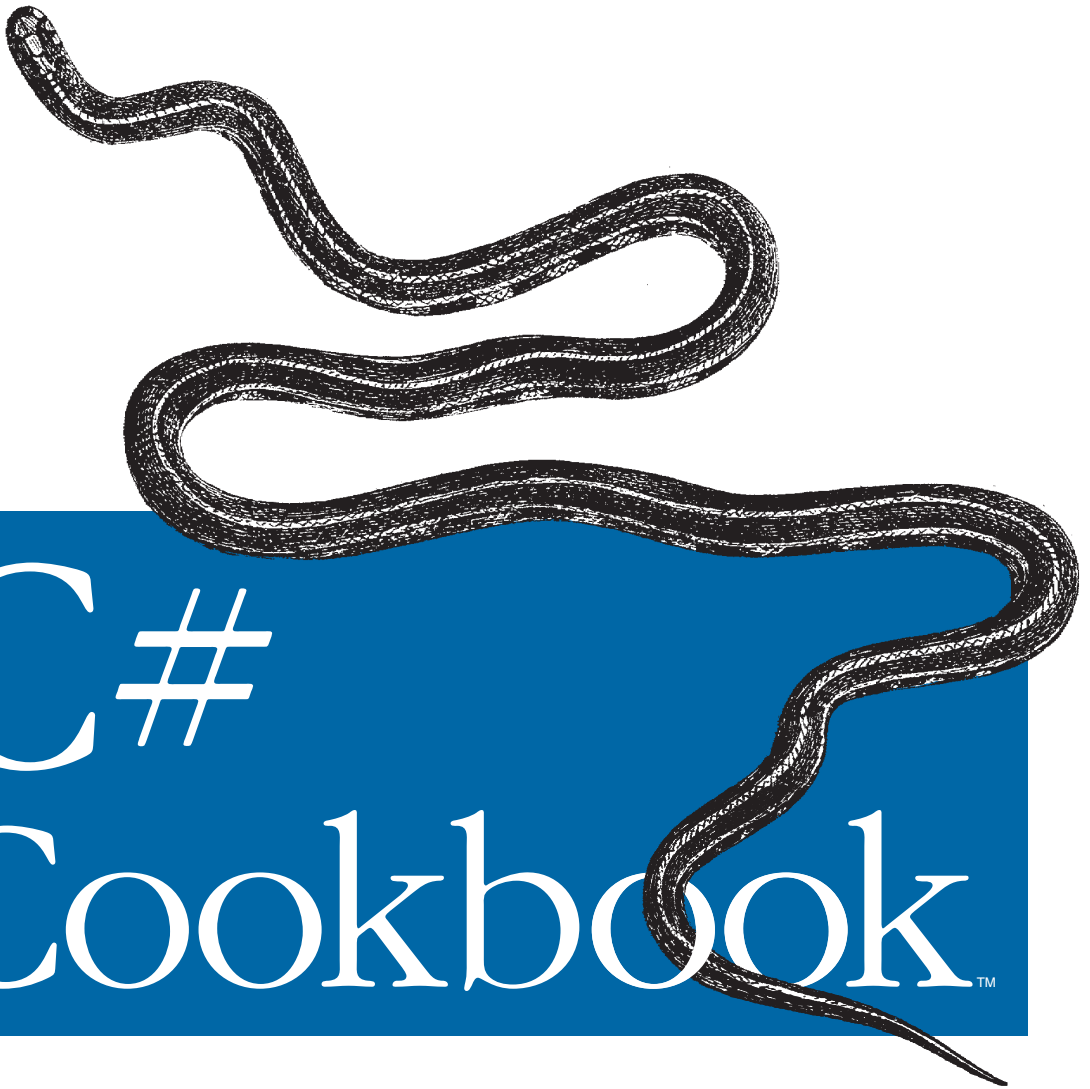


The Definitive Collection of Solutions and Examples



C# Cookbook™

O'REILLY®

Stephen Teilhet & Jay Hilyard

Regular Expressions

Included in the .NET Framework Class Library is the `System.Text.RegularExpressions` namespace that is devoted to creating, executing, and obtaining results from regular expressions executed against a string.

Regular expressions take the form of a pattern that can be matched to zero or more characters within a string. The simplest of these patterns, such as `.*` (match anything and everything) and `[A-Za-z]` (match any letter) are easy to learn, but more advanced patterns can be difficult to learn and even more difficult to implement correctly. Learning and understanding regular expressions can take considerable time and effort, but the work will pay off.

Regular expression patterns can take a simple form—such as a single word or character—or a much more complex pattern. The more complex patterns can recognize and match such things as the year portion of a date, all of the `<SCRIPT>` tags in an ASP page, or a phrase in a sentence that varies with each use. The .NET regular expression classes provide a very flexible and powerful way to do such things as recognize text, replace text within a string, and split up text into individual sections based on one or more complex delimiters.

Despite the complexity of regular expression patterns, the regular expression classes in the FCL are easy to use in your applications. Executing a regular expression consists of the following steps:

1. Create an instance of the `Regex` object that contains the regular expression pattern along with any options for executing that pattern.
2. Retrieve a reference to an instance of the `Match` object by calling the `Match` instance method if you want only the first match found, or to an instance of the `MatchesCollection` object by calling the `Matches` instance method if you want more than just the first match found.
3. If you've called the `Matches` method to retrieve a `MatchCollection` object, iterate over the `MatchCollection` using a `foreach` loop. Each iteration will allow access to every `Match` object that the regular expression produced.

8.1 Enumerating Matches

Problem

You need to find one or more substrings corresponding to a particular pattern within a string. You need to be able to inform the searching code to return either all matching substrings or only the matching substrings that are unique within the set of all matched strings.

Solution

Call the `FindSubstrings` method, which executes a regular expression and obtains all matching text. This method returns either all matching results or only the unique matches; this behavior is controlled by the `findAllUnique` parameter. Note that if the `findAllUnique` parameter is set to `true`, the unique matches are returned sorted alphabetically. Its source code is as follows:

```
using System;
using System.Collections;
using System.Text.RegularExpressions;

public static Match[] FindSubstrings(string source, string matchPattern,
                                     bool findAllUnique)
{
    SortedList uniqueMatches = new SortedList();
    Match[] retArray = null;

    Regex RE = new Regex(matchPattern, RegexOptions.Multiline);
    MatchCollection theMatches = RE.Matches(source);

    if (findAllUnique)
    {
        for (int counter = 0; counter < theMatches.Count; counter++)
        {
            if (!uniqueMatches.ContainsKey(theMatches[counter].Value))
            {
                uniqueMatches.Add(theMatches[counter].Value,
                                   theMatches[counter]);
            }
        }

        retArray = new Match[uniqueMatches.Count];
        uniqueMatches.Values.CopyTo(retArray, 0);
    }
    else
    {
        retArray = new Match[theMatches.Count];
        theMatches.CopyTo(retArray, 0);
    }

    return (retArray);
}
```

The following method searches for any tags in an XML string; it does this by searching for a block of text that begins with the < character and ends with the > character.

