Chapter 1

Hello, C#

My daughter has cycled through a number of musical instruments. With each one she is anxious to begin playing the classics—no, not Schubert or Schoenberg, but the Backstreet Boys and Britney Spears. Her various teachers, keen to keep her interest while grounding her in the fundamentals, have tended to indulge her. In a sense this chapter attempts the same precarious balance in presenting C#. In this context the classics are represented by Web Forms and Type Inheritance. The fundamentals are the seemingly mundane predefined language elements and mechanisms, such as scoping rules, arithmetic types, and namespaces. My approach is to introduce the language elements as they become necessary to implement a small first program. For those more traditionally minded, the chapter ends with a summary listing of the predefined language elements.

C# supports both integral and floating-point numeric types, as well as a Boolean type, a Unicode character type, and a high-precision decimal type. These are referred to as the *simple types.* Associated with these types is a set of operators, including addition (+), subtraction (–), equality (==), and inequality (!=). C# provides a predefined set of statements as well, such as the conditional `if` and `switch` statements and the looping `for`, `while`, and `foreach` statements. All of these, as well as the namespace and exception-handling mechanisms, are covered in this chapter.

1.1 A First C# Program

The traditional first program in a new language is one that prints *Hello, World!* on the user's console. In C# this program is implemented as follows:

```
// our first C# program
using System;
class Hello
{
      public static void Main()
      {
            Console.WriteLine( "Hello, World!" );
      }
}
```

When compiled and executed, this code generates the canonical

```
Hello, World!
```

Our program consists of four elements: (1) a comment, introduced by the double slash (`//`), (2) a `using` directive, (3) a class definition, and (4) a class *member function* (alternatively called a class *method*) named `Main()`.

A C# program begins execution in the class member function `Main()`. This is called the program entry point. `Main()` must be defined as `static`. In our example, we declare it as both `public` and `static`.

`public` identifies the level of access granted to `Main()`. A member of a class declared as `public` can be accessed from anywhere within the program. A class member is generally either a member function, performing a particular operation associated with the behavior of the class, or a data member, containing a value associated with the state of the class. Typically, class member functions are declared as `public` and data members are declared as `private`. (We'll look at member access levels again as we begin designing classes.)

Generally, the member functions of a class support the behavior associated with the class. For example, `WriteLine()` is a public member function of the `Console` class. `WriteLine()` prints its output to the user's console, followed by a new-line character. The `Console` class provides a `Write()` function as well. `Write()` prints its output to the terminal, but without inserting a new-line character. Typically, we use `Write()` when we wish the user to respond to a query posted to the console, and `WriteLine()` when we are simply displaying information. We'll see a relevant example shortly.

As C# programmers, our primary activity is the design and implementation of classes. What are classes? Usually they represent the entities in our applica-

tion domain. For example, if we are developing a library checkout system, we're likely to need classes such as `Book`, `Borrower`, and `DueDate` (an aspect of time).

Where do classes come from? Mostly from programmers like us, of course. Sometimes, it's our job to implement them. This book is designed primarily to make you an expert in doing just that. Sometimes the classes are already available. For example, the .NET System framework provides a `DateTime` class that is suitable for use in representing our `DueDate` abstraction. One of the challenges of becoming an expert C# programmer—and not a trivial one at that—is becoming familiar with the more than 1,000 classes defined within the .NET framework. I can't cover all of them here in this text, but we'll look at quite a number of classes, including support for regular expressions, threads, sockets, XML and Web programming, database support, and the new way of building a Windows application.

A challenging problem is how to logically organize a thousand or more classes so that users (that's us) can locate and make sense of them (and keep the names from colliding with one another). Physically, we can organize them within directories. For example, all the classes supporting Active Server Pages (ASP) can be stored in an `ASP.NET` directory under a root `System.NET` directory. This makes the organization reasonably clear to someone poking around the file directory structure.

What about within programs? As it turns out, there is an analogous organizing mechanism within C# itself. Rather than defining a physical directory, we identify a *namespace*. The most inclusive namespace for the .NET framework is called `System`. The `Console` class, for example, is defined within the `System` namespace.

Groups of classes that support a common abstraction are given their own namespace defined within the `System` namespace. For example, an `Xml` namespace is defined within the `System` namespace. (We say that the `Xml` namespace is nested within the `System` namespace.) Several namespaces in turn are nested within the `Xml` namespace. There is a `Serialization` namespace, for example, as well as `XPath`, `Xsl`, and `Schema` namespaces. These separate namespaces within the enclosing `Xml` namespace are factored

out to encapsulate and localize shared functionality within the general scope of XML. This arrangement makes it easier to identify the support, for example, that .NET provides for the World Wide Web Consortium (W3C) `XPath` recommendation. Other namespaces nested within the System namespace include `IO`, containing file and directory classes, `Collections`, `Threading`, `Web`, and so on.

In a directory structure, we indicate the relationship of contained and containing directories with the backslash (`\`), at least under Windows—for example,

```
System\Xml\XPath
```

With namespaces, similar contained and containing relationships are indicated by the *scope operator* (`.`) in place of a backslash—for example,

```
System.Xml.XPath
```

In both cases we know that `XPath` is contained within `Xml`, which is contained within `System`.

Whenever we refer to a name in a C# program, the compiler must resolve that name to an actual declaration of something somewhere within our program. For example, when we write

```
Console.WriteLine( "Hello, World" );
```

the compiler must somehow discover that `Console` is a class name and that `WriteLine()` is a member function within the `Console` class—that is, within the scope of the `Console` class definition. Because we have defined only the `Hello` class in our file, without our help the compiler is unable to resolve what the name `Console` refers to. Whenever the compiler cannot resolve what a name refers to, it generates a compile-time error, which stops our program from building:

```
C:\C#Programs\hello\hello.cs(7):

     The type or namespace name 'Console' does
     not exist in the class or namespace
```

The `using directive in our program`,

```
using System;
```

directs the compiler to look in the `System` namespace for any names that it cannot immediately resolve within the file it is processing—in this case, the file that contains the definition of our `Hello` class and its `Main()` member function.

Alternatively, we can explicitly tell the compiler where to look:

```
System.Console.WriteLine( "Hello, World" );
```

Some people—actually some very smart and otherwise quite decent people—believe that explicit listing of the *fully qualified name*—that is, the one that identifies the full set of namespaces in which a class is contained—is always preferable to a `using` directive. They point out that the fully qualified name clearly identifies where the class is found, and they believe that is useful information (even if it is repeated 14 times within 20 adjacent lines). I don't share that belief (and I really don't like all that typing). In my text—and this is one of the reasons we authors write books—the fully qualified name of a class is *never* used,[1] except to disambiguate the use of a type name (see Section 1.2 for an illustration of situations in which this is necessary).

Earlier I wrote that classes come mostly either from other programmers or from libraries provided by the development system. Where else do they come from? The C# language itself. C# predefines several heavily used data types, such as integers, single- and double-precision floating-point types, and strings. Each has an associated *type specifier* that identifies the type within C#: `int` represents the primitive integer type; `float`, the primitive single-precision type; `double`, the double-precision type; and `string`, the string type. (See Tables 1.2 and 1.3 in Section 1.18.2 for a list of the predefined numeric types.)

For example, an alternative implementation of our simple program defines a `string` object initialized with the `"Hello, World!"` string literal:

```
public static void Main() {
    string greeting = "Hello, World!"
    Console.WriteLine( greeting );
}
```

1. The Visual Studio wizards, such as Windows Forms and Web Forms, generate fully qualified names. However, because the names are machine generated, this does not really qualify as a counterexample.

`string` is a C# *keyword* — that is, a word reserved by the C# language and invested with special meaning. `public`, `static`, and `void` are also keywords in the language. `greeting` is referred to as an *identifier.* It provides a name for an object of type `string`. Identifiers in C# must begin with either an underscore (_) or an alphabet character. The names are case sensitive, so `greeting`, `Greeting`, and `Greeting1` each represent a unique identifier.

A common flash point among programmers centers on whether compound names should be separated by an underscore, as in `xml_text_reader`, or by capitalization of the first letter of each internal word, as in `xmlTextReader`. By convention, identifiers that represent class names usually begin with a capital letter, as in `XmlTextReader`.

Within a unit of program visibility referred to as a *declaration space*, or *scope*, identifiers must be unique. At *local scope*—that is, within a function body, such as within our definition of `Main()`—this is not a problem because we generally control the entire definition of any object within our function. As the extent of the scope widens—that is, as the number of programmers or organizations involved increases—the problem of unique identifiers becomes more difficult. This is where namespaces come into the picture.

1.2 Namespaces

Namespaces are a mechanism for controlling the visibility of names within a program. They are intended to help facilitate the combination of program components from various sources by minimizing name conflicts between identifiers. Before we look at the namespace mechanism, let's make sure we understand the problem that namespaces were invented to solve.

Names not placed within a namespace are automatically placed in a single unnamed global declaration space. These names are visible throughout the program, regardless of whether they occur in the same or a separate program file. Each name in the global declaration space must be unique for the program to build. Global names make it difficult to incorporate independent components into our programs.

For example, imagine that you develop a two-dimensional (2D) graphics component and name one of your global classes `Point`. You use your component,

and everything works fine. You tell some of your friends about it, and they naturally want to use it as well.

Meanwhile, I develop a three-dimensional (3D) graphics component and in turn name one of my global classes `Point`. I use my component, and everything also works fine. I show it to some of my friends. They're excited about it and wish to use it as well. So far, everyone is happy—well, at least about our coding projects.

Now imagine that we have a friend in common. She's implementing a 2D/3D game engine and would like use our two components, both of which come highly praised. Unfortunately, when she includes both within her application, the two independent uses of the `Point` identifier result in a compile-time error. Her game engine fails to build. Because she does not own either component, there is no easy fix for the two components to work together.

