CHAPTER 14

Chars, Strings, and Working with Text

In this chapter:

In this chapter, I'll explain the mechanics of working with individual characters and strings in the Microsoft .NET Framework. I'll start by talking about the `System.Char` structure and the various ways that you can manipulate a character. Then I'll go over the more useful `System.String` class, which allows you to work with immutable strings. (Once created, strings can't be modified in any way.) After examining strings, I'll show you how to perform various operations efficiently to build a string dynamically via the `System.Text.StringBuilder` class. With the string basics out of the way, I'll then describe how to format objects into strings and how to efficiently persist or transmit strings by using various encodings. Finally, I'll discuss the `System.Security.SecureString` class, which can be used to protect sensitive string data such as passwords and credit card information.

Characters

In the .NET Framework, characters are always represented in 16-bit Unicode code values, easing the development of global applications. A character is represented with an instance of the `System.Char` structure (a value type). The `System.Char` type is pretty simple. It offers two public read-only constant fields: `MinValue`, defined as `'\0'`, and `MaxValue`, defined as `'\uffff'`.

Given an instance of a `Char`, you can call the static `GetUnicodeCategory` method, which returns a value of the `System.Globalization.UnicodeCategory` enumerated type. This value indicates whether the character is a control character, a currency symbol, a lowercase letter, an uppercase letter, a punctuation character, a math symbol, or another character (as defined by the Unicode standard).

To ease developing, the Char type also offers several static methods, such as IsDigit, IsLetter, IsWhiteSpace, IsUpper, IsLower, IsPunctuation, IsLetterOrDigit, IsControl, IsNumber, IsSeparator, IsSurrogate, IsLowSurrogate, IsHighSurrogate, and IsSymbol. Most of these methods call GetUnicodeCategory internally and simply return true or false accordingly. Note that all of these methods take either a single character for a parameter or a String and the index of a character within the String as parameters.

In addition, you can convert a single character to its lowercase or uppercase equivalent in a culture-agnostic way by calling the static ToLowerInvariant or ToUpperInvariant method. Alternatively, the ToLower and ToUpper methods convert the character by using the culture information associated with the calling thread (which the methods obtain internally by querying the static CurrentCulture property of the System.Globalization.CultureInfo class). You can also specify a particular culture by passing an instance of the CultureInfo class to these methods. ToLower and ToUpper require culture information because letter casing is a culture-dependent operation. For example, Turkish considers the uppercase of U+0069 (LATIN LOWERCASE LETTER I) to be U+0130 (LATIN UPPERCASE LETTER I WITH DOT ABOVE), whereas other cultures consider the result to be U+0049 (LATIN CAPITAL LETTER I).

Besides these static methods, the Char type also offers a few instance methods of its own. The Equals method returns true if two Char instances represent the same 16-bit Unicode code point. The CompareTo methods (defined by the IComparable/IComparable<Char> interfaces) return a comparison of two Char instances; this comparison is not culture-sensitive. The ConvertFromUtf32 method produces a string consisting of one or two UTF-16 characters from a single UTF-32 character. The ConvertToUtf32 produces a UTF-32 character from a low/high surrogate pair or from a string. The ToString method returns a String consisting of a single character. The opposite of ToString is Parse/TryParse, which takes a single-character String and returns its UTF-16 code point.

The last method, GetNumericValue, returns the numeric equivalent of a character. I demonstrate this method in the following code.

```
using System;

public static class Program {
   public static void Main() {
      Double d;                           // '\u0033' is the "digit 3"
      d = Char.GetNumericValue('\u0033'); // '3' would work too
      Console.WriteLine(d.ToString());    // Displays "3"

      // '\u00bc' is the "vulgar fraction one quarter ('¼')"
      d = Char.GetNumericValue('\u00bc');
      Console.WriteLine(d.ToString());    // Displays "0.25"

      // 'A' is the "Latin capital letter A"
      d = Char.GetNumericValue('A');
      Console.WriteLine(d.ToString());    // Displays "-1"
   }
}
```

Finally, three techniques allow you to convert between various numeric types to `Char` instances and vice versa. The techniques are listed here in order of preference:

- **Casting** The easiest way to convert a `Char` to a numeric value such as an `Int32` is simply by casting. Of the three techniques, this is the most efficient because the compiler emits Intermediate Language (IL) instructions to perform the conversion, and no methods have to be called. In addition, some languages (such as C#) allow you to indicate whether the conversion should be performed using checked or unchecked code (discussed in Chapter 5, "Primitive, Reference, and Value Types").

- **Use the Convert type** The `System.Convert` type offers several static methods that are capable of converting a `Char` to a numeric type and vice versa. All of these methods perform the conversion as a checked operation, causing an `OverflowException` to be thrown if the conversion results in the loss of data.

- **Use the IConvertible interface** The `Char` type and all of the numeric types in the .NET Framework Class Library (FCL) implement the `IConvertible` interface. This interface defines methods such as `ToUInt16` and `ToChar`. This technique is the least efficient of the three because calling an interface method on a value type requires that the instance be boxed—`Char` and all of the numeric types are value types. The methods of `IConvertible` throw a `System.InvalidCastException` if the type can't be converted (such as converting a `Char` to a `Boolean`) or if the conversion results in a loss of data. Note that many types (including the FCL's `Char` and numeric types) implement `IConvertible`'s methods as explicit interface member implementations (described in Chapter 13, "Interfaces"). This means that you must explicitly cast the instance to an `IConvertible` before you can call any of the interface's methods. All of the methods of `IConvertible` except `GetTypeCode` accept a reference to an object that implements the `IFormatProvider` interface. This parameter is useful if for some reason the conversion needs to take culture information into account. For most conversions, you can pass `null` for this parameter because it would be ignored anyway.

The following code demonstrates how to use these three techniques.

```
using System;

public static class Program {
   public static void Main() {
      Char  c;
      Int32 n;

      // Convert number <-> character using C# casting
      c = (Char) 65;
      Console.WriteLine(c);                  // Displays "A"

      n = (Int32) c;
      Console.WriteLine(n);                  // Displays "65"

      c = unchecked((Char) (65536 + 65));
      Console.WriteLine(c);                  // Displays "A"
```

```
      // Convert number <-> character using Convert
      c = Convert.ToChar(65);
      Console.WriteLine(c);                  // Displays "A"

      n = Convert.ToInt32(c);
      Console.WriteLine(n);                  // Displays "65"


      // This demonstrates Convert's range checking
      try {
         c = Convert.ToChar(70000);          // Too big for 16 bits
         Console.WriteLine(c);               // Doesn't execute
      }
      catch (OverflowException) {
         Console.WriteLine("Can't convert 70000 to a Char.");
      }


      // Convert number <-> character using IConvertible
      c = ((IConvertible) 65).ToChar(null);
      Console.WriteLine(c);                  // Displays "A"

      n = ((IConvertible) c).ToInt32(null);
      Console.WriteLine(n);                  // Displays "65"
   }
}
```

The System.String Type

One of the most used types in any application is `System.String`. A `String` represents an immutable sequence of characters. The `String` type is derived immediately from `Object`, making it a reference type, and therefore, `String` objects (its array of characters) always live in the heap, never on a thread's stack. The `String` type also implements several interfaces (`IComparable`/`IComparable<String>`, `ICloneable`, `IConvertible`, `IEnumerable`/`IEnumerable<Char>`, and `IEquatable<String>`).

Constructing Strings

Many programming languages (including C#) consider `String` to be a primitive type—that is, the compiler lets you express literal strings directly in your source code. The compiler places these literal strings in the module's metadata, and they are then loaded and referenced at run time.

In C#, you can't use the `new` operator to construct a `String` object from a literal string.

```
using System;

public static class Program {
   public static void Main() {
      String s = new String("Hi there.");  // <-- Error
      Console.WriteLine(s);
   }
}
```

Instead, you must use the following simplified syntax.

```
using System;

public static class Program {
   public static void Main() {
      String s = "Hi there.";
      Console.WriteLine(s);
   }
}
```

If you compile this code and examine its IL (using ILDasm.exe), you'd see the following.

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       13 (0xd)
  .maxstack  1
  .locals init ([0] string s)
  IL_0000:  ldstr      "Hi there."
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000c:  ret
} // end of method Program::Main
```

The `newobj` IL instruction constructs a new instance of an object. However, no `newobj` instruction appears in the IL code example. Instead, you see the special `ldstr` (load string) IL instruction, which constructs a `String` object by using a literal string obtained from metadata. This shows you that the common language runtime (CLR) does, in fact, have a special way of constructing literal `String` objects.

If you are using unsafe code, you can construct a `String` object from a `Char*` or `SByte*`. To accomplish this, you would use C#'s `new` operator and call one of the constructors provided by the `String` type that takes `Char*` or `SByte*` parameters. These constructors create a `String` object, initializing the string from an array of `Char` instances or signed bytes. The other constructors don't have any pointer parameters and can be called using safe (verifiable) code written in any managed programming language.

C# offers some special syntax to help you enter literal strings into the source code. For special characters such as new lines, carriage returns, and backspaces, C# uses the escape mechanism familiar to C/C++ developers.

```
// String containing carriage-return and newline characters
String s = "Hi\r\nthere.";
```

Important Although the preceding example hard-codes carriage-return and newline characters into the string, I don't recommend this practice. Instead, the `System.Environment` type defines a read-only `NewLine` property that returns a string consisting of these characters when your application is running on Windows. However, the `NewLine` property is platform sensitive, and it returns the appropriate string required to obtain a newline by the underlying platform. So, for example, if the Common Language Infrastructure (CLI) is ported to a UNIX system, the `NewLine` property would return a string consisting of just a single character `\n`. Here's the proper way to define the previous string so that it works correctly on any platform.

```
String s = "Hi" + Environment.NewLine + "there.";
```

You can concatenate several strings to form a single string by using C#'s + operator as follows.

```
// Three literal strings concatenated to form a single literal string
String s = "Hi" + " " + "there.";
```

In this code, because all of the strings are literal strings, the C# compiler concatenates them at compile time and ends up placing just one string—`"Hi there."`—in the module's metadata. Using the + operator on nonliteral strings causes the concatenation to be performed at run time. To concatenate several strings together at run time, avoid using the + operator because it creates multiple string objects on the garbage-collected heap. Instead, use the `System.Text.StringBuilder` type (which I'll explain later in this chapter).

Finally, C# also offers a special way to declare a string in which all characters between quotes are considered part of the string. These special declarations are called verbatim strings and are typically used when specifying the path of a file or directory or when working with regular expressions. Here is some code showing how to declare the same string with and without using the verbatim string character (@).

```
// Specifying the pathname of an application
String file = "C:\\Windows\\System32\\Notepad.exe";

// Specifying the pathname of an application by using a verbatim string
String file = @"C:\Windows\System32\Notepad.exe";
```

You could use either one of the preceding code lines in a program because they produce identical strings in the assembly's metadata. However, the @ symbol before the string on the second line tells the compiler that the string is a verbatim string. In effect, this tells the compiler to treat backslash characters as backslash characters instead of escape characters, making the path much more readable in your source code.

Now that you've seen how to construct a string, let's talk about some of the operations you can perform on `String` objects.

Strings Are Immutable

The most important thing to know about a `String` object is that it is immutable. That is, once created, a string can never get longer, get shorter, or have any of its characters changed. Having immutable strings offers several benefits. First, it allows you to perform operations on a string without actually changing the string.

