CHAPTER 3

Strings and Regular Expressions

IN THIS CHAPTER

- Working with Strings
- Working with Regular Expressions

Regardless of what type of data you're working with or what kind of application you're creating, you will undoubtedly need to work with strings. No matter how the data is stored, the end user always deals in human-readable text. As such, knowing how to work with strings is part of the essential knowledge that any .NET developer needs to make rich and compelling applications.

In addition to showing you how to work with strings in the .NET Framework, this chapter will also introduce you to regular expressions. Regular expressions are format codes that not only allow you to verify that a particular string matches a given format, but you can also use regular expressions to extract meaningful information from what otherwise might be considered free-form text, such as extracting the first name from user input, or the area code from a phone number input, or the server name from a URL.

Working with Strings

Being able to work with strings is an essential skill in creating high-quality applications. Even if you are working with numeric or image data, end users need textual feedback. This section of the chapter will introduce you to .NET strings, how to format them, manipulate them and compare them, as well as other useful operations.

Introduction to the .NET String

Before the .NET Framework and the Common Language Runtime (CLR), developers used to have to spend considerable amount of effort working with strings. A reusable

library of string routines was a part of virtually every C and C++ programmer's toolbox. It was also difficult to write code that exchanged string data between different programming languages. For example, Pascal stores strings as an in-memory character array, where the first element of the array indicated the length of the string. C stores strings as an in-memory array of characters with a variable length. The end of the string was indicated by the ASCII null character (represented in C as `\0`).

In the .NET Framework, strings are stored as immutable values. This means that when you create a string in C# (or any other .NET language), that string is stored in memory in a fixed size to make certain aspects of the CLR run faster (you will learn more about this in Chapter 16, "Optimizing Your NET 2.0 Code"). As a result, when you do things such as concatenate strings or modify individual characters in a string, the CLR is actually creating multiple copies of your string.

Strings in C# are declared in the same way as other value types such as integer or float, as shown in the following examples:

```
string x = "Hello World";
string y;
string z = x;
```

Formatting Strings

One of the most common tasks when working with strings is formatting them. When displaying information to users, you often display things like dates, times, numeric values, decimal values, monetary values, or even things like hexadecimal numbers. C# strings all have the ability to display these types of information and much more. Another powerful feature is that when you use the standard formatting tools, the output of the formatting will be localization-aware. For example, if you display the current date in short form to a user in England, the current date in short form will appear different to a user in the United States.

To create a formatted string, all you have to do is invoke the `Format` method of the `string` class and pass it a format string, as shown in the following code:

```
string formatted = string.Format("The value is {0}", value);
```

The `{0}` placeholder indicates where a value should be inserted. In addition to specifying where a value should be inserted, you can also specify the format for the value.

Other data types also support being converted into strings via custom format specifiers, such as the `DateTime` data type, which can produce a custom-formatted output using

```
DateTime.ToString("format specifiers");
```

Table 3.1 illustrates some of the most commonly used format strings for formatting dates, times, numeric values, and more.

TABLE 3.1 Custom `DateTime` Format Specifiers

Specifier	Description
`d`	Displays the current day of the month.
`dd`	Displays the current day of the month, where values < 10 have a leading zero.
`ddd`	Displays the three-letter abbreviation of the name of the day of the week.
`dddd(+)`	Displays the full name of the day of the week represented by the given `DateTime` value.
`f(+)`	Displays the x most significant digits of the seconds value. The more f's in the format specifier, the more significant digits. This is *total* seconds, not the number of seconds passed since the last minute.
`F(+)`	Same as f(+), except trailing zeros are not displayed.
`g`	Displays the era for a given `DateTime` (for example, "A.D.")
`h`	Displays the hour, in range 1–12.
`hh`	Displays the hour, in range 1–12, where values < 10 have a leading zero.
`H`	Displays the hour in range 0–23.
`HH`	Displays the hour in range 0–23, where values < 10 have a leading zero.
`m`	Displays the minute, range 0–59.
`mm`	Displays the minute, range 0–59, where values < 10 have a leading zero.
`M`	Displays the month as a value ranging from 1–12.
`MM`	Displays the month as a value ranging from 1–12 where values < 10 have a leading zero.
`MMM`	Displays the three-character abbreviated name of the month.
`MMMM`	Displays the full name of the month.
`s`	Displays the number of seconds in range 0–59.
`ss(+)`	Displays the number of seconds in range 0–59, where values < 10 have a leading 0.
`t`	Displays the first character of the AM/PM indicator for the given time.
`tt(+)`	Displays the full AM/PM indicator for the given time.
`y/yy/yyyy`	Displays the year for the given time.
`z/zz/zzz(+)`	Displays the timezone offset for the given time.