Namespaces provide a general solution to the problem of global name collision. A namespace is given a name within which the classes and other types we define are encapsulated.[2] That is, the names placed within a namespace are not visible within the general program. We say that a namespace represents an independent declaration space or scope.

Let's help our mutual friend by providing separate namespaces for our two `Point` class instances:

```
namespace DisneyAnimation_2DGraphics
{
  public class Point { ... }

  // ...
}

namespace DreamWorksAnimation_3DGraphics
{
  public class Point { ... }

  // ...
}
```

2. Only namespaces and types can be declared within the global namespace. A function can be declared only as a class member. A data object can be either a class member or a local object within a function, such as our declaration of `greeting`.

The keyword `namespace` introduces the namespace definition. Following that is a name that uniquely identifies the namespace. (If we reuse the name of an existing namespace, the compiler assumes that we wish to add additional declarations to the existing namespace. The fact that the two uses of the namespace name do not collide allows us to spread the namespace declaration across files.)

The contents of each namespace are placed within a pair of curly braces. Our two `Point` classes are no longer visible to the general program; each is nested within its respective namespace. We say that each is a member of its respective namespace.

The `using` directive in this case is too much of a solution. If our friend opens both namespaces to the program—

```
using DisneyAnimation_2DGraphics;
using DreamWorksAnimation_3DGraphics;
```

the unqualified use of the `Point` identifier still results in a compile-time error. To unambiguously reference this or that `Point` class, we must use the fully qualified name—for example,

```
DreamWorksAnimation_3DGraphics.Point origin;
```

If we read it from right to left, this declares `origin` to be an instance of class `Point` defined within the `DreamWorksAnimation_3DGraphics` namespace.

The ambiguity within and between namespaces is handled differently depending on the perceived amount of control we have over the name conflict. In the simplest case, two uses of the same name occur within a single declaration space, triggering an immediate compile-time error when the second use of the name is encountered. The assumption is that the affected programmer has the ability to modify or rename identifiers within the working declaration space where the name conflict occurs.

It becomes less clear what should happen when the conflict occurs across namespaces. In one case, we open two independent namespaces, each of which contains a use of the same name, such as the two instances of `Point`. If we make explicit use of the multiply-defined `Point` identifier, an error is generated;

the compiler does not try to prioritize one use over the other or otherwise disambiguate the reference. One solution, as we did earlier, is to qualify each identifier's access. Alternatively, we can define an alias for either one or all of the multiply-defined instances. We do this with a variant of the `using` directive, as follows:

```
namespace GameApp
{
    // exposes the two instances of Point
    using DisneyAnimation_2DGraphics;
    using DreamWorksAnimation_3DGraphics;

    // OK: create unique identifiers for each instance
    using Point2D = DisneyAnimation_2DGraphics.Point;
    using Point3D = DreamWorksAnimation_3DGraphics.Point;

    class myClass
    {
           Point2D thisPoint;
           Point3D thatPoint;
    }
}
```

The alias is valid only within the current declaration space. That is, it doesn't introduce an additional permanent type name associated with the class. If we try to use it across namespaces, such as in the following:

```
namespace GameEngine
{
    class App
    {
           // error: not recognized
           private GameApp.Point2D origin;
    }
}
```

the compiler wants nothing to do with it, generating the following message:

```
The type or namespace name 'Point2D' does not exist in the class
or namespace 'GameApp'
```

When we use a namespace, we generally have no idea how many names are defined within it. It would be very disruptive if each time we added an additional

namespace, we had to hold our breath while we recompiled to see if anything would now break. The language, therefore, minimizes the disruption that opening a namespace can cause to our program.

If two or more instances of an identifier, such as our two `Point` classes, are made visible within our working declaration space through multiple `using` directives, an error is triggered only with an unqualified use of the identifier. If we don't access the identifier, the ambiguity remains latent, and neither an error nor a warning is issued.

If an identifier is made visible through a `using` directive that duplicates an identifier we have defined, our identifier has precedence. An unqualified use of the identifier always resolves to our defined instance. In effect, our instance hides the visibility of the identifier contained within the namespace. The program continues to work exactly as it had prior to our use of the additional namespace.

What sorts of names should be given to our namespaces? Generic names such as `Drawing`, `Data`, `Math`, and so on are unlikely to be unique. A recommended strategy is to add a prefix that identifies your organization or project group.

Namespaces are a necessary element of component development. As we've seen, they facilitate the reuse of our software in other program environments. For less ambitious programs, however, such as the `Hello` program at the start of this chapter, the use of a namespace is unnecessary.

1.3 Alternative Forms of the `Main()` Function

In the rest of this chapter we'll explore the predefined elements of the C# language as we implement a small program called `WordCount`. `WordCount` opens a user-specified text file and calculates the number of occurrences of each word within the file. The results are sorted in dictionary order and written to an output file. In addition, the program supports two command-line options:

1. `-t` causes the program to turn a trace facility on; by default, tracing is off.
2. `-s` causes the program to calculate and report the amount of time it takes to read the file, process the words, and write the results; by default, timings are not reported.

Our first task in `Main()` is to access the command-line arguments passed in to our program, if any. We do that by using a second form of `Main()`, which defines a one-parameter function signature:

```
class EntryPoint
{
      public static void Main( string [] args ) {}
}
```

`args` is defined as an array of string elements. `args` is automatically filled with any command-line arguments specified by the user. For example, if the user invoked our program as follows:

```
WordCount -s mytext.txt
```

the first element of `args` would hold `-s` and the second element would hold `mytext.txt`.

In addition, either form of the `Main()` function may optionally return a value of type `int`:

```
public static int Main() {}
public static int Main( string [] args ) {}
```

The return value is treated as the exit status of the program. By convention, a return value of `0` indicates that the program completed successfully. A nonzero value indicates some form of program failure. A `void` return type, paradoxically, internally results in a return status of `0`; that is, the execution environment always interprets the program as having succeeded. In the next section we look at how we can use this second form of `Main()`.

1.4 Making a Statement

The first thing we need to do is determine if the user specified any arguments. We do this by asking `args` the number of elements it contains.[3] For our program I decided that if the user doesn't supply the necessary command-line arguments, the program shuts down. (As an exercise, you may wish to reimplement the pro-

3. In C#, we cannot write `if (!args.Length)` to test whether the array is empty because `0` is not interpreted as meaning false.

gram to allow the user to interactively enter the desired options. The program is certainly friendlier that way.)

In my implementation, if `args` is empty, the program prints an explanation of the correct way to invoke `WordCount`, then exits using a `return` statement. (The `return` statement causes the function in which it occurs to terminate—that is, to return to the location from which it was invoked.)

```
public static void Main( string [] args )
{
 if ( args.Length == 0 )
 {
      display_usage();
      return;
 }
}
```

`Length` is a *property* of an array. It holds a count of the number of elements currently stored in the array. The test of `Length` is placed within the conditional test of the C# `if` statement. If the test evaluates to true, the statement immediately following the test is executed; otherwise it is ignored. If multiple statements need be executed, as in our example, they must be enclosed in curly braces (the text within the braces is called a *statement block*).

A common mistake that beginners make is to forget the statement block when they wish to execute two or more statements: [4]

```
// this is an incorrect usage of the if statement
if ( args.Length == 0 )
     display_usage();
     return;
```

The indentation of `return` reflects the programmer's intention. It does not, however, reflect the program's behavior. Without the statement block, only the function is conditionally executed; the `return` statement is executed whether or not the array is empty.

The `return` statement can also return a value. This value becomes the return value of the function—for example,

4. These code fragments all occur within the `Main()` function. To save space, I do not show the enclosing declaration of `Main()`.

```
public static int Main( string [] args )
{

      if ( args.Length == 0 )
      {
           display_usage();
           return -1;    // indicate failure
      }
}
```

The rule is that the value following the `return` statement must be compatible with the return type of the function. *Compatible* can mean one of two things. In the simplest case, the value being returned is the same type as that indicated as the return type of the function. The value `-1`, for example, is of type `int`. The second meaning of compatible requires that an implicit conversion exist between the actual return value and the function's return type.

The `if-else` statement allows us to select between alternative statements on the basis of the truth of a particular condition. The `else` clause represents a statement or statement block to be executed if the tested condition evaluates to false. For example, if we chose not to immediately return on discovering an empty `args` array, we could provide the following `if-else` statement instead:

```
if ( args.Length == 0 )
     display_usage();
else { /* do everything else here ... */ }
```

To access the individual command-line options, we'll use a `foreach` loop to iterate across the array, reading each element in turn. For example, the following loop statement prints each option to the user's console:

```
foreach ( string option in args )
      Console.WriteLine( option );
```

`option` is a read-only `string` object. It is visible only within the body of the `foreach` statement. Within each iteration of the loop, `option` is set to refer to the next element of the `args` array.

For our program, we'll compare each `string` element against the set of supported options. If the string does not match any of the options, we'll check to see if the string represents a text file. Whenever we are testing a series of mutu-

ally exclusive conditions, as we are in this case, we typically combine the tests into a chain of `if-else-if` statements—for example,

```
bool traceOn = false,
bool spyOn   = false;

foreach ( string option in args )
{
    if ( option.Equals( "-t" ))
         traceOn = true;
    else
    if ( option.Equals( "-s" ))
         spyOn = true;
    else
    if ( option.Equals( "-h" ))
       { display_usage(); return; }
    else
         check_valid_file_type( option );
}
```

The `bool` keyword represents a Boolean data type that can be assigned the literal values `true` or `false`. In our example, `traceOn` and `spyOn` represent two Boolean objects initialized to `false`.

`Equals()` is a nonstatic member function of `string`. *Nonstatic* member functions (also referred to as *instance* member functions) are invoked through an instance of the class for which the function is a member—in this case, the `string` object `option`. The expression

```
option.Equals( "-t" )
```

instructs the compiler to invoke the `string` instance method `Equals()` to compare the string stored within `option` with the string literal `"-t"`. If the two are equal, `Equals()` returns `true`; otherwise, it returns `false`.

