.NET 2.0 for Delphi Programmers

Jon Shemitz

Apress®

.NET 2.0 for Delphi Programmers

Copyright © 2006 by Jon Shemitz

ISBN-13: 978-1-59059-386-8

ISBN-10: 1-59059-386-3

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jim Sumser
Technical Reviewer: Hallvard Vassbotn
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole LeClerc
Copy Editor: Ami Knox
Assistant Production Director: Kari Brooks-Copony
Production Editor: Lori Bring
Compositor: Susan Glinert
Proofreader: Liz Welch
Indexer: Rebecca Plunkett
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `www.apress.com` in the Source Code section.

CHAPTER 11

Strings and Files

The FCL has its own style, and you'll find it easy to learn new parts of the library once you get a feel for the style. Strings are a good place to start—the `String` class offers a lot of standard functionality. Many classes offer methods that wrap core `String` methods, and so the core method's prototypes tend to propagate up the chain of wrappers. For example, the `Console` class has `Write` methods that look like `String.Format` because they wrap text file IO, which wrap `String.Format`. I speed over functionality that's familiar from Delphi, and lavish more attention on functionality that may be strange or new to Delphi programmers, such as the entirely new pattern language that `String.Format` uses, and the excellent regular expression implementation.

Learning the FCL

".NET programmers" know the FCL—learn once, work anywhere

The FCL documentation can be baffling at first. It's often more precise than clear, and new technical terms are often used without links to their definitions. The more of it you read, the clearer it all becomes.

Context-sensitive help on a class name will generally take you to the class's "Members" page. The Members page links to the class overview page and lists every member—all public constructors, all public properties, all public methods, &c. For all but the smallest, simplest classes, the class Members page is too long to be really useful as anything besides an entry point.

The class overview pages are generally worth reading, but often the best thing to do at a class Members page is to use the Help ➤ Sync Contents menu command (in 1.1—there is a tool button in both 1.1 and 2.0: see Figure 11-1) to show the class's entries in the Class Library reference section. In the table of contents, a class Members page is under the overview page, followed by pages that list all constructors on a single page, all properties on a single page, and so on. When the Members page goes on for several screens, the Methods page or the Properties page may be more manageable. Even so, some classes have so many methods that it takes several screen pages to list them all, which not only can make it easy to miss a method that might do what you want, but can also make it hard to get an idea of the sorts of things that the class can do.

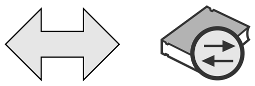

Figure 11-1. *The Sync Contents button in 1.1 and 2.0*

The `String` class is one of these potentially overwhelming classes with a large number of methods. The next section of this chapter, "Strings," breaks the `String` methods into functional groups. The intent is not to specify syntax—the Microsoft documentation is perfectly adequate for that—but to give you a feel for the sort of functionality that's available.

In general, these FCL chapters aim to explain enough of how various key FCL modules work that you can start using them and understand the documentation on edge conditions, &c. Easing into the documentation this way will hopefully help you understand it better when you start exploring aspects of the FCL that I don't cover.

■**Delphi Note** I do describe a few 2.0 methods, but my focus on key methods means that **most** of the methods I describe here are available in 1.1, and hence in Delphi 2006 (which can only use FCL 2.0 via the `--ClrVersion` switch to the command-line compiler).

You'll find that many FCL methods are heavily overloaded. For example, the `String` class's `Concat` method has nine public overloads. After a while you'll find that different methods are often overloaded in similar ways. They tend to follow a few common *models*. ("Model" is shorter than "pattern" and is not as heavily overloaded in developer talk. You know what I mean when I talk about following a model, but you aren't distracted by thoughts of *Design Patterns*.)

Learning to see these models in action is part of learning the FCL: when you see that a new type follows a familiar model, you know something about how the new type works.

Now, it's pretty clear that the FCL developers built these models into their code. They're much too ubiquitous not to be intentional. But the developers didn't give these models any public names, so I use short, made up names, like *Defaults* and *Smaller*, which I print italic and proper cased, so they stand out. I do this to make it easier for me to talk about these models, and to make it easier for you to think about them: giving a complicated subject a simple name can act as a sort of magnet to pull together disparate facts as you learn them.

Overall, *Defaults* is perhaps the most common overload model. Since C# doesn't support optional parameters, many overloads are just single statements that add various default parameters to their parameters, and pass the resulting expanded parameter list to the overload with real code. You'll see examples of the *Defaults* model throughout all these FCL chapters.

Strings

Look like Delphi, but (because of garbage collection) don't act like Delphi

As a Delphi programmer, you will find .NET strings look very familiar. A string variable is actually a reference to a string object (a reference type, on the heap) with a length field and an array of characters.

Because the string values have an explicit length field, operations like concatenation or right trimming do not need to calculate the string length by scanning for a null character, as in C-style string libraries.

Because strings are reference types, not value types, assigning a string value to a string variable is just a matter of changing the address in the string variable. Similarly, passing a string value to a method simply passes the address, not the actual string contents.

Delphi strings are in a syntactic class of their own and, in their own way, so are .NET strings. On .NET, strings are objects, but string objects **are** treated specially by the system. With every other type, every instance of the type is the same size as every other instance of the type. String objects are sized to fit their character buffer, but every string value is a `string` instance, not a string descendant:

```
/*  if  */ string AnyString = "The value doesn't matter";
/* then */ AnyString.GetType() == typeof(System.String)
```

There are larger differences between .NET strings and a native code Delphi's `AnsiString`. While sequences of operations are always relatively slow with long strings—so **many** cache-hostile character operations—creating lots of longish temporary strings is particularly expensive in a garbage-collected environment. This is because the frequent heap allocation triggers frequent garbage collection, and also because even modest-size strings end up on the Large Object Heap (and thus in generation 2), which makes the garbage collections more expensive. Chapter 3 has the details.

Also, .NET strings are not reference counted, the way that a native code Delphi's strings are. No reference counting means that passing string values from method to method doesn't add setup and teardown costs to method calls. No reference counting is also a reason that .NET strings are *immutable* in a way that native Delphi strings are not.

That is, Delphi's reference-counted strings make it easy to check at run time whether a given string value is unique or whether there are multiple references to the same value. If you change characters within a string value, the compiler emits calls to the run-time library that ensure the string value is unique, so when you change a character you only affect a single string value.

Doing a similar test on .NET would require a full sweep of the whole reference forest, from each root to every leaf, which is obviously impractical.[1] Thus, .NET strings have no methods that can change the actual character array. You can read a string character by character, but you can't change characters, nor can you insert or delete either characters or substrings.

1. The garbage collector sweeps from roots much less often than a unique string check might get called, and optimizes that as much as possible. You simply do not want application code doing this sweep—and especially not in a loop!

With immutable strings, it doesn't matter how unique your reference is, you can't change the string itself. Calling a method can't change a string parameter's value, unless it's explicitly passed as a `ref` or `out` parameter. Threads can't cause problems for each other by changing a shared string's contents. Certain types of buffer overflow bugs are simply not possible. Similarly, immutable strings eliminate security attacks that rely on changing a string buffer after it has passed various tests, but before some actual resource is accessed.

Instead of changing a character buffer, you create a new string. Code like `ThisString.Insert(4, AnotherString)` returns a new string, it doesn't change `ThisString`.

■ **Note** It takes an explicit assignment to change a string value. Forgetting to do the assignment is a common mistake.

DfN (Delphi for .NET) users should note that while indexed assignment **appears** to allow you to change characters within a string, this actually creates a new string by a three-way concatenation of a left substring, the replacement character, and the right substring. This concatenation is OK in extreme moderation, but is a very expensive operation within even comparatively short loops.

The `String` Class

Many standard operations are `System.String` methods

A Delphi `string` and a C# `string` are both a `System.String`. C# programmers need to—and Delphi programmers ought to—learn the various classes and methods of .NET's FCL (Framework Class Library) string code. The string code in the `System` namespace alone pretty much exceeds VCL units like `SysUtils` and `StrUtils`, and then there's the regex code in the `System.Text.RegularExpressions` namespace and the hash table code in the `System.Collections` and `System.Collections.Generic` namespaces. (I cover collections in the next chapter.)

Delphi programmers **can** mix Turbo Pascal procedures like `Str` and `Val` with VCL functions like `Trim` and `Format` and with FCL methods like `String.Split`, but you will probably find that the more you use the new FCL methods, the more you use FCL methods in place of their Delphi equivalents. Calls to `ThisString.Trim` go better with `ThisString.Split` than `Trim(ThisString)` does.

Concatenation Methods

In which we meet some common overload models

The `String.Concat` overloads set a pattern that you'll find repeated throughout this chapter. There are overloads that take one, two, three, or four `object` parameters, and there is an overload that takes a `params object[]` parameter. There are overloads that take two, three, or four string parameters, and there is an overload that takes a params string[] parameter. All Concat overloads return a single string.

Now, as per Chapter 9, the params array overloads mean that the overloads that take one, two, three, or four object parameters or two, three, or four string parameters are redundant. If they are not defined, the params array overload will be used. However, these overloads make for *Smaller* code, as it takes fewer bytes of CIL to push a handful of parameters than it does to create and populate an array. (String.Concat is heavily optimized, so that *Smaller* code is also faster; less widely used code will often just chain to the params array overload.)

The params array overloads also mean that you can concatenate any number of string or object parameters. As per Chapter 7, the C# compiler will automatically gather any extra type-compatible parameters into a temporary array, and pass that as the last params array parameter. Thus, in C# you can Concat any number of strings or any number of objects. Delphi does not support params parameters—a params array just looks like any other array to Delphi, and you have to use open array syntax to explicitly create an array once you have more than four parameters to concatenate. This makes String.Concat (and the other methods that overload like it) seem stranger and more discontinuous in Delphi than in C#—Delphi programmers see a syntax change at five parameters that C# programmers do not.

■ **Note** For the most part, the String.Concat overloads that add strings together are for compilers to use, in generating code for expressions like ThisString + ThatString. You may occasionally have an array of strings that you want to concatenate, but otherwise you will have little direct use for the string forms of String.Concat. The String.Join method (discussed later in this chapter) also concatenates its array of strings parameter, adding a separator string between each string in the array parameter.

The String.Concat overload that takes an object and returns a string simply calls the object's ToString method, returning an empty string on a null parameter. The overloads that take multiple objects and return a string all call each parameter's ToString method and concatenate the results.

For example, in C#, which boxes automatically, String.Concat(12, 34) returns "1234". Similarly, a string is an object—a string's ToString() method simply returns the string[2]—and String.Concat("(", 12, 34, ")") returns "(1234)".

Accepting object parameters instead of strings cuts boilerplate, making your code both smaller and *Simpler*. It's a lot easier to read code like String.Concat(This, That) than it is to read code like String.Concat(This.ToString(), That.ToString()). It's easier to write it, too.

Note that cutting boilerplate is not always the same thing as generating smaller CIL. In particular, C# code that takes advantage of params parameters is usually clearer—but not smaller—than code that explicitly creates an array. For example, String.Concat(0, 1, 2, 3, 4) is undoubtedly easier to read than String.Concat(new object[] {0, 1, 2, 3, 4})—but it compiles to the exact same CIL.

2. That is, Object.ReferenceEquals(ThisString, ThisString.ToString()).

The Format Method

Much like the SysUtils Format *function, but with a different pattern language*

The String.Format methods follow some of the same models as the String.Concat methods. Accepting object parameters instead of strings keeps calls to ToString() out of application code, and makes for *Simpler* code. String.Format accepts object parameters, calls ToString on each, and interpolates the results into a pattern string. This in turn becomes the *Format* model of passing parameters on to String.Format and doing various things with the results, which is followed by code from text streams on up to Console IO.

Accepting some small fixed number of parameters on the stack generates *Smaller* code, because calling CIL doesn't always have to construct a temporary array inline. String.Format also follows this model, adding a format string before the one, two, or three objects to be formatted. Again, because C# supports params arrays, these overloads are more noticeable in Delphi than in C#.

The format string is usually the first parameter to String.Format.[3] The result is the format string with any {} escape sequences replaced in various ways with one of the parameters. (Within a format string, use {{ and }} where you want the result to have { and } characters.) The simplest {} escapes are like {0} and {1}—just a single nonnegative integer between {} curly braces. Obviously enough, {0} is the first parameter in the main params array overload, and also in the overloads that take individual object parameters, while {1} is the second parameter and so on.

For example, String.Format("{0}{1}", 12, 34) returns "1234", while String.Format("{0} and {1}", "This", "That") returns "This and That" and String.Format("{2} and {0}", 1, 2, 3) returns "3 and 1". The format string does not have to include any escape sequences, but String.Format will raise an exception if any escape sequence in the format string refers to a nonexistent parameter.

You've probably got years of experience with variants of the C-style printf format language (like Delphi's Format function), and you may wonder what makes a format like "{0}{1}" enough better than "%d%d" that you should learn a new format language. There are two main reasons. First, an indexed escape like {0} handles repeats (and rearrangements) more clearly than a sequential language where each subsequent % escape refers to a subsequent parameter. (Think of the weird way that Delphi's Format('%d %d %d %0:d %d', [1, 2, 3, 4, 5]) returns '1 2 3 1 2', not '1 2 3 1 4'.) This can be particularly helpful when it comes to localization, as different languages may call for different term orders.

The second main reason is that the {} language is more extensible than the % language, because it has a start and a stop character. This means that it can have a sort of params parameter, an optional subsequence that runs to the end of the escape sequence and which can be fed to a parameter's ToString method to control the parameter's formatting. This makes extensibility much simpler (with many fewer reserved characters) than when everything must be shoehorned in between the % escape and the closing type code[4] character.