This method first displays all unique tag matches present in the XML string and then displays all tag matches within the string:

```
public static void TestFindSubstrings()
{
    string matchPattern = "<.*>";

    string source = @"<?xml version='1.0' encoding='UTF-8'?>
        <!-- my comment -->
        <![CDATA[<escaped> <><chars>>>>]]>
        <Window ID='Main'>
            <Control ID='TextBox'>
                <Property Top='0' Left='0' Text='BLANK' />
            </Control>
            <Control ID='Label'>
                <Property Top='0' Left='0' Caption='Enter Name Here' />
            </Control>
            <Control ID='Label'>
                <Property Top='0' Left='0' Caption='Enter Name Here' />
            </Control>
        </Window>";

    Console.WriteLine("UNIQUE MATCHES");
    Match[] x1 = FindSubstrings(source, matchPattern, true);
    foreach(Match m in x1)
    {
        Console.WriteLine(m.Value);
    }

    Console.WriteLine();
    Console.WriteLine("ALL MATCHES");
    Match[] x2 = FindSubstrings(source, matchPattern, false);
    foreach(Match m in x2)
    {
        Console.WriteLine(m.Value);
    }
}
```

The following text will be displayed:

```
UNIQUE MATCHES
<!-- my comment -->
<![CDATA[<escaped> <><chars>>>>]]>
</Control>
</Window>
<?xml version="1.0\" encoding=\"UTF-8\"?>
<Control ID="Label">
<Control ID="TextBox">
<Property Top="0" Left="0" Caption="Enter Name Here"/>
<Property Top="0" Left="0" Text="BLANK"/>
<Window ID="Main">
```

```

ALL MATCHES
<?xml version="1.0" encoding="UTF-8"?>
<!-- my comment -->
<![CDATA[<escaped> <><chars>>>>]]>
<Window ID="Main">
<Control ID="TextBox">
<Property Top="0" Left="0" Text="BLANK"/>
</Control>
<Control ID="Label">
<Property Top="0" Left="0" Caption="Enter Name Here"/>
</Control>
<Control ID="Label">
<Property Top="0" Left="0" Caption="Enter Name Here"/>
</Control>
</Window>

```

Discussion

As you can see, the regular expression classes in the FCL are quite easy to use. The first step is to create an instance of the `Regex` object that contains the regular expression pattern along with any options for running this pattern. The second step is to get a reference to an instance of the `Match` object, if you only need the first found match, or a `MatchCollection` object, if you need more than just the first found match. To get a reference to this object, the two instance methods `Match` and `Matches` can be called from the `Regex` object that was created in the first step. The `Match` method returns a single match object (`Match`) and `Matches` returns a collection of match objects (`MatchCollection`).

The `FindSubstrings` method returns an array of `Match` objects that can be used by the calling code. You might have noticed that the unique elements are returned sorted, and the nonunique elements are not sorted. A `SortedList`, which is used by the `FindSubstrings` method to store unique strings that match the regular expression pattern, automatically sorts its items when they are added.

The regular expression used in the `TestFindSubstrings` method is very simplistic and will work in most—but not all—conditions. For example, if two tags are on the same line, as shown here:

```
<tagData></tagData>
```

the regular expression will catch the entire line, not each tag separately. You could change the regular expression from `<.*>` to `<[^>]*>` to match only up to the closing `>` (`[^>]*` matches everything that is *not* a `>`). However, this will fail in the CDATA section, matching `<![CDATA[<escaped>, <>, and <chars> instead of <![CDATA[<escaped> <> <chars>>>>]]>`. The more complicated `@("<![CDATA.*>|<[^>]*>")` will match either `<![CDATA.*>` (a greedy match for everything within the CDATA section) or `<[^>]*>`, described previously.

See Also

See the “.NET Framework Regular Expressions” and “SortedList Class” topics in the MSDN documentation.

8.2 Extracting Groups from a MatchCollection

Problem

You have a regular expression that contains one or more named groups, such as the following:

```
\\(?:<TheServer>\w*)\\(?:<TheService>\w*)\\
```

where the named group `TheServer` will match any server name within a UNC string, and `TheService` will match any service name within a UNC string.

You need to store the groups that are returned by this regular expression in a keyed collection (such as a `Hashtable`) in which the key is the group name.

Solution

The `RegExUtilities` class contains a method, `ExtractGroupings`, that obtains a set of `Group` objects keyed by their matching group name:

```
using System;
using System.Collections;
using System.Text.RegularExpressions;

public static ArrayList ExtractGroupings(string source,
                                       string matchPattern,
                                       bool wantInitialMatch)
{
    ArrayList keyedMatches = new ArrayList();
    int startingElement = 1;
    if (wantInitialMatch)
    {
        startingElement = 0;
    }

    Regex RE = new Regex(matchPattern, RegexOptions.Multiline);
    MatchCollection theMatches = RE.Matches(source);

    foreach(Match m in theMatches)
    {
        Hashtable groupings = new Hashtable();

        for (int counter = startingElement;
            counter < m.Groups.Count; counter++)
        {
            // If we had just returned the MatchCollection directly, the
            // GroupNameFromNumber method would not be available to use
        }
    }
}
```

```

        groupings.Add(RE.GroupNameFromNumber(counter),
                    m.Groups[counter]);
    }

    keyedMatches.Add(groupings);
}

return (keyedMatches);
}

```

The ExtractGroupings method can be used in the following manner to extract named groups and organize them by name:

```

public static void TestExtractGroupings()
{
    string source = @"Path = "\\MyServer\MyService\MyPath;
                    \\MyServer2\MyService2\MyPath2\"";
    string matchPattern = @"\\(?:<TheServer>\w*)\\(?:<TheService>\w*)\\";

    foreach (Hashtable grouping in
        ExtractGroupings(source, matchPattern, true))
    {
        foreach (DictionaryEntry DE in grouping)
            Console.WriteLine("Key / Value = " + DE.Key + " / " +
                               DE.Value);
        Console.WriteLine("");
    }
}

```

This test method creates a source string and a regular expression pattern in the MatchPattern variable. The two groupings in this regular expression are highlighted here:

```

string matchPattern = @"\\(?:<TheServer>\w*)\\(?:<TheService>\w*)\\";

```

The names for these two groups are: TheServer and TheService. Text that matches either of these groupings can be accessed through these group names.

The source and matchPattern variables are passed in to the ExtractGroupings method, along with a Boolean value, which we will discuss shortly. This method returns an ArrayList containing Hashtable objects. These Hashtable objects contain the matches for each of the named groups in the regular expression, keyed by their group name.

This test method, TestExtractGroupings, returns the following:

```

Key / Value = 0 / \\MyServer\MyService\
Key / Value = TheService / MyService
Key / Value = TheServer / MyServer

Key / Value = 0 / \\MyServer2\MyService2\
Key / Value = TheService / MyService2
Key / Value = TheServer / MyServer2

```

If the last parameter to the `ExtractGroupings` method were to be changed to `false`, the following output would result:

```
Key / Value = TheService / MyService
Key / Value = TheServer / MyServer

Key / Value = TheService / MyService2
Key / Value = TheServer / MyServer2
```

The only difference between these two outputs are that the first grouping is not displayed when the last parameter to `ExtractGroupings` is changed to `false`. The first grouping is always the complete match of the regular expression.