If the mutually exclusive conditions are constant expressions[5], we can turn our chain of `if-else-if` statements into the somewhat more readable `switch` statement—for example,

5. A constant expression represents a value that can be evaluated at compile time. Typically, this means that the expression cannot contain a data object. (The value associated with a data object cannot be evaluated until runtime execution of our program.)

```
foreach ( string option in args )
    switch ( option )
    {
      case "-t":
            traceOn = true;
            break;

      case "-s":
            spyOn = true;
            break;

      case "-h":
                  display_usage();
                  return;

           default:
                  check_valid_file_type( option );
                  break;
    }
```

The `switch` statement can be used to test a value of an integral type, a `char` type, an enumeration, or a `string` type. The `switch` keyword is followed by the expression enclosed in parentheses. A series of `case` labels follows the `switch` keyword, each specifying a constant expression. Each `case` label must specify a unique value.

The result of the expression is compared against each `case` label in turn. If there is a match, the statements following the `case` label are executed. If there is no match and the `default` label is present, the statements associated with the `default` label are executed. If there is no match and no `default` label, nothing happens. (There can be only one `default` label.)

Each nonempty `case` label must be followed either by a `break` statement or by another terminating statement, such as a `return` or a `throw`; otherwise a compiler error results. (`throw` passes program control out of the current function into the runtime exception-handling mechanism. We look at exception handling in Section 1.17. The `break` statement passes program control to the statement following the terminating curly brace of the `switch` statement.)

An empty `case` label is the one exception to this rule. It does need not have a `break` statement. We do this typically when multiple values require the same action—for example,

```
switch ( next_char )
{
  case 'a':
  case 'A':
      acnt++;
      break;

  // to illustrate an alternative syntax ...
  case 'e':  case 'E': ecnt++; break;

  // ... the other vowels

  case '\0': return; // OK

  default: non_vowel_cnt++; break;
}
```

If we wish to execute the body of two `case` labels—the first of which is not empty—we must use a special `goto` statement that targets either an explicit `case` label or the `default` label:

```
switch ( text_word )
{
  case "C#":
  case "c#":

      csharp_cnt++;
      goto default;

  case "C++":
  case "c++":

      cplus_cnt++;
      goto default;

  case "Java":
  case "java":  goto case "C#";

  default:

      word_cnt++;
      break;
}
```

1.5 Opening a Text File for Reading and Writing

Let's assume that the user has entered a valid text file name for the program. Our job, then, is to open the file, read its contents, and after processing those contents, write the results out to a second file, which we need to create. Let's see how we do this.

Support for file input/output is encapsulated in the `System.IO` namespace. So the first thing we need to do is open the namespace to the compiler:

```
using System.IO;
```

Text files are read and written through the `StreamReader` and `StreamWriter` classes. There are various ways to create instances of these classes—for example,

```
string file_name = @"C:\fictions\gnome.txt";

StreamReader freader = File.OpenText( file_name );
StreamWriter fwriter =
          File.CreateText( @"C:\fictions\gnome.diag" );
```

`OpenText()` returns a `StreamReader` instance bound to the file represented by the string argument passed to it. In this case, it opens a text file stored in the `fictions` directory on drive `C:`. The file represented by the string must exist, and the user must have permission to open it; otherwise `OpenText()` throws an exception.

The `@` character identifies the string literal that follows it as a *verbatim* string literal. In an ordinary string literal, a backslash is treated as a special character. For example, when we write `"\n"`, the backslash and the `n` are interpreted as an escape sequence representing the new-line character. When we wish an actual backslash to appear in a string literal, we must *escape* it by preceding it with an additional backslash—for example,

```
string fname1 = "C:\\programs\\primer\\basic\\hello.cs";
```

Within a verbatim string literal, special characters, such as the backslash, do not need to be escaped.

A second difference between an ordinary and a verbatim string literal is the ability of a verbatim string literal to span multiple lines. The nested white space

within the multiple-line verbatim string literal, such as a new-line character or a tab, is preserved. This allows for the storage and generation of formatted text blocks. For example, here is how we might implement `display_usage()`:

```
public void display_usage()
{
   string usage =
      @"usage:  WordCount [-s] [-t] [-h] textfile.txt
          where [] indicates an optional argument
            -s prints a series of performance measurements
            -t prints a trace of the program
            -h prints this message";
    Console.WriteLine( usage );
}
```

`CreateText()` returns a `StreamWriter` instance. The file represented by the string argument passed to it, if it exists, is overwritten. To append to a file's existing text rather than overwriting it, we use `AppendText()`.

`StreamReader` provides a collection of read methods allowing us to read a single character, a block of characters, or, using `ReadLine()`, a line of text. (There is also a `Peek()` method to read one character ahead without extracting it from the file.) `StreamWriter` provides instances of both `WriteLine()` and `Write()`.

The following code segment reads each line of a text file in turn, assigning it to the string object `text_line`. `StreamReader` signals that it has reached the end of the file by returning a `null` string. The additional parentheses around the assignment of `text_line` are necessary because of what is called *operator precedence* (see Section 1.18.4 for a discussion):

```
string text_line;
while (( text_line = freader.ReadLine() ) != null )
{
      // write to output file
      fwriter.WriteLine( text_line );
}

// must explicitly close the readers
freader.Close();
fwriter.Close();
```

When we finish with the readers, we must invoke their associated `Close()` member functions in order to free the resources associated with them.[6]

1.6 Formatting Output

In addition to writing each line of text to the output file, let's extend the previous code segment to also write the line to the user's console. When writing to the console, however, we want to indicate the line number and the length of the line in characters, as well as the text itself. In addition, we don't want to echo an empty line. Here is the relevant portion of the modified code segment:

```
string text_line;
int    line_cnt = 1;

while (( text_line = freader.ReadLine() ) != null )
{
     // don't format empty lines
     if ( text_line.Length == 0 )
     {
            Console.WriteLine();
            continue;
     }


     // format output to console:
     // 1 (42): Master Timothy Gnome left home one morning

     Console.WriteLine( "{0} ({2}): {1}",
               line_cnt++, text_line, text_line.Length );
}
```

The `continue` statement allows us to short-circuit the remaining portion of the loop body. In this example, if the text line is empty, we write a new line to the console and then prematurely terminate this iteration of the loop body before writing the line of text to the console. The `continue` statement causes the next iteration of the loop to begin immediately.

6. If you are a C++ programmer, you are accustomed to having the class destructor automatically free resources. In a garbage-collected environment, however, destructors cannot provide that service. Rather we provide a `Dispose()` function, which is discussed in Section 4.8. `Close()` is an alternative form of `Dispose()`.

Similarly, a `break` statement causes the premature termination of the loop itself. For example, we might use the `break` statement when we are iterating through a collection searching for a value. Once the value has been found, we break out of the loop. An extreme example of this idiom is the nonterminating condition test for a loop—for example,

```
while ( true )
{
     // process until some condition occurs

     // ...

     if ( condition_occurs )
          break;
}
```

The assumption is that an internal state of the class or application causes the eventual invocation of the `break` statement, terminating the loop.

`WriteLine()` allows us to pass positional arguments within a literal string—for example,

```
Console.WriteLine( "Hello, {0}! Welcome to C#", user_name );
```

When a number enclosed within curly braces, such as `{0}`, is encountered within the literal string, it's treated as a placeholder for the associated value in the list of parameters that follows the literal string. `0` represents the first value, `1` represents the second value, and so on. The numbered placeholders can appear in any order, as in this example:

```
Console.WriteLine( "{0} ({2}): {1}",
     line_cnt++, text_line, text_line.Length );
```

The same position value can appear multiple times as well—for example,

```
Console.WriteLine( "Hello, {0}! Everyone, please welcome {0}",
                   user_name );
```

We can control numeric formatting of the built-in types through the addition of format characters. For example, a `C` interprets the value in terms of the local currency, `F` indicates a fixed-point format, `E` indicates an exponential (scientific)

format, and G leaves it to the system to pick the most compact form. Each may be followed by a value specifying the precision. For example, given the object

```
double d = 10850.795;
```

the WriteLine() statement

```
Console.WriteLine("{0} : {0:C2} : {0:F4} : {0:E2} : {0:G}",d);
```

generates the following output:

```
10850.795 : $10,850.80 : 10850.7950 : 1.09E+004 : 10850.795
```

We can use X or x to output the value in hexadecimal. X results in the uppercase values A through F; x results in lowercase a through f.

1.7 The `string` Type

Once we have read a line of text, we need to separate it into the individual words. The simplest method of doing that is to use the Split() method of string—for example,

```
string text_line;
string [] text_words;

while (( text_line = freader.ReadLine() ) != null )
{
      text_words = text_line.Split( null );
      // ...
}
```

Split() returns an array of string elements separated by a set of characters indicated by the user. If Split() is passed null, as it is in our example, it separates the elements of the original string using white space, such as a blank character or a tab. For example, the string

```
A beautiful fiery bird, he tells her, magical but untamed.
```

is split into an array of 10 string elements. Three of them, however—bird, her, and untamed—retain their adjacent punctuation. One strategy for removing the punctuation is to provide Split() with an explicit array of characters with which to separate the string—for example,

```
char [] separators = {
    ' ', '\n', '\t', // white space
    '.', '\"', ';', ',', '?', '!', ')', '(', '<', '>', '[', ']'
};

text_words = text_line.Split( separators );
```

A character literal is placed within single quotation marks. The new-line and tab characters are represented by the two-character escape sequences `\n` and `\t`. Each sequence represents a single character. The double quotation mark must also be escaped (`\"`) in order for it to be interpreted as a character rather than the beginning of a string literal.