```
if (s.ToUpperInvariant().Substring(10, 21).EndsWith("EXE")) {
  ...
}
```

Here, `ToUpperInvariant` returns a new string; it doesn't modify the characters of the string `s`. `Substring` operates on the string returned by `ToUpperInvariant` and also returns a new string, which is then examined by `EndsWith`. The two temporary strings created by `ToUpperInvariant` and `Substring` are not referenced for long by the application code, and the garbage collector will reclaim their memory at the next collection. If you perform a lot of string manipulations, you end up creating a lot of `String` objects on the heap, which causes more frequent garbage collections, thus hurting your application's performance.

Having immutable strings also means that there are no thread synchronization issues when manipulating or accessing a string. In addition, it's possible for the CLR to share multiple identical `String` contents through a single `String` object. This can reduce the number of strings in the system—thereby conserving memory usage—and it is what string interning (discussed later in the chapter) is all about.

For performance reasons, the `String` type is tightly integrated with the CLR. Specifically, the CLR knows the exact layout of the fields defined within the `String` type, and the CLR accesses these fields directly. This performance and direct access come at a small development cost: the `String` class is sealed, which means that you cannot use it as a base class for your own type. If you were able to define your own type, using `String` as a base type, you could add your own fields, which would break the CLR's assumptions. In addition, you could break some assumptions that the CLR team has made about `String` objects being immutable.

Comparing Strings

Comparing is probably the most common operation performed on strings. There are two reasons to compare two strings with each other. We compare two strings to determine equality or to sort them (usually for presentation to a user).

In determining string equality or when comparing strings for sorting, it is highly recommended that you call one of these methods (defined by the `String` class).

```
Boolean Equals(String value, StringComparison comparisonType)
static Boolean Equals(String a, String b, StringComparison comparisonType)

static Int32 Compare(String strA, String strB, StringComparison comparisonType)
static Int32 Compare(string strA, string strB, Boolean ignoreCase, CultureInfo culture)
static Int32 Compare(String strA, String strB, CultureInfo culture, CompareOptions options)
```

```
static Int32 Compare(String strA, Int32 indexA, String strB, Int32 indexB, Int32 length,
   StringComparison comparisonType)
static Int32 Compare(String strA, Int32 indexA, String strB, Int32 indexB, Int32 length,
   CultureInfo culture, CompareOptions options)
static Int32 Compare(String strA, Int32 indexA, String strB, Int32 indexB, Int32 length,
   Boolean ignoreCase, CultureInfo culture)

Boolean StartsWith(String value, StringComparison comparisonType)
Boolean StartsWith(String value,
   Boolean ignoreCase, CultureInfo culture)

Boolean EndsWith(String value, StringComparison comparisonType)
Boolean EndsWith(String value, Boolean ignoreCase, CultureInfo culture)
```

When sorting, you should always perform case-sensitive comparisons. The reason is that if two strings differing only by case are considered to be equal, they could be ordered differently each time you sort them; this would confuse the user.

The `comparisonType` argument (in most of the preceding methods) is one of the values defined by the `StringComparison` enumerated type, which is defined as follows.

```
public enum StringComparison {
   CurrentCulture = 0,
   CurrentCultureIgnoreCase = 1,
   InvariantCulture = 2,
   InvariantCultureIgnoreCase = 3,
   Ordinal = 4,
   OrdinalIgnoreCase = 5
}
```

The `options` argument (in two of the preceding methods) is one of the values defined by the `CompareOptions` enumerator type.

```
[Flags]
public enum CompareOptions {
   None = 0,
   IgnoreCase = 1,
   IgnoreNonSpace = 2,
   IgnoreSymbols  = 4,
   IgnoreKanaType = 8,
   IgnoreWidth = 0x00000010,
   Ordinal = 0x40000000,
   OrdinalIgnoreCase = 0x10000000,
   StringSort = 0x20000000
}
```

Methods that accept a `CompareOptions` argument also force you to explicitly pass in a culture. When passing in the `Ordinal` or `OrdinalIgnoreCase` flag, these `Compare` methods ignore the specified culture.

Many programs use strings for internal programmatic purposes such as path names, file names, URLs, registry keys and values, environment variables, reflection, Extensible Markup Language (XML) tags, XML attributes, and so on. Often, these strings are not shown to a user and are used only within the program. When comparing programmatic strings, you should always use

`StringComparison.Ordinal` or `StringComparison.OrdinalIgnoreCase`. This is the fastest way to perform a comparison that is not to be affected in any linguistic way because culture information is not taken into account when performing the comparison.

On the other hand, when you want to compare strings in a linguistically correct manner (usually for display to an end user), you should use `StringComparison.CurrentCulture` or `StringComparison.CurrentCultureIgnoreCase`.

Important For the most part, `StringComparison.InvariantCulture` and `StringComparison.InvariantCultureIgnoreCase` should not be used. Although these values cause the comparison to be linguistically correct, using them to compare programmatic strings takes longer than performing an ordinal comparison. Furthermore, the invariant culture is culture agnostic, which makes it an incorrect choice when working with strings that you want to show to an end user.

Important If you want to change the case of a string's characters before performing an ordinal comparison, you should use `String`'s `ToUpperInvariant` or `ToLowerInvariant` method. When normalizing strings, it is highly recommended that you use `ToUpperInvariant` instead of `ToLowerInvariant` because Microsoft has optimized the code for performing uppercase comparisons. In fact, the FCL internally normalizes strings to uppercase prior to performing case-insensitive comparisons. We use `ToUpperInvariant` and `ToLowerInvariant` methods because the `String` class does not offer `ToUpperOrdinal` and `ToLowerOrdinal` methods. We do not use the `ToUpper` and `ToLower` methods because these are culture sensitive.

Sometimes, when you compare strings in a linguistically correct manner, you want to specify a specific culture rather than use a culture that is associated with the calling thread. In this case, you can use the overloads of the `StartsWith`, `EndsWith`, and `Compare` methods shown earlier, all of which take `Boolean` and `CultureInfo` arguments.

Important The `String` type defines several overloads of the `Equals`, `StartsWith`, `EndsWith`, and `Compare` methods in addition to the versions shown earlier. Microsoft recommends that these other versions (not shown in this book) be avoided. Furthermore, `String`'s other comparison methods—`CompareTo` (required by the `IComparable` interface), `CompareOrdinal`, and the `==` and `!=` operators—should also be avoided. The reason for avoiding these methods and operators is because the caller does not explicitly indicate how the string comparison should be performed, and you cannot determine from the name of the method what the default comparison will be. For example, by default, `CompareTo` performs a culture-sensitive comparison, whereas `Equals` performs an ordinal comparison. Your code will be easier to read and maintain if you always indicate explicitly how you want to perform your string comparisons.

Now, let's talk about how to perform linguistically correct comparisons. The .NET Framework uses the `System.Globalization.CultureInfo` type to represent a language/country pair (as described by the RFC 1766 standard). For example, "en-US" identifies English as written in the United States, "en-AU" identifies English as written in Australia, and "de-DE" identifies German as written in Germany. In the CLR, every thread has two properties associated with it. Each of these properties refers to a `CultureInfo` object. The two properties are:

- **CurrentUICulture** This property is used to obtain resources that are shown to an end user. It is most useful for GUI or Web Forms applications because it indicates the language that should be used when displaying UI elements such as labels and buttons. By default, when you create a thread, this thread property is set to a `CultureInfo` object, which identifies the language of the Windows version the application is running on using the Win32 `GetUserDefaultUILanguage` function. If you're running a Multilingual User Interface (MUI) version of Windows, you can set this via the Regional And Language Options Control Panel Settings dialog box. On a non-MUI version of Windows, the language is determined by the localized version of the operating system installed (or the installed language pack) and the language is not changeable.

- **CurrentCulture** This property is used for everything that `CurrentUICulture` isn't used for, including number and date formatting, string casing, and string comparing. When formatting, both the language and country parts of the `CultureInfo` object are used. By default, when you create a thread, this thread property is set to a `CultureInfo` object, whose value is determined by calling the Win32 `GetUserDefaultLCID` method, whose value is set in the Regional And Language Control Panel applet.

For the two thread properties mentioned above, you can override the default value used by the system when a new thread gets created with AppDomain defaults by setting `CultureInfo`'s static `DefaultThreadCurrentCulture` and `DefaultThreadCurrentUICulture` properties.

On many computers, a thread's `CurrentUICulture` and `CurrentCulture` properties will be set to the same `CultureInfo` object, which means that they both use the same language/country information. However, they can be set differently. For example: an application running in the United States could use Spanish for all of its menu items and other GUI elements while properly displaying all of the currency and date formatting for the United States. To do this, the thread's `CurrentUICulture` property should be set to a `CultureInfo` object initialized with a language of "es" (for Spanish), while the thread's `CurrentCulture` property should be set to a `CultureInfo` object initialized with a language/country pair of "en-US."

Internally, a `CultureInfo` object has a field that refers to a `System.Globalization.CompareInfo` object, which encapsulates the culture's character-sorting table information as defined by the Unicode standard. The following code demonstrates the difference between performing an ordinal comparison and a culturally aware string comparison.

```
using System;
using System.Globalization;

public static class Program {
   public static void Main() {
```

```
      String s1 = "Strasse";
      String s2 = "Straße";
      Boolean eq;

      // CompareOrdinal returns nonzero.
      eq = String.Compare(s1, s2, StringComparison.Ordinal) == 0;
      Console.WriteLine("Ordinal  comparison: '{0}' {2} '{1}'", s1, s2,
         eq ? "==" : "!=");

      // Compare Strings appropriately for people
      // who speak German (de) in Germany (DE)
      CultureInfo ci = new CultureInfo("de-DE");

      // Compare returns zero.
      eq = String.Compare(s1, s2, true, ci) == 0;
      Console.WriteLine("Cultural comparison: '{0}' {2} '{1}'", s1, s2,
         eq ? "==" : "!=");
   }
}
```

Building and running this code produces the following output.

```
Ordinal  comparison: 'Strasse' != 'Straße'
Cultural comparison: 'Strasse' == 'Straße'
```

Note When the `Compare` method is not performing an ordinal comparison, it performs *character expansions*. A character expansion is when a character is expanded to multiple characters regardless of culture. In the above case, the German *Eszet* character 'ß' is always expanded to 'ss.' Similarly, the 'Æ' ligature character is always expanded to 'AE.' So in the code example, the second call to `Compare` will always return 0 regardless of which culture I actually pass in to it.

In some rare circumstances, you may need to have even more control when comparing strings for equality or for sorting. This could be necessary when comparing strings consisting of Japanese characters. This additional control can be accessed via the `CultureInfo` object's `CompareInfo` property. As mentioned earlier, a `CompareInfo` object encapsulates a culture's character comparison tables, and there is just one `CompareInfo` object per culture.

When you call `String`'s `Compare` method, if the caller specifies a culture, the specified culture is used, or if no culture is specified, the value in the calling thread's `CurrentCulture` property is used. Internally, the `Compare` method obtains the reference to the `CompareInfo` object for the appropriate culture and calls the `Compare` method of the `CompareInfo` object, passing along the appropriate options (such as case insensitivity). Naturally, you could call the `Compare` method of a specific `CompareInfo` object yourself if you need the additional control.

The `Compare` method of the `CompareInfo` type takes as a parameter a value from the `CompareOptions` enumerated type (as shown earlier). You can OR these bit flags together to gain significantly greater control when performing string comparisons. For a complete description of these symbols, consult the .NET Framework documentation.

The following code demonstrates how important culture is to sorting strings and shows various ways of performing string comparisons.