Take a look at the following lines of code, which demonstrate using string format specifiers to create custom-formatted date and time strings:

```
DateTime dt = DateTime.Now;

Console.WriteLine(string.Format("Default format: {0}", dt.ToString()));
Console.WriteLine(dt.ToString("dddd dd MMMM, yyyy g"));
Console.WriteLine(string.Format("Custom Format 1: {0:MM/dd/yy hh:mm:sstt}", dt));
Console.WriteLine(string.Format("Custom Format 2: {0:hh:mm:sstt G\\MT zz}", dt));
```

Here is the output from the preceding code:

```
Default format: 9/24/2005 12:59:49 PM
Saturday 24 September, 2005 A.D.
```

```
Custom Format 1: 09/24/05 12:59:49PM
Custom Format 2: 12:59:49PM GMT -06
```

You can also provide custom format specifiers for numeric values as well. Table 3.2 describes the custom format specifiers available for numeric values.

TABLE 3.2 Numeric Custom Format Specifiers

Specifier	Description
`0`	The zero placeholder.
`#`	The digit placeholder. If the given value has a digit in the position indicated by the # specifier, that digit is displayed in the formatted output.
`.`	Decimal point.
`,`	Thousands separator.
`%`	Percentage specifier. The value being formatted will be multiplied by 100 before being included in the formatted output.
`E0/E+0/e/e+0/e-0/E`	Scientific notation.
`'XX'` or `"XX"`	Literal strings. These are included literally in the formatted output without translation in their relative positions.
`;`	Section separator for conditional formatting of negative, zero, and positive values.

If multiple format sections are defined, conditional behavior can be implemented for even more fine-grained control of the numeric formatting:

- Two sections—If you have two formatting sections, the first section applies to all positive (including 0) values. The second section applies to negative values. This is extremely handy when you want to enclose negative values in parentheses as is done in many accounting software packages.
- Three sections—If you have three formatting sections, the first section applies to all positive (*not* including 0) values. The second section applies to negative values, and the third section applies to zero.

The following few lines of code illustrate how to use custom numeric format specifiers.

```
double dVal = 59.99;
double dNeg = -569.99;
double zeroVal = 0.0;
double pct = 0.23;

string formatString = "{0:$#,###0.00;($#,###0.00);nuttin}";
Console.WriteLine(string.Format(formatString, dVal));
Console.WriteLine(string.Format(formatString, dNeg));
Console.WriteLine(string.Format(formatString, zeroVal));
Console.WriteLine(pct.ToString("00%"));
```

The output generated by the preceding code is shown in the following code:

```
$59.99
($569.99)
nuttin
23%
```

Manipulating and Comparing Strings

In addition to displaying strings that contain all kinds of formatted data, other common string-related tasks are string manipulation and comparison. An important thing to keep in mind is that the string is actually a class in the underlying Base Class Library of the .NET Framework. Because it is a class, you can actually invoke methods on a string, just as you can invoke methods on any other class.

You can invoke these methods both on string literals or on string variables, as shown in the following code:

```
int x = string.Length();
int y = "Hello World".Length();
```

Table 3.3 is a short list of some of the most commonly used methods that you can use on a string for obtaining information about the string or manipulating it.