The `string` type supports the subscript operator (`[]`)—but for read operations only. Indexing begins at zero, and extends to `Length-1` characters—for example,

```
for ( int ix = 0; ix < text_line.Length; ++ix )
   if ( text_line[ ix ] == '.' ) // OK: read access
    text_line[ ix ]  = '_';          // error: no write access
```

The `string` type does not support use of the `foreach` loop to iterate across the characters of its string.[7] The reason is the somewhat nonintuitive immutable aspect of a string object. Before we make sense of what that means, let me briefly make sense of the C# for loop statement.

The `for` loop consists of the following elements:

```
for ( init-statement; condition; expression )
      statement
```

`init-statement` is executed once before the loop is executed. In our example, `ix` is initialized to `0` before the loop begins executing.

`condition` serves as the loop control. It is evaluated before each iteration of the loop. For as many iterations as `condition` evaluates as true, `statement` is executed. `statement` can be either a single statement or a statement block. If `condition` evaluates to false on the first iteration, `statement` is

7. The technical reason is that the `String` class does not implement the `IEnumerable` interface. See Chapter 4 for a full discussion of the `IEnumerable` interface and interfaces in general.

never executed. In our example, `condition` tests whether `ix` is less than `text_line.Length`—that is, the count of the number of characters contained within the string. While `ix` continues to index a valid character element, the loop continues to execute.

`expression` is evaluated after each iteration of the loop. It is typically used to modify the objects initialized within `init-statement` and tested in `condition`. If `condition` evaluates to false on the first iteration, `expression` is never executed. In our example, `ix` is incremented following each loop iteration.

The reason we cannot write to the individual characters of the underlying string literal is that string objects are treated as immutable. Whenever it seems as if we are changing the value of a string object, what has actually happened is that a new string object containing those changes has been created.

For example, to do an accurate occurrence count of words within a text file, we'll want to recognize *A* and *a* as being the same word. One way to do that is to change each string to all lowercase before we store it:

```
while (( text_line = freader.ReadLine() ) != null )
{
      // oops: this doesn't work as we intended ...
      text_line.ToLower();
      text_words = text_line.Split( null );

      // ...
}
```

`ToLower()` correctly changes all uppercase characters within `text_line` to lowercase. (There is a `ToUpper()` member function as well.) The problem is that the new representation is stored in a new string object that is returned by the call to `ToLower()`. Because we do not assign the new string object to anything, it's just tossed away. `text_line` is not changed, and it won't change unless we reassign the return value to it, as shown here:

```
text_line = text_line.ToLower();
```

To minimize the number of new `string` objects generated when we are making multiple modifications to a `string`, we use the `StringBuilder` class. See Section 4.9 for an illustration.

1.8 Local Objects

A data object must be defined within either a function or a class; it cannot exist as an independent object either in a namespace or within the global declaration space. Objects that are defined within a function are called *local objects*. A local object comes into existence when its enclosing function begins execution. It ceases to exist when the function terminates. A local object is not provided with a default initial value.

Before a local object can be read or written to, the compiler must feel sure that the object has been assigned to. The simplest way to reassure the compiler is to initialize the local object when we define it—for example,

```
int ival = 1024;
```

This statement defines an integer object `ival` and initializes it with a value of `1024`.

iSometimes it doesn't make sense to initialize an object because we don't use it until after it has the target of an assignment. For example, consider `user_name` in the following program fragment:

```
static int Main()
{
      string    user_name;
      int       num_tries = 0;
      const int max_tries = 4;

      while ( num_tries < max_tries )
      {
            // generate user message ...

            ++num_tries;
            user_name = Console.ReadLine();

            // test whether entry is valid
      }

      // compiler error here!
      // use of unassigned local variable user_name
      Console.WriteLine( "Hello, {0}", user_name );
      return 0;
}
```

By inspection, we see that `user_name` must always be assigned to within the `while` loop. We know this because `num_tries` is initialized to `0`. The `while` loop is always evaluated at least once. The compiler, however, flags the use of `user_name` in the `WriteLine()` statement as the illegal use of an unassigned local object. What do we know that it doesn't?

Each time we access a local object, the compiler checks that the object has been *definitely assigned* to. It determines this through *static* flow analysis—that is, an analysis of what it can know at compile time. The compiler cannot know the value of a nonconstant object, such as `num_tries`, even if its value is painfully obvious to us. The static flow analysis carried out by the compiler assumes that a nonconstant object can potentially hold any value. Under that assumption, the `while` loop is not guaranteed to execute. Therefore, `user_name` is not guaranteed to be assigned to, and the compiler thus issues the error message.

The compiler can fully evaluate only constant expressions, such as the literal values `7` or `'c'`, and nonwritable constant objects, such as `max_tries`. Nonconstant objects and expressions can be definitely known only during runtime. This is why the compile treats all nonconstants as potentially holding any value. It's the most conservative and therefore safest approach.

One fix to our program, of course, is to provide a throwaway initial value:

```
string user_name = null;
```

An alternative solution is to use the fourth of our available loop statements in C#, the `do-while` loop statement. The `do-while` loop always executes its loop body at least once before evaluating a condition. If we rewrite our program to use the `do-while` loop, even the compiler can recognize that `user_name` is guaranteed to be assigned because its assignment is independent of the value of the nonconstant `num_tries`:

```
do
{
    // generate user message ...
    ++num_tries;
    user_name = Console.ReadLine();
    // test whether entry is valid
} while ( num_tries < max_tries );
```

Local objects are treated differently from other objects in that their use is order dependent. A local object cannot be used until it has been declared. There is also a subtle extension to the rule: Once a name has been used within a local scope, it is an error to change the meaning of that use by introducing a new declaration of that name. Let's look at an example.

```
public class EntryPoint
{
   private string str = "hello, field";
   public void local_member()
   {
          // OK: refers to the private member
          /* 1 */ str = "set locally";

          // error: This declaration changes the
          // meaning of the previous statement
          /* 2 */ string str = "hello, local";
    }
}
```

At `1`, the assignment of `str` is resolved to the private member of the class `EntryPoint`. At `2`, however, the meaning of `str` changes with the declaration of a local `str` string object. C# does not allow this sort of change in the meaning of a local identifier. The occurrence of the local definition of `str` triggers a compile-time error.

What if we move the declaration of the private `str` data member so that it occurs after the definition of the method? That doesn't change the behavior. The entire class definition is inspected before the body of each member function is evaluated. The name and type of each class member are recorded within the class declaration space for subsequent lookup. The order of member declarations is not significant—for example,

```
public class EntryPoint
{
   // OK: let's place this first
   public void local_member()
   {
          // still refers to the private class member
          /* 1 */ str = "set locally";
```

```
            // still the same error
            /* 2 */ string str = "hello, local";
        }

        // position of member does not change its visibility
        private string str = "hello, field";
    }
```

Each local block maintains a declaration space. Names declared within the local block are not visible outside of it. The names are visible, however, within any blocks nested with the containing block—for example,

```
public void example()
{   // top-level local declaration space
    int ival = 1024;

    {
        // ival is still in scope here

        double ival = 3.14; // error: reuse of name
        string str = "hello";
    }

    {
        // ival still in scope, str is not!
        double str = 3.14159; // OK
    }

    // what would happen if we defined a str object here?
}
```

If we added a local declaration of `str` at the end of the function, what would happen? Because this declaration occurs at the top-level local declaration space, the two previous legal uses of the identifier `str` within the local nested blocks would be invalidated, and a compiler error would be generated.

Why is there such strict enforcement against multiple uses of a name within the local declaration spaces? In part because local declaration spaces are considered under the ownership of the programmer. That is, the enforcement of a strict policy is not considered onerous for the programmer. She can quickly go in and locally modify one or another of the identifiers. And by doing so, the thinking goes, she is improving the clarity of her program.

1.9 Value and Reference Types

Types in C# are categorized as either value or reference types. The behavior when copying or modifying objects of these types is very different.

An object of a *value* type stores its associated data directly within itself. Any changes to that data does not affect any other object. For example, the predefined arithmetic types, such as `int` and `double`, are value types. When we write

```
double pi = 3.14159;
```

the value `3.14159` is directly stored within `pi`.

When we initialize or assign one value type with another, the data contained in the one is copied to the second. The two objects remain independent.

For example, when we write

```
double shortPi = pi;
```

although both `pi` and `shortPi` now hold the same value, the values are distinct instances contained in independent objects. We call this a *deep copy*.

If we change the value stored by `shortPi`,

```
shortPi = 3.14;
```

the value of `pi` remains unchanged. Although this may seem obvious—perhaps to the point of tedium—this is not what happens when we copy and modify reference types!

A *reference* type is separated into two parts:

1. A named *handle* that we manipulate directly.
2. An unnamed object of the handle's type stored on what is referred to as the *managed heap*. This object must be created with the `new` expression (see Section 1.11 for a discussion).

The handle either holds the address of an object on the heap or is set to `null`; that is, it currently refers to no object. When we initialize or assign one reference type to another, only the address stored within the handle is copied.

Both instances of the reference type now refer to the same object on the heap. Modifications made to the object through either instance are visible to both. We call this a *shallow copy*.

All class definitions are treated as reference types. For example, when we write

```
class Point
{
      float x, y;
      // ...
}

Point origin;
```

`Point` is a reference type, and `origin` reflects reference behavior.

A `struct` definition allows us to introduce a user-defined value type. For example, when we write

```
struct Point
{
      float x, y;
      // ...
}

Point origin;
```

`Point` is now a value type, and `origin` reflects value behavior. In terms of performance, a value type is generally more efficient, at least for small, heavily used objects. We look at this topic in more detail in Section 2.19.

The predefined C# array is a reference type. The discussion of the array in the next section should clarify reference type behavior.