3. The one exception is the overload that takes an IFormatProvider parameter before the pattern string. Explicitly passing a format provider lets you control the locale, or *culture*, without changing the current thread's culture settings. You may need to juggle several different locales; you may only need to format a few values in a particular locale; or you may not have permission to change a thread's culture.

4. Type codes like %s for strings and %d for decimals.

Extensibility is important to String.Format because, unlike printf and Delphi's Format function, String.Format does not actually format any of its parameters. That is, when a printf-like function evaluates an escape sequence like %5.3d, it rips that into a width of 5 and a precision of 3, and passes the width and precision to internal integer formatting code. Getting a bit ahead of myself, when String.Format evaluates an escape sequence like {0,5:z3}, it rips that into parameter 0 with a width of 5. String.Format will use the field width to align the formatted parameter—but String.Format doesn't know anything about precision, or long and short formats, or the like. If parameter 0 supports custom formatting, String.Format will pass the z3 string to the appropriate ToString overload; otherwise, String.Format will just use the basic, no-parameter ToString overload.

■Note The String.Format pattern language is untyped. Every parameter is a System.Object, responsible for formatting itself with a ToString overload. The good side of this is that when your escape sequences just specify a parameter, with no format string—like {0} or {7}—you can change the type you're formatting without having to change the Format pattern string (which sure isn't true of printf-like methods). The bad side of relegating formatting to the type itself is that passing the wrong format string can give you spectacularly wrong results, or even an exception.

Escape Sequence Specifics

One mandatory component, and two optional components

Every {} escape sequence has an *index component*. The index component is a sequence of digits—that is, it is always an integer >= 0. The index component may be followed by an optional *alignment component* (a comma followed by an integer), which may be followed by an optional *formatting component* (everything between a : [colon] and the closing } character). For example,

```
{0}
{1, 5}
{3:g7}
{0, -5:z3}
```

The alignment component **must** precede any formatting component, because the formatting component is everything from the colon to the end of the escape sequence—including commas followed by an integer.

An alignment component specifies the field width. A negative integer is a left-aligned field, while a nonnegative field width is a right-aligned field. For example, String.Format("{0, 3}", 7) returns space-space-seven, while String.Format("{0, -3}", 7) returns seven-space-space.

Any padding is done with blanks: a three-character string in a five-character field will be padded with two blanks. Wide parameters are **not** clipped: a seven-character string in a five-character field will not be clipped to five characters, and will slide any subsequent fields to the right. When there is no alignment component, the field width is 0—or just as wide as necessary, with no pad characters.

A formatting component starts with a colon; everything between the colon and the closing } character is a format string that is fed to a parameter object's `ToString(string, IFormatProvider)` overload, if the parameter object supports `IFormattable`.

■ **Note** Any white space before and after an alignment component's comma is ignored, as is any white space before an alignment component's colon—but any white space **after** an alignment component's colon is part of the format string.

When there is no format string (or the parameter object doesn't support `IFormattable`), `String.Format` ignores any formatting component and calls each parameter object's `ToString()` overload, the overload that takes no parameters.

Standard types like numbers and dates define standard and custom formatting strings, and custom types can declare their own formatting strings. For numbers and dates, not specifying a format string is the same as explicitly specifying the `:G` ("general") format.

There is no guarantee that a custom type's `ToString` methods will honor locale settings, but the standard types always get elements like thousands separators, national currency symbols, and date-time pictures from an `IFormatProvider`. Most `String.Format` overloads give localized results by getting the `IFormatProvider` from

```
/* System.Threading. */ Thread.CurrentThread.CurrentCulture
```

but there is an overload that allows you to explicitly supply the `IFormatProvider` and get results localized for another culture. You can specify a customer's culture, or an *invariant* (canonical) culture, which is useful for text files that will be shared by users with different culture settings.

Numeric Formats

How numeric types interpret the optional formatting component

Standard numeric formats consist of a single alphabetic character, the *format specifier*, and an optional *precision specifier*, which is an integer from 0 to 99.[5] A format string like `j6`, which does fit this template but does not contain a supported format specifier, will throw an exception. Any format string that does not fit this template is a custom numeric format, which I cover (briefly!) after Table 11-1.

Remember that these numeric formats are implemented by the numeric types' `ToString` methods. Calling `ThisFloat.ToString ("R")` will return the same string as `String.Format("{0:R}", ThisFloat)`. (The right and left padding of the alignment component, however, comes from `String.Format`, not the `ToString` methods.)

5. In regex terms, this is two capture groups—the mandatory format specifier is `([a-zA-Z])`, and the optional precision specifier is `(\d{0,2})`. That is, the regex `([a-zA-Z])(\d{0,2})` matches (and rips) a standard numeric format.

Table 11-1 summarizes standard numeric formats, and aims only to show you what's available; when it comes to details, I encourage you to experiment and to read the FCL documentation. The pages for "Standard Numeric Format Strings" and "Custom Numeric Format Strings" are particularly helpful.

■ **Note** Most numeric format specifiers are case insensitive: `c` has the same effect as `C`. With exponential formats the specifier's case controls the case of the `E`, and with hexadecimal formats the specifier's case controls the case of the digits `A` through `F`.

Table 11-1. *Standard Numeric Formats*

Format	Notes
`C`—Currency	The number is formatted using the current (or supplied) culture's rules for currency symbol, thousands and decimal separators, &c. An explicit precision specifier overrides the culture's default currency precision.
`D`—Decimal	Integer only. An explicit precision specifier will cause left padding with `0` characters.
`E`—Exponential	Scientific notation with six digits after the decimal point, unless you explicitly specify the precision. The exponent is always signed, and contains at least three digits; your choice of `E` or `e` controls the case of the `E` in the output. You can get more precise control with a custom numeric format string.
`F`—Fixed-point	A floating point number, with a fixed number of digits to the right of the decimal point. The number of digits is specified by the current culture, if you don't explicitly supply a precision specifier.
`G`—General	This is the default when you don't specify a format—the most compact string representation of the number. The precision specifier has complex effects; see the SDK "Standard Numeric Format Strings" help page.
`N`—Number	Integer and float point. Thousands separators (i.e., `10,001`, not `10001`) and fixed-point decimal, as with the `F` format. Use `N0` (that's "n-zero," not "n-oh") to format integers with thousands separators and no decimal point.
`P`—Percent	The number multiplied by 100, and formatted according to the appropriate culture's percentage conventions.
`R`—Round-trip	Floating point only. `System.Double.Parse` will precisely re-create an `R`-formatted `double`, and `System.Single.Parse` will precisely re-create an `R`-formatted `float`.
`X`—Hexadecimal	Integer formatting. The specifier case controls the digit case; the optional precision controls the minimum number of result digits desired, with low values zero-padded on the left.

In the rare cases where the standard numeric formats don't give you the formatting you need, you can "draw" a custom numeric format "picture." For example, `:E3` will format 1234.1234 as `1.234E+003`. To get the terser `1.234E3`, you could use the custom `:#.###E0` format.

Custom numeric formatting uses a familiar sort of picture language, with # and 0 placeholders that allow you to control rounding, scaling, left-padding with 0s, and so on. You can specify multiple pictures, separated by semicolons, to format positive and negative numbers differently; you can also special-case 0 formatting.

Most people will have little use for custom numeric formatting, and know enough when they know that it exists. If you do need precise control over numeric output, the SDK "Custom Numeric Format Strings" topic covers it pretty well.

Date Formats

How the `DateTime` type interprets the optional formatting component

The `DateTime` object supports standard and custom formatting strings, much like the system numeric types. Standard date formatting strings are single characters: unsupported characters generate exceptions, while longer strings are custom date formats. As with the numeric types, if you don't specify a formatting component, you get the `:G` format.

Table 11-2 is a brief overview of the standard `DateTime` formats; for more details, see the SDK "Date and Time Format Strings" section. Note that many `DateTime` format specifiers are case sensitive, with an uppercase letter specifying a long form and a lowercase letter specifying a short form.

Table 11-2. *Standard `DateTime` Formats*

Format	Notes
`D, d`	Date. Uppercase `D` uses the current culture's **long** date pattern, while lowercase `d` uses the **short** date pattern.
`T, t`	Time. Uppercase `T` uses the current culture's **long** time pattern, while lowercase `t` uses the **short** time pattern.
`F, f`	Full. Current culture's long date; space; and either the long time (`F` format) or the short time (`f` format).
`G, g`	General. Current culture's short date; space; and either the long time (`G` format) or the short time (`g` format). `G` is the default format that you get if you don't specify any formatting component.
`M, m`	Day of the month, but neither year or time. No difference between `M` and `m` formats.
`Y, y`	Month of the year, but neither date nor time. No difference between `Y` and `y` formats.
`R, r`	RFC1123-compatible, OS- and culture-independent date/time format. Note that while the result will say `GMT`, `R` formatting does **not** convert local time to GMT; this conversion is your responsibility.
`s, u, U`	Sortable date/time pattern. These three are **not** equivalent; experiment and read the documentation to find the format most suitable for your applications.

The `String.Format` function does use a very different pattern language than Delphi's `Format` function and other `printf`-like functions. But the new pattern language is easy to learn, and not only is more extensible than the old language but also handles repeated and/or out-of-order use of parameters much better.

Substrings

Another common overload model

The `String.Substring` methods are similar to Delphi's `Copy` function—they extract a string from an existing string. There are two overloads. The first,

```
public string Substring(int startIndex);
```

copies from an offset to the end of the string, while the second,

```
public string Substring(int startIndex, int length);
```

copies a specific number of bytes, starting at an offset.

■ **Caution** The `Substring` methods act differently from Delphi's `Copy` function. If you try to `Copy()` from an starting offset that's past the end of the string, you'll get an empty string; the `Substring` methods will throw an exception. Similarly, if you try to `Copy()` past the end of the string, you'll get everything from the starting character to the end of the string; the `Substring` methods will throw an exception.

Now, extracting long substrings is always an expensive operation, especially in a loop. Copying thousands of characters from one location to another slows the CPU to memory rates and, as per Chapter 3, repeatedly allocating large heap blocks makes full garbage collections much too common.

The FCL makes it easy to avoid extracting most substrings. Methods that search or replace within a string usually have overloads that search or replace within a substring. These overloads honor what I'll call the *Substring* model, with overloads that take a string and either one or two integers. The one-integer overloads specify a right tail operation, where each scan starts where the last left off. The two-integer overloads specify an interior slice, like all the text between two matching XML tags. Just as with the eponymous `Substring` method, accessing any character outside the base string generally raises an exception.

The Chapter11\Substrings C# project contains a `Substring` class that encapsulates a few common uses of the *Substring* model, including especially the `Regex` methods I cover later in this chapter. `Substring` instances contain both a reference to a `Base` string, and the two integers that define a substring: the starting `Index`[6] and the substring `Length`. The point is to avoid actually calling `String.Substring` as much as possible, using instead references to portions of a single base string.

6. A substring's `Index` is the address of the first character, the substring's offset from the start of the base string.

Compare Methods

You can choose to heed or ignore both case and culture

The static `String.Compare` method is similar to the Delphi `AnsiCompare` methods. It compares two strings and returns an integer, where a negative value indicates that the first parameter is less than (sorts before) the second parameter, zero indicates that the two parameters are equal, and a positive value indicates that the first parameter is greater than the second parameter. The comparison defaults to case and culture sensitive, but there are overloads that allow you to specify a case- and/or culture-insensitive search (this is, finally, an example of the *Defaults* model).

`Compare` also has overloads that let you compare substrings. It may not be clear that `Compare` follows the *Substring* model: since `String.Compare` only compares equal-length substrings, `String.Compare` only takes a single length parameter. The case-sensitive substring overload takes a string and then an offset, followed by another string and then an offset, followed by a length:

```
static int String.Compare(String, Int32, String, Int32, Int32 Length)
```

The following method from the Chapter11\Substrings C# project uses this overload (which specifies case sensitivity) to compare a `Substring` to a `string`:

```
public int Compare(string CompareTo, bool IgnoreCase)
{
  return String.Compare(Base, Index, CompareTo, 0, Length, IgnoreCase);
}
```

Should this method do any error checking? For example, what if the `CompareTo` string's `Length` is less than the `Length` parameter? My `Substring` methods generally leave error checking to the FCL routines, which raise exceptions when you try to index out of a substring. There's no reason to replicate those error checks, and every reason not to—a replicated test might be (subtly, one hopes) wrong, or might become so in some future FCL version.

When you want to do a culture-**insensitive** string comparison, without regard for cultural rules that may, e.g., equate `e` with `é` or put `é` before `f`, you use the `String.CompareOrdinal` methods, which (like the `String.Compare` methods) are static methods[7] that compare two strings. The `CompareOrdinal` methods compare each character (which is usually, but not always, the same as a single Unicode code point) by its *ordinal value*—that is, they compare characters as 16-bit unsigned integers in much the same way as a native code Delphi evaluates expressions like `ThisString >= ThatString`.

You can compare either whole strings or substrings:

```
static int String.CompareOrdinal (String, String)
static int String.CompareOrdinal (String, Int32, String, Int32, Int32 Length)
```

7. In Delphi, you have to call `String` static methods like `Compare` as `System.String.Compare`, not just `String.Compare` or even `&String.Compare`.

Finally (though the String class's comparison repertoire is far from exhausted), I should mention that one common thing to ask about a string instance is whether it sorts ahead of, with, or behind some other string instance. The CompareTo instance method does a case- and culture-sensitive, whole-string comparison. CompareTo returns zero if the strings match, a positive number if the string instance is lexically greater than (after) the string parameter, and a negative number if the string instance is lexically less than (before) the string parameter.

Search and Replace

Find and/or replace strings and characters within the instance string

The FCL's equivalent of Delphi's Pos procedure is the String.IndexOf method. The IndexOf methods are instance methods, so ThisString.IndexOf(ThatString) searches for ThatString in ThisString. There are overloads to search in substrings of the instance string. All IndexOf searches are sensitive to both case and culture.