Discussion

Groups within a regular expression can be defined in one of two ways. The first way is to add parentheses around the subpattern that you wish to define as a grouping. This type of grouping is sometimes labeled as *unnamed*. This grouping can later be easily extracted from the final text in each `Match` object returned by running the regular expression. The regular expression for this recipe could be modified, as follows, to use a simple unnamed group:

```
string matchPattern = @"\\(\\w*)\\(\\w*)\\";
```

After running the regular expression, you can access these groups using a numeric integer value starting with 1.

The second way to define a group within a regular expression is to use one or more *named groups*. A named group is defined by adding parentheses around the subpattern that you wish to define as a grouping and, additionally, adding a named value to each grouping, using the following syntax:

```
(?<Name>\\w*)
```

The *Name* portion of this syntax is the name you specify for this group. After executing this regular expression, you can access this group by the name *Name*.

To access each group, you must first use a loop to iterate each `Match` object in the `MatchCollection`. For each `Match` object, you access the `GroupCollection`'s indexer, using the following unnamed syntax:

```
string group1 = m.Groups[1].Value;
string group2 = m.Groups[2].Value;
```

or the following named syntax where `m` is the `Match` object:

```
string group1 = m.Groups["Group1_Name"].Value;
string group2 = m.Groups["Group2_Name"].Value;
```

If the `Match` method was used to return a single `Match` object instead of the `MatchCollection`, use the following syntax to access each group:

```
// Un-named syntax
string group1 = theMatch.Groups[1].Value;
string group2 = theMatch.Groups[2].Value;
```

```
// Named syntax
string group1 = theMatch.Groups["Group1_Name"].Value;
string group2 = theMatch.Groups["Group2_Name"].Value;
```

where theMatch is the Match object returned by the Match method.

See Also

See the “.NET Framework Regular Expressions” and “Hashtable Class” topics in the MSDN documentation.

8.3 Verifying the Syntax of a Regular Expression

Problem

You have either constructed a regular expression dynamically from your code or based on user input. You need to test the validity of this regular expression’s syntax before you actually use it.

Solution

Use the following method to test the validity of a regular expression’s syntax:

```
using System;
using System.Text.RegularExpressions;

public static bool VerifyRegEx(string testPattern)
{
    bool isValid = true;

    if ((testPattern != null) && (testPattern.Trim().Length > 0))
    {
        try
        {
            Regex.Match("", testPattern);
        }
        catch (ArgumentException)
        {
            // BAD PATTERN: Syntax error
            isValid = false;
        }
    }
    else
    {
        //BAD PATTERN: Pattern is null or blank
        isValid = false;
    }

    return (isValid);
}
```

To use this method, pass it the regular expression that you wish to verify:

```
public static void TestUserInputRegex(string regex)
{
    if (VerifyRegex(regex))
        Console.WriteLine("This is a valid regular expression.");
    else
        Console.WriteLine("This is not a valid regular expression.");
}
```

Discussion

The `VerifyRegex` method calls the static `Regex.Match` method, which is useful for running quick regular expressions against a string. The static `Regex.Match` method returns a single `Match` object. By using this static method to run a regular expression against a string (in this case a blank string), we can determine whether the regular expression is invalid by watching for a thrown exception. The `Regex.Match` method will throw an `ArgumentException` if the regular expression is not syntactically correct. The `Message` property of this exception contains the reason the regular expression failed to run, and the `ParamName` property contains the regular expression passed to the `Match` method. Both of these properties are read-only.

Before testing the regular expression with the static `Match` method, the regular expression is tested to see if it is null or blank. A null regular expression string returns an `ArgumentNullException` when passed in to the `Match` method. On the other hand, if a blank regular expression is passed in to the `Match` method, no exception is thrown (as long as a valid string is also passed to the first parameter of the `Match` method).

8.4 Quickly Finding Only the Last Match in a String

Problem

You need to find the last pattern match in a string, but you do not want the overhead of finding all matches in a string and having to move to the last match in the collection of matches.

Solution

Using the `RegexOptions.RightToLeft` option, the match starts at the end of the string and proceeds toward the beginning. The first found match is the last match in the string. You supply the `RegexOptions.RightToLeft` constant as an argument to the `Match` method. The instance `Match` method can be used as follows:

```
Regex RE = new Regex(Pattern, RegexOptions.RightToLeft);
Match theMatch = RE.Match(Source);
```

or use the static `Regex.Match` method:

```
Match theMatch = Regex.Match(Source, Pattern, RegexOptions.RightToLeft);
```

where *Pattern* is the regular expression pattern and *Source* is the string against which to run the pattern.

Discussion

The `RegexOptions.RightToLeft` regular expression option will force the regular expression engine to start searching for a pattern starting with the end of the string and proceeding backward toward the beginning of the string. The first match encountered will be the match closest to the end of the string—in other words, the last match in the string.

See Also

See the “.NET Framework Regular Expressions” topic in the MSDN documentation.

8.5 Replacing Characters or Words in a String

Problem

You are given a string in which a complex pattern of characters needs to be replaced with a new string.

Solution

Using the `Replace` instance method on the `Regex` class allows for easy replacement of text within a string. The following overloaded `Replace` methods accept a *source* string that contains characters or words to be replaced, a *matchPattern* to match the replaceable text in the *source* parameter, and a *replaceStr* string to replace the text matched by *matchPattern*. In addition there are two parameters, *count* and *startPos*, to control the number of replacements allowed and where the replacements start from in the *source* string, respectively:

```
using System;
using System.Text.RegularExpressions;

public static string Replace(string source, char matchPattern,
                             string replaceStr)
{
    return (Replace(source, matchPattern.ToString(), replaceStr, -1, 0));
}

public static string Replace(string source, char matchPattern,
                             string replaceStr, int count)
{
    return (Replace(source.ToString(), matchPattern.ToString(), replaceStr,
                    count, 0));
}

public static string Replace(string source, char matchPattern,
                             string replaceStr, int count, int startPos)
```

```

    {
        return (Replace(source.ToString(), matchPattern.ToString(), replaceStr,
            count, startPos));
    }

    public static string Replace(string source, string matchPattern,
        string replaceStr)
    {
        return (Replace(source, matchPattern, replaceStr, -1, 0));
    }

    public static string Replace(string source, string matchPattern,
        string replaceStr, int count)
    {
        return (Replace(source, matchPattern, replaceStr, count, 0));
    }

    public static string Replace(string source, string matchPattern,
        string replaceStr, int count, int startPos)
    {
        Regex RE = new Regex(matchPattern);
        string newString = RE.Replace(source, replaceStr, count, startPos);

        return (newString);
    }