1.10 The C# Array

The built-in array in C# is a fixed-size container holding elements of a single type. When we declare an array object, however, the actual size of the array is *not* part of its declaration. In fact, providing an explicit size generates a compile-time error—for example,

```
string []     text; // OK
string [ 10 ] text; // error
```

We declare a multidimensional array by marking each additional dimension with a comma, as follows:

```
string [,]    two_dimensions;
string [,,]   three_dimensions;
string [,,,]  four_dimensions;
```

When we write

```
string [] messages;
```

`messages` represents a handle to an array object of string elements, but it is not itself the array object. By default, `messages` is set to `null`. Before we can store elements within the array, we have to create the array object using the `new` expression. This is where we indicate the size of the array:

```
messages = new string[ 4 ];
```

messages now refers to an array of four string elements, accessed through index 0 for the first element, 1 for the second, and so on:

```
messages[ 0 ] = "Hi. Please enter your name: ";
messages[ 1 ] = "Oops. Invalid name. Please try again: ";
// ...
messages[ 3 ] = "Well, that's enough. Bailing out!";
```

An attempt to access an undefined element, such as the following indexing of an undefined fifth element for our `messages` array:

```
messages[ 4 ] = "Well, OK: one more try";;
```

results in a runtime exception being thrown rather than a compile-time error:

```
Exception occurred: System.IndexOutOfRangeException:
    An exception of type
    System.IndexOutOfRangeException was thrown.
```

1.11 The `new` Expression

We use the `new` expression to allocate either a single object:

```
Hello myProg = new Hello(); // () are necessary
```

or an array of objects:

```
messages = new string[ 4 ];
```

on the program's managed heap.

The name of a type follows the keyword `new`, which is followed by either a pair of parentheses (to indicate a single object) or a pair of brackets (to indicate an array object). We look at the allocation of a single reference object in Section 2.7 in the discussion of class constructors. For the rest of this section, we focus on the allocation of an array.

Unless we specify an initial value for each array element, each element of the array object is initialized to its default value. (The default value for numeric types is `0`. For reference types, the default value is `null`.) To provide initial values for an array, we specify a comma-separated list of constant expressions within curly braces following the array dimension:

```
string[] m_message = new string[4]
{
    "Hi. Please enter your name: ",
    "Oops. Invalid name. Please try again: ",
    "Hmm. Wrong again! Is there a problem? Please retry: ",
    "Well, that's enough. Bailing out!",
};

int [] seq = new int[8]{ 1,1,2,3,5,8,13,21 };
```

The number of initial values must match the dimension length exactly. Too many or too few is flagged as an error:

```
int [] ia1;

ia1 = new int[128] { 1, 1 };  // error: too few
ia1 = new int[3]{ 1,1,2,3 };  // error: too many
```

We can leave the size of the dimension empty when providing an initialization list. The dimension is then calculated based on the number of actual values:

```
ia1 = new int[]{ 1,1,2,3,5,8 }; // OK: 6 elements
```

A shorthand notation for declaring local array objects allows us to leave out the explicit call to the `new` expression—for example,

```
string[] m_message =
{
    "Hi. Please enter your name: ",
    "Oops. Invalid name. Please try again: ",
```

```
    "Hmm. Wrong again! Is there a problem? Please retry: ",
    "Well, that's enough. Bailing out!",
};

int [] seq = { 1,1,2,3,5,8,13,21 };
```

Although `string` is a reference type, we don't allocate strings using the `new` expression. Rather, we initialize them using value type syntax—for example,

```
// a string object initialized to literal Pooh
string winnie = "Pooh";
```

1.12 Garbage Collection

We do not explicitly delete objects allocated through the `new` expression. Rather, these objects are cleaned up by garbage collection in the runtime environment. The garbage collection algorithm recognizes when an object on the managed heap is no longer referenced. That object is marked as available for collection.

When we allocate a reference type on the managed heap, such as the following array object:

```
int [] fib =
     new int[6]{ 1,1,2,3,5,8 };
```

the heap object is recognized as having an active reference. In this example, the array object is referred to by `fib`.

Now let's initialize a second array handle with the object referred to by `fib`:

```
int [] notfib = fib;
```

The result is a shallow copy. Rather than `notfib` addressing a separate array object with its own copy of the six integer elements, `notfib` refers to the array object addressed by `fib`.

If we modify an element of the array through `notfib`, as in

```
notfib [ 0 ] = 0;
```

that change is also visible through `fib`. If this sort of indirect modification (sometimes called *aliasing*) is not acceptable, we must program a deep copy:

```
// allocate a separate array object
notfib = new int [6];
```

```
// copy the elements of fib into notfib
// beginning at element 0 of notfib
fib.CopyTo( notfib, 0 );
```

`notfib` no longer addresses the same array object referred to by `fib`. If we now modify an element of the array through `notfib`, the array referred to by `fib` is unaffected. This is the semantic difference between a shallow copy and a deep copy.

If we now reassign `fib` to also address a new array object—for example, one that contains the first 12 values of the *Fibonacci sequence*:

```
fib = new int[12]{ 1,1,2,3,5,8,13,21,34,55,89,144 };
```

the array object previously referred to by `fib` no longer has an active reference. It may now be marked for deletion—when and if the garbage collector becomes active.

1.13 Dynamic Arrays: The `ArrayList` Collection Class

As the lines of text are read from the file, I prefer to store them rather than process them immediately. A string array would be the container of choice to do this, but the C# array is a fixed-size container. The required size varies with each text file that is opened, so the C# array is too inflexible.

The `System.Collections` namespace provides an `ArrayList` container class that grows dynamically as we either insert or delete elements. For example, here is our earlier read loop revised to add elements to the container:

```
using System.Collections;
private void readFile()
{
    ArrayList m_text = new ArrayList();
    string    text_line;

    while (( text_line = m_reader.ReadLine() ) != null )
    {
           if ( text_line.Length == 0 )
                 continue;

           // insert the line at the back of the container
           m_text.Add( text_line );
    }
```

```
        // let's see how many we actually added ...
        Console.WriteLine( "We inserted {0} lines", text.Count );
    }
```

The simplest and most efficient way to insert a single element is to use the `Add()` function. It inserts the new element at the back of the list:

```
text.Add( text_line );
```

`Count` returns the number of elements held in the `ArrayList` object:

```
Console.WriteLine( "We inserted {0} lines", text.Count );
```

Just as we do for all reference types, we create an `ArrayList` object on the managed heap using the `new` expression:

```
ArrayList text = new ArrayList();
```

The elements of an `ArrayList` are stored in a chunk of contiguous memory. When that memory becomes full, a larger chunk of contiguous memory has to be allocated (usually twice the size) and the existing elements are copied into this new chunk. We call this chunk the *capacity* of the `ArrayList` object.

The capacity of an `ArrayList` represents the total number of elements that can be added before a new memory chunk needs to be allocated. The count of an `ArrayList` represents the number of elements currently stored within the `ArrayList` object. By default, an empty `ArrayList` object begins life with a capacity of `16` elements.

To override the default capacity, we pass in an alternative capacity when we create the `ArrayList` object—for example,

```
ArrayList text = new ArrayList( newCapacity );
```

where `newCapacity` represents a reasoned integer value. `Capacity` returns the current capacity of the `ArrayList` object:

```
Console.WriteLine( "Count {0} Capacity {1}",
                   text.Count, text.Capacity );
```

Once we've completed our element insertion, we can trim the capacity of the `ArrayList` to the actual element count using the `TrimToSize()` method:

```
text.TrimToSize();
```

Trimming an `ArrayList` object does not restrict our ability to insert additional elements. If we do, however, we once again increase the capacity.

1.14 The Unified Type System

When we define an object, we must specify its type. The type determines the kind of values the object can hold and the permissible range of those values. For example, `byte` is an unsigned integral type with a size of `8` bits. The definition

```
byte b;
```

declares that `b` can hold integral values, but that those values must be within the range of `0` to `255`. If we attempt to assign `b` a floating-point value:

```
b = 3.14159;  // compile-time error
```

a string value:

```
b = "no way"; // compile-time error
```

or an integer value outside its range:

```
b = 1024;     // compile-time error
```

each of those assignments is flagged as a type error by the compiler. This is true of the C# array type as well. So why is an `ArrayList` container able to hold objects of any type?

The reason is the *unified type system*. C# predefines a reference type named `object`. Every reference and value type—both those predefined by the language and those introduced by programmers like us—is a kind of `object`. This means that any type we work with can be assigned to an instance of type `object`. For example, given

```
object o;
```

each of the following assignments is legal:

```
o = 10;
o = "hello, object";
o = 3.14159;
```

```
o = new int[ 24 ];
o = new WordCount();
o = false;
```

We can assign any type to an `ArrayList` container because its elements are declared to be of type `object`.

`object` provides a fistful of public member functions. The most frequently used method is `ToString()`, which returns a string representation of the actual type—for example,

```
Console.WriteLine( o.ToString() );
```

1.14.1 Shadow Boxing

Although it may not be immediately apparent, there is something very strange about assigning an `object` type with an object of type `int`. The reason is that `object` is a reference type, while `int` is a value type.

In case you don't remember, a reference type consists of two parts: the named handle that we manipulate in our program, and an unnamed object allocated on the managed heap by the `new` expression. When we initialize or assign one reference type with another, the two handles now refer to the same unnamed object on the heap. This is the shallow copy that was introduced earlier.

A value type is not represented as a handle/object pair. Rather the declared object directly contains its data. A value type is not allocated on the managed heap. It is neither reference-counted nor garbage-collected. Rather its lifetime is equivalent to the extent of its containing environment. A local object's lifetime is the length of time that the function in which it is defined is executing. A class member's lifetime is equal to the lifetime of the class object to which it belongs.

The strangeness of assigning a value type to an `object` instance should seem a bit clearer now. The `object` instance is a reference type. It represents a handle/object pair. A value type just holds its value and is not stored on the heap. How can the handle of the object instance refer to a value type?