The IndexOf method also has overloads to search for an individual character. If you need to search for any of several characters, the IndexOfAny methods search the instance string (or an instance substring) for the first occurrence of any of the characters in the char[] anyOf parameter.[8] While string search with IndexOf is culture **sensitive**, character search with both IndexOf and IndexOfAny is culture **insensitive**—characters are compared by ordinal value as in the CompareOrdinal methods.

■ **Tip** You might use IndexOfAny as part of a search tree that finds the first occurrence of any of a set of strings, but it's much easier to use a Regex (discussed later) for this and other complex searches.

The StartsWith and EndsWith methods are convenience methods. StartsWith(Value) is roughly equivalent to IndexOf(Value) == 0, while EndsWith(Value) is roughly equivalent to IndexOf(Value) == Length - Value.Length. Both StartsWith and EndsWith are case and culture sensitive, and both handle empty strings and other edge conditions properly.

There are also LastIndexOf and LastIndexOfAny methods, which find the last occurrence of a string or character in a string or substring, instead of the first occurrence.

Remember, because strings are immutable, you can't replace characters or substrings within a string. The String.Replace methods are instance methods that return new strings and leave the instance untouched.

```
string Replace(char Target, char Replacement)
string Replace(string Target, string Replacement)
```

8. This array is **not** marked with the params keyword, so you're always aware that you are passing an array, in C# as in Delphi.

For example, in Delphi, `ThisString.Replace('\', '\\')` returns a new string, where every `\` in `ThisString` is replaced with `\\`. You would need to code `ThisString := ThisString.Replace('\', '\\')` to actually change `ThisString`.

The `Replace` methods always do case- and culture-sensitive matching. The `Replace` methods are ideally suited for tasks like escaping characters (converting `\` to `\\` as shown earlier, or using HTML escapes like `&amp;` or `&lt;`) and converting template text like `%CurrentDirectory%` to the actual current directory.

■ **Tip** The `Regex.Match` and `Replace` methods (later in this chapter) give much more control over both matching and replacing than these `String` methods, but the `Regex` methods are slower and more complicated. The Chapter11\Replacement C# project shows that repeated calls to `String.Replace` are usually faster than a single `Regex.Replace` with an alternation pattern (like `"This|That"`, which matches `"This"` or `"That"`).

Split and Join

From delimited strings to string arrays, and back again

The `String.Split` methods break a string instance into an array of strings. The string is scanned for characters (or, in 2.0, substrings) that match any of the characters or strings in an array parameter. If the array is `null` (`Nil`, in Delphi) or has a `Length` of 0, `Split` supplies an array of white space characters and splits on (broadly speaking) words. To split `ThisString` on lines, use the C# `ThisString.Split('\n')` or the Delphi `ThisString.Split([^M])`.[9]

Character and substring matching is a matter of bit equality—a case-sensitive and culture-insensitive comparison.

The extracted strings do not contain the break characters (or strings). The first element in the result is the substring to the left of the first delimiter. If the string you `Split` doesn't contain any of the delimiters, you get back a one-element string array, which contains the `Split` string. Similarly, if the `Split` string starts with a delimiter, the first result string will be empty, with a `Length` of 0.

The last element in the result is the substring to the right of the first delimiter. If the `Split` string ends with a delimiter, the last result string will be empty, with a `Length` of 0.

Every other result element is the text between two delimiter elements. An empty string (one with a `Length` of 0) is the text 'between' a pair of delimiter elements. For example, when you `Split` on white space, there is an empty string 'within' every CR-LF. Because of these empty strings, there is one result string for each occurrence of any of the delimiter elements, plus one result string that holds the 'remainder,' the text to the right of the last delimiter, if any. (In 2.0, you can optionally suppress the empty strings between delimiters.)

For example, there are five path delimiter characters in the filename `@"c:\Program Files\Shazzubt\Read Me.txt"`: one colon, three backslashes, and one dot. Thus, splitting the string `@"c:\Program Files\Shazzubt\Read Me.txt"` on the `char[] delims = {':', '\\', '.'}` gives a

9. The overload that takes only a single `char[]` parameter takes a `params` array.

six-element string array. The second string has a `Length` of zero—it is the empty string between the `:` and the `\`. The first string is `"c"` and the second to last string is `"Read Me"`.

If you care about what break character preceded or followed a particular result string, you have to look at the original `Split` string. To get the break character **before** the *Nth* string, you sum the lengths of all previous result strings, and add `N - 1`. This is the offset of the break character within the `Split` string. Similarly, the offset of the break character after the *Nth* string is the sum of the lengths of the first `N` result strings, plus `N`. (The Chapter11\Split C# project demonstrates the `Common\Utilities.cs` implementation of both of these methods.)

■**Note** `Split` is comparatively expensive, and often one doesn't need all the delimited strings. `Regex.Matches` (see later) is not only more flexible than `Split`, it can actually be faster when you only need one or two substrings from a long string, because it doesn't have to examine the whole string and because it doesn't have to do a character-by-character copy of each substring. See, for example, the Chapter11\BenchmarkSplitVsRegex C# project.

`Split` produces an array of strings from a single string. I've already covered two ways to go the other way, to turn an array of strings into a single string:

1. `String.Concat` concatenates each element in an array of strings (or objects) without any interpolation; the first character of the second string follows immediately after the second character of the first string.
2. `String.Format` offers great control over both interpolation and selection from the string array.

`String.Join` is sort of midway between `Concat` and `Format`—`Join` concatenates each element in an array of strings, adding a delimiter string between each element. Both this C# code

```
string MethodName = "System.String.Split";
string[] MethodComponents = MethodName.Split('.');
string MethodName2 = String.Join(".", MethodComponents);
Debug.Assert( ! Object.ReferenceEquals(MethodName, MethodName2)); // different addr
Debug.Assert(MethodName == MethodName2); // same value
```

and its Delphi equivalent (from the Chapter11\InverseFunction Delphi project)

```
var
  MethodName: string = 'System.String.Split';
  MethodComponents: array of string;
  MethodName2: string;
begin
  MethodComponents := MethodName.Split(['.']);
  MethodName2 := System.String.Join('.', MethodComponents);
  Assert(not TObject.ReferenceEquals(MethodName, MethodName2)); // different addr
  Assert(MethodName = MethodName2); // same value
end.
```

`Split` the string `"System.String.Split"` on a `'.'` character, then `Join` the resulting string array with a `"."` separator string, getting a new string, `"System.String.Split"`. As these examples may suggest, `Join` can be a very useful method.

You may have noticed that `Join` is **almost** an inverse function for `Split`. `Join` is not quite a true inverse function for `Split` because you can `Split` strings on multiple delimiter characters, and not just a single delimiter.

Miscellaneous Methods

More standard functionality

The `String` class supports most standard string-handling operations. Table 11-3 briefly summarizes `String` methods that mirror familiar Delphi `System` and `SysUtils` routines.

Table 11-3. *Miscellaneous `String` Methods*

Method	Delphi Equivalent	Notes
`String.Insert`	`System.Insert`	Insert one string into another, at a specific position.
`String.Remove`	`System.Delete`	Remove a substring.
`String.ToLower`	`SysUtils.AnsiLowerCase`	Lowercase every character. Culture sensitive.
`String.ToUpper`	`SysUtils.AnsiUpperCase`	Uppercase every character. Culture sensitive.
`String.Trim`	`SysUtils.Trim`	Remove white space from beginning and end; overload lets you remove **any** characters.
`String.TrimEnd`	`SysUtils.TrimRight`	Remove white space from end; overload lets you remove **any** characters.
`String.TrimStart`	`SysUtils.TrimLeft`	Remove white space from start; overload lets you remove **any** characters.

Always remember that where the Delphi procedure `Insert(ThisString, 3, ThatString)` modifies `ThisString`, inserting `ThatString` before the third character, the equivalent FCL function `ThisString.Insert(2, ThatString)` returns a new string, and does not affect `ThisString` in any way: you have to code `ThisString = ThisString.Insert(2, ThatString)`. Also, never forget that the first character in a .NET string is at position 0, not at position 1.

Constructors

The explicit constructors are quite specialized

Most strings are loaded from disk or the network, or created from the myriad permutations of the various string operations on loaded strings and `ToString()` results and on string literals embedded in code. (The next subsection, “Interning,” has details on string literals.) String values **are** instances of the System.String class, though, and there are String class constructors that you may need to call at various times (see, for example, the "File IO" subsection, later in this chapter).

The simplest constructor—String(Char, Int32)—creates a string with a specific number of copies of a single character. You can thus create a string with, say, three spaces, or five tabs, or two zeroes, and so on.

```
new string('\t', 5);          // C# for five tabs
System.String.Create(^T, 5); // Delphi for five tabs
```

More complicated constructors allow you to copy all or part of a character array to a new string. These constructors apply the *Substring* model to the character array: you can copy either the whole array or an interior slice. For example,

```
const
  CharacterArray: array[0..3] of char = 'test';
var
  test, es: string;
begin
  test := System.String.Create(CharacterArray); // 'test'
  es   := System.String.Create(CharacterArray, 1, 2); // 'es'
end.
```

You might use the character array constructor to build a string character by character: you allocate a character array; use character indexing to populate it; then pass the array to the appropriate string constructor. This is not a technique you will have much occasion to use directly, as the StringBuilder class, discussed later, uses exactly these techniques and may well provide all the functionality that you'll ever need.

There are also unsafe versions of the character array constructors, which can copy C-style null-terminated strings (or substrings thereof) to managed System.String values. (The point of the unsafe constructors is interfacing with legacy code. Chapter 10 and Appendix 0 cover unsafe code.) By default, text is assumed to be UTF8, but an optional Encoding parameter allows you to specify that the SByte* parameter points to null-terminated strings in formats from ASCII and UTF7 to UTF8, UTF16, and UTF32.

Interning

Only one copy of each string literal

Two strings are equal if they have the same number of characters and each character matches—which **has to be** true if they both refer to the same String object. Obviously, comparing two references is faster than comparing the lengths and then comparing each character (especially with long strings), so String.Equals(string, string) checks for reference equality first, only comparing lengths and characters if it's comparing two distinct String objects.

All string literals are *interned* to save space and comparison time. Interning means that the strings are stored in an internal hash table (Chapter 12) whose keys and values are both strings. When an assembly is loaded, all its string literals are loaded into memory, as interned strings, and the assembly's string literal table is set to point to the interned strings.[10] If a literal matches an existing interned string, the assembly's string literal table will refer to the existing string; if a literal does not match an existing interned string, the run time creates a new interned string. The CIL `ldstr` instruction turns a metadata string token—essentially, an index into the assembly's string literal table—into a normal string reference.

The intern table is maintained on a per-process basis. This means that

1. Strings are interned across assembly boundaries. If assembly A contains a string literal `<p>` and assembly B also contains a string literal `<p>`, applications that use both assemblies A and B will only use a single copy of the string literal `<p>`.
2. Strings are also interned across `AppDomain` boundaries (Chapter 14). This can save space, and makes it cheaper to marshal interned strings from one `AppDomain` to another: all that needs to be marshaled is the string reference, not the string value.

The `String.IsInterned` static method allows you to look up a string value in the intern table. `String.IsInterned( StringValue)` returns `null` (or `Nil`, in Delphi) if `StringValue` is not in the intern table and returns the interned string if `StringValue` **is** already in the intern table. Note that a non-`null` return will always `Equal` the passed `StringValue`—but will not necessarily `ReferenceEqual` the passed `StringValue`.

C# allows you to `switch` on string values. The generated code uses `String.IsInterned` to check the string you're switching on against the interned string literals. If `IsInterned` returns `null`, the string you're switching on **can't** match any of the string literals in the `case` clauses. If, however, `IsInterned` returns a non-`null` value, the `switch` code can use a series of `beq` CIL instructions to simply compare the reference to the interned switch string to each interned string literal.

The `String.Intern` method lets you add strings to the intern table. `Intern` takes a string and returns a string; unlike `IsInterned`, the result is always `Equal` to the input. If the input string is already interned, `Intern` returns the interned value; otherwise, it adds the new string to the intern table, and then returns it.

Interning a string can save space and speed up comparisons with a set of standard strings—but it also means that the value will persist until the process terminates. Note that interning a newly created string only makes it immortal by adding it to the intern table; it does not change its garbage collection generation. An immortal interned string will incur garbage collection costs every time it is relocated.

10. In 2.0, the `[assembly:StringFreezing]` attribute allows NGEN-ed applications (Chapter 4) to turn off interning on an assembly-by-assembly basis. String literals in an assembly with frozen strings aren't interned when the assembly is loaded. Rather, the frozen literals are stored in the native code at NGEN time in a format compatible with run-time `String` objects: instead of turning a token into a reference to a native string, the NGEN-ed code simply loads an address in the NGEN-ed code, and uses that as a string reference.

■ **Tip** If you want to save space and comparison time without making strings immortal, you can use your own hash table (see Chapter 12).

String Conversions

Formatting and parsing numbers and dates

As you've already seen, the various overrides and overloads of the `Object.ToString` method are the FCL equivalent of Delphi's functions like `IntToStr` and `IntToHex`. A 'bare' `ToString()` call gives the default format, which is generally the most compact format. Numbers and dates also support a `ToString(FormatString)` overload, which allows you to format values as in `String.Format`.

That is, values are responsible for formatting themselves. Similarly, each built-in value type is responsible for validating and converting a string to an appropriately typed value. For example, the `Int32.Parse` methods convert strings to 32-bit integers in various ways (or raise an exception), while the `Int64.Parse` and `DateTime.Parse` methods do the same, except that they return a 64-bit integer or a `DateTime`.

The various `Parse` methods are the most direct way to convert a string containing a formatted value to an actual value type. There are overloads that understand cultural conventions (so you can parse numbers like `123.456,789`, even in the US) and that can parse hexadecimal strings.