TABLE 3.3 Commonly Used String Instance Methods

Method	Description
`CompareTo`	Compares this string instance with another string instance.
`Contains`	Returns a Boolean indicating whether the current string instance contains the given substring.
`CopyTo`	Copies a substring from within the string instance to a specified location within an array of characters.
`EndsWith`	Returns a Boolean value indicating whether the string ends with a given substring.
`Equals`	Indicates whether the string is equal to another string. You can use the `'=='` operator as well.
`IndexOf`	Returns the index of a substring within the string instance.
`IndexOfAny`	Returns the first index occurrence of any character in the substring within the string instance.
`PadLeft`	Pads the string with the specified number of spaces or another Unicode character, effectively right-justifying the string.
`PadRight`	Appends a specified number of spaces or other Unicode character to the end of the string, creating a left-justification.
`Remove`	Deletes a given number of characters from the string.
`Replace`	Replaces all occurrences of a given character or string within the string instance with the specified replacement.
`Split`	Splits the current string into an array of strings, using the specified character as the splitting point.

TABLE 3.3 Continued

Method	Description
`StartsWith`	Returns a Boolean value indicating whether the string instance starts with the specified string.
`SubString`	Returns a specified portion of the string, given a starting point and length.
`ToCharArray`	Converts the string into an array of characters.
`ToLower`	Converts the string into all lowercase characters.
`ToUpper`	Converts the string into all uppercase characters.
`Trim`	Removes all occurrences of a given set of characters from the beginning and end of the string.
`TrimStart`	Performs the `Trim` function, but only on the beginning of the string.
`TrimEnd`	Performs the `Trim` function, but only on the end of the string.

Take a look at the following code, which illustrates some of the things you can do with strings to further query and manipulate them:

```
string sourceString = "Mary Had a Little Lamb";
string sourceString2 = "   Mary Had a Little Lamb       ";
Console.WriteLine(sourceString.ToLower());
Console.WriteLine(string.Format("The string '{0}' is {1} chars long.",
    sourceString,sourceString.Length));
Console.WriteLine(string.Format("Fourth word in sentence is : {0}",
    sourceString.Split(' ')[3]));
Console.WriteLine(sourceString2.Trim());
Console.WriteLine("Two strings equal? " + (sourceString == sourceString2.Trim()));
```

The output of the preceding code looks as follows:

```
mary had a little lamb
The string 'Mary Had a Little Lamb' is 22 chars long.
Fourth word in sentence is : Little
Mary Had a Little Lamb
Two strings equal? True
```

Introduction to the `StringBuilder`

As mentioned earlier, strings are immutable. This means that when you concatenate two strings to form a third string, there will be a short period of time where the CLR will actually have all three strings in memory. So, for example, when you concatenate as shown in the following code:

```
string a = "Hello";
string b = "World";
string c = a + " " + c;
```

You actually end up with four strings in memory, including the space. To alleviate this performance issue with string concatenation as well as to provide you with a tool to make concatenation easier, the .NET Framework comes with a class called the `StringBuilder`.

By using a `StringBuilder` to dynamically create strings of variable length, you get around the immutable string fact of CLR strings and the code can often become more readable as a result. Take a look at the `StringBuilder` in action in the following code:

```
StringBuilder sb = new StringBuilder();
sb.Append("Greetings!\n");
formatString = "{0:$#,###0.00;($#,###0.00);Zero}";
dVal = 129.99;
sb.AppendFormat(formatString, dVal);
sb.Append("\nThis is a big concatenated string.");
Console.WriteLine(sb.ToString());
```

The output of the preceding code looks like the following:

```
Greetings!
$129.99
This is a big concatenated string.
```

Note that the `\n` from the preceding code inserts a newline character into the string.

Working with Regular Expressions

Regular expressions allow the fast, efficient processing of text. The text being processed can be something as small as an email address or as large as a multiline input box. The use of regular expressions not only allows you to validate text against a defined pattern, but it also allows you to extract data from text that matches a given pattern.

You can think of a regular expression as an extremely powerful wildcard. Most of us are familiar enough with wildcards to know that when we see an expression like "SAMS *", everything that begins with the word SAMS is a match for that expression. Regular expressions give you additional power, control, and functionality above and beyond simple wildcards.

This section provides you with a brief introduction to the classes in the .NET Framework that support the use of regular expressions. For more information on regular expressions themselves, you might want to check out *Regular Expression Pocket Reference* (O'Reilly Media, ISBN: 059600415X) or *Mastering Regular Expressions, 2nd Edition* (O'Reilly Media, ISBN: 0596002890). These books will give you the information you need in order to create your own regular expressions as well as a list of commonly used expressions. Regular expressions themselves are beyond the scope of this chapter.