Through an *implicit* conversion process called *boxing*, the compiler allocates a heap address to assign to the `object` instance. When we assign a literal value or an object of a value type to an `object` instance, the following steps take place: (1) an `object` box is allocated on the heap to hold the value, (2) the value

is copied into the box, and (3) the object instance is assigned the heap address of the box.

1.14.2 Unboxing Leaves Us Downcast

There is not much we can do with an `object` except invoke one of its public member functions. We cannot access any of the methods or properties of the original type. For example, when we assign a `string` object to an `object`:

```
string s = "cat";
object o = s;

// error: string property Length is not available
//        through the object instance ...
if ( o.Length != 3 )
```

all knowledge of its original type is unavailable to the compiler. If we wish to make use of the `Length` property, we must first return the `object` back to a `string`. However, an `object` is not automatically converted to another type:

```
// error: no implicit conversion of an object type
//        to any other type ...

string str = o;
```

A conversion is carried out automatically only if it can be guaranteed to be safe. For the compiler to determine that, it must know both the source and the target types. With an `object` instance, all type information is absent—at least for the compiler. (The type and environment information, however, is available both to the runtime environment and to us, the programmers, during program execution. We look at accessing that information in Chapter 8.)

For any conversion for which the compiler cannot guarantee safety, the user is required to do an explicit type cast—for example,

```
string str = ( string ) o;
```

The explicit cast directs the compiler to perform the type conversion even though a compile-time analysis suggests that it is potentially unsafe. What if the programmer is wrong? Does this mean we have a hard bug to dig out?

Actually, no.

The full type information is available to the runtime environment, and if it turns out that `o` really does not represent a `string` object, the type mismatch is recognized and a runtime exception is thrown. So if the programmer is incorrect with an explicit cast, we have a bug, but because of the automatic runtime check, not one that is difficult to track down.

Two operators can help us determine the correctness of our cast: `is` and `as`. We use the `is` operator to ask if a reference type is actually a particular type—for example,

```
string str;

if ( o is string )
     str = ( string ) o;
```

The `is` operator is evaluated at runtime and returns `true` if the actual object is of the particular type. This does not relieve us of the need for the explicit cast, however. The compiler does not evaluate our program's logic.

Alternatively, we can use the `as` operator to perform the cast at runtime if the actual object is of the particular type that interests us—for example,

```
string str = o as string;
```

If `o` is not of the appropriate type, the conversion is not applied and `str` is set to `null`. To discover whether the downcast has been carried out, we test the target of the conversion:

```
if ( str != null )
     // OK: o does reference a string ...
```

In converting an `object` instance to a particular reference type, the only work required is setting the handle to the `object`'s heap address. Converting an `object` instance to a particular value type requires a bit more work because an object of a value type directly contains its data.

This additional work in converting a reference type back to a value type is called *unboxing*. The data copied into the previously generated box is copied back into the object of the target value type. The reference count of the associated box on the managed heap is decremented by `1`.

1.15 Jagged Arrays

Now that we've stored each line of text within an `ArrayList` container, we next want to iterate across the elements, separating each line into an array of the separate words. We'll need to store these arrays because they become fodder for the function implementing the word count. But this storage proves something of a problem—or at least a puzzle. Problem or puzzle, *jagged arrays* provide a solution.

If we are only reading the elements of the container, the `foreach` loop is the preferred iteration method. It spares us the explicit cast of the `object` element to an object of its actual type. Any other element assignment requires the cast:

```
for( int ix = 0; ix < text.Count; ++ix ){
     string str = ( string )text[ ix ];
     // ...
}

// read-only access ...
foreach ( string str in text ){ ... }
```

Splitting one line of text into an array of its individual words is simple:

```
     string [] words = str.Split( null );
```

The next part is also simple, but it is sometimes initially confusing. What we want to do is store the collection of these arrays themselves in an array. That is, we want an array of arrays.

The outer array represents the actual text. The array at the first index represents the first line, at the second index the second line, and so on. An ordinary multidimensional array cannot support what we want because it requires that both dimensions be fixed.

In our case, the first dimension is fixed—it's the number of lines in the text file—but the second dimension varies with the number of words contained within each line of text. This is the situation a jagged array addresses. Each of its array elements can be an individual dimension. The syntax is an empty bracket pair for each dimension. For example, our array of arrays is two-dimensional, so its declaration looks like this:

```
string [][] sentences;
```

We initialize the array in two steps. In the first step we allocate the first dimension. This is the number of lines stored in the `ArrayList` object:

```
sentences = new string[ text.Count ][];
```

This statement says that `sentences` is an array of size `text.Count` that holds one-dimensional arrays of `string` elements. That is exactly what we want.

Next we need to individually initialize each of these elements with the actual string array. We'll do this by iterating across the `ArrayList` and assigning the resulting `Split()` of each of its strings:

```
string str;
for( int ix = 0; ix < text.Count; ++ix )
{
    str = ( string )text[ ix ];
    sentences[ ix ] = str.Split( null );
}
```

The individual string arrays are accessed through the first dimension. For example, to print out both the number of elements in the individual string arrays and the elements themselves, we could write the following:

```
// returns length of first dimension ...
int dim1_length = sentences.GetLength( 0 );
Console.WriteLine( "There are {0} arrays stored in sentences",
                        dim1_length );

for( int ix = 0; ix < dim1_length; ++ix )
{
     Console.WriteLine( "There are {0} words in array {1}",
                         sentences[ ix ].Length, ix+1 );

     foreach ( string s in sentences[ ix ])
                Console.Write( "{0} ", s );

     Console.WriteLine();
}
```

All the C# array types have access to the public members of the `Array` class defined in the `System` namespace. `GetLength()`, illustrated here, is one such member. The majority of the member functions, however, such as `Sort()`, `Reverse()`, and `BinarySearch()`, support arrays of only one dimension.

1.16 The `Hashtable` Container

The `System.Collections` namespace provides a `Hashtable` container. A `Hashtable` represents a key/value pair in which the key is used for fast lookup. In other languages, we might call our table `map` or `Dictionary`. We'll use the `Hashtable` object to hold the occurrence count of the individual words.

We'll use the word as the key. The occurrence count is the value. If the word is not yet entered in the table, we add the pair, setting the occurrence count to `1`. Otherwise, we use the key to retrieve and increment the occurrence count. Here is what this looks like:

```
Hashtable words = new Hashtable();
int dim1_length = sentences.GetLength( 0 );

for( int ix = 0; ix < dim1_length; ++ix )
{
     foreach ( string st in sentences[ ix ] )
     {
         // normalize each word to lowercase
         string key = st.ToLower();

         // is the word currently in Hashtable?
         // if not, then we add it ...

            if ( ! words.Contains( key ))
                   words.Add( key, 1 );

         // otherwise, we increment the count
            else
                 words[ key ] = (int) words[ key ] + 1;
     }
}
```

To discover if a key is present in a `Hashtable`, we invoke its predicate `Contains()` method, which returns `true` if the entry is found. We add the key/value pair to a `Hashtable` either by an explicit assignment:

```
words[ key ] = 1;
```

or through the `Add() method`:

```
words.Add( key, 1 );
```

Typically, the key type is a `string` or one of the built-in numeric types. The value type, which holds the actual data to be retrieved, is generally more application specific—often a class implemented to support the problem domain. The `Hashtable` can support different key and value types because it declares both its key and value members to be of type `object`.

However, because `Hashtable` stores its value as an `object` type, we must explicitly cast it back to its original type. For a value type, recall, this requires unboxing. Any modification to the unboxed value is not automatically reflected in the stored instance, so we must reset it:

```
words[ key ] = (int) words[ key ] + 1;
```

Keeping track of trivial words, such as *a*, *an*, *if*, *but*, and so on, may or may not be useful. One strategy for eliminating these words from our occurrence count is to create a second `Hashtable` of common words—for example,

```
Hashtable common_words = new Hashtable();

common_words.Add( "the", 0 );
common_words.Add( "but", 0 );
// ...
common_words.Add( "and", 0  );
```

and to check each word against this table before entering it into our main table:

```
foreach ( string st in sentences[ ix ])
{
    string key = st.ToLower();
    if ( common_words.Contains( key ))
         continue;
}
```

All that's left for the completion of our program is to print out the occurrence count of the words to the output file in dictionary order. How do we do that?

Our first thought is to iterate across the `Hashtable` using a `foreach` loop. To do that, we access each key/value pair in turn through a `DictionaryEntry` object, which provides a `Key` and a `Value` pair of properties:

```
foreach ( DictionaryEntry de in word )
     fwriter.WriteLine( "{0} : {1}", de.Key, de.Value );
```

As with many programming solutions, this both works and doesn't work. The good news is that it prints out the word and occurrence count of each element. The bad news is that it doesn't do so in dictionary order. For example, here are the first few entries:

```
lime-tinted : 1
waist : 1
bold : 1
```

The problem is that the key is inserted within the `Hashtable` based on the key's hash value, and we cannot directly override that. One solution is to access the key values, place them in a sortable container, and then sort and iterate over that container. For each now sorted key, we retrieve the associated value:

```
ArrayList aKeys = new ArrayList( words.Keys );
aKeys.Sort();

foreach ( string key in aKeys )
    fwriter.WriteLine( "{0} :: {1}", key, words[ key ] );
```

This solves the final sticking point in our solution:

```
apron :: 1
around :: 1
blossoms :: 3
```

The `IDictionary` interface provides an abstract model for the storage and retrieval of key/value pairs. (We look at interfaces in detail in Chapter 4.) The `Hashtable` implements this interface, in which both the key and the value are defined to be of type object. Several strongly typed, specialized classes defined under `System.Collections.Specialized` are also available. These include

- `StringDictionary`: a hash table with the key strongly typed to be a string rather than an object. The key is treated as case insensitive.
- `ListDictionary`: an `IDictionary` implementation using a singly-linked list. If the number of elements is ten or less, it is smaller and faster than a `Hashtable`.
- `NameValueCollection`: a sorted collection of string keys and string values. These can be accessed through either a key hash or an index. `NameValueCollection` stores multiple string values under a single key.