Alternatively, the higher-level `Convert` class contains methods that will interconvert a variety of types. For example, it has a `ToInt32` method that is overloaded to convert booleans, strings, and various numbers to an `Int32`. There are also `ToBoolean`, `ToDouble`, and `ToString` overloads that take an `Int32`. The various `Parse` methods offer fine-grained control, while the `Convert` class offers "one-stop shopping" and a consistent interface. `Convert` excels at tasks like converting floating point values to integer values in a language-independent way, and can handle tasks that the `Parse` methods don't support, like parsing binary and octal strings or doing base-64 (MIME) conversion of data blocks.

The `StringBuilder` Class

Concatenation and replacement

.NET strings are immutable—you can read individual characters, but you can't change them. Delphi allows you to write code like `ThisString[Index] := UpCase(ThisString[Index])`, but this is implemented as two `Substring` operations and a `Concat`, and is comparatively expensive.

The efficient way to change several characters is to use the `String.ToCharArray` method, change the character array, and then pass the changed character array to a string constructor. However, as with building a string character by character, you will seldom write code to do this directly, as the `(System.Text) StringBuilder` class encapsulates all the necessary logic, in a fairly general way.

A `StringBuilder` is basically a (private) character array with a `Length` and a `Capacity`. The `Capacity` is the number of characters in the array, while the `Length` is the number of those characters that have actually been set and that belong in the string. You can create empty string builders with a default Capacity; there are also constructor overloads that allow you to specify an initial string and/or initial Capacity. When you know (even roughly) how big the final string will be, passing the Capacity to the constructor minimizes both character copying and memory management overhead.

You can Append characters or strings to the StringBuilder, and it will copy them, character by character, to the private character array, incrementing the Length and increasing the Capacity as necessary. Append is heavily overloaded, and you can append many standard types directly, without having to either call ToString (e.g., you can Append(ThisInt) and don't have to Append(ThisInt.ToString())) or box common value types. There is also an AppendFormat method, which is the same as Append(String.Format()) but *Smaller* and *Simpler*.

The StringBuilder.ToString method passes the first Length characters to a String constructor, returning a new string. (ToString also has a *Substring* overload, which lets you extract a slice of the StringBuilder.) The ToString method does not affect the character array; you can call ToString, make some changes, and then call ToString again, if necessary. The StringBuilder.Clear method sets the Length to 0, but you can also set the Length directly: decreasing it truncates the string, increasing it pads it with blanks.

You can read and write characters directly, using array indexing. The StringBuilder.Replace methods change all instances of a character or string with another character or string; there are overloads that allow you to Replace within a *Substring* region of the character array. You can also Insert strings within the character array, or Remove substrings.

■**Note** The Insert, Remove, and Replace methods are loosely modeled on their String class equivalents. That is, they return a StringBuilder reference, so code like Builder = Builder.Replace("this", "that") will compile. However, they do modify the StringBuilder instance; they do **not** create new StringBuilder instances. I presume the point of returning a reference is that you can chain code like Builder.Replace("\n", "\\n").Replace("\r", "\\r").

There's surprisingly little overhead involved in creating a StringBuilder and then calling its ToString method. The Chapter11\BenchBuilder C# project shows that to change a single character, it's actually faster to use code like

```
StringBuilder Builder = new StringBuilder(ThisString);
Builder[0] = Char.ToUpper(Builder[0]);
ThisString = Builder.ToString();
```

than the much simpler

```
ThisString = Char.ToUpper(ThisString[0]) + ThisString.Substring(1);
```

The difference isn't great, however, and it's not a bad decision to emphasize clarity over speed in a case like this, reserving the StringBuilder class for code that does extensive surgery on a string.

Regular Expressions

What and how

Regular expressions are much more central to Unix programming than they have been to Windows programming. Under Unix, regular expressions are everywhere, from Perl and grep to shell filename expansions. There's even a POSIX regular expression API so Kylix applications can easily use regular expressions. On Windows, though, many Delphi programmers use regular expressions only to form the occasional complex search/replace string in the code editor—if they use regular expressions at all. There are a few Delphi regular expression libraries, but they're not widely used and there's no standard syntax.

Those days are over. The FCL `System.Text.RegularExpressions` namespace includes a standard `Regex` class with a Perl-compatible pattern language[11] and a comprehensive set of search and replace methods.

The next two subsections are a brief introduction to regular expressions, mostly for Delphi programmers who are new to regular expressions. The "Regex Introduction" subsection covers when you would and wouldn't use a regex. "The Regex Engine" gives you just enough background on the regex engine to understand the syntax and to work through mysterious behavior. You may benefit from this background if your regex skills are just a handful of puzzling tips—but most of you with even a little regex experience will probably want to skip ahead to either "Regex Pattern Language," which covers the FCL regex pattern language; or "The `Regex` Class," which explains how to actually use the FCL `Regex` class.

Regex Introduction

What a regex does, and when you use a regex

A regular expression (or *regex*) is a description of certain types of text, expressed as a set of algebraic rules. Text that follows the rules is *regular*, while text that does not follow the rules is not regular. When you create and use a new instance of the `Regex` class, your pattern string gets compiled to a state machine.[12] You can then apply the state machine (your compiled regex) to a bit of text to get all the substrings that match—or you can replace all the matching substrings.

What makes regexs so useful is that the syntax allows you to do much more than just search for keywords. You can do things like find the next < character followed by an identifier;

11. The FCL regex pattern language embraces and extends Perl's regex pattern language. Chapter 12's `Hashtable` and `Dictionary<K, V>` are similarly more general than Perl's hashes: a Perl hash is `reference[string]`, while an FCL hash is `object[object]` or `K[V]`. Between hashes, regexs, and boxing, you can write all sorts of symbolic code—C# and Delphi stuff that feels like LISP.

12. This is related to the `Compiled` option, which I cover later in this section, but is not the same thing. A Perl-style regex is compiled to a set of tables that a state machine rockets around until it matches or doesn't match. This is the default behavior for .NET regexs, too. A normal `Regex` instance refers to private tables in normal (collectable) managed data. The state machine in a `Compiled` .NET `Regex` doesn't interpret tables. Instead, all its decisions are implemented in custom CIL that gets jitted and run just like any compiler-generated CIL loaded from disk. Jitted code stays in memory until the application domain unloads at (or before—see Chapter 14) process termination. That is, the default is fairly fast, and uses collectable data; the `Compiled` option **is** faster, but uses non-GC code space.

followed maybe by white space and any number of identifier=value pairs, where a value is either a string of letters or a quoted string; all followed by `>` or `/>`. And then you can easily extract—or modify—the HTML tag, and all its attributes, and the closing `>` or `/>`.

A regular expression consists of literal text—which must be matched exactly—mixed freely with various expressions that describe which characters are acceptable at this point in the match. The expressions can be as specific as ‘match this character only’ or as broad as ‘match any character.’ You can specify sets of characters to match, and there are various predefined sets for matching alphanumeric characters, white space, and so on. In addition, you can group elements with parentheses, and specify exactly how many times an element should or may appear.

The pattern language is very terse: `a+` means “at least one `a` character” while `a*` means “any number of `a` characters, including no (zero) `a` characters.” `\d` means “any digit” and `\d{3}` means any three digits,[13] so `\d{3}-\d{2}-\d{4}` matches US Social Security numbers.

In some ways, regexs are an inverse function for `String.Format`. `Format` takes a ‘picture’ of the output and pours data into it; regexs take a ‘picture’ of the input and pull data out of it. Neither ‘picture’ is a model of clarity, but both save significant amounts of your time.

In fact, using regexs can save both programmer time and run time. You wouldn’t use a regex where `String.IndexOf` will do. However, regex compilation does produce an efficient matcher, and as per the earlier “Split and Join” subsection, regexs are competitive with comparatively simple matching jobs like `String.Split`, especially when you only need the first or last few matches.

But it’s complex matches (like finding unquoted attribute values in HTML tags) or even only slightly complicated matches (like finding all identifiers between `%` characters) where a regex really shines. The latter is `% [a-z_] [a-z_0-9]* %`, ignoring case and pattern white space. With just a little experience you can see that this is four easy-to-read elements, four simple states: find the literal `%`; followed by a letter or underscore; followed by any number of letters, digits, or underscores; followed by another literal `%`.

Imagine writing a method that would do that match efficiently: would it take you fifteen minutes? Would it take less than the 44 lines that the Chapter11\ManualMatch C# project does? Would it work the first time, the way my code did? Would it be easy to read, or would it be a state machine—like the regex compiler builds? Would it be slower than a regex, the way my code is?[14]

13. As you can see, regex elements come in different lengths.

14. Yes, I was shocked to find that my hand-built matcher is nearly 50% slower than even the ‘interpreted’ regex. It turns out that a large chunk of the discrepancy lies in my use of the `Char.IsLetter` and `Char.IsDigit` methods, which don’t even exactly match the *a to z* rules I use in the regex. (For example, in many countries, é is a letter.) When I replace those calls with substantially bulkier inline code like `(ThisChar >= 'a' && ThisChar <= 'z') || (ThisChar >= 'A' && ThisChar <= 'Z')`, my manual matcher becomes **almost** as fast as an interpreted regex—but still slower.

I’m sure I could bum a few cycles—for instance, it’s possible that `String.IndexOf(Char)` is faster than my `FindFirstLiteral` state—but then I’d be spending even more time writing complex code in a perhaps futile attempt to beat the performance I got with fifteen seconds’ work writing a one-line regex.

■**Note** You can think of regex patterns as a state machine–specification language. What you get is about as efficient as what you'd write by hand—and the pattern is much easier to write, much easier to read, much easier to debug, and much easier to maintain.

In addition to flexible matching and good performance, regexs support a wide range of operations. You can simply test whether some string contains any substrings that match the regex. You can find the first substring that matches, and extract the whole match or specific portions of it. You can find the next match; you can find the last match; you can get all matches. You can replace all matches with the same string—or you can use the match text to generate a replacement: examples include lowercasing XML tags, or replacing `%token%` strings by looking up `token` in a table.

The Regex Engine

How patterns match text

A regular expression is a stream of *elements*. When the regex engine matches a regex against a string, it looks for a text stream that matches each element in turn. If the regex finds a match, the match may be the whole string, or the match may be a substring, or there may even be several substring matches.

The regex engine looks for the first character of the string that matches the first element of the regex. If it finds one, it finds all the characters that match the first element of the regex, then compares the next character of the string with the second element of the regex, and so on. If it makes it to the last element of the regex, it has a match, and the *next match* operation would start just after the last character of this match. If the engine doesn't make it to the last element of the regex, it *backtracks*, looking for the last character that it might have interpreted differently, and giving that a try. (It may help to think of this as a depth-first traversal of the match space.) If the engine backtracks all the way to the first character of the match, it starts all over, looking for the next character that matches the first element of the regex, and so on.

For example, imagine matching the pattern **“.*”** against the string `There are “two” quotes in “this” string`. There are three elements in the **“.*”** regex—the literal “, the wild card **.***, and the literal ”. The literal “ can only match a “ in the string, so you go into a nice tight loop, looking for the first “ in the string. The wild card **.*** matches 0 or more of any character, so you skip to the end of the string. At the end of the string, you proceed to the next element in the regex, the ” literal. There's no match for the ” literal at the end of the string, so you backtrack to the last ” in the string, and match the `“two” quotes in “this”` substring.

If you were matching against a string like `this string has a “mismatched quote`, you would backtrack all the way to the first “, and conclude that there was no match.

Now, while writing a regex **is** always faster than writing the corresponding matching code by hand, it can be maddening at times, with seemingly fine distinctions that can mean the difference between your regex matching as you expect it to and your regex matching *almost* as you expect it to. That is, mismatching is usually not an all or nothing proposition: just as with a hand-coded string parser, you can write a regex that gets part of the match right and part wrong. Or that matches some of the strings that you want it to, but not all of them. Or that matches some of the strings that you don't want it to. Or all three.

When a regex doesn't work as expected, just take your regex an element at a time. At each step, compare what you **wanted** the engine to do with what you **actually** told the engine to do, and you should be fine. Look especially carefully at *-ed elements, which can cause trouble in two ways. On the one hand, they are *greedy*, and may consume more than you expect. On the other hand, they are optional and they always match. You can get some unexpected results when an optional element matches an empty string between two characters.

Regex Pattern Language

Perl compatible, with Microsoft extensions

Regexs have been around for a long time, and the pattern language has gradually gotten more complex. I try to cover the basic elements (that you really have to understand to use regexs) pretty thoroughly, and pretty much ignore the advanced elements (like *look ahead* and *look behind* assertions) that it's OK to look up as the need arises.

For more information, there's the Microsoft documentation—plus, since the `Regex` pattern language is a superset of the Perl regex language, standard regexs from the various "regex cookbook" sites should work just fine in your .NET programs. Perl compatibility also means that Google and standard regex books like *Mastering Regular Expressions* can help you with FCL regexs.[15]

■ **Note** The Chapter11\RegexExplorer project lets you experiment with regexs interactively. You can load a text sample, and build regexs to match it. The explorer shows all the matches in the sample, and updates this visible result set with each change to the regex. This makes it easy to see the effect of various changes, as well as what is right and wrong with each part of the regex. (All of the 'documentation' is in the mouseover tool tips.)

White Space

Makes patterns more legible

When you compile a regular expression, you can specify options that change some of the default behavior. For example, the `Singleline` and `Multiline` options affect how your pattern matches strings with new-line (`\n` or `^J`) characters. I'll talk about these in the appropriate places—I bring this up now, because one of the available options controls how white space in the pattern affects a match.