```
using System;
using System.Text;
using System.Windows.Forms;
using System.Globalization;
using System.Threading;

public sealed class Program {
   public static void Main() {
      String output = String.Empty;
      String[] symbol = new String[] { "<", "=", ">" };
      Int32 x;
      CultureInfo ci;

      // The code below demonstrates how strings compare
      // differently for different cultures.
      String s1 = "coté";
      String s2 = "côte";

      // Sorting strings for French in France.
      ci = new CultureInfo("fr-FR");
      x = Math.Sign(ci.CompareInfo.Compare(s1, s2));
      output += String.Format("{0} Compare: {1} {3} {2}",
         ci.Name, s1, s2, symbol[x + 1]);
      output += Environment.NewLine;

      // Sorting strings for Japanese in Japan.
      ci = new CultureInfo("ja-JP");
      x = Math.Sign(ci.CompareInfo.Compare(s1, s2));
      output += String.Format("{0} Compare: {1} {3} {2}",
         ci.Name, s1, s2, symbol[x + 1]);
      output += Environment.NewLine;

      // Sorting strings for the thread's culture
      ci = Thread.CurrentThread.CurrentCulture;
      x = Math.Sign(ci.CompareInfo.Compare(s1, s2));
      output += String.Format("{0} Compare: {1} {3} {2}",
         ci.Name, s1, s2, symbol[x + 1]);
      output += Environment.NewLine + Environment.NewLine;

      // The code below demonstrates how to use CompareInfo.Compare's
      // advanced options with 2 Japanese strings. One string represents
      // the word "shinkansen" (the name for the Japanese high-speed
      // train) in hiragana (one subtype of Japanese writing), and the
      // other represents the same word in katakana (another subtype of
      // Japanese writing).
      s1 = "";  // ("\u3057\u3093\u304B\u3093\u305b\u3093")
      s2 = "";  // ("\u30b7\u30f3\u30ab\u30f3\u30bb\u30f3")

      // Here is the result of a default comparison
      ci = new CultureInfo("ja-JP");
      x = Math.Sign(String.Compare(s1, s2, true, ci));
      output += String.Format("Simple {0} Compare: {1} {3} {2}",
         ci.Name, s1, s2, symbol[x + 1]);
      output += Environment.NewLine;
```

```
      // Here is the result of a comparison that ignores
      // kana type (a type of Japanese writing)
      CompareInfo compareInfo = CompareInfo.GetCompareInfo("ja-JP");
      x = Math.Sign(compareInfo.Compare(s1, s2, CompareOptions.IgnoreKanaType));
      output += String.Format("Advanced {0} Compare: {1} {3} {2}",
         ci.Name, s1, s2, symbol[x + 1]);

      MessageBox.Show(output, "Comparing Strings For Sorting");
   }
}
```

Note This source code file can't be saved in ANSI or the Japanese characters will be lost. To save this file in Microsoft Visual Studio, go to the Save File As dialog box, click the down arrow that is part of the Save button and select Save With Encoding. I selected Unicode (UTF-8 With Signature) – Codepage 65001. Microsoft's C# compiler can successfully parse source code files by using this code page.

Building and running this code produces the output shown in Figure 14-1.

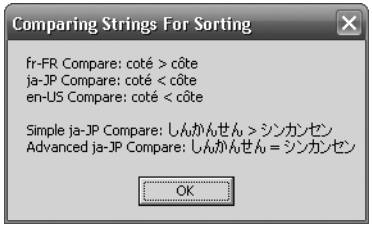

FIGURE 14-1 String sorting results.

In addition to `Compare`, the `CompareInfo` class offers the `IndexOf`, `LastIndexOf`, `IsPrefix`, and `IsSuffix` methods. Because all of these methods offer overloads that take a `CompareOptions` enumeration value as a parameter, they give you more control than the `Compare`, `IndexOf`, `LastIndexOf`, `StartsWith`, and `EndsWith` methods defined by the `String` class. Also, you should be aware that the FCL includes a `System.StringComparer` class that you can also use for performing string comparisons. This class is useful when you want to perform the same kind of comparison repeatedly for many different strings.

String Interning

As I said in the preceding section, checking strings for equality is a common operation for many applications—this task can hurt performance significantly. When performing an ordinal equality check, the CLR quickly tests to see if both strings have the same number of characters. If they don't, the strings are definitely not equal; if they do, the strings might be equal, and the CLR must then compare each individual character to determine for sure. When performing a culturally aware comparison, the CLR must always compare all of the individual characters because strings of different lengths might be considered equal.

In addition, if you have several instances of the same string duplicated in memory, you're wasting memory because strings are immutable. You'll use memory much more efficiently if there is just one instance of the string in memory and all variables needing to refer to the string can just point to the single string object.

If your application frequently compares strings for equality by using case-sensitive, ordinal comparisons, or if you expect to have many string objects with the same value, you can enhance performance substantially if you take advantage of the string interning mechanism in the CLR. When the CLR initializes, it creates an internal hash table in which the keys are strings and the values are references to `String` objects in the managed heap. Initially, the table is empty (of course). The `String` class offers two methods that allow you to access this internal hash table.

```
public static String Intern(String str);
public static String IsInterned(String str);
```

The first method, `Intern`, takes a `String`, obtains a hash code for it, and checks the internal hash table for a match. If an identical string already exists, a reference to the already existing `String` object is returned. If an identical string doesn't exist, a copy of the string is made, the copy is added to the internal hash table, and a reference to this copy is returned. If the application no longer holds a reference to the original `String` object, the garbage collector is able to free the memory of that string. Note that the garbage collector can't free the strings that the internal hash table refers to because the hash table holds the reference to those `String` objects. `String` objects referred to by the internal hash table can't be freed until the AppDomain is unloaded or the process terminates.

As does the `Intern` method, the `IsInterned` method takes a `String` and looks it up in the internal hash table. If a matching string is in the hash table, `IsInterned` returns a reference to the interned string object. If a matching string isn't in the hash table, however, `IsInterned` returns `null`; it doesn't add the string to the hash table.

By default, when an assembly is loaded, the CLR interns all of the literal strings described in the assembly's metadata. Microsoft learned that this hurts performance significantly due to the additional hash table lookups, so it is now possible to turn this "feature" off. If an assembly is marked with a `System.Runtime.CompilerServices.CompilationRelaxationsAttribute` specifying the `System.Runtime.CompilerServices.CompilationRelaxations.NoStringInterning` flag value, the CLR may, according to the ECMA specification, choose not to intern all of the strings defined in that assembly's metadata. Note that, in an attempt to improve your application's performance, the C# compiler always specifies this attribute/flag whenever you compile an assembly.

Even if an assembly has this attribute/flag specified, the CLR may choose to intern the strings, but you should not count on this. In fact, you really should never write code that relies on strings being interned unless you have written code that explicitly calls the `String`'s `Intern` method yourself.

The following code demonstrates string interning.

```
String s1 = "Hello";
String s2 = "Hello";
Console.WriteLine(Object.ReferenceEquals(s1, s2));   // Should be 'False'

s1 = String.Intern(s1);
s2 = String.Intern(s2);
Console.WriteLine(Object.ReferenceEquals(s1, s2));   // 'True'
```

In the first call to the `ReferenceEquals` method, `s1` refers to a `"Hello"` string object in the heap, and `s2` refers to a different `"Hello"` string object in the heap. Because the references are different, `False` should be displayed. However, if you run this on version 4.5 of the CLR, you'll see that `True` is displayed. The reason is because this version of the CLR chooses to ignore the attribute/flag emitted by the C# compiler, and the CLR interns the literal `"Hello"` string when the assembly is loaded into the AppDomain. This means that `s1` and `s2` refer to the single `"Hello"` string in the heap. However, as mentioned previously, you should never write code that relies on this behavior because a future version of the CLR might honor the attribute/flag and not intern the `"Hello"` string. In fact, version 4.5 of the CLR does honor the attribute/flag when this assembly's code has been compiled using the NGen.exe utility.

Before the second call to the `ReferenceEquals` method, the `"Hello"` string has been explicitly interned, and `s1` now refers to an interned `"Hello"`. Then by calling `Intern` again, `s2` is set to refer to the same `"Hello"` string as `s1`. Now, when `ReferenceEquals` is called the second time, we are guaranteed to get a result of `True` regardless of whether the assembly was compiled with the attribute/flag.

So now, let's look at an example to see how you can use string interning to improve performance and reduce memory usage. The `NumTimesWordAppearsEquals` method below takes two arguments: a word and an array of strings in which each array element refers to a single word. This method then determines how many times the specified word appears in the wordlist and returns this count.

```
private static Int32 NumTimesWordAppearsEquals(String word, String[] wordlist) {
   Int32 count = 0;
   for (Int32 wordnum = 0; wordnum < wordlist.Length; wordnum++) {
      if (word.Equals(wordlist[wordnum], StringComparison.Ordinal))
         count++;
   }
   return count;
}
```

As you can see, this method calls `String`'s `Equals` method, which internally compares the strings' individual characters and checks to ensure that all characters match. This comparison can be slow. In addition, the wordlist array might have multiple entries that refer to multiple `String` objects containing the same set of characters. This means that multiple identical strings might exist in the heap and are surviving ongoing garbage collections.

Now, let's look at a version of this method that was written to take advantage of string interning.

```
private static Int32 NumTimesWordAppearsIntern(String word, String[] wordlist) {
   // This method assumes that all entries in wordlist refer to interned strings.
   word = String.Intern(word);
   Int32 count = 0;
   for (Int32 wordnum = 0; wordnum < wordlist.Length; wordnum++) {
      if (Object.ReferenceEquals(word, wordlist[wordnum]))
         count++;
   }
   return count;
}
```

This method interns the word and assumes that the wordlist contains references to interned strings. First, this version might be saving memory if a word appears in the wordlist multiple times because, in this version, `wordlist` would now contain multiple references to the same single `String` object in the heap. Second, this version will be faster because determining if the specified word is in the array is simply a matter of comparing pointers.

Although the `NumTimesWordAppearsIntern` method is faster than the `NumTimesWordAppearsEquals` method, the overall performance of the application might be slower when using the `NumTimesWordAppearsIntern` method because of the time it takes to intern all of the strings when they were added to the `wordlist` array (code not shown). The `NumTimesWordAppearsIntern` method will really show its performance and memory improvement if the application needs to call the method multiple times using the same wordlist. The point of this discussion is to make it clear that string interning is useful, but it should be used with care and caution. In fact, this is why the C# compiler indicates that it doesn't want string interning to be enabled.

String Pooling

When compiling source code, your compiler must process each literal string and emit the string into the managed module's metadata. If the same literal string appears several times in your source code, emitting all of these strings into the metadata will bloat the size of the resulting file.

To remove this bloat, many compilers (including the C# compiler) write the literal string into the module's metadata only once. All code that references the string will be modified to refer to the one string in the metadata. This ability of a compiler to merge multiple occurrences of a single string into a single instance can reduce the size of a module substantially. This process is nothing new—C/C++ compilers have been doing it for years. (Microsoft's C/C++ compiler calls this string pooling.) Even so, string pooling is another way to improve the performance of strings and just one more piece of knowledge that you should have in your repertoire.

Examining a String's Characters and Text Elements

Although comparing strings is useful for sorting them or for detecting equality, sometimes you need just to examine the characters within a string. The `String` type offers several properties and methods to help you do this, including `Length`, `Chars` (an indexer in C#), `GetEnumerator`, `ToCharArray`, `Contains`, `IndexOf`, `LastIndexOf`, `IndexOfAny`, and `LastIndexOfAny`.

In reality, a `System.Char` represents a single 16-bit Unicode code value that doesn't necessarily equate to an abstract Unicode character. For example, some abstract Unicode characters are a combination of two code values. When combined, the U+0625 (the Arabic letter *Alef* with *Hamza* below) and U+0650 (the Arabic Kasra) characters form a single abstract character or *text element*.

In addition, some Unicode text elements require more than a 16-bit value to represent them. These text elements are represented using two 16-bit code values. The first code value is called the high surrogate, and the second code value is called the low surrogate. High surrogates have a value between U+D800 and U+DBFF, and low surrogates have a value between U+DC00 and U+DFFF. The use of surrogates allows Unicode to express more than a million different characters.

Surrogates are rarely used in the United States and Europe but are more commonly used in East Asia. To properly work with text elements, you should use the `System.Globalization.StringInfo` type. The easiest way to use this type is to construct an instance of it, passing its constructor a string. Then you can see how many text elements are in the string by querying the `StringInfo`'s `LengthInTextElements` property. You can then call `StringInfo`'s `SubstringByTextElements` method to extract the text element or the number of consecutive text elements that you desire.

In addition, the `StringInfo` class offers a static `GetTextElementEnumerator` method, which acquires a `System.Globalization.TextElementEnumerator` object that allows you to enumerate through all of the abstract Unicode characters contained in the string. Finally, you could call `StringInfo`'s static `ParseCombiningCharacters` method to obtain an array of `Int32` values. The length of the array indicates how many text elements are contained in the string. Each element of the array identifies an index into the string where the first code value for a new text element can be found.

The following code demonstrates the various ways of using the `StringInfo` class to manipulate a string's text elements.