1.17 Exception Handling

That pretty much wraps up the `WordCount` program—at least in terms of its functionality. The primary remaining issue is that of error detection. For example, what if the user has specified a file that doesn't exist? Or what if the file is in a format we don't currently support? Checking that is simple. To find out if a file exists, we can invoke the `Exists()` function of the `File` class, passing to it the string supplied to us by the user:

```
using System.IO;
if ( ! File. Exists( file_name ))
      // oops ...
```

The harder part is choosing how to handle and/or report the problem. In the .NET environment, the convention is to report all program anomalies through *exception handling*. And that is what we will do.

Exception handling consists of two primary components: (1) the recognition and raising of an exception through a `throw` expression and (2) the handling of the exception within a `catch` clause. Here is a `throw` expression:

```
public StreamReader openFile( string file_name )
{
   if ( file_name == null )
        throw new ArgumentNullException();

   // reach here only if no ArgumentNullException thrown
   if ( ! File.Exists( file_name ))
   {
          string msg = "Invalid file name: " + file_name;
          throw new ArgumentException( msg );
   }

   // reach here if file_name not null and file exists
   if ( ! file_name.EndsWith( ".txt" ))
   {
          string msg = "Sorry. ";
          string ext = Path.GetExtension( file_name );

          if ( ext != String.Empty )
                msg += "We currenly do not support " +
                             ext + " files."
```

```
            msg = "\nCurrenly we only support .txt files.";
            throw new Exception( msg );
        }

        // OK: here only if no exceptions thrown
        return File.OpenText( file_name );
    }
```

The object of a `throw` expression is always an instance of the `Exception` class hierarchy. (We look at inheritance and class hierarchies in our discussion of object-oriented programming in Chapter 3.) The `Exception` class is defined in the `System` namespace. It is initialized with a string message identifying the nature of the exception. The `ArgumentException` class represents a subtype of the `Exception` class. It more precisely identifies the category of exception. The `ArgumentNullException` class is in turn a subtype of `ArgumentException`. It is the most specific of these three exception objects.

Once an exception has been thrown, normal program execution is suspended. The exception-handling facility searches the method *call chain* in reverse order for a catch clause capable of handling the exception.

We handle an exception by matching the type of the exception object using one or a series of `catch` clauses. A `catch` clause consists of three parts: the keyword `catch`, the declaration of the exception type within parentheses, and a set of statements within curly braces that actually handle the exception.

`catch` clauses are associated with `try` blocks. A `try` block begins with the `try` keyword, followed by a sequence of program statements enclosed within curly braces. The `catch` clauses are positioned at the end of the `try` block. For example, consider the following code sequence:

```
StreamReader freader = openFile( fname );
string textline;

while (( textline = freader.ReadLine() ) != null )
```

We know that `openFile()` throws three possible exceptions. `ReadLine()` throws one exception, that of the `IOException` class. As written, this code does not handle any of those four exceptions. To correct that, we place the code inside a `try` block and associate the relevant set of catch clauses:

```
try
{
    StreamReader freader = openFile( fname );
    string textline;

    while (( textline = freader.ReadLine() ) != null )
    {
           // do the work here ...
    }
    catch ( IOException ioe )
    { ... }

    catch ( ArgumentNullException ane )
    { ... }

    catch ( ArgumentException ae )
    { ... }

    catch ( Exception e )
    { ... }
```

What happens when an exception is thrown? The exception-handling mechanism looks at the site of the `throw` expression and asks, "Has this occurred within a `try` block?" If it has, the type of the exception object is compared against the exception type declaration of each associated `catch` clause in turn. If the types match, the body of the catch clause is executed.

This represents a complete handling of the exception, and normal program execution resumes. If the `catch` clause does not specify a `return` statement, execution begins again at the first statement following the set of `catch` clauses. Execution does *not* resume at the point where the exception was thrown.

What if the type of the exception object does not match one of the `catch` clauses or if the code does not occur within a `try` block? The currently executing function is terminated, and the exception-handling mechanism resumes its search within the function that invoked the function just terminated.

If one of the three `if` statements of `openFile()` throws an exception, the assignment of `freader` and the remainder of the `try` block are not executed. Rather the exception-handling mechanism assumes program control, examining each associated `catch` clause in turn, trying to match the exception type.

What if the chain of function calls is unwound to the `Main()` program entry point, and still no appropriate `catch` clause is found? The program itself is then terminated. The unhandled exception is propagated to the runtime debugger. The user can either debug the program or let it prematurely terminate.

In addition to a set of `catch` clauses, we can place a `finally` block after the last `catch` clause. The code associated with the `finally` block is always executed before the function exits:

- If an exception is handled, first the `catch` clause and then the `finally` clause is executed before normal program execution resumes.
- If an exception occurs but there is no matching `catch` clause, the `finally` clause is executed; the remainder of the function is discarded.
- If no exception occurs, the function terminates normally. The `finally` clause is executed following the last non-`return` statement of the function.

The primary use of the `finally` block is to reduce code duplication due to differing exception/no-exception exit points that need to execute the same code.

1.18 A Basic Language Handbook for C#

The three things left undone for a complete implementation of our `WordCount` application are (1) to illustrate how to implement the timing diagnostics, (2) to show how to conditionally output tracing statements, and (3) to package our code into the `WordCount` class. This last item is so important that it deserves its own chapter—Chapter 2 to be exact.

We look at how to generate the tracing output in Section 5.5.2 in the discussion of the `TraceListener` class hierarchy. The timing support is presented in Section 8.6.3 in the discussion of interoperability with the Win32 API. The remainder of this section provides a brief handbook of the C# basic language in the form of a set of tables with brief commentary.

1.18.1 Keywords

Keywords are identifiers reserved by the language. They represent concepts or facilities of the basic C# language. `abstract`, `virtual`, and `override`, for example, specify different categories of dynamic functions that support object-

oriented programming. `delegate`, `class`, `interface`, `enum`, `event`, and `struct` represent a variety of complex types that we can define. The full set of keywords is listed in Table 1.1.

Table 1.1 The C# Keywords

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
while				

A name cannot begin with a number. For example, `1_name` is illegal but `name_1` is OK. A name must also not match a language keyword, with the one proverbial exception: We can reuse a C# keyword name by prefixing it with `@`—for example,

```
class @class
{
   static void @static(bool @bool)
   {
         if (@bool)
               Console.WriteLine( "true" );
          else Console.WriteLine( "false" );
   }
}
```

```
class Class1
{
    static void M { @class.@static( true ); }
}
```

This prefix option may prove useful in interfacing with other languages, in particular when a C# keyword is used as an identifier in the other programming language. (The `@` character is not part of the identifier. Rather it provides a context for interpreting the keyword as an identifier.)

1.18.2 Built-in Numeric Types

Integral literals, such as `42` and `1024`, are of type `int`. To indicate an unsigned literal value, the lower- or uppercase letter `U` is added as a suffix—for example, `42u`, `1024U`. To indicate a literal value of type `long`, we add a lower- or uppercase `L` as a suffix—for example, `42L`, `1024l`. (For readability, the upper case `L` is the preferred usage.) To specify a literal value that is both unsigned and `long`, we combine the two suffixes—for example, `42UL`, `1024LU`.

The sizes (whether the value is signed or unsigned) determine the range of values a type can hold. A signed byte, for example, holds the range `-128` through `127`, an unsigned byte the range `0` through `255`, and so on.

However, using types smaller than `int` can be nonintuitive in some circumstances, and I find myself avoiding them except occasionally as class members. For example, the code sequence

```
sbyte s1 = 0;
s1 = s1 + 1; // error!
```

is flagged as an error with the following message:

```
Cannot implicitly convert type 'int' to 'byte'
```

The rule is that the built-in numeric types are implicitly converted to a type as large as or larger. So `int` is implicitly promoted to `double`. But any conversion from a larger to a smaller type requires an explicit user conversion. For example, the compiler does not implicitly allow the conversion of `double` to `int`. That conversion must be explicit:

```
s1 = (sbyte) ( s1 + 1 ); // OK
```

Why, though, you might ask, is a conversion necessary when `s1` is simply being incremented by `1`? In C#, integral operations are carried out minimally through signed 32-bit precision values. This means that an arithmetic use of `s1` results in a promotion of its value to `int`. When we mix operands of different types, the two are promoted to the smallest common type. For example, the type of the result of `s1+1` is `int`.

When we assign a value to an object, as with our assignment of `s1` here, the type of the right-hand value must match the type of the object. If the two types do not match, the right-hand value must be converted to the object's type or the assignment is flagged as an error.

By default, a floating-point literal constant, such as `3.14`, is treated as type `double`. Adding a lower- or uppercase `F` as a suffix to the value turns it into a single-precision `float` value. Character literals, such as `'a'`, are placed within single quotes. Decimal literals are given a suffix of a lower- or uppercase M. (The decimal type is probably unfamiliar to most readers; Section 4.3.1 looks at it in more detail.) Table 1.2 lists the built-in numeric types.