Normally, a white character is a literal character, to be matched like any other. This can make multi-element regexs hard to read, with no visual break between elements. The `IgnorePatternWhitespace` option allows you to override this default behavior, and include white space in your pattern—adding spaces between elements, or adding line breaks between

15. The *Programming Perl* "camel book" is another great source of information about writing and debugging regexs. The famous *Compilers* "dragon book" is particularly strong on explaining how regexs are evaluated. All three are in the bibliography (Appendix 5).

groups of elements. When you want to use this "extended mode" but need to match a space or new-line character, you can use the character escape mechanism (discussed later in this chapter) and match `\x20` or `\n`, or you can just match any whitespace character with `\s`.

The extended mode is much more legible than the default mode, especially since you can also add comments, which run from a # character to the end of the line. You'll find yourself using the extended mode pretty routinely, and the Chapter11\RegexExplorer project sets it by default. What's more, most of the examples in this chapter do use white space to make them more readable, and you should assume that they need to be compiled with the `IgnorePatternWhitespace` option.

Pattern Elements

Each element can match one or more characters

Each pattern element may be either a literal character or a command in the regex pattern language. A trivial regex (that consists of nothing but literal characters) does a case-sensitive match, much like `TargetString.IndexOf(TrivialRegex)` would. The `IgnoreCase` option makes literals match in a culture-sensitive, case-insensitive way, and the pattern `the` will match `the`, `The`, `THE`, and so on.

There are three basic types of commands in the pattern language: special characters that don't match as literals but act like keywords, affecting the match in various ways; user-defined character classes, where you list the acceptable characters between square brackets; and various backslash escapes, which either name predefined character classes or act like extra keywords.

The `.`, `^`, and `$` characters have special meaning as pattern elements (see Table 11-4). If you want to match `.`, `^`, or `$` literally, you have to escape them with a backslash—`\.`, `\^`, and `\$`. (There **are** other characters with special meanings—like the backslash itself—that have to be escaped if you want to match literally. I will cover these as they come up.)

Table 11-4. *Regex Pattern Characters*

Character	Matches
`.`	By default, the `.` is a wildcard that matches any character except `\n` (`^J`). In `Singleline` mode (i.e., when you specify the `Singleline` option), the `.` matches any character **including** `\n`. I'll show some examples of the `.` wildcard later in this subsection, under the "Quantifiers" heading.
`^`	By default, the `^` matches only the empty string to the left of the first character. That is, `anywhere` matches every string that contains `anywhere`, anywhere in the string, while `^Start` matches only strings that begin with `Start`. In `Multiline` mode, the `^` also matches the empty string to the left of every line—that is, between a `\n` and the first character of the line—and `^Start` would also match lines that begin with `Start`.
`$`	By default, the `$` matches only the empty string to the right of the last character. That is, `end$` matches only strings that end with `end`. In `Multiline` mode, the `$` also matches the empty string to the right of every line—that is, between the last character of the line and a `\n`, and `end$` would also match lines that end with `end`.
`\`	All multiletter commands start with `\`. To match a literal backslash, use `\\`.

You start a pattern with `^` and end it with `$` when you want to match only a whole string, or line, with no nonmatching text before or after the matching text. For example, it's not enough that an integer contain a stream of digits—it must not contain any other characters (except, maybe, leading or trailing white space).

A simple user-defined character class consists of a list of letters between square brackets: `b[ai]t` will match `bat` and `bit`. (Note that it will **not** match `bait`!) The list of letters can also include an ordinal range of characters by separating a pair of characters with a dash: `[0-9]` will match any digit. Similarly, using the * ("any number of") quantifier, `[a-zA-Z_] [a-zA-Z_0-9]*` will match a programming language identifier: a letter or underscore, followed by any number of alphanumeric characters or underscores.

You can also specify that an element match every character **except** the ones you specify by starting the class with a ^ (caret): `[^0-9]` matches any nondigit. Similarly, `[^"]` matches every character except a double-quote, and `" [^"]* "` matches a simple double-quoted string: a " character, followed by any number of nonquote characters, followed by a " character.

If your character class needs to include a ^ character, you can simply not list it first—`[$^]` matches the $ or ^ characters—or you can use a \^ escape, as you would to match a literal ^. (Where a character like **.**, \, or ^ has a special meaning, escaping it with a \ turns the character into a literal.) So, `a\*` matches `a*`, not any number of `a` characters; both `[\^$]` and `[$^]` will match either $ or ^; and `[ \[ \] ]`will match either [or].[16]

A backslash before characters that **don't** have special meaning is a sort of compound keyword. These may be either a predefined character class that matches any character in the class or a special *assertion* that (like ^ and $) matches an empty string if certain conditions are true. The six character classes in Table 11-5 are all from Perl: note that they **are** case sensitive, with the uppercase versions including all characters **not** in the corresponding lowercase version.

Table 11-5. *Perl-compatible Predefined Character Classes*

Escape	Matches
`\w`	Word (alphanumeric) characters: letters, digits, and underscores
`\W`	Nonword characters—any characters that **don't** match `\w`
`\s`	White space
`\S`	Nonwhite space—any character that puts pixels on the background
`\d`	Digit characters
`\D`	Nondigit characters

By default, these predefined classes include Unicode characters; in `ECMAScript` *mode, these predefined classes include only 7-bit ASCII characters.*

In addition to the predefined character classes in Table 11-5, .NET's regex pattern language also includes `\p{name}`, which matches any member of the Unicode character group *name*, and `\P{name}`, which matches any character that's **not** in the named Unicode character group. For example, `\p{Lu}` matches any uppercase letter, and `\P{Ll}` matches anything except lowercase letters.

Both the character classes in Table 11-5 and `\p{}` and `\P{}` Unicode character group escapes can be used **either** in place of literals (in the same way as **.**, ^, and $) **or** within square brackets, as part of a character class. Thus, `[a-z\d]` is the same as `[a-z0-9]`, while `\p{Lu} \p{Ll}*` matches

16. The `Regex.Escape` method can take any string and turn it into a regex literal, escaping all characters as necessary. See "The `Regex.Escape` Method," later in this chapter.

any Proper Cased substring, and `[\p{Lu}\p{Ll}]+` (the `+` quantifier means "at least one") matches any sequence of uppercase or lowercase letters.

■ **Tip** See the `System.Globalization.UnicodeCategory` enum for a list of Unicode character groups.

In 2.0, regexs support *character class subtraction*. Character classes can now include a `-[`*`class`*`]` between the 'positive' class and the closing bracket. For example, `[a-z - [aeiou]]` matches any lowercase consonant. Similarly, `[\p{Ll} -[a-z]]` matches any lowercase character that's not on an American keyboard, like `é`.

Table 11-6 contains six escape sequences that act as assertions that match zero-length strings when some conditions are satisfied. When the conditions are **not** satisfied, the assertion does not match, and so (unless you've carelessly made the assertion optional) neither does the regex. For example, `\b` asserts that there is a `\w` character on the left or the right, but not on both sides—`\b verb` matches the first four characters of `verbose` but not the last four characters of `adverb`.

Table 11-6. *Two-character Regex Assertions*

Escape	Effect
`\A`	Matches empty string to the left of the first character in the string. Like `^`, except not affected by the `Multiline` option.
`\z`	Matches empty string to the right of the last character in the string. Like `$`, except not affected by the `Multiline` option.
`\Z`	Like `\z`, except will ignore a single trailing `\n`.
`\G`	Matches empty string before the first character and after the previous match. That is, there can be no unmatched characters between matches. For example, you only get three matches when you match `\G\w+\s+` against `Three words but then, no more.`—because `then,` doesn't fit the `\w+\s+` pattern, and `no` doesn't match, even though it does fit the `\w+\s+` pattern.
`\b`	Matches the empty string between a `\w` character and anything but a `\w` character. That is, `\b` matches between `\w` and `\W` characters, and also at the start (or end) of the string if the first (or last) character is a `\w`.
`\B`	Matches whenever `\b` does not.

Finally, there are several ways to specify literals that don't appear on the keyboard:

- `\a`, `\f`, `\n`, `\r`, `\t`, and `\v` have the same meanings as in C# literals (Table 5-4), while `\e` matches the ASCII escape character `1B` (`#27` or `^[`, in Delphi), and (though only within a character class) `\b` matches the ASCII backspace character `08` (`#8` or `^H`, in Delphi).
- You can specify (most) ASCII control characters with `\c`—for example, `\cH` is `^H` and `\c^` is `^^`, but `\c[` is **not** `^[`.

- You can specify ASCII literals in hexadecimal with `\x` followed by two hexadecimal digits—for example, `\x20` is a space, and `\x1B` is the ASCII escape character, `\e`.
- You can specify Unicode literals in hexadecimal with `\u` followed by four hexadecimal digits—for example, `\u00A3` is the British currency symbol, £, and `\u20ac` is the euro symbol, .

■ **C# Note** Don't be confused by the parallelism between regex escapes like `\n` and C# character and string escapes. In C#, escapes are processed as string (and character) literals are compiled, and if you use `"\\"` as a regex pattern, your regex pattern contains a **single** backslash character—which is not a valid regex, and will raise an exception. To match a single backslash, you need to use the regex pattern `"\\\\"`—or `@"\\"`. Similarly, `"\n"` is a single new-line character, not the `\n` escape. To match `\n`, you need to use `"\\n"`—or `@"\n"`. In general, C# `@""` literals are your best choice for regex patterns, both because you don't have to double backslashes (though you **do** have to double double-quotes) and because they can span multiple lines.

Quantifiers

Each element can have an optional repeat count

By itself, every element matches once. It doesn't matter whether the element is an assertion that matches empty strings, or whether the element is a literal or a wildcard that matches a single character: if the element doesn't match, neither does the regex; while when an element does match, the regex engine tries to match the next element to the next character. However, every pattern element can be followed by an optional *quantifier*, which specifies a minimum and maximum number of times the element can match. When an element can appear at least 0 times, it is optional.

There are single character abbreviations (+, ?, and *) for the three most common quantifiers. The + quantifier means that an element is mandatory, but may repeat: the element may appear one or more times. For example, `um+` will match `um`, `umm`, `ummm`, and so on. You can use parentheses to group a series of elements into a compound element: `(ha)+` will match `ha`, `haha`, `hahaha`, and so on.[17]

The ? and * quantifiers make an element optional. A ? element may appear 0 or 1 times, while an * element can appear any number of times. For example, `Sam(uel)?` will match both `Sam` and `Samuel`. Similarly, `umm*` will match `um`, `umm`, `ummm`, and so on.

Both * and + are *greedy* and will match as much as possible. For example, matching "`.*`" against `This "string" has "three" "" quotes` will match "`string" has "three" "`". You can specify *lazy* (nongreedy) repeats with *? and +?. Lazy repeats match as little as possible, and matching "`.+?`" against `This "string" has "three" "" quotes` will match "`string`" and "`three`", while matching "`.*?`" against the same string will match "`string`", "`three`", and "".

17. Because parentheses have meaning grouping elements, but not within character classes, you have to escape them to do a literal match, but you do not have to escape them within a character class—`\( [^)]+ \)` matches a (character followed by at least one character besides a), followed by a) character.

Less-common quantifiers use a `{}` syntax. `{n}` specifies **exactly** *n* repeats—`\d{2}` is the same as `\d\d`, and matches exactly two digits. `{n,}` specifies **at least** *n* repeats—`\d{2,}` is the same as `\d\d+` or `\d\d\d*` and matches at least two digits. `{n,m}` specifies at least *n* but no more than *m* repeats—`\d{1,3}` is the same as `\d\d?\d?`, and matches one, two, or three digits.[18]

You can also specify lazy repeats with the `{}?` syntax. `{n}?` is supported, but is exactly the same as `{n}`—exactly *n* repeats. `{n,}?` is as few as possible, but at least *n* repeats, while `{n,m}?` is as few as possible, but at least *n* and no more than *m* repeats.

Capture Groups

Substrings within the match

When you match a regex against a string, you get a `Match` object that contains information about the match. If the `Match` object's `Success` property is `true`, the match succeeded and the `Value` property is a string containing the substring of the match string that matches the regex.[19] For example, when you match `\d{3}-\d{2}-\d{4}` against

```
The Social Security Administration says "Any number beginning with 000 will NEVER
be a valid SSN" and so numbers like 000-00-0000 or 000-45-6789 can safely be printed
in books or used as dummy data.
```

you get two valid matches, with a `Value` of `000-00-0000` and `000-45-6789`. (You can either call the `Regex.Matches` method to get [a class that acts like] an array of `Match` objects, or you can call the `Regex.Match` method to get a single `Match` object, and then call `Match.NextMatch` while the `Match.Success` property is `true`. See "The `Regex` Class," later in this chapter.)

If you care about fields within each match, the match operation can also divide each match into fields—you don't have to create a regex that describes each field, and then apply each field regex to each match. By default, every matching pair of parentheses creates a *capture group* in addition to providing logical grouping. For example, when you match `(\d{3}) - (\d{2}) - (\d{4})` against the preceding SSN text, you still get two matches with `Value` properties of `000-00-0000` and `000-45-6789`, but now each `Match` object has four `Groups`.

The first group corresponds to the substring that matched the whole regex—`AnyMatch.Value == AnyMatch.Groups[0].Value`. Subsequent `Groups` entries contain any capture groups, so when you match `(\d{3}) - (\d{2}) - (\d{4})` against `000-45-6789` you get four groups:

`Groups[0].Value`	`000-45-6789`
`Groups[1].Value`	`000`
`Groups[2].Value`	`45`
`Groups[3].Value`	`6789`

18. The ?, +, and * quantifiers are equivalent to longer `{}` quantifiers involving 0 and 1—? is the same as `{0,1}`, while * is the same as `{0,}` and + is the same as `{1,}`. No quantifier at all is the same as `{1}`.

19. Note that the `Match` object does not actually contain the substring. Rather, it contains a private reference to the string that was matched, and the `Index` and `Length` of the match substring within the base string. Reading the `Value` property actually calls `String.Substring`.