```

To use the overloaded Replace methods to replace the word FOO with the word BAR in a sentence, you could write the following:

```

public static void TestReplace()
{
    string source = "Replace the FOO in this text block of text FOO.";
    string matchPattern = "FOO";
    string replaceStr = "BAR";

    Console.WriteLine(Replace(source, matchPattern, replaceStr));
    Console.WriteLine(Replace(source, matchPattern, replaceStr, -1));
    Console.WriteLine(Replace(source, matchPattern, replaceStr, -1, 0));
    Console.WriteLine(Replace(source, matchPattern, replaceStr, 1));
    Console.WriteLine(Replace(source, matchPattern, replaceStr, 1, 0));
    Console.WriteLine(Replace(source, matchPattern, replaceStr, 1));
    Console.WriteLine(Replace(source, matchPattern, replaceStr, 1, 20));

    Console.WriteLine(Replace(source, matchPattern, replaceStr, -1, 0));
    Console.WriteLine(Replace(source, matchPattern, replaceStr, 1, 0));
    Console.WriteLine(Replace(source, matchPattern, replaceStr, 1, 20));
}

```

which would produce the following output:

```

Replace the BAR in this text block of text BAR.
Replace the BAR in this text block of text BAR.
Replace the BAR in this text block of text BAR.
Replace the BAR in this text block of text FOO.
Replace the BAR in this text block of text FOO.
Replace the BAR in this text block of text FOO.

```

Replace the FOO in this text block of text BAR.
Replace the BAR in this text block of text BAR.
Replace the BAR in this text block of text FOO.
Replace the FOO in this text block of text BAR.

This code looks for the word "FOO", and each time this pattern is found, the string "BAR" is substituted for the matched string ("FOO").

Discussion

Using the overloaded instance `Replace` method on the `Regex` class, we can easily define a replacement string that is substituted for a regular expression pattern each time that pattern is found. Several overloads of this method provide even more flexibility in determining where to replace matches and how many matches will be replaced.

An overloaded static `Replace` method is also provided on the `Regex` class. This method is somewhat different than its instance method counterpart. This static `Replace` method does not allow for the flexibility of a *startPos* or a *count* parameter. In lieu of these parameters, an *options* parameter is used. This parameter allows for modification of the *RegexOptions* options. If you require that the regular expression options (*RegexOptions*) be controllable, rather than using the less flexible static `Regex.Replace` method, you can modify the overloaded `Replace` methods as follows:

```
// Constant to provide a default set of options for the regular expression
const RegexOptions defaultOptions = RegexOptions.IgnorePatternWhitespace |
    RegexOptions.Multiline;

public static string Replace(string source, char matchPattern,
    string replaceStr)
{
    return (Replace(source, matchPattern.ToString(), replaceStr, -1, 0,
        defaultOptions));
}

public static string Replace(string source, char matchPattern,
    string replaceStr, int count)
{
    return (Replace(source.ToString(), matchPattern.ToString(), replaceStr,
        count, 0, defaultOptions));
}

public static string Replace(string source, char matchPattern,
    string replaceStr, int count, int startPos)
{
    return (Replace(source.ToString(), matchPattern.ToString(), replaceStr,
        count, startPos, defaultOptions));
}

public static string Replace(string source, char matchPattern,
    string replaceStr, int count, int startPos,
    RegexOptions options)
{
    return (Replace(source.ToString(), matchPattern.ToString(), replaceStr,
```

```

        count, startPos, options));
    }

    public static string Replace(string source, string matchPattern,
                                string replaceStr)
    {
        return (Replace(source, matchPattern, replaceStr, -1, 0,
                        defaultOptions));
    }

    public static string Replace(string source, string matchPattern,
                                string replaceStr, int count)
    {
        return (Replace(source, matchPattern, replaceStr, count, 0,
                        defaultOptions));
    }

    public static string Replace(string source, string matchPattern,
                                string replaceStr, int count, int startPos)
    {
        return (Replace(source, matchPattern, replaceStr, count, startPos,
                        defaultOptions));
    }

    public static string Replace(string source, string matchPattern,
                                string replaceStr, int count, int startPos,
                                RegexOptions options)
    {
        Regex RE = new Regex(matchPattern, options);
        string newString = RE.Replace(source, replaceStr, count, startPos);

        return (newString);
    }
}

```

An *options* parameter of type `RegexOptions` has been added to the end of each method's parameter list. The last `Replace` method uses this *options* parameter to define how the `Regex` object will use the regular expression. Note also that a constant `defaultOptions` of type `RegexOptions` has been defined to provide a uniform way to represent the default set of options in each overloaded method.

See Also

See the “.NET Framework Regular Expressions” topic in the MSDN documentation.

8.6 Augmenting the Basic String Replacement Function

Problem

You need to replace character patterns within the target string with a new string. However, in this case, each replacement operation has a unique set of conditions

that must be satisfied in order to allow the replacement to occur. Consider, for example, that you receive a string in the form of XML (or possibly HTML). You wish to modify an attribute of a specific XML tag to a particular number, but only if that number is within a specified range (or possibly outside of a particular range).

Solution

Use the overloaded instance Replace method that accepts a MatchEvaluator delegate along with its other parameters. The MatchEvaluator delegate, which is a callback method that overrides the default behavior of the Replace method, is shown here:

```
using System;
using System.Text.RegularExpressions;

public static string MatchHandler(Match theMatch)
{
    // Handle Top property of the Property tag
    if (theMatch.Value.StartsWith("<Property"))
    {
        long topPropertyValue = 0;

        // Obtain the numeric value of the Top property
        Match topPropertyMatch = Regex.Match(theMatch.Value,
                                             "Top=\\([-]*\\d*");
        if (topPropertyMatch.Success)
        {
            if (topPropertyMatch.Groups[1].Value.Trim().Equals(""))
            {
                // If blank, set to zero
                return (theMatch.Value.Replace("Top=\\\"", "Top=\\\"0\\\""));
            }
            else if (topPropertyMatch.Groups[1].Value.Trim().Equals("-"))
            {
                // If only a negative sign (syntax error), set to zero
                return (theMatch.Value.Replace("Top=\\\"-\\\"", "Top=\\\"0\\\""));
            }
            else
            {
                // We have a valid number
                // Convert the matched string to a numeric value
                topPropertyValue = long.Parse(
                    topPropertyMatch.Groups[1].Value,
                    System.Globalization.NumberStyles.Any);

                // If the Top property is out of the specified
                // range, set it to zero
                if (topPropertyValue < 0 || topPropertyValue > 5000)
                {
                    return (theMatch.Value.Replace("Top=\\\" +
                                                    topPropertyValue +
                                                    \"\\\"", "Top=\\\"0\\\""));
                }
            }
        }
    }
}
```

```

    }
}

return (theMatch.Value);
}

```

The callback method for the Replace method is shown here:

```

public static void ComplexReplace(string matchPattern, string source)
{
    MatchEvaluator replaceCallback = new MatchEvaluator(MatchHandler);
    Regex RE = new Regex(matchPattern, RegexOptions.Multiline);
    string newString = RE.Replace(source, replaceCallback);

    Console.WriteLine("Replaced String = " + newString);
}

```

To use this callback method with the static Replace method, modify the previous ComplexReplace method as follows:

```

public void ComplexReplace(string matchPattern, string source)
{
    MatchEvaluator replaceCallback = new MatchEvaluator(MatchHandler);
    string newString = Regex.Replace(source, matchPattern,
    replaceCallback);

    Console.WriteLine("Replaced String = " + newString);
}

```

where *source* is the original string to run the replace operation against, and *matchPattern* is the regular expression pattern to match in the *source* string.