```
using System;
using System.Text;
using System.Globalization;
using System.Windows.Forms;


public sealed class Program {
   public static void Main() {
      // The string below contains combining characters
      String s = "a\u0304\u0308bc\u0327";
      SubstringByTextElements(s);
      EnumTextElements(s);
      EnumTextElementIndexes(s);
   }
```

```
   private static void SubstringByTextElements(String s) {
      String output = String.Empty;

      StringInfo si = new StringInfo(s);
      for (Int32 element = 0; element < si.LengthInTextElements; element++) {
         output += String.Format(
            "Text element {0} is '{1}'{2}",
            element, si.SubstringByTextElements(element, 1),
            Environment.NewLine);
      }
      MessageBox.Show(output, "Result of SubstringByTextElements");
   }

   private static void EnumTextElements(String s) {
      String output = String.Empty;

      TextElementEnumerator charEnum =
         StringInfo.GetTextElementEnumerator(s);
      while (charEnum.MoveNext()) {
         output += String.Format(
            "Character at index {0} is '{1}'{2}",
            charEnum.ElementIndex, charEnum.GetTextElement(),
            Environment.NewLine);
      }
      MessageBox.Show(output, "Result of GetTextElementEnumerator");
   }

   private static void EnumTextElementIndexes(String s) {
      String output = String.Empty;

      Int32[] textElemIndex = StringInfo.ParseCombiningCharacters(s);
      for (Int32 i = 0; i < textElemIndex.Length; i++) {
         output += String.Format(
            "Character {0} starts at index {1}{2}",
            i, textElemIndex[i], Environment.NewLine);
      }
      MessageBox.Show(output, "Result of ParseCombiningCharacters");
   }
}
```

Building and running this code produces the message boxes shown in Figures 14-2, 14-3, and 14-4.

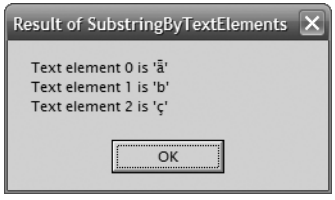

FIGURE 14-2 Result of `SubstringByTextElements`.

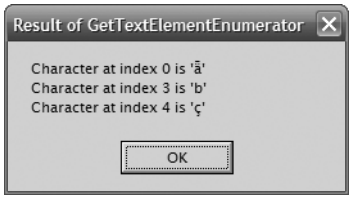

FIGURE 14-3 Result of `GetTextElementEnumerator`.

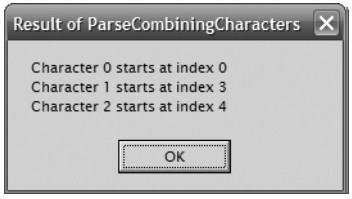

FIGURE 14-4 Result of `ParseCombiningCharacters`.

Other String Operations

The `String` type also offers methods that allow you to copy a string or parts of it. Table 14-1 summarizes these methods.

TABLE 14-1 Methods for Copying Strings

Member	Method Type	Description
`Clone`	Instance	Returns a reference to the same object (`this`). This is OK because `String` objects are immutable. This method implements `String`'s `ICloneable` interface.
`Copy`	Static	Returns a new duplicate string of the specified string. This method is rarely used and exists to help applications that treat strings as tokens. Normally, strings with the same set of characters are interned to a single string. This method creates a new string object so that the references (pointers) are different even though the strings contain the same characters.
`CopyTo`	Instance	Copies a portion of the string's characters to an array of characters.
`Substring`	Instance	Returns a new string that represents a portion of the original string.
`ToString`	Instance	Returns a reference to the same object (`this`).

In addition to these methods, `String` offers many static and instance methods that manipulate a string, such as `Insert`, `Remove`, `PadLeft`, `Replace`, `Split`, `Join`, `ToLower`, `ToUpper`, `Trim`, `Concat`, `Format`, and so on. Again, the important thing to remember about all of these methods is that they return new string objects; because strings are immutable, after they're created, they can't be modified (using safe code).

Constructing a String Efficiently

Because the `String` type represents an immutable string, the FCL provides another type, `System.Text.StringBuilder`, which allows you to perform dynamic operations efficiently with strings and characters to create a `String`. Think of `StringBuilder` as a fancy constructor to create a `String` that can be used with the rest of the framework. In general, you should design methods that take `String` parameters, not `StringBuilder` parameters.

Logically, a `StringBuilder` object contains a field that refers to an array of `Char` structures. `StringBuilder`'s members allow you to manipulate this character array, effectively shrinking the string or changing the characters in the string. If you grow the string past the allocated array of characters, the `StringBuilder` automatically allocates a new, larger array, copies the characters, and starts using the new array. The previous array is garbage collected.

When finished using the `StringBuilder` object to construct your string, "convert" the `StringBuilder`'s character array into a `String` simply by calling the `StringBuilder`'s `ToString` method. This creates a new `String` object in the heap that contains the string that was in the `StringBuilder` at the time you called `ToString`. At this point, you can continue to manipulate the string inside the `StringBuilder`, and later you can call `ToString` again to convert it into another `String` object.

Constructing a `StringBuilder` Object

Unlike with the `String` class, the CLR has no special information about the `StringBuilder` class. In addition, most languages (including C#) don't consider the `StringBuilder` class to be a primitive type. You construct a `StringBuilder` object as you would any other non-primitive type.

```
StringBuilder sb = new StringBuilder();
```

The `StringBuilder` type offers many constructors. The job of each constructor is to allocate and initialize the state maintained by each `StringBuilder` object:

- **Maximum capacity** An `Int32` value that specifies the maximum number of characters that can be placed in the string. The default is `Int32.MaxValue` (approximately 2 billion). It's unusual to change this value. However, you might specify a smaller maximum capacity to ensure that you never create a string over a certain length. Once constructed, a `StringBuilder`'s maximum capacity value can't be changed.
- **Capacity** An `Int32` value indicating the size of the character array being maintained by the `StringBuilder`. The default is 16. If you have some idea of how many characters you'll place in the `StringBuilder`, you should use this number to set the capacity when constructing the `StringBuilder` object.

  When appending characters to the character array, the `StringBuilder` detects if the array is trying to grow beyond the array's capacity. If it is, the `StringBuilder` automatically doubles the capacity field, allocates a new array (the size of the new capacity), and copies the characters from the original array into the new array. The original array will be garbage collected

in the future. Dynamically growing the array hurts performance; avoid this by setting a good initial capacity.

- **Character array** An array of `Char` structures that maintains the set of characters in the "string." The number of characters is always less than or equal to the capacity and maximum capacity values. You can use the `StringBuilder`'s `Length` property to obtain the number of characters used in the array. The `Length` is always less than or equal to the `StringBuilder`'s capacity value. When constructing a `StringBuilder`, you can pass a `String` to initialize the character array. If you don't specify a string, the array initially contains no characters—that is, the `Length` property returns 0.

`StringBuilder` Members

Unlike a `String`, a `StringBuilder` represents a mutable string. This means that most of `StringBuilder`'s members change the contents in the array of characters and don't cause new objects to be allocated on the managed heap. A `StringBuilder` allocates a new object on only two occasions:

- You dynamically build a string whose length is longer than the capacity you've set.
- You call `StringBuilder`'s `ToString` method.

Table 14-2 summarizes `StringBuilder`'s members.

TABLE 14-2 `StringBuilder` Members

Member	Member Type	Description
`MaxCapacity`	Read-only property	Returns the largest number of characters that can be placed in the string.
`Capacity`	Read/write property	Gets or sets the size of the character array. Trying to set the capacity smaller than the string's length or bigger than `MaxCapacity` throws an `ArgumentOutOfRangeException`.
`EnsureCapacity`	Method	Guarantees that the character array is at least the size specified. If the value passed is larger than the `StringBuilder`'s current capacity, the current capacity increases. If the current capacity is already larger than the value passed to this method, no change occurs.
`Length`	Read/write property	Gets or sets the number of characters in the "string." This will likely be smaller than the character array's current capacity. Setting this property to 0 resets the `StringBuilder`'s contents to an empty string.
`ToString`	Method	The parameterless version of this method returns a `String` representing the `StringBuilder`'s character array.
`Chars`	Read/write indexer property	Gets or sets the character at the specified index into the character array. In C#, this is an indexer (parameterful property) that you access using array syntax (`[]`).
`Clear`	Method	Clears the contents of the `StringBuilder` object, the same as setting its `Length` property to `0`.

Member	Member Type	Description
Append	Method	Appends a single object to the end of the character array, growing the array if necessary. The object is converted to a string by using the general format and the culture associated with the calling thread.
Insert	Method	Inserts a single object into the character array, growing the array if necessary. The object is converted to a string by using the general format and the culture associated with the calling thread.
AppendFormat	Method	Appends the specified objects to the end of the character array, growing the array if necessary. The objects are converted to strings by using the formatting and culture information provided by the caller. AppendFormat is one of the most common methods used with StringBuilder objects.
AppendLine	Method	Appends a blank line or a string with a blank line to the end of the character array, increasing the capacity of the array if necessary.
Replace	Method	Replaces one character with another or one string with another from within the character array.
Remove	Method	Removes a range of characters from the character array.
Equals	Method	Returns true only if both StringBuilder objects have the same maximum capacity, capacity, and characters in the array.
CopyTo	Method	Copies a subset of the StringBuilder's characters to a Char array.

One important thing to note about StringBuilder's methods is that most of them return a reference to the same StringBuilder object. This allows a convenient syntax to chain several operations together.

```
StringBuilder sb = new StringBuilder();
String s = sb.AppendFormat("{0} {1}", "Jeffrey", "Richter").
   Replace(' ', '-').Remove(4, 3).ToString();
Console.WriteLine(s);  // "Jeff-Richter"
```

You'll notice that the String and StringBuilder classes don't have full method parity; that is, String has ToLower, ToUpper, EndsWith, PadLeft, PadRight, Trim, and so on. The StringBuilder class doesn't offer any of these methods. On the other hand, the StringBuilder class offers a richer Replace method that allows you to replace characters or strings in a portion of the string (not the whole string). It's unfortunate that there isn't complete parity between these two classes because now you must convert between String and StringBuilder to accomplish certain tasks. For example, to build up a string, convert all characters to uppercase, and then insert a string requires code like the following.