    This is also true of the `Group` and `Capture` objects, discussed later. `Match` descends from `Group`, which descends from `Capture`; both `Group` and `Match` inherit the `Index`, `Length`, and `Value` properties from `Capture`.

In general, group numbers correspond to left parentheses, so when `\. ((\d) (\d))` matches `.45`, you get

`Groups[0].Value`	`.45`	The regex as a whole
`Groups[1].Value`	`45`	The leftmost parenthesis
`Groups[2].Value`	`4`	The first nested capture group
`Groups[3].Value`	`5`	The second nested capture group

It's not particularly hard to count parentheses in these simple regexs, and thus to know which group will contain which field. However, with longer regexs, it does get easier to make a mistake. More importantly, if you add any parenthesized expressions to a regex, any capture groups to the right of the new expression get renumbered—and this is not uncommon, since parentheses serve both as logical groups (applying a single quantifier to a stream of pattern elements) **and** as capture groups.

Accordingly, you can name a capture group by adding a `?<name>` (or `?'name'`) between the left parenthesis and the first pattern element in the group. Thus,

```
\. ((?<first> \d)  (?'second' \d))
```

matches just like `\. ((\d) (\d))` except that you can read `Groups["first"]` and `Groups["second"]` as synonyms for `Groups[2]` and `Groups[3]`. Using named capture groups is slightly slower than using numbered capture groups, but using named capture groups can be much clearer and more stable than using numbered capture groups.

■ **Note** You can access a named capture group by name **or** number.

Creating a capture `Group` object takes time and memory. The `ExplicitCapture` option changes parentheses so that they serve only as logical groupings; only named capture groups will actually capture fields and create capture `Group` objects. Alternatively (since named capture groups can make a regex harder to read), you can use `(?: )` as an explicit *noncapture group*—the parentheses serve only as logical groupings.

For example, matching `\. (?: (\d) (\d))+` against `.4567` gives only three groups—`.4567`, `6`, and `7`. The noncapture group that encloses the two `(\d)` groups is just a logical grouping that lets the `+` quantifier apply to the pair of `(\d)` capture groups. This example also illustrates that when a quantifier makes a capture group match repeatedly, the group `Value` is the **last** match. You can read the other match(es) from the group's `Captures` collection:

Group	`Value`	`Captures.Count`	`Captures[0].Value`	`Captures[1].Value`
`Groups[0]`	`.4567`	1	`.4567`	
`Groups[1]`	`6`	2	`4`	`6`
`Groups[2]`	`7`	2	`5`	`7`

For a more complex and realistic example, this regex

```
#[ExplicitCapture | IgnorePatternWhitespace]20
< (?<Tag> [a-zA-Z_] \w+ )            # the tag
  ( \s+ (?<Attribute> [a-zA-Z_] \w+) = # optional attribute=
        ((" (?<Value> [^"]* ) ")       # a quoted value
        |(?<Value>  [^"\s]+) )         # or, an unquoted value
  )* \s*                               # trailing white space
>
```

will match an HTML tag, and parse it into the tag name (like `div` or `table` or whatever) and an optional collection of `attribute=value` pairs, where the values may or may not be quoted. When this regex matches, there are always four capture groups. The first group is the tag, from < to >; the second group is the tag name; the third group is the attribute names, if any; and the fourth group is the attribute values, if any.

■ **Note** This regex is included in the Chapter11\RegexExplorer project, which makes it easy to examine the results with different HTML tags.

When this regex matches a simple HTML tag like `<html>`, the first group's `Value` is `<html>`; the second group's value is `html`; while the third and fourth groups do not match (i.e., `Success` is `false`). When this regex matches an HTML tag with attributes:

```
<table border=0 width="100%" cellspacing=0 cellpadding=8>
```

the first group's `Value` is the whole tag; the second group's value is `table`; and the third and fourth groups each have four captures:

Group	`Captures[0]`	`Captures[1]`	`Captures[2]`	`Captures[3]`	`Value`
`Groups[2]`	`border`	`width`	`cellspacing`	`cellpadding`	`cellpadding`
`Groups[3]`	`0`	`100%`	`0`	`8`	`8`

20. I use this `#[ ExplicitCapture | IgnorePatternWhitespace]` syntax (with `RegexOption` member names in square brackets, separated by |) to show the options a particular regex needs both because it seems clear enough, and the Chapter11\RegexExplorer project understands this syntax as a sort of pragma—if there's one and only one such comment, and each name **is** a `RegexOption`, the Regex Explorer will use the specified options. (This allows the right-click menu to paste pattern **and** options.)

Note that these pragmas are **not** regex syntax—the `Regex` class ignores everything between a # and the end of the line when it compiles a pattern in the `IgnorePatternWhitespace` (aka "extended") mode.

Alternation

Match multiple subpatterns

So far, a regex has been a series of pattern elements, each with an optional quantifier. Pattern elements can be grouped with parentheses, to form a compound element that can have its own quantifier, but each is joined by an implicit concatenation operator—each pattern element must match.

The *alternation operator*, `|` (the C "or" operator), allows a regex to contain two (or more) different ways to match. For example, in the complex HTML parsing regex shown earlier,

```
(("  (?<Value> [^"]* ) ")      # a quoted value
| (?<Value>  \S+) )            # or, an unquoted value
```

matches either a quoted value **or** an unquoted sequence of at least one non-white-space character.

The alternation operator has lower precedence than the implicit concatenation operator, so `this|that` is **not** the same as `thi [st] hat`. That is, `this that` matches `this` or `that`, not `thishat` or `thithat` (though it will match the `this` in `thishat` and the `that` in `thithat`.)

The regex engine will always try to match the left side of the alternation operator before the right side. Thus, when you match `that|th([a-z]*)t` against `that thought`, you will get two matches, each with two capture groups:

Match	`Groups[0]`	`Groups[1]`
`Matches[0]`	`that`	No match!
`Matches[1]`	`thought`	`ough`

Back References

Matching captured text

Sometimes you want to look for a repeat of a captured substring. For example, you might want to match HTML like `<h1>text</h1>` or `<b>text</b>`. You can do this with *back references*: `\N` matches capture group number *N*, and `\k<Name>` matches the capture group named *Name*.

Thus, `<(?<Tag>[a-zA-Z] \w*) [^>]*>` matches an HTML tag, saving the tag text in the named capture group, `Tag`, and ignoring any attributes. The more complex variant

```
#[SingleLine | IgnorePatternWhitespace]
< (?<Tag>[a-zA-Z] \w*) [^>]*  > # the opening HTML tag
(?<Text> .*?)                   # text within the tag
</ \k<Tag> >                    # the matching </ tag
```

matches text from `<tag>` to `</tag>`, saving the text between the tags in the named capture group, `Text`.

The Capture Stack

Matching nested text

The preceding example works pretty well until you try to match nested text like

```
<div class="Outer">
  <div class="Inner">
  </div>
  This text will not be part of the match.
</div>
```

The match will stop at the first `</div>`, skipping everything between that and the end of the outer `<div>`. An inability to match nested text is a traditional, well-known limitation of the regular expression engines in Perl, Linux, and JavaScript. The FCL `Regex` class includes an extension that **does** allow you to match nested text.

For example, the following regex will match nested parentheses:

```
#[IgnorePatternWhitespace]
\(                     # a literal (

 (?:                   # non-capture group
     (?<Stack>  \( )   # on nested (, push empty capture
   | (?<-Stack> \) )   # on nested ), pop empty capture
   | [^()]             # anything except ( or )
 )*                    # any number of chars between parens

(?(Stack)              # if stack not empty:
    ^                  # then, match beginning of string (ie, fail)
  | \) )               # else, match literal )
```

Step by key step:

1. The regex scans for a `(` character, at which point it proceeds to a normal, greedy noncapture group—`\( (?: )*`—which consists of three alternate subexpressions.

2. The first alternative, `(?<Stack>  \( )`, is a normal, named capture that matches a literal `(` and saves it to the `Stack` capture. As earlier, .NET's regex implementation saves **all** captures, not just the last one.

3. The second alternative, `(?<-Stack> \) )` uses a new operator to match a literal `)` and delete the most recent `Stack` capture. It fails if there isn't a literal `)`, thus preventing a match of an unpaired left parenthesis.

4. The `(?(Stack)` uses another new operator: `(?(Name) a | b)` matches `a` if the `Name` capture group has captured any values, and matches `b` if the `Name` capture group is empty. `Stack` is only nonempty if there is an unpaired `(`, so we fail by matching `^` (the start of the string) and match `\)` if the stack is empty.

That is, matching nested text relies on using a named capture group as a stack (pushing on a begin string and popping on an end string) and then using `?()` to fail if the stack is not empty. The Chapter11\RegexExplorer project also includes a **much** more complex version of this regex, which is a complete HTML tag parser: it captures the `Tag` and any `Attribute` and `Value` strings; it handles nested tags properly; and it captures the `Text` between the `<Tag>` and the `</Tag>`.

■**Note** The regex pattern language includes several useful operators that I don't discuss here, like the four *zero-width positive/negative lookahead/lookbehind assertions*. While these can be confusing, I think I've explained enough that you can pick up these advanced topics on your own. The Regex Explorer's HTML parsing regex uses an example of the `(?> )` *greedy subexpression* and the Chapter11\Replace project uses lookahead and lookbehind assertions; otherwise, I leave you to experimentation, Google, and the Microsoft documentation.

The `Regex` Class

The `System.Text.RegularExpressions` namespace

The three previous subsections have talked about why you should use regexs, how regexs work, and how to write a regex. What I haven't covered is the actual mechanics of using a regex in your C# or Delphi code.

A regex pattern is a sort of program: a description of a text-matching state machine. Like other programs, a pattern has to be compiled before you can use it. By default, when you create a `Regex` object, the pattern is compiled to a set of tables that define the states the matcher can be in, and the various conditions that can move it from one state to another. Table interpretation is very cheap and efficient, but it is still interpretation; there are two different ways to compile a regex to jittable CIL instead of to tables, which I talk about in the "Regex Options" and "Precompiled Regexs" topics, in this subsection.

Because regex creation and first use is much more expensive than subsequent uses, you generally want to reuse a regex whenever you can. At the same time, it **does** feel wrong to create a static class member (visible to all of an object's methods) just so that a `Regex` used only in one method can be reused. This is why the most commonly used `Regex` match and replace overloads come in both static and instance versions. While the instance versions use the regex that you passed to the `Regex` constructor, the static overloads allow you to pass a regex as a pattern string (and, perhaps, a `RegexOptions` bitmap). The point of the static overloads is that the system maintains a `Regex` object cache (keyed by pattern and regex options) that lets it reuse an already compiled regex.

■**Note** The caching strategy changed in 2.0, where only the static overloads use the regex cache. Creating two `Regex` objects with the same pattern and the same options means the same regex will be compiled twice. In 1.1, explicitly creating a `Regex` instance was also cached. Presumably the change was made to minimize the size of the cache, and to allow `Regex` instances to be garbage collected when you're through with them.

While the `Regex` cache is not a documented part of the `Regex` class, and may vary from version to version (and may act differently on CE [or Mono] than on Windows), the cache lookup is not particularly expensive, and you should probably use the static methods most of the time. They make your code a bit simpler and easier to read, without adding a tremendous run-time burden. The three times you would want to use the instance overloads instead of the static overloads are

1. When you want to use `Regex` functionality that doesn't have a static overload. For example, the static overloads only allow you to match a regex against an entire target string, but there are instance overloads that follow the *Substring* model and that allow you to match a regex against the right tail (or an interior slice) of a target string. These are useful in applications like processing the text between HTML tags without calling `String.Substring` to create a new string. Similarly, the various methods that give you access to the collection of capture group names are only available through `Regex` instances.

2. When you don't want a `Regex` object to stick around in the cache, but want it to be garbage collected after it's used. (For example, it may be used only once, or it might only be used twice a day in a program that runs for days or months.) In this case, you would explicitly create a `Regex` object as a local variable, and let it be automatically scavenged after the last use.

3. When you repeatedly call a method that uses a regex, in a loop where speed is absolutely critical. In this case, you would either create a `Regex` object as a class member or a local variable, or you would precompile your regexs to a separate assembly.

The `Regex` constructor has two overloads. The overload that takes just a pattern string compiles the regex using the default options: interpretive matching; `.` doesn't match `\n`; `^` and `$` match only at the beginning and end of the search string; white space is significant; and so on. To override any of these options, you use the overload that takes a pattern string and a `RegexOptions` bitmap (see the "Regex Options" topic, later in this chapter, for details).

When you explicitly create a `Regex` object, you have full control over its lifetime. A regex that will be used frequently can be declared as a static member,

```
private static Regex Explicit =
  new Regex(Pattern, Options); // created when class 1st referenced
```

which is created and compiled when the class is first referenced, and which lasts until the program exits. Once a `Regex` has been created, calling `Regex` methods directly through a class member (or a local variable) **is** slightly faster than calling them indirectly through `Regex` static methods and the `Regex` cache.

The `Match` Object

Capture groups and other information about a match operation

The `Match` object contains the results of a single match operation. If the regex matches the text, the `Match.Success` property will be `true`. If the match succeeded, the `Index` and `Length` properties specify a substring within the target string; the `Value` property and the `ToString` method call `String.Substring` to return the substring. As always, calling `String.Substring` is slower than working with `Index` and `Length`.

The `Match.Groups` property is a collection of `Groups` objects, containing information about each capture group in the regex. You can index `Groups` by name or number—it's faster to index by number, but it's safer to index by name. `Regex.GroupNumberFromName`[21] allows you to do the lookup **once**.