If the ComplexReplace method is called from the following code:

```

public static void TestComplexReplace()
{
    string matchPattern = "<.*>";
    string source = @"<?xml version=""1.0"" encoding=""UTF-8"">
<Window ID=""Main">
    <Control ID=""TextBox">
        <Property Top=""-100"" Left=""0"" Text=""BLANK""/>
    </Control>
    <Control ID=""Label">
        <Property Top=""99990"" Left=""0"" Caption=""Enter Name Here""/>
    </Control>
</Window>";

    ComplexReplace(matchPattern, source);
}

```

only the Top attributes of the Property tags are changed from their original values to 0.

The result of this replace action will change only the Top property value of a Property tag to zero if it is less than zero or greater than 5000. Any other tag that contains a Top property will remain unchanged. The following two lines of the source string will be changed from:

```
<Property Top="100" Left="0" Text="BLANK"/>
<Property Top="99999" Left="0" Caption="Enter Name Here"/>
```

to:

```
<Property Top="100" Left="0" Text="BLANK"/>
<Property Top="0" Left="0" Caption="Enter Name Here"/>
```

Discussion

The `MatchEvaluator` delegate, which is automatically invoked when it is supplied as a parameter to the `Regex` class's `Replace` method, allows for custom replacement of each string that conforms to the regular expression pattern.

If the current `Match` object is operating on a `Property` tag whose `Top` property is out of the specified range, the code within the `MatchHandler` callback method returns a new modified string. Otherwise, the currently matched string is returned unchanged. This ability allows you to override the default `Replace` functionality by replacing only that part of the source string that meets certain criteria. The code within this callback method gives you some idea of what can be accomplished using this replacement technique.

To make use of this callback method, we need a way to call it from the `ComplexReplace` method. First, a variable of type `System.Text.RegularExpressions.MatchEvaluator` is created. This variable (`replaceCallback`) is the delegate that is used to call the `MatchHandler` method:

```
MatchEvaluator replaceCallback = new MatchEvaluator(MatchHandler);
```

Finally, the `Replace` method is called with the reference to the `MatchEvaluator` delegate passed in as a parameter:

```
string newString = RE.Replace(source, replaceCallback);
```

See Also

See the “.NET Framework Regular Expressions” topic in the MSDN documentation.

8.7 A Better Tokenizer

Problem

A simple method of *tokenizing*—or breaking up a string into its discrete elements—was presented in Recipe 2.6. However, this is not powerful enough to handle all your string-tokenizing needs. You need a tokenizer—also referred to as a *lexer*—that can split up a string based on a well-defined set of characters.

Solution

Using the `Split` method of the `Regex` class, we can use a regular expression to indicate the types of tokens and separators that we are interested in gathering. This technique

works especially well with equations, since the tokens of an equation are well-defined. For example, the code:

```
using System;
using System.Text.RegularExpressions;

public static string[] Tokenize(string equation)
{
    Regex RE = new Regex(@"([\+\-\*\(\)\^\\]");
    return (RE.Split(equation));
}
```

will divide up a string according to the regular expression specified in the `Regex` constructor. In other words, the string passed in to the `Tokenize` method will be divided up based on the delimiters `+`, `-`, `*`, `(`, `)`, `^`, or `\`. The following method will call the `Tokenize` method to tokenize the equation: $(y - 3)(3111x^{21} + x + 320)$:

```
public void TestTokenize()
{
    foreach(string token in Tokenize("(y - 3)(3111*x^21 + x + 320)"))
        Console.WriteLine("String token = " + token.Trim());
}
```

which displays the following output:

```
String token =
String token = (
String token = y
String token = -
String token = 3
String token = )
String token =
String token = (
String token = 3111
String token = *
String token = x
String token = ^
String token = 21
String token = +
String token = x
String token = +
String token = 320
String token = )
String token =
```

Notice that each individual operator, parenthesis, and number has been broken out into its own separate token.

Discussion

The tokenizer created in Recipe 2.6 would be useful in specific controlled circumstances. However, in real-world projects, we do not always have the luxury of being able to control the set of inputs to our code. By making use of regular expressions, we can take the original tokenizer and make it flexible enough to allow it to be applied to any type or style of input we desire.

The key method used here is the `Split` instance method of the `Regex` class. The return value of this method is a string array whose elements include each individual token of the source string—the equation, in this case.

Notice that the static method allows `RegexOptions` enumeration values to be used, while the instance method allows for a starting position to be defined and a maximum amount of matches to occur. This may have some bearing on whether you choose the static or instance method.

See Also

See Recipe 2.6; see the “.NET Framework Regular Expressions” topic in the MSDN documentation.

8.8 Compiling Regular Expressions

Problem

You have a handful of regular expressions to execute as quickly as possible over many different strings. Performance is of the utmost importance.

Solution

The best way to do this task is to use compiled regular expressions. However, there are some drawbacks to using this technique, which we will examine.

There are two ways to compile regular expressions. The easiest way is to use the `RegexOptions.Compiled` enumeration value in the `Options` parameter of the static `Match` or `Matches` methods on the `Regex` class:

```
Match theMatch = Regex.Match(source, pattern, RegexOptions.Compiled);
```

```
MatchCollection theMatches = Regex.Matches(source, pattern, RegexOptions.Compiled);
```

If more than a few expressions will be compiled and/or the expressions need to be shared across applications, consider precompiling all of these expressions into their own assembly. Do this by using the static `CompileToAssembly` method on the `Regex` class. The following method accepts an assembly name and compiles two simple regular expressions into this assembly:

```
public static void CreateRegExDLL(string assmName)
{
    RegexCompilationInfo[] RE = new RegexCompilationInfo[2]
    {
        new RegexCompilationInfo("PATTERN", RegexOptions.Compiled,
            "CompiledPATTERN", "Chapter_Code", true),
        new RegexCompilationInfo("NAME", RegexOptions.Compiled,
            "CompiledNAME", "Chapter_Code", true)};

    System.Reflection.AssemblyName aName =
        new System.Reflection.AssemblyName();
    aName.Name = assmName;
}
```

```
    Regex.CompileToAssembly(RE, aName);  
}
```

Now that the expressions are compiled to an assembly, the assembly can be added as a reference to your project and used as follows:

```
Chapter_Code.CompiledNAME CN = new Chapter_Code.CompiledNAME();  
Match mName = CN.Match("Get the NAME from this text.");  
Console.WriteLine("mName.Value = " + mName.Value);
```

This code displays the following text:

```
mName.Value = NAME
```

Discussion

Compiling regular expressions allows the expression to run faster. To understand how, we need to examine the process that an expression goes through as it is run against a string. If an expression is not compiled, the regular expression engine converts the expression to a series of internal codes that are recognized by the regular expression engine; it is not converted to MSIL. As the expression runs against a string, the engine interprets the series of internal codes. This can be a slow process, especially as the source string becomes very large and the expression becomes much more complex.