Table 1.2 The C# Numeric Types

Keyword	Type	Usage
sbyte	Signed 8-bit int	`sbyte sb = 42;`
short	Signed 16-bit int	`short sv = 42;`
int	Signed 32-bit int	`int iv = 42;`
long	Signed 64-bit int	`long lv = 42, lv2 = 42L, lv3 = 42l;`
byte	Unsigned 8-bit int	`byte bv = 42, bv2 = 42U, bv3 = 42u;`
ushort	Unsigned 16-bit int	`ushort us = 42;`
uint	Unsigned 32-bit int	`uint ui = 42;`
long	Unsigned 64-bit int	`ulong ul = 42, ul2 = 4ul, ul3 = 4UL;`
float	Single-precision	`float f1 = 3.14f, f3 = 3.14F;`
double	Double-precision	`double d = 3.14;`
bool	Boolean	`bool b1 = true, b2 = false;`
char	Unicode char	`char c1 = 'e', c2 = '\0';`
decimal	Decimal	`decimal d1 = 3.14M, d2 = 7m;`

The keywords for the C# predefined types are aliases for types defined within the `System` namespace. For example, `int` is represented under .NET by the `System.Int32` type, and `float` by the `System.Single` type.

This underlying representation of the prebuilt types within C# is the same set of programmed types as for all other .NET languages. This means that although the simple types can be manipulated simply as values, we can also program them as class types with a well-defined public set of methods. It also makes combining our code with other .NET languages considerably more direct. One benefit is that we do not have to translate or modify types in order to have them recognized in the other language. A second benefit is that we can directly reference or extend types built in a different .NET language. The underlying `System` types are listed in Table 1.3.

Table 1.3 The Underlying `System` Types

C# Type	System Type	C# Type	System Type
sbyte	System.SByte	byte	System.Byte
short	System.Int16	ushort	System.UInt16
int	System.Int32	uint	System.UInt32
long	System.Int64	ulong	System.UInt64
float	System.Single	double	System.Double
char	System.Char	bool	System.Boolean
decimal	System.Decimal		
object	System.Object	string	System.String

1.18.3 Arithmetic, Relational, and Conditional Operators

C# predefines a collection of arithmetic, relational, and conditional operators that can be applied to the built-in numeric types. The arithmetic operators are listed in Table 1.4, together with examples of their use; the relational operators are listed in Table 1.5, and the conditional operators in Table 1.6.

The binary numeric operators accept only operands of the same type. If an expression is made up of mixed operands, the types are implicitly promoted to the smallest common type. For example, the addition of a `double` and an `int` results in the promotion of the `int` to `double`. The `double` addition operator is

then executed. The addition of an `int` and an unsigned `int` results in both operands being promoted to `long` and execution of the `long` addition operator.

Table 1.4 The C# Arithmetic Operators

Operator	Description	Usage
*	Multiplication	`expr1 * expr2;`
/	Division	`expr1 / expr2;`
%	Remainder	`expr1 % expr2;`
+	Addition	`expr1 + expr2;`
–	Subtraction	`expr1 - expr2;`
++	Increment by 1	`++expr1; expr2++;`
––	Decrement by 1	`--expr1; expr2--;`

The division of two integer values yields a whole number. Any remainder is truncated; there is no rounding. The remainder is accessed by the remainder operator (`%`):

```
5 / 3 evaluates to 1, while 5 % 3 evaluates to 2
5 / 4 evaluates to 1, while 5 % 4 evaluates to 1
5 / 5 evaluates to 1, while 5 % 5 evaluates to 0
```

Integral arithmetic can occur in either a checked or unchecked context. In a *checked* context, if the result of an operation is outside the range of the target type, an `OverflowException` is thrown. In an *unchecked* context, no error is reported.

Floating-point operations never throw exceptions. A division by zero, for example, results in either negative or positive infinity. Other invalid floating-point operations result in NAN (not a number).

The relational operators evaluate to the Boolean values `false` or `true`. We cannot mix operands of type `bool` and the arithmetic types, so relational operators do not support concatenation. For example, given three variables—`a`, `b`, and `c`—that are of type `int`, the compound inequality expression

```
// illegal
a != b != c;
```

is illegal because the `int` value of `c` is compared for inequality with the Boolean result of `a != b`.

Table 1.5 The C# Relational Operators

Operator	Description	Usage
<	Less than	`expr1 < expr2;`
>	Greater than	`expr1 > expr2;`
<=	Less than or equal to	`expr1 <= expr2;`
>=	Greater than or equal to	`expr1 >= expr2;`
==	Equality	`expr1 == expr2;`
!=	Inequality	`expr1 != expr2;`

Table 1.6 The C# Conditional Operators

Op	Description	Usage
!	Logical NOT	`! expr1`
||	Logical OR (short circuit)	`expr1 || expr2;`
&&	Logical AND (short circuit)	`expr1 && expr2;`
|	Logical OR (`bool`)—evaluate both sides	`bool1 | bool2;`
&	Logical AND (`bool`)—evaluate both sides	`bool1 & bool2;`
?:	Conditional	`cond_expr ? expr1 : expr2;`

The conditional operator takes the following general form:

```
expr
   ? execute_if_expr_is_true
   : execute_if_expr_is_false;
```

If `expr` evaluates to `true`, the expression following the question mark is evaluated. If `expr` evaluates to `false`, the expression following the colon is evaluated. Both branches must evaluate to the same type. Here is how we might use the conditional operator to print either a space or a comma followed by a space, depending on whether `last_elem` is `true`:

```
Console.Write( last_elem ? " " : ", " )
```

Because the result of an assignment operator (=) is the value assigned, we can concatenate multiple assignments. For example, the following assigns `1024` to both the `val1` and the `val2` objects:

```
// sets both to 1024
val1 = val2 = 1024;
```

Compound assignment operators provide a shorthand notation for applying arithmetic operations when the object to be assigned is also being operated upon. For example, rather than writing

```
cnt = cnt + 2;
```

we typically write

```
// add 2 to the current value of cnt
cnt += 2;
```

A compound assignment operator is associated with each arithmetic operator:

```
+=, -=, *=, /=, and %=.
```

When an object is being added to or subtracted from by `1`, the C# programmer uses the increment and decrement operators:

```
cnt++; // add 1 to the current value of cnt
cnt--; // subtract 1 from the current value of cnt
```

Both operators have prefix and postfix versions. The *prefix* version returns the value after the operation. The *postfix* version returns the value before the operation. The value of the object is the same with either the prefix or the postfix version. The return value, however, is different.

1.18.4 Operator Precedence

There is one "gotcha" to the use of the built-in operators: When multiple operators are combined in a single expression, the order in which the expressions are evaluated is determined by a predefined precedence level for each operator. For example, the result of `5+2*10` is always `25` and never `70` because the multiplication operator has a higher precedence level than that of addition; as a result, in this expression `2` is always multiplied by `10` before the addition of `5`.

We can override the built-in precedence level by placing parentheses around the operators we wish to be evaluated first. For example, `(5+2)*10` evaluates to `70`.

Here is the precedence order for the more common operators; each operator has a higher precedence than the operators under it. Operators on the same line have equal precedence. In the case of equal precedence, the order of evaluation is left to right:

```
Logical NOT (!)
Arithmetic *, /, and %
Arithmetic + and -
Relational <, >, <=, and >=
Relational == and !=
Logical AND (&& and &)
Logical OR (|| and |)
Assignment (=)
```

For example, consider the following statement:

```
if (textline = Console.ReadLine() != null) ... // error!
```

Our intention is to test whether `textline` is assigned an actual string or `null`. Unfortunately, the higher precedence of the inequality operator over that of the assignment operator causes a quite different evaluation. The subexpression

```
Console.ReadLine() != null
```

is evaluated first and results in either a `true` or `false` value. An attempt is then made to assign that Boolean value to `textline`. This is an error because there is no implicit conversion from `bool` to `string`.

To evaluate this expression correctly, we must make the evaluation order explicit by using parentheses:

```
if ((textline = Console.ReadLine()) != null) ... // OK!
```

1.18.5 Statements

C# supports four loop statements: `while`, `for`, `foreach`, and `do-while`. In addition, C# supports the conditional `if` and `switch` statements. These are all detailed in Tables 1.7, 1.8, and 1.9.

Table 1.7 The C# Loop Statements

Statement	Usage

while	`while ( ix < size ){`
	`  iarray[ ix ] = ix;`
	`  ix++;`
	`}`

for	`for (int ix = 0; ix<size; ++ix)` `  iarray[ ix ] = ix;`
foreach	`foreach ( int val in iarray )` `  Console.WriteLine( val );`

do-while	`int ix = 0;`
	`do`
	`{`
	`  iarray[ ix ] = ix;`
	`  ++ix;`
	`}`
	`while ( ix < size );`

Table 1.8 The C# Conditional `if` Statements

Statement	Usage

if	`if (usr_rsp=='N' || usr_rsp=='n')`
	`  go_for_it = false;`
	
	`if ( usr_guess == next_elem )`
	`{ // begins statement block`
	`  num_right++;`
	`  got_it = true;`
	`} // ends statement block`

if-else	`if ( num_tries == 1 )`
	`  Console.WriteLine( " ... " );`
	`else`
	`if ( num_tries == 2 )`
	`  Console.WriteLine( " ... " );`
	`else`
	`if ( num_tries == 3 )`
	`  Console.WriteLine( " ... " );`
	`else Console.WriteLine( " ... " );`

Table 1.9 The C# `switch` Statements

Statement	Usage

switch	`// equivalent to if-else-if clauses above`
	`switch ( num_tries )`
	`{`
	`  case 1:`
	`  Console.WriteLine( " ... " );`
	`  break;`
	
	`  case 2:`
	`  Console.WriteLine( " ... " );`
	`  break;`
	
	`  case 3:`
	`  Console.WriteLine( " ... " );`
	`  break;`
	
	`  default:`
	`  Console.WriteLine( " ... " );`
	`  break;`
	`}`
	
	`// can use string as well`
	`switch ( user_response )`
	`{`
	`  case "yes":`
	`  // do something`
	`  goto case "maybe";`
	
	`  case "no":`
	`  // do something`
	`  goto case "maybe";`
	
	`  case "maybe":`
	`  // do something`
	`  break;`
	
	`  default:`
	`  // do something;`
	`  break;`
	`}`