Each `Group` has a `Success` property, as well as `Value`, `Index`, and `Length` properties, just like the `Match` object. (In fact, the `Match` class descends from the `Group` class.) As per the earlier "Capture Groups" topic, a group's `Value` is the **last** substring that matched the capture group. The `Group.Captures` property is a collection of `Capture` objects, which contains **each** substring—`Value`, `Index`, and `Length`—that matched the capture group. (The `Group` class descends from the `Capture` class.) A `Capture` object does not have a `Success` property: `Capture` objects are only found within groups that matched successfully.

The `Match.Result` method interpolates capture group values into a pattern string using a Perl-compatible language where `$0` is replaced by `Groups[0].Value`, `$1` is replaced by `Groups[1].Value`, and so on. Thus, for a `Match M`,

```
M.Result("$1.$2") == String.Format("{0}.{1}", M.Groups[1], M.Groups[2])
```

Finally, there is a certain amount of redundancy in this result object hierarchy, which can be confusing to new users. The first group, `Groups[0]`, contains the substring that matched the whole regex, which is also contained in the `Match` object's `Value`, `Index`, and `Length` properties. In fact, because a `Match` object `is` a `Group`, in a successful `Match M`, `M` and `M.Groups[0]` are the same object.

That is, `Object.ReferenceEquals(M, M.Groups[0])`, and referring to `Groups[0]` is just a waste of CPU cycles. This smacks suspiciously of bad design: it seems like someone wanted a `Match` to have `Value` and `Success` properties, so that you didn't have to index into the `Groups` array, and the easiest way to do this was to make `Match` inherit from `Group`, even if few people would say that a "a Match is-a Group." Another consequence of this decision is that a `Match` has a `Captures` property, just like every other `Group`, even though a `Match` will never have more than one `Capture`, which is yet another alias for the match itself. (So `Object.ReferenceEquals(M, M.Captures[0])`. See the Chapter11\MatchResult C# project.)

■ **Tip** As a rule of thumb, don't refer to a `Match` object's `Captures` property, and don't refer to `Groups[0]`.

---

21. `Group` objects do **not** have a name property—the mapping from capture group names to capture group numbers is the responsibility of the `Regex` object. See the `GetGroupNames`, `GetGroupNumbers`, `GroupNameFromNumber`, and `GroupNumberFromName` methods.

Regex Match Methods

From binary tests to getting every match

There are three different matching methods: `IsMatch`, `Match`, and `Matches`. If all you really care about is whether the regex matched or not, the `IsMatch` method is a convenient shorthand for calling the `Match` method and examining the `Match` object it returns. That is, code like `ThisRegex.IsMatch(ThisText)` is a bit smaller and easier to read than `ThisRegex.Match(ThisText).Success`. (Since all you care about is `Success` and not all the details of the `Match`, calling `IsMatch` is also slightly faster than calling `Match`.)

The simplest `Regex.Match` overloads will return the **first** match in a target string (if any). When you want to process each match in turn, you can either call `Match.NextMatch`, which returns the next match in the original target string, or you can use one of the `Regex.Match` method's *Substring* overloads, which allows you to restrict the search to a portion of the target string.

You can also call `Regex.Matches` to get a `MatchCollection` containing all the matches in the target string. The `MatchCollection` class acts a lot like an array of `Match` objects, but implements optimizations that defer calling `Regex.Match` as long as possible. The `MatchCollection` maintains a private `ArrayList` (Chapter 12) that contains every `Match` that you've read from the `Items` property.[22] If you ask for a `Match` that's not in the list, the `MatchCollection` (effectively) calls `Match.NextMatch` until it has enough `Match` items, or until a `NextMatch` operation fails. (Reading the `Count` property forces the `MatchCollection` to create every `Match`.)

That is, calling `Matches` creates a `MatchCollection` but doesn't actually do any matching. In particular, this means that it's not especially expensive to break out of a `foreach` loop that examines `Matches`—you haven't 'paid for' any matches that you haven't seen.

Regex Replace Methods

Two main overloads, with permutations

There are a lot of `Regex.Replace` overloads, but there are only two basic kinds, and the plethora of overloads is produced by multiplying these two kinds by static and instance overloads, and then multiplying again by various *Substring* overloads. Both kinds find all regex matches within a target string, and return a new string where each match is replaced with a string based on the match. One kind of `Replace` method takes a `ReplacementPattern` string and replaces each `Match M` with `M.Result(ReplacementPattern)`. The other kind of `Replace` method passes each `Match` to a delegate, and replaces the match with the delegate's result.

The `Match.Result` form is useful for format conversions. For example, with a regex that matches unquoted attribute values within HTML tags and captures the attribute name and value, you can add quotes to every unquoted HTML attribute value in a whole document (see the `QuoteHtmlAttributes` method in the Chapter11\Replace project). Similarly, with a regex that matches US phone numbers like `(555) 555-1212`, `555-555-1212`, and `555.555.1212` and captures the area code and phone number fields, you can convert all recognized phone numbers to a single canonical form.

22. Surprisingly, 2.0 does still use an `ArrayList` and not a `List<Match>`.

The delegate form offers maximum flexibility. A regex that matches tokens between percent signs and captures the tokens can look up the captured tokens in a `Hashtable` or `Dictionary<,>` (Chapter 12) and return a string from the symbol table. Or, as in this method from the Chapter11\Replace project, you can use the delegate form to substitute the `{number}` syntax of `String.Format` for the `$number` syntax of `Match.Result`:

```
public static string Replace(string Input, Regex R, string Pattern)
{
  // Create an object[] with an entry for every capture group in R
  object[] Groups = new object[R.GetGroupNumbers().Length];

  // Pass Regex.Replace a C# 2.0 anonymous method
  return R.Replace(Input, delegate(Match M)
  {
    M.Groups.CopyTo(Groups, 0);
    return String.Format(Pattern, Groups);
  } );
}
```

The `Regex.Split` Method

More flexible than `String.Split`

The `Split` method uses regex matches to split a string into an array of strings. The result array contains the text **outside of** the matches, plus any capture groups. That is, the first item in the result is the text to the left of the first match, if any. If there is at least one match, and its `Groups.Count` is greater than 1, any successful captures are added to the results. And so on, to the last result item, which is the text to the right of the last match, if any.

For example,

```
Regex.Split("Tom, Dick, and Harry", @", \s* (and)? \s*",
  RegexOptions.IgnorePatternWhitespace|RegexOptions.ExplicitCapture);
```

returns a three string array, `{"Tom", "Dick", "Harry"}`.

If you care about what the delimiters are, you can have the regex capture them. For example,

```
Regex.Split("(555) 555-1212", @"\s* ([().-]) \s*",
  RegexOptions.IgnorePatternWhitespace);
```

returns a seven-string array, `{"", "(", "555", ")", "555", "-", "1212"}`. Note that the first string is empty, because there is no text to the left of the first regex match.

■**Note** At the risk of being repetitious, `Regex.Split` is a comparatively expensive operation because of all the `Substring` calls it entails.

Regex Options

Details on all nine option bits

Creating a `Regex` (or using one of the static methods) with just a pattern is the equivalent of explicitly specifying `RegexOptions.None`—you get all the default behavior. To get the optional behaviors, you construct a `RegexOptions` bitmap by *or*-ing together various `RegexOptions` values. For example,

```
RegexOptions.IgnorePatternWhitespace | RegexOptions.Compiled
```

I've mentioned some of these options already: this topic contains a quick summary of each option, in alphabetical order.

`Compiled`

Compile the regex state machine to actual CIL, instead of to a set of tables that define the various match states and the conditions that move the matcher from one state to another. Since the code must be compiled and jitted before first use, setup costs are even greater for the compiled option than for the interpretive default, but a compiled regex does match significantly faster than an interpreted regex.

In general, you should only use the compiled option when you will be using a regex frequently. Because of compiled regexs' greater speed, you may be tempted to use them even in one-shot cases like splitting a MIME `multipart` email message on the `boundary` string. While the greater execution speed may indeed pay for the greater setup costs when applied to a multi-megabyte message, a compiled regex (like all other CIL code) cannot be unloaded once it's been jitted. Compiling single-use regexs to CIL thus represents a sort of memory leak that may become significant in a long-running program; the tables for a normal, interpreted regex are reclaimed when (if) the `Regex` object is reclaimed.

`CultureInvariant`

When you specify `IgnoreCase`, case mapping is done using the current thread's `CurrentCulture` property. If you are matching, say, 7-bit ASCII Internet protocol text, you may not want this behavior. The `CultureInvariant` option forces case mapping to be done via `CultureInfo.InvariantCulture` (i.e., English, without any special country rules).

■ **Note** This option only matters when you also specify `IgnoreCase`.

`ECMAScript`

Restricts standard character classes (like `\s` and `\w`) to 7-bit ASCII values.

Note This option can be used **only** with the `IgnoreCase`, `Multiline`, and `Compiled` options—any other combination causes an exception.

`ExplicitCapture`

'Bare' parentheses are just logical groupings, not capture groups. To specify a capture group, you have to give it a name with `(?<Name> capture pattern)`. In general, capture groups are not free, and you should only capture substrings that you actually use. There is usually a tradeoff between the awkwardness of having to name the capture groups and the awkwardness of using `(?: non-capture pattern)` to specify a noncapture group.

`IgnoreCase`

Literals and character classes are not case sensitive. The case mapping is done in a culture-sensitive way; see the preceding `CultureInvariant` option. This corresponds to the Perl `i` option (except for the culture sensitivity).

`IgnorePatternWhitespace`

Ignore white space in patterns, and enable comments between a `#` and the end of the line. To match white space, use either `\s` or a more specific escape like `\x20`, `\t`, &c; to match a `#`, use `\#`. This corresponds exactly to the Perl `x` ("extended") option; I highly recommend you use it for all but the simplest regexs.

`Multiline`

Changes `^` to match the start of any line, not just the start of the string, and changes `$` to match the end of any line, not just the end of the string. This corresponds exactly to the Perl `m` option.

Note `Multiline` does not affect `Singleline` in any way and vice versa. `Multiline` controls the behavior of the `^` and `$` assertions; `Singleline` controls the behavior of the `.` wildcard.

`None`

All the default behaviors, as in Table 11-7. `None` can also be used as a null value when you are composing a bitmap by *or*-ing in individual option bits:

```
RegexOptions Bitmap = RegexOptions.None;
foreach (RegexOptions Option in SomeCollection) Bitmap |= Option;
```

Table 11-7. *Default* `Regex` *Behaviors, and Their* `RegexOptions` *Overrides*

Default Behavior	Override
Interpreted match, using tables that can be garbage collected.	`Compiled`
Ignoring case is culture sensitive.	`CultureInvariant`
Standard character classes include non-ASCII characters.	`ECMAScript`
All parentheses are both logical groupings and capture groups.	`ExplicitCapture`
All literals are case sensitive.	`IgnoreCase`
White-space characters are treated as literals, comments are not supported.	`IgnorePatternWhitespace`
`^` and `$` match the start and end of the string, not the start and end of a line.	`Multiline`
Match left to right.	`RightToLeft`
`.` matches every character except `\n`.	`Singleline`

`RightToLeft`

Starts matching the **last** pattern element to the **last** target character, and works backwards. The effect is to reverse the order of the `Matches`. Very useful when you want only the last match (or the last *N* matches) instead of the first match.

`Singleline`

Changes `.` to match any character, not any character except `\n`. This corresponds exactly to the Perl `s` option.

Note `Singleline` does not affect `Multiline` in any way and vice versa. `Singleline` controls the behavior of the `.` wildcard; `Multiline` controls the behavior of the `^` and `$` assertions.

The `Regex.Escape` Method

An easy way to match literal text

There are a number of characters that have special meaning within a regex pattern. If you want to match them literally, you have to escape them, turning, e.g., `Read Me.txt` into `Read\x20Me\.txt`. It's generally easy enough to do this when you're composing a regex by hand, but when you are composing a regex programmatically and want to match some string literally, you need a way to escape every special character in the string. This is what the `Regex.Escape` method does: it replaces every character in a string that has special meaning in regexs with an escape sequence that allows the string to match as a literal.

For example,

```
String.Format("^--{0}$(.*?)^--{0}$", Regex.Escape(BoundaryString))
```

generates a regex pattern that will match one part of a MIME multipart email message. (This regex needs both the `Singleline` and `Multiline` modes, so that `.` matches new-line characters and so that `^` and `$` match the start and end of lines.)

Precompiled Regexs

Compile regexs to CIL ahead of time, so they only have to be jitted

In programs that use a large number of `Regex` static members, compiling all the regexs can add significantly to startup time. When you have a lot of `Regex` static members, it can make sense to use `Regex.CompileToAssembly` to place your regexs in a separate assembly. `CompileToAssembly` takes an array of `RegexCompilationInfo` objects (each of which specifies a pattern, options, a name, and a namespace) and creates an assembly containing precompiled regexs. As per the Chapter11\RegexAssembly\Consumer C# project, `CompileToAssembly` creates `Regex` descendant classes, which you have to create before you can use the regex. However, these regexs are always compiled to CIL (even if you don't specify `RegexOptions.Compiled`), and the compilation is done at the time the assembly is created, not when the regex is first used. It takes about as long to create a precompiled regex object as to pass a pattern string to the `Regex` constructor—but first use is much faster.

You're probably most likely to use `CompileToAssembly` as in the Chapter11\RegexAssembly C# project, to create an assembly that you install with your application. Alternatively, you might use the dynamic assembly-loading techniques in the Chapter11\DemandCreate C# project (and Chapter 14) to create the regex assembly the first time your application runs on a particular machine. Dynamic type creation **is** a bit more expensive than simply calling a constructor from an assembly you have a reference to, but it's still faster than compiling even a simple regex like `.+`.

Files

The `System.IO` namespace

As you would probably expect, FCL file IO is thoroughly object oriented. No `FindFirst/FindNext`, no "handles" or "cookies" or "opaque types." There are methods to probe and manipulate the file system; there are objects that represent directory entries; there are objects that represent open files.