Validating Input

One extremely common use of regular expressions is to validate user input against some predefined format. For example, rules are often enforced to ensure that passwords have

certain characteristics that make them harder to break. These rules are typically defined as regular expressions. Regular expressions are also often used to validate simple input such as email addresses and phone numbers.

The key class provided by the .NET Framework for working with regular expressions is the `RegEx` class. This class provides a static method called `IsMatch` that returns a Boolean indicating whether the specified input string matches a given regular expression.

In the following code, a common regular expression used to test for valid email addresses is used:

```
string emailPattern =
@"^([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)¦[ccc]
(([\w-]+\.)+))([a-zA-Z]{2,4}¦[0-9]{1,3})(\]?)$";
Console.Write("Enter an e-mail address:");
string emailInput = Console.ReadLine();
bool match = Regex.IsMatch(emailInput, emailPattern);
if (match)
    Console.WriteLine("E-mail address is valid.");
else
    Console.WriteLine("Supplied input is not a valid e-mail address.");
```

Don't worry if this regular expression doesn't make much sense to you. The basic idea behind the email pattern is that it requires some alphanumeric characters, then an @-sign, and then some combination of characters followed by a ".", and at least two characters following that. Try out the preceding code on different inputs and see what results you get. Even if you don't understand the regular expressions themselves, knowing that they exist and that you can use them to validate input is going to be extremely helpful in the creation of your applications.

Extracting Data from Input

The other common use for regular expressions is in parsing text according to the expression and using that to extract data (called Group matches) from user input.

Regular expressions include a particular feature called *groups*. A group allows you to put a named identifier on a particular section of the regular expression. When you call `Match()` to compare input data against the pattern, the results actually separate the matches by group, allowing you to extract the portion of the input that matched each group.

For example, in the preceding example we could have created a group called `username` that would have allowed us to extract all of the data that precedes the @ symbol in an email address. Then, when performing a match, we could have extracted the username from the input using the regular expression's named group.

Take a look at the following code, which illustrates how to extract both the protocol name and the port number from a URL entered by the user at the console. The great thing about regular expressions is that they are their own language, so they don't have a particular affinity toward C, C++, C#, VB.NET, or any other language. This makes it easy

to borrow regular expressions from samples and reference guides on the Internet and in publications. In the following code, the regular expression was borrowed from an MSDN example:

```
string urlPattern = @"^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/";
Console.WriteLine();
Console.Write("Enter a URL for data parsing: ");
string url = Console.ReadLine();
Regex urlExpression = new Regex(urlPattern, RegexOptions.Compiled);
Match urlMatch = urlExpression.Match(url);
Console.WriteLine("The Protocol you entered was " +
  urlMatch.Groups["proto"].Value);
Console.WriteLine("The Port Number you entered was " +
  urlMatch.Groups["port"].Value);
```

When you run the preceding code against a URL without a port number, you will notice that you don't get any group values. This is because the input doesn't actually match the regular expression at all. When there are no matches, you obviously can't extract meaningful data from the named groups. When you run the preceding code against a URL with port numbers that match the regular expression, you will get output that looks like the following text:

```
Enter a URL for data parsing: http://server.com:2100/home.aspx
The Protocol you entered was http
The Port Number you entered was :2100
```

Summary

In this chapter you have seen that the days of having to carry around your own library of string routines like a portable life preserver are long gone. With C# and the .NET Framework, strings are a native part of the library of base classes, and as such provide you with a full host of utility methods for comparison, manipulation, formatting, and much more. You also saw that the `StringBuilder` class provides you with an easy-to-use set of utility methods for dynamically building strings without the performance penalty of native string concatenation.

Finally, this chapter gave you a brief introduction into the power of regular expressions and how that power can be harnessed with the `Regex` class. After reading through this chapter and testing out the sample code, you should be familiar with some of the things that you can do with strings and regular expressions to make your applications more powerful.

This page intentionally left blank