```
// Construct a StringBuilder to perform string manipulations.
StringBuilder sb = new StringBuilder();

// Perform some string manipulations by using the StringBuilder.
sb.AppendFormat("{0} {1}", "Jeffrey", "Richter").Replace(" ", "-");

// Convert the StringBuilder to a String in
// order to uppercase all the characters.
String s = sb.ToString().ToUpper();
```

```
// Clear the StringBuilder (allocates a new Char array).
sb.Length = 0;

// Load the uppercase String into the StringBuilder,
// and perform more manipulations.
sb.Append(s).Insert(8, "Marc-");

// Convert the StringBuilder back to a String.
s = sb.ToString();

// Display the String to the user.
Console.WriteLine(s);  // "JEFFREY-Marc-RICHTER"
```

It's inconvenient and inefficient to have to write this code just because `StringBuilder` doesn't offer all of the operations that `String` does. In the future, I hope that Microsoft will add more string operation methods to `StringBuilder` to make it a more complete class.

Obtaining a String Representation of an Object: `ToString`

You frequently need to obtain a string representation of an object. Usually, this is necessary when you want to display a numeric type (such as `Byte`, `Int32`, and `Single`) or a `DateTime` object to the user. Because the .NET Framework is an object-oriented platform, every type is responsible for providing code that converts an instance's value to a string equivalent. When designing how types should accomplish this, the designers of the FCL devised a pattern that would be used consistently throughout. In this section, I'll describe this pattern.

You can obtain a string representation for any object by calling the `ToString` method. A public, virtual, parameterless `ToString` method is defined by `System.Object` and is therefore callable using an instance of any type. Semantically, `ToString` returns a string representing the object's current value, and this string should be formatted for the calling thread's current culture; that is, the string representation of a number should use the proper decimal separator, digit-grouping symbol, and other elements associated with the culture assigned to the calling thread.

`System.Object`'s implementation of `ToString` simply returns the full name of the object's type. This value isn't particularly useful, but it is a reasonable default for the many types that can't offer a sensible string. For example, what should a string representation of a `FileStream` or a `Hashtable` object look like?

All types that want to offer a reasonable way to obtain a string representing the current value of the object should override the `ToString` method. Many of the core types built into the FCL (`Byte`, `Int32`, `UInt64`, `Double`, and so on) override their `ToString` method and return a culturally aware string. In the Visual Studio debugger, a datatip is displayed when the mouse is placed over a particular variable. The text shown in the datatip is obtained by calling the object's `ToString` method. So, when you define a class, you should always override the `ToString` method so that you get good debugging support.

Specific Formats and Cultures

The parameterless `ToString` method has two problems. First, the caller has no control over the formatting of the string. For example, an application might want to format a number into a currency string, decimal string, percent string, or hexadecimal string. Second, the caller can't easily choose to format a string by using a specific culture. This second problem is more troublesome for server-side application code than for client-side code. On rare occasions, an application needs to format a string by using a culture other than the culture associated with the calling thread. To have more control over string formatting, you need a version of the `ToString` method that allows you to specify precise formatting and culture information.

Types that offer the caller a choice in formatting and culture implement the `System.IFormattable` interface.

```
public interface IFormattable {
   String ToString(String format, IFormatProvider formatProvider);
}
```

In the FCL, all of the base types (`Byte`, `SByte`, `Int16/UInt16`, `Int32/UInt32`, `Int64/UInt64`, `Single`, `Double`, `Decimal`, and `DateTime`) implement this interface. In addition, some other types, such as `Guid`, implement it. Finally, every enumerated type definition will automatically implement the `IFormattable` interface so that a meaningful string symbol from an instance of the enumerated type can be obtained.

`IFormattable`'s `ToString` method takes two parameters. The first, `format`, is a string that tells the method how the object should be formatted. `ToString`'s second parameter, `formatProvider`, is an instance of a type that implements the `System.IFormatProvider` interface. This type supplies specific culture information to the `ToString` method. I'll discuss how shortly.

The type implementing the `IFormattable` interface's `ToString` method determines which format strings it's going to recognize. If you pass a format string that the type doesn't recognize, the type is supposed to throw a `System.FormatException`.

Many of the types Microsoft has defined in the FCL recognize several formats. For example, the `DateTime` type supports "d" for short date, "D" for long date, "g" for general, "M" for month/day, "s" for sortable, "T" for long time, "u" for universal time in ISO 8601 format, "U" for universal time in full date format, "Y" for year/month, and others. All enumerated types support "G" for general, "F" for flags, "D" for decimal, and "X" for hexadecimal. I'll cover formatting enumerated types in more detail in Chapter 15, "Enumerated Types and Bit Flags."

Also, all of the built-in numeric types support "C" for currency, "D" for decimal, "E" for exponential (scientific) notation, "F" for fixed-point, "G" for general, "N" for number, "P" for percent, "R" for round-trip, and "X" for hexadecimal. In fact, the numeric types also support picture format strings just in case the simple format strings don't offer you exactly what you're looking for. Picture format strings contain special characters that tell the type's `ToString` method exactly how many digits to

show, exactly where to place a decimal separator, exactly how many digits to place after the decimal separator, and so on. For complete information about format strings, see "Formatting Types" in the .NET Framework SDK.

For most types, calling `ToString` and passing `null` for the format string is identical to calling `ToString` and passing "G" for the format string. In other words, objects format themselves using the "General format" by default. When implementing a type, choose a format that you think will be the most commonly used format; this format is the "General format." By the way, the `ToString` method that takes no parameters assumes that the caller wants the "General format."

So now that format strings are out of the way, let's turn to culture information. By default, strings are formatted using the culture information associated with the calling thread. The parameterless `ToString` method certainly does this, and so does `IFormattable`'s `ToString` if you pass `null` for the `formatProvider` parameter.

Culture-sensitive information applies when you're formatting numbers (including currency, integers, floating point, percentages, dates, and times). The `Guid` type has a `ToString` method that returns only a string representing its value. There's no need to consider a culture when generating the `Guid`'s string because GUIDs are used for programmatic purposes only.

When formatting a number, the `ToString` method sees what you've passed for the `formatProvider` parameter. If `null` is passed, `ToString` determines the culture associated with the calling thread by reading the `System.Globalization.CultureInfo.CurrentCulture` property. This property returns an instance of the `System.Globalization.CultureInfo` type.

Using this object, `ToString` reads its `NumberFormat` or `DateTimeFormat` property, depending on whether a number or date/time is being formatted. These properties return an instance of `System.Globalization.NumberFormatInfo` or `System.Globalization.DateTimeFormatInfo`, respectively. The `NumberFormatInfo` type defines a bunch of properties, such as `CurrencyDecimalSeparator`, `CurrencySymbol`, `NegativeSign`, `NumberGroupSeparator`, and `PercentSymbol`. Likewise, the `DateTimeFormatInfo` type defines an assortment of properties, such as `Calendar`, `DateSeparator`, `DayNames`, `LongDatePattern`, `ShortTimePattern`, and `TimeSeparator`. `ToString` reads these properties when constructing and formatting a string.

When calling `IFormattable`'s `ToString` method, instead of passing `null`, you can pass a reference to an object whose type implements the `IFormatProvider` interface.

```
public interface IFormatProvider {
   Object GetFormat(Type formatType);
}
```

Here's the basic idea behind the `IFormatProvider` interface: when a type implements this interface, it is saying that an instance of the type is able to provide culture-specific formatting information and that the culture information associated with the calling thread should be ignored.

The `System.Globalization.CultureInfo` type is one of the very few types defined in the FCL that implements the `IFormatProvider` interface. If you want to format a string for, say, Vietnam, you'd construct a `CultureInfo` object and pass that object in as `ToString`'s `formatProvider` parameter. The following code obtains a string representation of a `Decimal` numeric value formatted as currency appropriate for Vietnam.

```
Decimal price = 123.54M;
String s = price.ToString("C", new CultureInfo("vi-VN"));
MessageBox.Show(s);
```

If you build and run this code, the message box shown in Figure 14-5 appears.

FIGURE 14-5 Numeric value formatted correctly to represent Vietnamese currency.

Internally, `Decimal`'s `ToString` method sees that the `formatProvider` argument is not `null` and calls the object's `GetFormat` method as follows.

```
NumberFormatInfo nfi = (NumberFormatInfo)
   formatProvider.GetFormat(typeof(NumberFormatInfo));
```

This is how `ToString` requests the appropriate number-formatting information from the (`CultureInfo`) object. Number types (such as `Decimal`) request only number-formatting information. But other types (such as `DateTime`) could call `GetFormat` like the following.

```
DateTimeFormatInfo dtfi = (DateTimeFormatInfo)
   formatProvider.GetFormat(typeof(DateTimeFormatInfo));
```

Actually, because `GetFormat`'s parameter can identify any type, the method is flexible enough to allow any type of format information to be requested. The types in the .NET Framework call `GetFormat`, requesting only number or date/time information; in the future, other kinds of formatting information could be requested.

By the way, if you want to obtain a string for an object that isn't formatted for any particular culture, you should call `System.Globalization.CultureInfo`'s static `InvariantCulture` property and pass the object returned as `ToString`'s `formatProvider` parameter.

```
Decimal price = 123.54M;
String s = price.ToString("C", CultureInfo.InvariantCulture);
MessageBox.Show(s);
```

If you build and run this code, the message box shown in Figure 14-6 appears. Notice the first character in the resulting string: ¤. This is the international sign for currency (U+00A4).

FIGURE 14-6 Numeric value formatted to represent a culture-neutral currency.

Normally, you wouldn't display a string formatted by using the invariant culture to a user. Typically, you'd just save this string in a data file so that it could be parsed later.

In the FCL, just three types implement the `IFormatProvider` interface. The first is `CultureInfo`, which I've already explained. The other two are `NumberFormatInfo` and `DateTimeFormatInfo`. When `GetFormat` is called on a `NumberFormatInfo` object, the method checks whether the type being requested is a `NumberFormatInfo`. If it is, `this` is returned; if it's not, `null` is returned. Similarly, calling `GetFormat` on a `DateTimeFormatInfo` object returns `this` if a `DateTimeFormatInfo` is requested and `null` if it's not. These two types implement this interface simply as a programming convenience. When trying to obtain a string representation of an object, the caller commonly specifies a format and uses the culture associated with the calling thread. For this reason, you often call `ToString`, passing a string for the format parameter and `null` for the `formatProvider` parameter. To make calling `ToString` easier for you, many types offer several overloads of the `ToString` method. For example, the `Decimal` type offers four different `ToString` methods.

```
// This version calls ToString(null, null).
// Meaning: General numeric format, thread's culture information
public override String ToString();

// This version is where the actual implementation of ToString goes.
// This version implements IFormattable's ToString method.
// Meaning: Caller-specified format and culture information
public String ToString(String format, IFormatProvider formatProvider);

// This version simply calls ToString(format, null).
// Meaning: Caller-specified format, thread's culture information
public String ToString(String format);

// This version simply calls ToString(null, formatProvider).
// This version implements IConvertible's ToString method.
// Meaning: General format, caller-specified culture information
public String ToString(IFormatProvider formatProvider);
```

Formatting Multiple Objects into a Single String

So far, I've explained how an individual type formats its own objects. At times, however, you want to construct strings consisting of many formatted objects. For example, the following string has a date, a person's name, and an age.

```
String s = String.Format("On {0}, {1} is {2} years old.",
   new DateTime(2012, 4, 22, 14, 35, 5), "Aidan", 9);
Console.WriteLine(s);
```

If you build and run this code where "en-US" is the thread's current culture, you'll see the following line of output.

```
On 4/22/2012 2:35:05 PM, Aidan is 9 years old.
```

`String`'s static `Format` method takes a format string that identifies replaceable parameters by using numbers in braces. The format string used in this example tells the `Format` method to replace `{0}` with the first parameter after the format string (the date/time), replace `{1}` with the second parameter after the format string ("Aidan"), and replace `{2}` with the third parameter after the format string (7).

Internally, the `Format` method calls each object's `ToString` method to obtain a string representation for the object. Then the returned strings are all appended and the complete, final string is returned. This is all fine and good, but it means that all of the objects are formatted by using their general format and the calling thread's culture information.

You can have more control when formatting an object if you specify format information within braces. For example, the following code is identical to the previous example except that I've added formatting information to replaceable parameters 0 and 2.

```
String s = String.Format("On {0:D}, {1} is {2:E} years old.",
   new DateTime(2012, 4, 22, 14, 35, 5), "Aidan", 9);
Console.WriteLine(s);
```

If you build and run this code where "en-US" is the thread's current culture, you'll see the following line of output.