To fix this performance problem, you can compile the expression so that it gets converted directly to a series of MSIL instructions, which perform the pattern matching for the specific regular expression. Once the Just-In-Time (JIT) compiler is run on this MSIL, the instructions are converted to machine code. This allows for an extremely fast execution of the pattern against a string.

There are two drawbacks to using the `RegexOptions.Compiled` enumerated value to compile regular expressions. The first is that the first time an expression is used with the `Compiled` flag, it performs very slowly, due to the compilation process. Fortunately, this is a one-time expense since every unique expression is compiled only once. The second drawback is that an in-memory assembly gets generated to contain the IL, which can never be unloaded. An assembly can never be unloaded from an `AppDomain`. The garbage collector cannot remove it from memory. If large numbers of expressions are compiled, the amount of heap resources that will be used up and not released will be larger. So use this technique wisely.

Compiling regular expressions into their own assembly immediately gives you two benefits. First, precompiled expressions do not require any extra time to be compiled while your application is running. Second, they are in their own assembly and therefore can be used by other applications.



Consider precompiling regular expressions and placing them in their own assembly rather than using the `RegexOptions.Compiled` flag.

To compile one or more expressions into an assembly, the static `CompileToAssembly` method of the `Regex` class must be used. To use this method, a `RegexCompilationInfo` array must be created and filled with `RegexCompilationInfo` objects. The next step is to create the assembly in which the expression will live. An instance of the `AssemblyName` class is created using the default constructor. Next, this assembly is given a name (do not include the `.dll` file extension in the name, it is added automatically). Finally, the `CompileToAssembly` method can be called with the `RegexCompilationInfo` array and the `AssemblyName` object supplied as arguments.



In our example, this assembly is placed in the same directory that the executable was launched from.

See Also

See the “.NET Framework Regular Expressions” and “AssemblyName Class” topics in the MSDN documentation.

8.9 Counting Lines of Text

Problem

You need to count lines of text within a string or within a file.

Solution

Read in the entire file and count the number of linefeeds, as shown in the following method:

```
using System;
using System.Text.RegularExpressions;
using System.IO;

public static long LineCount(string source, bool isFileName)
{
    if (source != null)
    {
        string text = source;

        if (isFileName)
        {
            FileStream FS = new FileStream(source, FileMode.Open,
                FileAccess.Read, FileShare.Read);
            StreamReader SR = new StreamReader(FS);
            text = SR.ReadToEnd();
            SR.Close();
            FS.Close();
        }
    }
}
```

```

Regex RE = new Regex("\n", RegexOptions.Multiline);
MatchCollection theMatches = RE.Matches(text);

// Needed for files with zero length
// Note that a string will always have a line terminator
// and thus will always have a length of 1 or more
if (isFileName)
{
    return (theMatches.Count);
}
else
{
    return (theMatches.Count) + 1;
}
}
else
{
    // Handle a null source here
    return (0);
}
}

```

An alternative version of this method uses the `StreamReader.ReadLine` method to count lines in a file and a regular expression to count lines in a string:

```

public static long LineCount2(string source, bool isFileName)
{
    if (source != null)
    {
        string text = source;
        long numOfLines = 0;

        if (isFileName)
        {
            FileStream FS = new FileStream(source, FileMode.Open,
                FileAccess.Read, FileShare.Read);
            StreamReader SR = new StreamReader(FS);

            while (text != null)
            {
                text = SR.ReadLine();

                if (text != null)
                {
                    ++numOfLines;
                }
            }

            SR.Close();
            FS.Close();
            return (numOfLines);
        }
        else
        {

```

```

        Regex RE = new Regex("\n", RegexOptions.Multiline);
        MatchCollection theMatches = RE.Matches(text);

        return (theMatches.Count + 1);
    }
}
else
{
    // Handle a null source here
    return (0);
}
}

```

The following method counts the lines within a specified text file and a specified string:

```

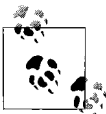
public static void TestLineCount()
{
    // Count the lines within the file TestFile.txt
    LineCount(@"C:\TestFile.txt", true);

    // Count the lines within a string
    // Notice that a \r\n characters start a new line
    // as well as just the \n character
    LineCount("Line1\r\nLine2\r\nLine3\nLine4", false);
}

```

Discussion

Every line ends with a special character. For Windows files, the line terminating characters are a carriage return followed by a linefeed. This sequence of characters is described by the regular expression pattern `\r\n`. Unix files terminate their lines with just the linefeed character (`\n`). The regular expression `"\n"` is the lowest common denominator for both sets of line-terminating characters. Consequently, this method runs a regular expression that looks for the pattern `"\n"` in a string or file.



Macintosh files usually end with a carriage-return character (`\r`). To count the number of lines in this type of file, the regular expression should be changed to the following in the constructor of the `Regex` object:

```
Regex RE = new Regex("\r", RegexOptions.Multiline);
```

Simply running this regular expression against a string returns the number of lines minus one because the last line does not have a line-terminating character. To account for this, one is added to the final count of linefeeds in the string.

The `LineCount` method accepts two parameters. The first is a string that either contains the actual text that will have its lines counted or the path and name of a text file whose lines are to be counted. The second parameter, `isFileName`, determines whether the first parameter (source) is a string or a file path. If this parameter is true, the source parameter is a file path; otherwise, it is simply a string.

See Also

See the “.NET Framework Regular Expressions,” “FileStream Class,” and “StreamReader Class” topics in the MSDN documentation.

8.10 Returning the Entire Line in Which a Match Is Found

Problem

You have a string or file that contains multiple lines. When a specific character pattern is found on a line, you want to return the entire line, not just the matched text.

Solution

Use the `StreamReader.ReadLine` method to obtain each line in a file in which to run a regular expression against:

```
public static ArrayList GetLines(string source, string pattern,
                                bool isFileName)
{
    string text = source;
    ArrayList matchedLines = new ArrayList();

    // If this is a file, get the entire file's text
    if (isFileName)
    {
        FileStream FS = new FileStream(source, FileMode.Open,
                                      FileAccess.Read, FileShare.Read);
        StreamReader SR = new StreamReader(FS);

        while (text != null)
        {
            text = SR.ReadLine();

            if (text != null)
            {
                // Run the regex on each line in the string
                Regex RE = new Regex(pattern, RegexOptions.Multiline);
                MatchCollection theMatches = RE.Matches(text);

                if (theMatches.Count > 0)
                {
                    // Get the line if a match was found
                    matchedLines.Add(text);
                }
            }
        }
    }
}
```

```

        SR.Close();
        FS.Close();
    }
    else
    {
        // Run the regex once on the entire string
        Regex RE = new Regex(pattern, RegexOptions.Multiline);
        MatchCollection theMatches = RE.Matches(text);

        // Get the line for each match
        foreach (Match m in theMatches)
        {
            int lineStartPos = GetBeginningOfLine(text, m.Index);
            int lineEndPos = GetEndOfLine(text, (m.Index + m.Length - 1));
            string line = text.Substring(lineStartPos,
                                        lineEndPos - lineStartPos);
            matchedLines.Add(line);
        }
    }

    return (matchedLines);
}

public static int GetBeginningOfLine(string text, int startPointOfMatch)
{
    if (startPointOfMatch > 0)
    {
        --startPointOfMatch;
    }

    if (startPointOfMatch >= 0 && startPointOfMatch < text.Length)
    {
        // Move to the left until the first '\n' char is found
        for (int index = startPointOfMatch; index >= 0; index--)
        {
            if (text[index] == '\n')
            {
                return (index + 1);
            }
        }

        return (0);
    }

    return (startPointOfMatch);
}

public static int GetEndOfLine(string text, int endPointOfMatch)
{
    if (endPointOfMatch >= 0 && endPointOfMatch < text.Length)
    {
        // Move to the right until the first '\n' char is found
        for (int index = endPointOfMatch; index < text.Length; index++)
        {