■ **Note** Files and sockets are not unified; sockets have their own object model, in the `System.Net.Sockets` namespace. I don't cover sockets or networking in this book, because I'm not trying to be exhaustive. I'm trying to give just enough detail on just enough classes for you to see the recurring themes in FCL design. Before the end of these FCL chapters, you'll have expectations about how the various models will come into play in each new set of FCL classes that you use, especially when you have used similar code before.

Most of the filename manipulation and directory enumeration code is pretty straightforward, and the only real issue is knowing where in the System.IO namespace to find the code you want. However, reading and writing files is not very like any of Delphi's file models. FCL file IO involves stream reader and stream writer classes that access an open file (file stream) class. Accordingly, the "File System Information" subsection speeds through the simple stuff, while the "File IO" subsection goes into somewhat more detail on reading and writing file streams.

File System Information

Both low-level, filename-oriented classes and higher-level objects

The System.IO namespace provides two types of access to files. The Path, Directory, and File classes are static classes (in 1.1, they're sealed, abstract classes) with static methods. They provide logical grouping, but are only minimally object oriented. The DirectoryInfo and FileInfo classes are regular classes: each instance represents a file or directory, and there are properties to read and write timestamps and the like.

The Path class doesn't have a PathInfo counterpart. While Path has the GetTempFileName method that creates temporary files and the GetTempPath method that returns the name of the temporary directory, most Path members are direct analogs to various "file name utilities" in the Delphi SysUtils unit—they deal with path strings simply as strings that follow certain rules, and not as pointers to file system entries. Table 11-8 lists a few key Path members to give you an idea what sort of things to look in Path for.

Table 11-8. *Selected* Path *Members*

Path **Member**	SysUtils **Equivalent**	**Purpose**
DirectorySeparatorChar	PathDelim	Either \ or /, depending on OS.
ChangeExtension	ChangeFileExt	Takes a filename string; returns a filename string with a different extension. **Does not rename a file.**
GetDirectoryName	ExtractFileDir	Returns a string containing only the path, with no filename.
GetFileName	ExtractFileName	Returns a string containing only the filename, with no path.
GetFullPath	ExpandFileName	Gets a rooted, absolute filename.

The Directory and DirectoryInfo classes offer roughly parallel functionality, as do the File and FileInfo classes. When you only need to do a single thing with a file or directory, it will generally be faster and easier to use the low-level File and Directory classes. The static methods don't incur the overhead of creating an object that represents the file system entry. However, when you need to do several operations on each file or directory, it will generally be faster and easier to use the higher-level Info classes. Creating an Info object means that path string validation and security checks only have to be done once, and calling instance methods on file objects means that you don't have to keep passing the path string.

The Directory class has static methods like CreateDirectory and SetCurrentDirectory that do exactly what you'd expect. There are also methods to test whether a directory exists, delete directories, and get and set various timestamps. Where the FCL differs from both the VCL and the Win32 API is in enumerating directories. There's no FindFirst/FindNext: the Directory.GetDirectories methods return an array of strings containing the names of subdirectories; the GetFiles methods return an array of strings containing filenames; and the GetFileSystemEntries methods return an array of strings containing both file and directory names.

For example, the following FileNames method from the Chapter11\RecursiveEnumeration project enumerates every filename in a given directory. You can use this C# 2.0 method in a foreach statement like foreach (string FileName in Enumerate.FileNames(Path, Pattern)) that gives you each filename under Path that matches Pattern.

```
public static IEnumerable<string> FileNames(string Path, string Pattern)
{
  string[] Entries = Directory.GetFileSystemEntries(Path, Pattern);
  foreach (string Entry in Entries)
  {
    if (! Directory.Exists(Entry))
      yield return Entry;
    else
      foreach (string Child in FileNames(Entry, Pattern))
        yield return Child;
  }
}
```

This code is pretty straightforward except for the way it has to call Directory.Exists on every filename. We know the filename exists—but we have to use Directory.Exists or File.Exists to tell whether it names a directory or file.

The DirectoryInfo class has a constructor that takes a path name. Much like the Directory class, there are instance methods like Create and Delete and properties like Exists and Parent that do exactly what you'd expect. The enumeration methods return an array of Info objects, instead of an array of strings. As you'd expect, the GetFiles methods return an array of FileInfo objects, while the GetDirectories methods return an array of DirectoryInfo objects. The GetFileSystemInfos methods are confusingly described as returning an "array of strongly typed FileSystemInfo" objects—which **does** only mean that each entry in the FileSystemInfo[] is either a FileInfo or a DirectoryInfo, and you can tell them apart with the is operator or via GetType.

The FileSystemInfo version of the preceding FileNames method is not all that different:

```
public static IEnumerable<FileSystemInfo> Files
  (DirectoryInfo Directory, string Pattern)
{
  FileSystemInfo[] Entries = Directory.GetFileSystemInfos(Pattern);
  foreach (FileSystemInfo Entry in Entries)
  {
    if (Entry is FileInfo)
      yield return Entry;
    else
      foreach (FileSystemInfo Child in Files((DirectoryInfo)Entry, Pattern))
        yield return Child;
    }
}
```

As you can see, this is much like the string version that uses `Directory` methods instead of `DirectoryInfo` methods—except that the file vs. directory test is more straightforward. It's clearer that we are simply distinguishing files from directories with the `Entry is FileInfo` test[23] than with the `Directory.Exists(Entry)` test. These two examples capture in miniature the major difference between the `Directory` and `DirectoryInfo` classes—operations on `Info` objects are smaller and clearer, but the string version is three or four times faster than the `Info` object's version (because we're doing so little with each `Info` object we create).

There are other, less-important differences between the static and instance classes. For example, when you create a `DirectoryInfo` or `FileInfo` object, the constructor calls `Path.GetFullPath` to convert a partial name, like `.\this` or `..\that`, to an absolute path, rooted at a drive or share name. Thus, if you do `foreach (FileSystemInfo File in Enumerate.Files( @"\..", "*"))`, no `File.FullName` will have a `\..` in it. If you do `foreach (string FileName in Enumerate.FileNames( @"\..", "*"))`, each `FileName` will have a `\..` in it.

File IO

Reading and writing text and binary files

The `File` class has static methods like `Exists` and `Delete` that do exactly what you'd expect. It also has static methods like `Open` that return a new `FileStream` object; a `FileStream` object represents an open file. Similarly, the `FileInfo` class has an `Exists` property and a `Delete` method as well as various `Open` overloads that all return a new `FileStream`.

By itself, the `FileStream` only offers byte-by-byte access to a file. There are blocking and asynchronous methods to read and write individual bytes or byte arrays, but usually you will use a `FileStream` through a Reader or Writer object. The `StreamReader` and `StreamWriter` classes are character-oriented classes that read and write text files; the `BinaryReader` and `BinaryWriter` read and write binary streams.

When you create a Reader or Writer object, you pass it either an open `FileStream` or a filename and let it create the `FileStream` for you. When you are done reading and writing the file, you should `Close` the Reader or Writer object. This closes the underlying `FileStream`. The Reader and Writer classes implement `IDisposable`, and call `Close` in their `Dispose` method. Thus, creating a Reader and Writer object within the to-be-`Dispose()`d expression of a `uses` statement guarantees that the file will be closed at the end of the `uses` statement.

For example, the following method reads a text file to a string:

23. We could also read the `Entry.Attributes` property, and test whether the `Directory` bit is set or do `Entry.GetType() == typeof(DirectoryInfo)`—any speed difference between the three tests is swamped by the overhead of creating `FileSystemInfo` objects plus the overhead of actual file IO.

```
public static string Read(string FileName)
{
  using (StreamReader Reader = new StreamReader(FileName))
  {
    int StreamLength = (int) Reader.BaseStream.Length;
    char[] Text = new char[StreamLength];
    Reader.Read(Text, 0, StreamLength);
    return new String(Text);
  }
}
```

This `Read` method creates a `StreamReader` that opens the `FileName` file, and reads the `Length` of the `BaseStream`, which is the actual open file. It casts this to an `int`[24] as `Stream.Length` is a `long`; creates a `char` array that will hold all the characters in the file; reads them in with `Reader.Read()`; and calls the `String` constructor that turns a character array to a string. This passes over each character twice (reading it to the character array[25] and then into the string), but this can't be helped, as there is no way to create a string and then modify its character array.

The following Delphi function from the Chapter11\ReadText project also reads a text file to a string:

```
function ReadFile(FileName: string): string;
var
  Reader: StreamReader;
begin
  Reader := StreamReader.Create(FileName);
  try
    Result := Reader.ReadToEnd;
  finally
    Reader.Close;
  end;
end;
```

(as, for that matter, does the FCL 2.0 method `File.ReadAll`), but the C# method illustrates the actual techniques involved, and will generally be faster than `StreamReader.ReadToEnd`, which creates a `StringBuilder` and reads the file in a series of chunks.

The `StreamReader` class also has a `ReadLine` method that allows you to read a text file a line at a time.

When you create a `StreamReader` (or a `StreamWriter`) without an explicit `Encoding` parameter, it defaults to `System.Text.Encoding.UTF8`, which is, basically, the standard Latin-1 8-bit character set, with escape sequences for multibyte Unicode characters. The default `UTF8` encoding reads standard 8-bit text files and converts each character to 16-bit Unicode characters. You can use an explicit `Encoding` parameter to read or write UTF7 or 16-bit or 32-bit Unicode files.

For example, the following method writes a string to a 16-bit Unicode file:

24. Raising an exception if the stream is too long to fit in a string.

25. Reading is further complicated when the input stream may contain UTF escape sequences.

```
public static void Write16(string FileName, string FileText)
{
  // The second, false parameter to the StreamWriter constructor makes FileText
  // replace any existing text in the file; a true parameter would append, instead.
  using (StreamWriter Writer = new StreamWriter(FileName, false, Encoding.Unicode))
    Writer.Write(FileText);
}
```

The preceding method writes a string to a file, then closes the file. The `Write` (and the `WriteLine` which, like Delphi's `WriteLn`, appends a line break after the `Write`) method is heavily overloaded, and you can `Write` numbers, booleans, and objects to the stream. As with `String.Concat`, these methods are *Simpler* and *Smaller* than peppering your code with calls to `ToString`. Both `Write` and `WriteLine` also have overloads that follow the *Format* model, taking a format string and any number of `object` parameters, passing them to `String.Format`, and writing the result to the stream.

`StreamReader` and `StreamWriter` are implementations of the abstract `TextReader` and `TextWriter` classes. Their 'sibling' classes, `StringReader` and `StringWriter`, allow you to use the text stream methods to read and write strings. These classes are useful when the same code needs to write to either a file or a string,[26] but should probably be avoided when you only need to read and write strings. A `StringWriter` is basically just a wrapper around a `StringBuilder` and doesn't really offer any extra functionality to make up for the extra cost. Similarly, while a `StringReader` does allow you to process a string line by line, you can get similar functionality from `String.Split`, and you can get better performance with substring techniques that don't create a new string for each line.

Binary streams follow the same general pattern as text streams—an `IDisposable` reader or writer class that gives access to a `Stream` instance, and closes the stream when you close the reader or writer—except that the `BinaryReader` and `BinaryWriter` constructors don't have overloads that take a filename. You always have to explicitly open a stream and pass the `Stream` object to the reader or writer.

The binary reader and writer classes have methods to read and write the standard CLR primitives (including strings), but there are no methods to write compound types like records or classes. Field alignment and layout within a compound type is entirely up to the jitter, and can vary from CPU to CPU, and from CLR version to CLR version. You have to read and write element by element, in a way that's probably quite familiar: read and write in the same order, and start every file with some sort of format identifier, preferably one containing version information.

■ **Note** .NET also includes serialization support (Chapter 14) that makes it very easy to persist data structures, or to pass them from one machine to another. You will generally use raw binary streams only when you need to read or write standard file formats, or when IO speed really matters.

---

26. For example, automating a console application by reading input from a string.

The .NET Console

Standard input and standard output

The `System.Console` class is used much like Delphi's `ReadLn` and `WriteLn`, but is somewhat broader. The `Console` class has `ReadLine` and `WriteLine` (and `Read` and `Write`) methods that parallel the preceding `TextReader` and `TextWriter` classes and that (by default) read and write standard input and standard output, but it also has methods and properties to control the appearance of the console window, like `Title`, `WindowTop`, and `ForegroundColor`.

The `Console` class's static `In`, `Out`, and `Error` properties contain `TextReader` and `TextWriter` objects that are, by default, mapped to the standard input, output, and error streams. You can substitute any `TextReader` and `TextWriter` objects that you like; the `OpenStandardX` methods will (re)acquire the standard streams.

While in Win32 Delphi, `WriteLn` from a forms application will raise an exception,[27] on .NET any application can use `Console.Write` to write to the `Console.Out` stream (which, again, is usually standard output), even if it doesn't have an open console window. This can be useful for various 'daemon' processes that normally run invisibly but that nonetheless log their progress to an optional trace window. A "windows application" that doesn't create any forms is invisible, but any windows application can use the same Win32 API calls to create a console that Delphi or C# console applications use automatically.

The FCL offers no methods to manually create a console, but you can use the methods in the `Shemitz.GuiConsole` namespace to create and release a console on demand. See the Chapter11\ConsoleTest C# project for an example.

Key Points

String and file IO offer standard functionality and a complex but consistent interface

- Many FCL methods are heavily overloaded. These overloads tend to follow a few common models whose permutations account for much of this multiplicity.
- The `String` class contains a lot of functionality that's long been common in various string libraries but that may be new to Delphi programmers.
- If you already know regexs, the FCL regex classes are a joy to use. If you don't already know regexs, they're a wonderful place to start.
- File IO is object oriented, but the organizing principles should be very familiar.

27. Unless you've used Win32 calls to create a console.