```
On Sunday, April 22, 2012, Aidan is 9.000000E+000 years old.
```

When the `Format` method parses the format string, it sees that replaceable parameter 0 should have its `IFormattable` interface's `ToString` method called passing `"D"` and `null` for its two parameters. Likewise, `Format` calls replaceable parameter 2's `IFormattable ToString` method, passing `"E"` and `null`. If the type doesn't implement the `IFormattable` interface, `Format` calls its parameterless `ToString` method inherited from `Object` (and possibly overridden), and the default format is appended into the resulting string.

The `String` class offers several overloads of the static `Format` method. One version takes an object that implements the `IFormatProvider` interface so that you can format all of the replaceable parameters by using caller-specified culture information. Obviously, `Format` calls each object's `IFormattable.ToString` method, passing it whatever `IFormatProvider` object was passed to `Format`.

If you're using `StringBuilder` instead of `String` to construct a string, you can call `StringBuilder`'s `AppendFormat` method. This method works exactly as `String`'s `Format` method except that it formats a string and appends to the `StringBuilder`'s character array. As does `String`'s `Format`, `AppendFormat` takes a format string, and there's a version that takes an `IFormatProvider`.

`System.Console` offers `Write` and `WriteLine` methods that also take format strings and replaceable parameters. However, there are no overloads of `Console`'s `Write` and `WriteLine` methods that allow you to pass an `IFormatProvider`. If you want to format a string for a specific culture, you have to call `String`'s `Format` method, first passing the desired `IFormatProvider` object and then passing the resulting string to `Console`'s `Write` or `WriteLine` method. This shouldn't be a big deal because, as I said earlier, it's rare for client-side code to format a string by using a culture other than the one associated with the calling thread.

Providing Your Own Custom Formatter

By now it should be clear that the formatting capabilities in the .NET Framework were designed to offer you a great deal of flexibility and control. However, we're not quite finished. It's possible for you to define a method that `StringBuilder`'s `AppendFormat` method will call whenever any object is being formatted into a string. In other words, instead of calling `ToString` for each object, `AppendFormat` can call a function you define, allowing you to format any or all of the objects in any way you want. What I'm about to describe also works with `String`'s `Format` method.

Let me explain this mechanism by way of an example. Let's say that you're formatting HTML text that a user will view in an Internet browser. You want all `Int32` values to appear in bold. To accomplish this, every time an `Int32` value is formatted into a `String`, you want to surround the string with HTML bold tags: <B> and </B>. The following code demonstrates how easy it is to do this.

```
using System;
using System.Text;
using System.Threading;

public static class Program {
   public static void Main() {
      StringBuilder sb = new StringBuilder();
      sb.AppendFormat(new BoldInt32s(), "{0} {1} {2:M}", "Jeff", 123, DateTime.Now);
      Console.WriteLine(sb);
   }
}

internal sealed class BoldInt32s : IFormatProvider, ICustomFormatter {
   public Object GetFormat(Type formatType) {
      if (formatType == typeof(ICustomFormatter)) return this;
      return Thread.CurrentThread.CurrentCulture.GetFormat(formatType);
   }

   public String Format(String format, Object arg, IFormatProvider formatProvider) {
      String s;

      IFormattable formattable = arg as IFormattable;
```

```
      if (formattable == null) s = arg.ToString();
      else s = formattable.ToString(format, formatProvider);

      if (arg.GetType() == typeof(Int32))
         return "<B>" + s + "</B>";
      return s;
   }
}
```

When you compile and run this code where "en-US" is the thread's current culture, it displays the following output (your date may be different, of course).

```
Jeff <B>123</B> September 1
```

In `Main`, I'm constructing an empty `StringBuilder` and then appending a formatted string into it. When I call `AppendFormat`, the first parameter is an instance of the `BoldInt32s` class. This class implements the `IFormatProvider` interface that I discussed earlier. In addition, this class implements the `ICustomFormatter` interface.

```
public interface ICustomFormatter {
   String Format(String format, Object arg,
      IFormatProvider formatProvider);
}
```

This interface's `Format` method is called whenever `StringBuilder`'s `AppendFormat` needs to obtain a string for an object. You can do some pretty clever things inside this method that give you a great deal of control over string formatting. Let's look inside the `AppendFormat` method to see exactly how it works. The following pseudocode shows how `AppendFormat` works.

```
public StringBuilder AppendFormat(IFormatProvider formatProvider,
   String format, params Object[] args) {

   // If an IFormatProvider was passed, find out
   // whether it offers an ICustomFormatter object.
   ICustomFormatter cf = null;

   if (formatProvider != null)
      cf = (ICustomFormatter)
         formatProvider.GetFormat(typeof(ICustomFormatter));

   // Keep appending literal characters (not shown in this pseudocode)
   // and replaceable parameters to the StringBuilder's character array.
   Boolean MoreReplaceableArgumentsToAppend = true;
   while (MoreReplaceableArgumentsToAppend) {
      // argFormat refers to the replaceable format string obtained
      // from the format parameter
      String argFormat = /* ... */;

      // argObj refers to the corresponding element
      // from the args array parameter
      Object argObj = /* ... */;
```

```
      // argStr will refer to the formatted string to be appended
      // to the final, resulting string
      String argStr = null;

      // If a custom formatter is available, let it format the argument.
      if (cf != null)
         argStr = cf.Format(argFormat, argObj, formatProvider);

      // If there is no custom formatter or if it didn't format
      // the argument, try something else.
      if (argStr == null) {
         // Does the argument's type support rich formatting?
         IFormattable formattable = argObj as IFormattable;
         if (formattable != null) {
            // Yes; pass the format string and provider to
            // the type's IFormattable ToString method.
            argStr = formattable.ToString(argFormat, formatProvider);
         } else {
            // No; get the default format by using
            // the thread's culture information.
            if (argObj != null) argStr = argObj.ToString();
            else argStr = String.Empty;
         }
      }
      // Append argStr's characters to the character array field member.
      /* ... */

      // Check if any remaining parameters to format
      MoreReplaceableArgumentsToAppend = /* ... */;
   }
   return this;
}
```

When `Main` calls `AppendFormat`, `AppendFormat` calls my format provider's `GetFormat` method, passing it the `ICustomFormatter` type. The `GetFormat` method defined in my `BoldInt32s` type sees that the `ICustomFormatter` is being requested and returns a reference to itself because it implements this interface. If my `GetFormat` method is called and is passed any other type, I call the `GetFormat` method of the `CultureInfo` object associated with the calling thread.

Whenever `AppendFormat` needs to format a replaceable parameter, it calls `ICustomFormatter`'s `Format` method. In my example, `AppendFormat` calls the `Format` method defined by my `BoldInt32s` type. In my `Format` method, I check whether the object being formatted supports rich formatting via the `IFormattable` interface. If the object doesn't, I then call the simple, parameterless `ToString` method (inherited from `Object`) to format the object. If the object does support `IFormattable`, I then call the rich `ToString` method, passing it the format string and the format provider.

Now that I have the formatted string, I check whether the corresponding object is an `Int32` type, and if it is, I wrap the formatted string in <B> and </B> HTML tags and return the new string. If the object is not an `Int32`, I simply return the formatted string without any further processing.

Parsing a String to Obtain an Object: Parse

In the preceding section, I explained how to take an object and obtain a string representation of that object. In this section, I'll talk about the opposite: how to take a string and obtain an object representation of it. Obtaining an object from a string isn't a very common operation, but it does occasionally come in handy. Microsoft felt it necessary to formalize a mechanism by which strings can be parsed into objects.

Any type that can parse a string offers a public, static method called `Parse`. This method takes a `String` and returns an instance of the type; in a way, `Parse` acts as a factory. In the FCL, a `Parse` method exists on all of the numeric types as well as for `DateTime`, `TimeSpan`, and a few other types (such as the SQL data types).

Let's look at how to parse a string into a number type. Almost all of the numeric types (`Byte`, `SByte`, `Int16/UInt16`, `Int32/UInt32`, `Int64/UInt64`, `Single`, `Double`, `Decimal`, and `BigInteger`) offer at least one `Parse` method. Here I'll show you just the `Parse` method defined by the `Int32` type. (The `Parse` methods for the other numeric types work similarly to `Int32`'s `Parse` method.)

```
public static Int32 Parse(String s, NumberStyles style,
   IFormatProvider provider);
```

Just from looking at the prototype, you should be able to guess exactly how this method works. The `String` parameter, `s`, identifies a string representation of a number you want parsed into an `Int32` object. The `System.Globalization.NumberStyles` parameter, `style`, is a set of bit flags that identify characters that `Parse` should expect to find in the string. And the `IFormatProvider` parameter, `provider`, identifies an object that the `Parse` method can use to obtain culture-specific information, as discussed earlier in this chapter.

For example, the following code causes `Parse` to throw a `System.FormatException` because the string being parsed contains a leading space.

```
Int32 x = Int32.Parse(" 123", NumberStyles.None, null);
```

To allow `Parse` to skip over the leading space, change the `style` parameter as follows.

```
Int32 x = Int32.Parse(" 123", NumberStyles.AllowLeadingWhite, null);
```

See the .NET Framework SDK documentation for a complete description of the bit symbols and common combinations that the `NumberStyles` enumerated type defines.

Here's a code fragment showing how to parse a hexadecimal number.

```
Int32 x = Int32.Parse("1A", NumberStyles.HexNumber, null);
Console.WriteLine(x);  // Displays "26"
```

This Parse method accepts three parameters. For convenience, many types offer additional overloads of Parse so you don't have to pass as many arguments. For example, Int32 offers four overloads of the Parse method.

```
// Passes NumberStyles.Integer for style
// and thread's culture's provider information.
public static Int32 Parse(String s);

// Passes thread's culture's provider information.
public static Int32 Parse(String s, NumberStyles style);

// Passes NumberStyles.Integer for the style parameter.
public static Int32 Parse(String s, IFormatProvider provider);

// This is the method I've been talking about in this section.
public static Int32 Parse(String s, NumberStyles style,
   IFormatProvider provider);
```

The DateTime type also offers a Parse method.

```
public static DateTime Parse(String s,
   IFormatProvider provider, DateTimeStyles styles);
```

This method works just as the Parse method defined on the number types except that DateTime's Parse method takes a set of bit flags defined by the System.Globalization.DateTimeStyles enumerated type instead of the NumberStyles enumerated type. See the .NET Framework SDK documentation for a complete description of the bit symbols and common combinations the DateTimeStyles type defines.

For convenience, the DateTime type offers three overloads of the Parse method.

```
// Passes thread's culture's provider information
// and DateTimeStyles.None for the style
public static DateTime Parse(String s);

// Passes DateTimeStyles.None for the style
public static DateTime Parse(String s, IFormatProvider provider);

// This is the method I've been talking about in this section.
public static DateTime Parse(String s,
   IFormatProvider provider, DateTimeStyles styles);
```

Parsing dates and times is complex. Many developers have found the Parse method of the DateTime type too forgiving in that it sometimes parses strings that don't contain dates or times. For this reason, the DateTime type also offers a ParseExact method that accepts a picture format string that indicates exactly how the date/time string should be formatted and how it should be parsed. For more information about picture format strings, see the DateTimeFormatInfo class in the .NET Framework SDK.

Note Some developers have reported the following back to Microsoft: when their application calls `Parse` frequently, and `Parse` throws exceptions repeatedly (due to invalid user input), performance of the application suffers. For these performance-sensitive uses of `Parse`, Microsoft added `TryParse` methods to all of the numeric data types, `DateTime`, `DateTimeOffset`, `TimeSpan`, and even `IPAddress`. This is what one of the two `Int32`'s two `TryParse` method overloads looks like.