```

```

        if (text[index] == '\n')
        {
            return (index);
        }
    }

    return (text.Length);
}

return (endPointOfMatch);
}

```

The following method shows how to call the `GetLines` method with either a filename or a string:

```

public static void TestGetLine()
{
    // Get each line within the file TestFile.txt as a separate string
    Console.WriteLine();
    ArrayList lines = GetLines(@"C:\TestFile.txt", "\n", true);
    foreach (string s in lines)
        Console.WriteLine("MatchedLine: " + s);

    // Get the lines matching the text "Line" within the given string
    Console.WriteLine();
    lines = GetLines("Line1\r\nLine2\r\nLine3\nLine4", "Line", false);
    foreach (string s in lines)
        Console.WriteLine("MatchedLine: " + s);
}

```

Discussion

The `GetLines` method accepts three parameters:

source

The string or filename in which to search for a pattern.

pattern

The regular expression pattern to apply to the *source* string.

isFileName

Pass in `true` if the *source* is a filename or `false` if *source* is a string.

This method returns an `ArrayList` of strings that contains each line in which the regular expression match was found.

The `GetLines` method can obtain the lines on which matches occur, within a string or a file. When running a regular expression against a file whose name is passed in to the *source* parameter (when *isFileName* equals `true`) in the `GetLines` method, the file is opened and read line-by-line. The regular expression is run against each line and if a match is found, that line is stored in the `matchedLines` `ArrayList`. Using the `ReadLine` method of the `StreamReader` object saves us from having to determine where

each line starts and ends. Determining where a line starts and ends in a string requires some work, as you shall see.

Running the regular expression against a string passed in to the *source* parameter (when *isFileName* equals *false*) in the `GetLines` method produces a `MatchCollection`. Each `Match` object in this collection is used to obtain the line on which it is located in the *source* string. The line is obtained by starting at the position of the first character of the match in the *source* string and moving one character to the left until either a `'\n'` character is found or the beginning of the *source* string is found (this code is found in the `GetBeginningOfLine` method). This gives you the beginning of the line, which is placed in the variable `LineStartPos`. Next, the end of the line is found by starting at the last character of the match in the *source* string and moving to the right until either a `'\n'` character is found or the end of the *source* string is found (this code is found in the `GetEndOfLine` method). This ending position is placed in the `LineEndPos` variable. All of the text between the `LineStartPos` and `LineEndPos` will be the line in which the match is found. Each of these lines is added to the `matchedLines` `ArrayList` and returned to the caller.

Something interesting you can do with the `GetLines` method is to pass in the string `"\n"` in the pattern parameter of this method. This trick will effectively return each line of the string or file as a string in the `ArrayList`.

Note that if more than one match is found on a line, each matching line will be added to the `ArrayList`.

See Also

See the “.NET Framework Regular Expressions,” “`FileStream` Class,” and “`StreamReader` Class” topics in the MSDN documentation.

8.11 Finding a Particular Occurrence of a Match

Problem

You need to find a specific occurrence of a match within a string. For example, you want to find the third occurrence of a word or the second occurrence of a Social Security Number. In addition, you may need to find every third occurrence of a word in a string.

Solution

To find a particular occurrence of a match in a string, simply subscript the array returned from `Regex.Matches`:

```
public static Match FindOccurrenceOf(string source, string pattern,
                                     int occurrence)
```

```

{
    if (occurrence < 1)
    {
        throw (new ArgumentException("Cannot be less than 1",
            "occurrence"));
    }

    // Make occurrence zero-based
    --occurrence;

    // Run the regex once on the source string
    Regex RE = new Regex(pattern, RegexOptions.Multiline);
    MatchCollection theMatches = RE.Matches(source);

    if (occurrence >= theMatches.Count)
    {
        return (null);
    }
    else
    {
        return (theMatches[occurrence]);
    }
}

```

To find each particular occurrence of a match in a string, build an `ArrayList` on the fly:

```

public static ArrayList FindEachOccurrenceOf(string source, string pattern,
                                             int occurrence)
{
    ArrayList occurrences = new ArrayList();

    // Run the regex once on the source string
    Regex RE = new Regex(pattern, RegexOptions.Multiline);
    MatchCollection theMatches = RE.Matches(source);

    for (int index = (occurrence - 1);
         index < theMatches.Count; index += occurrence)
    {
        occurrences.Add(theMatches[index]);
    }

    return (occurrences);
}

```

The following method shows how to invoke the two previous methods:

```

public static void TestOccurrencesOf()
{
    Match matchResult = FindOccurrenceOf(
        "one two three one two three one two three one"
        + " two three one two three one two three", "two", 2);
    if (matchResult != null)
        Console.WriteLine(matchResult.ToString() + "\t" +
            matchResult.Index);
}

```

```
Console.WriteLine();
ArrayList results = FindEachOccurrenceOf(
    "one one two three one two three one two" +
    " three one two three", "one", 2);
foreach (Match m in results)
    Console.WriteLine(m.ToString() + "\t" + m.Index);
}
```

Discussion

This recipe contains two similar but distinct methods. The first method, `FindOccurrenceOf`, returns a particular occurrence of a regular expression match. The occurrence you want to find is passed in to this method via the occurrence parameter. If the particular occurrence of the match does not exist—for example, you ask to find the second occurrence, but only one occurrence exists—a null is returned from this method. Because of this, you should check that the returned object of this method is not null before using that object. If the particular occurrence exists, the `Match` object that holds the match information for that occurrence is returned.

The second method in this recipe, `FindEachOccurrenceOf`, works similar to the `FindOccurrenceOf` method, except that it continues to find a particular occurrence of a regular expression match until the end of the string is reached. For example, if you ask to find the second occurrence, this method would return an `ArrayList` of zero or more `Match` objects. The `Match` objects would correspond to the second, fourth, sixth, and eighth occurrence of a match and so on until the end of the string is reached.

See Also

See the “.NET Framework Regular Expressions” and “ArrayList Class” topics in the MSDN documentation.

8.12 Using Common Patterns

Problem

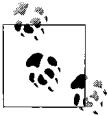
You need a quick list from which to choose regular expression patterns that match standard items. These standard items could be a Social Security Number, a zip code, a word containing only characters, an alphanumeric word, an email address, a URL, dates, or one of many other possible items used throughout business applications.