```
public static Boolean TryParse(String s, NumberStyles style,
   IFormatProvider provider, out Int32 result);
```

As you can see, this method returns `true` or `false` indicating whether the specified string can be parsed into an `Int32`. If the method returns `true`, the variable passed by reference to the result parameter will contain the parsed numeric value. The `TryXxx` pattern is discussed in Chapter 20, "Exceptions and State Management."

Encodings: Converting Between Characters and Bytes

In Win32, programmers all too frequently have to write code to convert Unicode characters and strings to Multi-Byte Character Set (MBCS) characters and strings. I've certainly written my share of this code, and it's very tedious to write and error-prone to use. In the CLR, all characters are represented as 16-bit Unicode code values and all strings are composed of 16-bit Unicode code values. This makes working with characters and strings easy at run time.

At times, however, you want to save strings to a file or transmit them over a network. If the strings consist mostly of characters readable by English-speaking people, saving or transmitting a set of 16-bit values isn't very efficient because half of the bytes written would contain zeros. Instead, it would be more efficient to encode the 16-bit values into a compressed array of bytes and then decode the array of bytes back into an array of 16-bit values.

Encodings also allow a managed application to interact with strings created by non-Unicode systems. For example, if you want to produce a file readable by an application running on a Japanese version of Windows 95, you have to save the Unicode text by using the Shift-JIS (code page 932) encoding. Likewise, y ou'd use Shift-JIS encoding to read a text file produced on a Japanese Windows 95 system into the CLR.

Encoding is typically done when you want to send a string to a file or network stream by using the `System.IO.BinaryWriter` or `System.IO.StreamWriter` type. Decoding is typically done when you want to read a string from a file or network stream by using the `System.IO.BinaryReader` or `System.IO.StreamReader` type. If you don't explicitly select an encoding, all of these types default to using UTF-8. (UTF stands for Unicode Transformation Format.) However, at times, you might want to explicitly encode or decode a string. Even if you don't want to explicitly do this, this section will give you more insight into the reading and writing of strings from and to streams.

Fortunately, the FCL offers some types to make character encoding and decoding easy. The two most frequently used encodings are UTF-16 and UTF-8:

- UTF-16 encodes each 16-bit character as 2 bytes. It doesn't affect the characters at all, and no compression occurs—its performance is excellent. UTF-16 encoding is also referred to as Unicode encoding. Also note that UTF-16 can be used to convert from little-endian to big-endian and vice versa.

- UTF-8 encodes some characters as 1 byte, some characters as 2 bytes, some characters as 3 bytes, and some characters as 4 bytes. Characters with a value below 0x0080 are compressed to 1 byte, which works very well for characters used in the United States. Characters between 0x0080 and 0x07FF are converted to 2 bytes, which works well for European and Middle Eastern languages. Characters of 0x0800 and above are converted to 3 bytes, which works well for East Asian languages. Finally, surrogate pairs are written out as 4 bytes. UTF-8 is an extremely popular encoding, but it's less efficient than UTF-16 if you encode many characters with values of 0x0800 or above.

Although the UTF-16 and UTF-8 encodings are by far the most common, the FCL also supports some encodings that are used less frequently:

- UTF-32 encodes all characters as 4 bytes. This encoding is useful when you want to write a simple algorithm to traverse characters and you don't want to have to deal with characters taking a variable number of bytes. For example, with UTF-32, you do not need to think about surrogates because every character is 4 bytes. Obviously, UTF-32 is not an efficient encoding in terms of memory usage and is therefore rarely used for saving or transmitting strings to a file or network. This encoding is typically used inside the program itself. Also note that UTF-32 can be used to convert from little-endian to big-endian and vice versa.

- UTF-7 encoding is typically used with older systems that work with characters that can be expressed using 7-bit values. You should avoid this encoding because it usually ends up expanding the data rather than compressing it. The Unicode Consortium has deprecated this encoding.

- ASCII encodes the 16-bit characters into ASCII characters; that is, any 16-bit character with a value of less than 0x0080 is converted to a single byte. Any character with a value greater than 0x007F can't be converted, so that character's value is lost. For strings consisting of characters in the ASCII range (0x00 to 0x7F), this encoding compresses the data in half and is very fast (because the high byte is just cut off). This encoding isn't appropriate if you have characters outside of the ASCII range because the character's values will be lost.

Finally, the FCL also allows you to encode 16-bit characters to an arbitrary code page. As with the ASCII encoding, encoding to a code page is dangerous because any character whose value can't be expressed in the specified code page is lost. You should always use UTF-16 or UTF-8 encoding unless you must work with some legacy files or applications that already use one of the other encodings.

When you need to encode or decode a set of characters, you should obtain an instance of a class derived from `System.Text.Encoding`. `Encoding` is an abstract base class that offers several static `readonly` properties, each of which returns an instance of an `Encoding`-derived class.

Here's an example that encodes and decodes characters by using UTF-8.

```
using System;
using System.Text;

public static class Program {
   public static void Main() {
      // This is the string we're going to encode.
      String s = "Hi there.";

      // Obtain an Encoding-derived object that knows how
      // to encode/decode using UTF8
      Encoding encodingUTF8 = Encoding.UTF8;

      // Encode a string into an array of bytes.
      Byte[] encodedBytes = encodingUTF8.GetBytes(s);

      // Show the encoded byte values.
      Console.WriteLine("Encoded bytes: " +
         BitConverter.ToString(encodedBytes));

      // Decode the byte array back to a string.
      String decodedString = encodingUTF8.GetString(encodedBytes);

      // Show the decoded string.
      Console.WriteLine("Decoded string: " + decodedString);
   }
}
```

This code yields the following output.

```
Encoded bytes: 48-69-20-74-68-65-72-65-2E
Decoded string: Hi there.
```

In addition to the UTF8 static property, the `Encoding` class also offers the following static properties: `Unicode`, `BigEndianUnicode`, UTF32, UTF7, ASCII, and `Default`. The `Default` property returns an object that is able to encode/decode using the user's code page as specified by the Language For Non-Unicode Programs option of the Region/Administrative dialog box in Control Panel. (See the `GetACP` Win32 function for more information.) However, using the `Default` property is discouraged because your application's behavior would be machine-setting dependent, so if you change the system's default code page or if your application runs on another machine, your application will behave differently.

In addition to these properties, `Encoding` also offers a static `GetEncoding` method that allows you to specify a code page (by integer or by string) and returns an object that can encode/decode using the specified code page. You can call `GetEncoding`, passing `"Shift-JIS"` or `932`, for example.

When you first request an encoding object, the `Encoding` class's property or `GetEncoding` method constructs a single object for the requested encoding and returns this object. If an already-requested encoding object is requested in the future, the encoding class simply returns the object it previously constructed; it doesn't construct a new object for each request. This efficiency reduces the number of objects in the system and reduces pressure in the garbage-collected heap.

Instead of calling one of `Encoding`'s static properties or its `GetEncoding` method, you could also construct an instance of one of the following classes: `System.Text.UnicodeEncoding`, `System.Text.UTF8Encoding`, `System.Text.UTF32Encoding`, `System.Text.UTF7Encoding`, or `System.Text.ASCIIEncoding`. However, keep in mind that constructing any of these classes creates new objects in the managed heap, which hurts performance.

Four of these classes, `UnicodeEncoding`, `UTF8Encoding`, `UTF32Encoding`, and `UTF7Encoding`, offer multiple constructors, providing you with more control over the encoding and preamble. (Preamble is sometimes referred to as a *byte order mark* or *BOM*.) The first three aforementioned classes also offer constructors that let you tell the class to throw exceptions when decoding an invalid byte sequence; you should use these constructors when you want your application to be secure and resistant to invalid incoming data.

You might want to explicitly construct instances of these encoding types when working with a `BinaryWriter` or a `StreamWriter`. The `ASCIIEncoding` class has only a single constructor and therefore doesn't offer any more control over the encoding. If you need an `ASCIIEncoding` object, always obtain it by querying `Encoding`'s `ASCII` property; this returns a reference to a single `ASCIIEncoding` object. If you construct `ASCIIEncoding` objects yourself, you are creating more objects on the heap, which hurts your application's performance.

After you have an `Encoding`-derived object, you can convert a string or an array of characters to an array of bytes by calling the `GetBytes` method. (Several overloads of this method exist.) To convert an array of bytes to an array of characters or a string, call the `GetChars` method or the more useful `GetString` method. (Several overloads exist for both of these methods.) The preceding code demonstrated calls to the `GetBytes` and `GetString` methods.

All `Encoding`-derived types offer a `GetByteCount` method that obtains the number of bytes necessary to encode a set of characters without actually encoding. Although `GetByteCount` isn't especially useful, you can use this method to allocate an array of bytes. There's also a `GetCharCount` method that returns the number of characters that would be decoded without actually decoding them. These methods are useful if you're trying to save memory and reuse an array.

The `GetByteCount`/`GetCharCount` methods aren't that fast because they must analyze the array of characters/bytes in order to return an accurate result. If you prefer speed to an exact result, you can call the `GetMaxByteCount` or `GetMaxCharCount` method instead. Both methods take an integer specifying the number of bytes or number of characters and return a worst-case value.

Each `Encoding`-derived object offers a set of public read-only properties that you can query to obtain detailed information about the encoding. See the .NET Framework SDK documentation for a description of these properties.

To illustrate most of the properties and their meanings, I wrote the following program that displays the property values for several different encodings.

```
using System;
using System.Text;

public static class Program {
   public static void Main() {
      foreach (EncodingInfo ei in Encoding.GetEncodings()) {
         Encoding e = ei.GetEncoding();
         Console.WriteLine("{1}{0}" +
            "\tCodePage={2}, WindowsCodePage={3}{0}" +
            "\tWebName={4}, HeaderName={5}, BodyName={6}{0}" +
            "\tIsBrowserDisplay={7}, IsBrowserSave={8}{0}" +
            "\tIsMailNewsDisplay={9}, IsMailNewsSave={10}{0}",

            Environment.NewLine,
            e.EncodingName, e.CodePage, e.WindowsCodePage,
            e.WebName, e.HeaderName, e.BodyName,
            e.IsBrowserDisplay, e.IsBrowserSave,
            e.IsMailNewsDisplay, e.IsMailNewsSave);
      }
   }
}
```

Running this program yields the following output (abridged to conserve paper).

```
IBM EBCDIC (US-Canada)
        CodePage=37, WindowsCodePage=1252
        WebName=IBM037, HeaderName=IBM037, BodyName=IBM037
        IsBrowserDisplay=False, IsBrowserSave=False
        IsMailNewsDisplay=False, IsMailNewsSave=False

OEM United States
        CodePage=437, WindowsCodePage=1252
        WebName=IBM437, HeaderName=IBM437, BodyName=IBM437
        IsBrowserDisplay=False, IsBrowserSave=False
        IsMailNewsDisplay=False, IsMailNewsSave=False

IBM EBCDIC (International)
        CodePage=500, WindowsCodePage=1252
        WebName=IBM500, HeaderName=IBM500, BodyName=IBM500
        IsBrowserDisplay=False, IsBrowserSave=False
        IsMailNewsDisplay=False, IsMailNewsSave=False

Arabic (ASMO 708)
        CodePage=708, WindowsCodePage=1256
        WebName=ASMO-708, HeaderName=ASMO-708, BodyName=ASMO-708
        IsBrowserDisplay=True, IsBrowserSave=True
        IsMailNewsDisplay=False, IsMailNewsSave=False

Unicode
        CodePage=1200, WindowsCodePage=1200
        WebName=utf-16, HeaderName=utf-16, BodyName=utf-16
        IsBrowserDisplay=False, IsBrowserSave=True
        IsMailNewsDisplay=False, IsMailNewsSave=False
```

```
Unicode (Big-Endian)
        CodePage=1201, WindowsCodePage=1200
        WebName=unicodeFFFE, HeaderName=unicodeFFFE, BodyName=unicodeFFFE
        IsBrowserDisplay=False, IsBrowserSave=False
        IsMailNewsDisplay=False, IsMailNewsSave=False

Western European (DOS)
        CodePage=850, WindowsCodePage=1252
        WebName=ibm850, HeaderName=ibm850, BodyName=ibm850
        IsBrowserDisplay=False, IsBrowserSave=False
        IsMailNewsDisplay=False, IsMailNewsSave=False