These patterns can be useful in making sure that a user has input the correct data and that it is well-formed. These patterns can also be used as an extra security measure to keep hackers from attempting to break your code by entering strange or malformed input (e.g., SQL injection or cross-site-scripting attacks). Note that these regular expressions are not a silver bullet that will stop all attacks on your system; rather, they are an added layer of defense.

Solution

- Match only alphanumeric characters along with the characters -, +, ., and any whitespace:

```
^([\w\.\+-]|\s)*$
```



Be careful using the - character within a character class—a regular expression enclosed within [and]. That character is also used to specify a range of characters, as in a-z for a through z inclusive. If you want to use a literal - character, either escape it with \ or put it at the end of the expression, as shown in the previous and next examples.

- Match only alphanumeric characters along with the characters -, +, ., and any whitespace, with the stipulation that there is at least one of these characters and no more than 10 of these characters:

```
^([\w\.\+-]|\s){1,10}$
```

- Match a date in the form ##/##/#### where the day and month can be a one- or two-digit value, and year can either be a two- or four-digit value:

```
^\d{1,2}\d{1,2}\d{2,4}$
```

- Match a time to be entered with an optional am or pm extension (note that this regular expression also handles military time):

```
^\d{1,2}:\d{2}\s?([ap]m)?$
```

- Match an IP address:

```
^[0-2]?[0-5]?[0-5]\.){3}[0-2]?[0-5]?[0-5]$
```

- Verify that an email address is in the form *name@address* where *address* is not an IP address:

```
^[A-Za-z0-9_-\.\.]+@(((A-Za-z0-9\-\.))+([A-Za-z\-\.]))+$
```

- Verify that an email address is in the form *name@address* where *address* is an IP address:

```
^[A-Za-z0-9_-\.\.]+@[0-2]?[0-5]?[0-5]\.){3}[0-2]?[0-5]?[0-5]$
```

- Match only a dollar amount with the optional \$ and + or - preceding characters (note that any number of decimal places may be added):

```
^\$?[+-]?[\d,]*\.\d{0,2}$
```

This is similar to the previous regular expression except that only up to two decimal places are allowed:

```
^\$?[+-]?[\d,]*\.\d{0,2}$
```

- Match a credit card number to be entered as four sets of four digits separated with a space, -, or no character at all:

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

- Match a zip code to be entered either as five digits with an optional four-digit extension:

```
^\d{5}(-\d{4})?$
```

- Match a North American phone number with an optional area code and an optional - character to be used in the phone number and no extension:

```
^\(\?[0-9]{3}\)\)?\-\?[0-9]{3}\-\?[0-9]{4}$
```

- Match a phone number similar to the previous regular expression, but allow an optional five-digit extension prefixed with either ext or extension:

```
^\(\?[0-9]{3}\)\)?\-\?[0-9]{3}\-\?[0-9]{4}(\s*ext(ension)?[0-9]{5})?$
```

- Match a full path beginning with the drive letter and optionally match a file-name with a three-character extension (note that no .. characters signifying to move up the directory hierarchy are allowed, nor is a directory name with a . followed by an extension):

```
^[a-zA-Z]:[\\\/]([_a-zA-Z0-9]+[\\\/]?)*([_a-zA-Z0-9]+\.[_a-zA-Z0-9]{0,3})?$
```

Discussion

Regular expressions are effective at finding specific information, and they have a wide range of uses. Many applications use them to locate specific information within a larger range of text, as well as to filter out bad input. The filtering action is very useful in tightening the security of an application and preventing an attacker from attempting to use carefully formed input to gain access to a machine on the Internet or a local network. By using a regular expression to allow only good input to be passed to the application, you can reduce the likelihood of many types of attacks, such as SQL injection or cross-site-scripting.

The regular expressions presented in this recipe only provide a minute cross-section of what can be accomplished with them. By taking these expressions and manipulating parts of them, you can easily modify them to work with your application. Take, for example, the following expression which allows only between 1 and 10 alphanumeric characters, along with a few symbols to be allowed as input:

```
^([\w\.\+-]|\s){1,10}$
```

By changing the {1,10} part of the regular expression to {0,200}, this expression will now match a blank entry or an entry of the specified symbols up to and including 200 characters.

Note the use of the ^ character at the beginning of the expression and the \$ character at the end of the expression. These characters start the match at the beginning of the text and match all the way to the end of the text. Adding these characters forces the regular expression to match the entire string or none of it. By removing these characters, you can search for specific text within a larger block of text. For example, the following regular expression matches only a string containing nothing but a U.S. zip code (there can be no leading or trailing spaces):

```
^\d{5}(-\d{4})?$
```

This version matches only a zip code with leading or trailing spaces (notice the addition of the \s* to the start and end of the expression):

```
^\s*\d{5}(-\d{4})?\s*$
```

However, this modified expression matches a zip code found anywhere within a string (including a string containing just a zip code):

```
\d{5}(-\d{4})?
```

Use the regular expressions in this recipe and modify them to suit your needs.

See Also

Two good books that cover regular expressions are *Regular Expression Pocket Reference* by Tony Stubblebine (O'Reilly) and *Mastering Regular Expressions*, Second Edition, by Jeffrey Friedl (O'Reilly).

8.13 Documenting Your Regular Expressions

Problem

You have one or more complex regular expressions that may exist in a file outside of your code. You need a way to place comments within the regular expression itself. These comments will aid others in being able to read and maintain your regular expressions later on.

Solution

Add comments to the regular expression using the # comment character:

```
string matchPattern = @"\\\\" # Find this: \\
                    (?<TheServer>\w*) # Server name
                    \\ # Find this: \
                    (?<TheService>\w*)\\ # Service name";
```

or add C#-style comments outside of the regular expression string:

```
string matchPattern = @"\\\\" + // Find this: \\
                    @"(?<TheServer>\w*)" + // Server name
                    @"\\" + // Find this: \
                    @"(?<TheService>\w*)\\"; // Service name
```

When using these expressions in a `Regex` object, the `RegexOptions.IgnorePatternWhitespace` enumeration value must be added to the options parameter of the `Regex` object constructor:

```
Regex RE = new Regex(matchPattern,
                    RegexOptions.Multiline | RegexOptions.IgnorePatternWhitespace);
MatchCollection theMatches = RE.Matches("The source text goes here...");
```

Discussion

With large and complex regular expressions, it is desirable to break up the expression into manageable pieces and to identify what each piece does. For example, the regular expression in the Solution section will pull the server and service pieces out

of a UNC string. By breaking up the regular expression onto separate lines and adding comments to each line, we have allowed other developers (who might not be familiar with regular expressions) to more quickly and easily read and maintain our regular expression.

Typically, you would use the string concatenation and C#-style commenting to comment a regular expression string. However, if you are retrieving the regular expression from an external source, such as a text file, regular expression style commenting (#) is the type to use.

With simpler regular expressions, you can get away with adding a C# comment outside of the regular expression string to indicate what it does. But adding comments to the regular expression itself greatly aids in understanding it.