Unicode (UTF-8)
        CodePage=65001, WindowsCodePage=1200
        WebName=utf-8, HeaderName=utf-8, BodyName=utf-8
        IsBrowserDisplay=True, IsBrowserSave=True
        IsMailNewsDisplay=True, IsMailNewsSave=True
```

Table 14-3 covers the most commonly used methods offered by all `Encoding`-derived classes.

TABLE 14-3 Methods of the `Encoding`-Derived Classes

Method	Description
`GetPreamble`	Returns an array of bytes indicating what should be written to a stream before writing any encoded bytes. Frequently, these bytes are referred to as BOM bytes. When you start reading from a stream, the BOM bytes automatically help detect the encoding that was used when the stream was written so that the correct decoder can be used. For some `Encoding`-derived classes, this method returns an array of 0 bytes—that is, no preamble bytes. A `UTF8Encoding` object can be explicitly constructed so that this method returns a 3-byte array of 0xEF, 0xBB, 0xBF. A `UnicodeEncoding` object can be explicitly constructed so that this method returns a 2-byte array of 0xFE, 0xFF for big-endian encoding or a 2-byte array of 0xFF, 0xFE for little-endian encoding. The default is little-endian.
`Convert`	Converts an array of bytes specified in a source encoding to an array of bytes specified by a destination encoding. Internally, this static method calls the source encoding object's `GetChars` method and passes the result to the destination encoding object's `GetBytes` method. The resulting byte array is returned to the caller.
`Equals`	Returns `true` if two `Encoding`-derived objects represent the same code page and preamble setting.
`GetHashCode`	Returns the encoding object's code page.

Encoding and Decoding Streams of Characters and Bytes

Imagine that you're reading a UTF-16 encoded string via a `System.Net.Sockets.NetworkStream` object. The bytes will very likely stream in as chunks of data. In other words, you might first read 5 bytes from the stream, followed by 7 bytes. In UTF-16, each character consists of 2 bytes. So calling `Encoding`'s `GetString` method passing the first array of 5 bytes will return a string consisting of just two characters. If you later call `GetString`, passing in the next 7 bytes that come in from the stream, `GetString` will return a string consisting of three characters, and all of the code points will have the wrong values!

This data corruption problem occurs because none of the Encoding-derived classes maintains any state in between calls to their methods. If you'll be encoding or decoding characters/bytes in chunks, you must do some additional work to maintain state between calls, preventing any loss of data.

To decode chunks of bytes, you should obtain a reference to an Encoding-derived object (as described in the previous section) and call its GetDecoder method. This method returns a reference to a newly constructed object whose type is derived from the System.Text.Decoder class. Like the Encoding class, the Decoder class is an abstract base class. If you look in the .NET Framework SDK documentation, you won't find any classes that represent concrete implementations of the Decoder class. However, the FCL does define a bunch of Decoder-derived classes. These classes are all internal to the FCL, but the GetDecoder method can construct instances of these classes and return them to your application code.

All Decoder-derived classes offer two important methods: GetChars and GetCharCount. Obviously, these methods are used for decoding an array of bytes and work similarly to Encoding's GetChars and GetCharCount methods, discussed earlier. When you call one of these methods, it decodes the byte array as much as possible. If the byte array doesn't contain enough bytes to complete a character, the leftover bytes are saved inside the decoder object. The next time you call one of these methods, the decoder object uses the leftover bytes plus the new byte array passed to it—this ensures that the chunks of data are decoded properly. Decoder objects are very useful when reading bytes from a stream.

An Encoding-derived type can be used for stateless encoding and decoding. However, a Decoder-derived type can be used only for decoding. If you want to encode strings in chunks, call GetEncoder instead of calling the Encoding object's GetDecoder method. GetEncoder returns a newly constructed object whose type is derived from the abstract base class System.Text.Encoder. Again, the .NET Framework SDK documentation doesn't contain any classes representing concrete implementations of the Encoder class. However, the FCL does define some Encoder-derived classes. As with the Decoder-derived classes, these classes are all internal to the FCL, but the GetEncoder method can construct instances of these classes and return them to your application code.

All Encoder-derived classes offer two important methods: GetBytes and GetByteCount. On each call, the Encoder-derived object maintains any leftover state information so that you can encode data in chunks.

Base-64 String Encoding and Decoding

As of this writing, the UTF-16 and UTF-8 encodings are quite popular. It is also quite popular to encode a sequence of bytes to a base-64 string. The FCL does offer methods to do base-64 encoding and decoding, and you might expect that this would be accomplished via an Encoding-derived type. However, for some reason, base-64 encoding and decoding is done using some static methods offered by the System.Convert type.

To encode a base-64 string as an array of bytes, you call Convert's static FromBase64String or FromBase64CharArray method. Likewise, to decode an array of bytes as a base-64 string, you call

Convert's static ToBase64String or ToBase64CharArray method. The following code demonstrates how to use some of these methods.

```
using System;

public static class Program {
   public static void Main() {
      // Get a set of 10 randomly generated bytes
      Byte[] bytes = new Byte[10];
      new Random().NextBytes(bytes);

      // Display the bytes
      Console.WriteLine(BitConverter.ToString(bytes));

      // Decode the bytes into a base-64 string and show the string
      String s = Convert.ToBase64String(bytes);
      Console.WriteLine(s);

      // Encode the base-64 string back to bytes and show the bytes
      bytes = Convert.FromBase64String(s);
      Console.WriteLine(BitConverter.ToString(bytes));
   }
}
```

Compiling this code and running the executable file produces the following output (your output might vary from mine because of the randomly generated bytes).

```
3B-B9-27-40-59-35-86-54-5F-F1
O7knQFk1hlRf8Q==
3B-B9-27-40-59-35-86-54-5F-F1
```

Secure Strings

Often, `String` objects are used to contain sensitive data such as a user's password or credit-card information. Unfortunately, `String` objects contain an array of characters in memory, and if some unsafe or unmanaged code is allowed to execute, the unsafe/unmanaged code could snoop around the process's address space, locate the string containing the sensitive information, and use this data in an unauthorized way. Even if the `String` object is used for just a short time and then garbage collected, the CLR might not immediately reuse the `String` object's memory (especially if the `String` object was in an older generation), leaving the `String`'s characters in the process's memory, where the information could be compromised. In addition, because strings are immutable, as you manipulate them, the old copies linger in memory and you end up with different versions of the string scattered all over memory.

Some governmental departments have stringent security requirements that require very specific security guarantees. To meet these requirements, Microsoft added a more secure string class to the FCL: `System.Security.SecureString`. When you construct a `SecureString` object, it internally allocates a block of unmanaged memory that contains an array of characters. Unmanaged memory is used so that the garbage collector isn't aware of it.

These string's characters are encrypted, protecting the sensitive information from any malicious unsafe/unmanaged code. You can append, insert, remove, or set a character in the secure string by using any of these methods: `AppendChar`, `InsertAt`, `RemoveAt`, and `SetAt`. Whenever you call any of these methods, internally, the method decrypts the characters, performs the operation in place, and then re-encrypts the characters. This means that the characters are in an unencrypted state for a very short period of time. This also means that the performance of each operation is less than stellar, so you should perform as few of these operations as possible.

The `SecureString` class implements the `IDisposable` interface to provide an easy way to deterministically destroy the string's secured contents. When your application no longer needs the sensitive string information, you simply call `SecureString`'s `Dispose` method or use a `SecureString` instance in a `using` construct. Internally, `Dispose` zeroes out the contents of the memory buffer to make sure that the sensitive information is not accessible to malicious code, and then the buffer is freed. Internally, a `SecureString` object has a field to a `SafeBuffer`-derived object, which maintains the actual string. Because the `SafeBuffer` class is ultimately derived from `CriticalFinalizerObject`, discussed in Chapter 21, "The Managed Heap and Garbage Collection," the string's characters are guaranteed to be zeroed out and have its buffer freed when it is finalized. Unlike a `String` object, when a `SecureString` object is finalized, the encrypted string's characters will no longer be in memory.

Now that you know how to create and modify a `SecureString` object, let's talk about how to use one. Unfortunately, the most recent FCL has limited support for the `SecureString` class. In other words, there are only a few methods that accept a `SecureString` argument. In the .NET Framework 4, you can pass a `SecureString` as a password when

- Working with a cryptographic service provider (CSP). See the `System.Security.Cryptography.CspParameters` class.
- Creating, importing, or exporting an X.509 certificate. See the `System.Security.Cryptography.X509Certificates.X509Certificate` and `System.Security.Cryptography.X509Certificates.X509Certificate2` classes.
- Starting a new process under a specific user account. See the `System.Diagnostics.Process` and `System.Diagnostics.ProcessStartInfo` classes.
- Constructing an event log session. See the `System.Diagnostics.Eventing.Reader.EventLogSession` class.
- Using the `System.Windows.Controls.PasswordBox` control. See this class's `SecurePassword` property.

Finally, you can create your own methods that can accept a `SecureString` object parameter. Inside your method, you must have the `SecureString` object create an unmanaged memory buffer that contains the decrypted characters before your method uses the buffer. To keep the window of opportunity for malicious code to access the sensitive data as small as possible, your code should require access to the decrypted string for as short a period of time as possible. When finished using the string, your code should zero the buffer and free it as soon as possible. Also, never put the contents

of a SecureString into a String: if you do, the String lives unencrypted in the heap and will not have its characters zeroed out until the memory is reused after a garbage collection. The SecureString class does not override the ToString method specifically to avoid exposing the sensitive data (which converting it to a String would do).

Here is some sample code demonstrating how to initialize and use a SecureString (when compiling this, you'll need to specify the /unsafe switch to the C# compiler).

```
using System;
using System.Security;
using System.Runtime.InteropServices;

public static class Program {
   public static void Main() {
      using (SecureString ss = new SecureString()) {
         Console.Write("Please enter password: ");
         while (true) {
            ConsoleKeyInfo cki = Console.ReadKey(true);
            if (cki.Key == ConsoleKey.Enter) break;

            // Append password characters into the SecureString
            ss.AppendChar(cki.KeyChar);
            Console.Write("*");
         }
         Console.WriteLine();

         // Password entered, display it for demonstration purposes
         DisplaySecureString(ss);
      }
      // After 'using', the SecureString is Disposed; no sensitive data in memory
   }

   // This method is unsafe because it accesses unmanaged memory
   private unsafe static void DisplaySecureString(SecureString ss) {
      Char* pc = null;
      try {
         // Decrypt the SecureString into an unmanaged memory buffer
         pc = (Char*) Marshal.SecureStringToCoTaskMemUnicode(ss);

         // Access the unmanaged memory buffer that
         // contains the decrypted SecureString
         for (Int32 index = 0; pc[index] != 0; index++)
            Console.Write(pc[index]);
      }
      finally {
         // Make sure we zero and free the unmanaged memory buffer that contains
         // the decrypted SecureString characters
         if (pc != null)
            Marshal.ZeroFreeCoTaskMemUnicode((IntPtr) pc);
      }
   }
}
```

The System.Runtime.InteropServices.Marshal class offers five methods that you can call to decrypt a SecureString's characters into an unmanaged memory buffer. All of these methods are static, all accept a SecureString argument, and all return an IntPtr. Each of these methods has a corresponding method that you must call in order to zero the internal buffer and free it. Table 14-4 shows the System.Runtime.InteropServices.Marshal class's methods to decrypt a SecureString into a memory buffer and the corresponding method to zero and free the buffer.

TABLE 14-4 Methods of the Marshal Class for Working with Secure Strings

Method to Decrypt SecureString to Buffer	Method to Zero and Free Buffer
SecureStringToBSTR	ZeroFreeBSTR
SecureStringToCoTaskMemAnsi	ZeroFreeCoTaskMemAnsi
SecureStringToCoTaskMemUnicode	ZeroFreeCoTaskMemUnicode
SecureStringToGlobalAllocAnsi	ZeroFreeGlobalAllocAnsi
SecureStringToGlobalAllocUnicode	ZeroFreeGlobalAllocUnicode