CHAPTER 1

Introduction to C# and .NET

TIPS IN THIS CHAPTER

- When to Use C# and C++ 8
- Distinguishing .NET from Other Environments 11
- Using the Common Language Runtime 13
- Viewing Intermediate Language Code 16
- Taking Advantage of Just-In-Time Debugging 18
- Exploiting the .NET Interoperability with COM 20
- Using .NET Versioning to Handle Software Updates 22
- Querying Class Capabilities Through .NET Reflection 24

The primary language for developing applications for small computer systems over the past 20 or so years has been the C programming language, developed by Dennis Ritchie for the UNIX operating system in the 1970s. C offered programmers much of the control over the physical computer that they had in Assembly, but it also offered the ease of programming afforded by a high-level language such as FORTRAN.

In the last decade, much of the development has been done in the C++ language developed by Bjarne Stroustrup. C++ turned the C language into a modern, object-oriented language while retaining the basic, low-level qualities of the C language. Today, when most people mention programming in C, they actually mean C++.

C and C++ have many similarities. One of these is that both languages have a long learning curve. Despite intensive study, it can take years to learn the features and nuances of C++ and to use the power of the language to write large applications.

Over the last few years, Microsoft introduced and developed Visual Basic, which rapidly became a favorite among programmers who wanted to develop programs quickly for specific purposes. Visual Basic is based loosely on the syntax and statements of BASIC, and offers a compiled language with a shorter learning curve than C++. Visual Basic for Applications is commonly used with many applications to provide a means for the user to write *macros*.

From its beginning, the developers of C# (pronounced "C Sharp") wanted to create a language that combines "the high productivity of Visual Basic and the raw power of C++." This sounds slightly oxymoronic. With such a claim come tradeoffs, and usually "high productivity" comes at the cost of "raw power." This is also true of C#. You will experience the "high productivity" but at the cost of power. To get at the "raw power of C++," you will have to revert to features in C# that let you write C++ code from within C# and call library modules written in C++. If you do much of this, though, your application probably is more suited to C++ than C#.

The developers also used the word "modern" to describe the language, but in many ways C# is a throwback to earlier days of programming for small computer systems. It is an object-oriented and "type-safe" language loosely derived from C++. While much of the syntax and keywords will be familiar to C++ programmers, C# is *not* a refinement of the C++ language.

C# is one of the languages supported by Microsoft's Visual Studio 7.0 development environment, which also supports Visual Basic, Visual C++, VBScript, and JScript. C#—along with the current version of Visual Basic and "managed" C++—is based on the Next Generation Windows Services (NGWS) platform, now called the .NET software development environment.

The "services" part of NGWS should give you a clue as to the focus of the language. C# is "component-oriented." The .NET environment itself is component-oriented, and thus the underlying design of C# is to make it easier to write components. Over the last few years we have had access to a number of component-oriented tools such as the Component Object Model (COM) and ActiveX. C# and the .NET framework simplifies component writing. You do not have to use an IDL (interface definition language) file to create a component, and you do not have to create type libraries to use the components in a program. When you create a component with C#, the component contains all of the information—the *metadata*—it needs to describe itself.

Along the way, you will see many of the terms usually used with components to describe C# concepts. You will see *methods* instead of *functions* and *properties* and *fields* instead of *variables*.

The .NET framework defines a Common Language Specification (you may sometimes see this as Common Language Subset) and provides a Common Language Runtime (CLR) module. Programs that support the .NET framework are compiled to intermediate code modules, and the CLR provides the translation to the native language for the computer.

Unlike the pseudocode (or P-code that most of us have learned to hate), the intermediate language (IL) in the .NET environment provides more than an interpreter. When you run a program that has been compiled into IL, the .NET framework recompiles the intermediate code into native code.

Looking at C#

One of the biggest advantages to C# is that programmers who have invested a lot of time and work in learning the C++ language do not have to discard that knowledge to begin developing programs in a new language. There are some new concepts and techniques, and some new function names to learn, but generally the syntax is similar to C++.

In addition, programmers do not need to discard code they have already written. C# contains mechanisms to call library functions in existing code and in system libraries. You still may call the same Windows functions that you used in C++, and in many cases you may choose between the Windows function and the .NET framework. The following short program, *MsgBox.cs*, for example, displays two message boxes, one right after the other. The first message box results from a call the Windows API *MessageBox()* function, and the second results by a call to the CLR's *MessageBox* class *Show()* method. You will have difficulty distinguishing between the two. (In the final analysis, both message boxes are generated by the Windows function.)

```
//
//  MsgBox.cs -- demonstrate using the Windows API message box and the
//               .NET framework message box.
//
//               Compile this program with the following command line:
//                   C:>csc msgbox.cs
//
namespace nsMsgBox
{
    using System.Windows.Forms;
    using System.Runtime.InteropServices;
    class clsMain
    {
//  Import the dll and prototype the MessageBox function.
        [DllImport("User32.dll")]
        public static extern int MessageBox (int hwnd, string Message,
                                             string Caption, int Flags);
        static public void Main ()
        {
```

```
//  Call the Windows API function
            MessageBox (0, "Hello C# World", "Howdy",
                           (int) MessageBoxButtons.OK
                         | (int) MessageBoxIcon.Exclamation);
//  Call the CLR method
            System.Windows.Forms.MessageBox.Show ("Hello, C# World!",
                         "Howdy",
                         MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
        }
    }
}
```

Although C# is a new language, it is difficult to find any new concepts in the language. It is largely patterned after Java, a technology developed by Sun Microsystems, and is based on the Visual Basic programming model. Along the way, it borrows heavily from the syntax and keywords of C and C++, and you will find many of the concepts of the Smalltalk programming language in C#. That is not surprising, however, because C++ has a lot of similarities with Smalltalk.

The Java language attempts to provide developers with a platform-independent language. Through the Java Virtual Machine (JVM) intermediate language interpreter, Java's goal was to provide a "write-once, run-anywhere" language that would free developers from having to write a different program for every computer platform and every operating system. The developer would use the same code, and the JVM would provide the interface between the program and the platform and operating system.

Through C# and the .NET framework, Microsoft is attempting to achieve the same goal. Rather than compile programs into native code for computers running Windows on the Intel processor platform, languages such as C# that use the .NET platform compile into an IL. The CLR then dynamically transforms this IL code into the native code for a computer platform. Theoretically, at least, an operating system need only implement the CLR to give developers the ability to run their programs.

That is amazingly similar to the Java concept and has led many in the programming industry to make statement such as "C# is simply Java by another name." Whether C# is a Sun in Microsoft disguise is not important. What is important is the .NET platform. The technology offered in the .NET program goes beyond the JVM.

While C# appears based on the Visual Basic programming model, it does not yield completely to that model. Instead, C# draws the concepts of the C-family of languages and Visual Basic to meet somewhere in the middle. The development probably is better for Visual Basic than it is for the C languages. Visual Basic cedes several statements and keywords, such as the *Option Base* keyword and the *Variant* data types, to work with .NET.

Figure 1-1 illustrates how the C# developers built the language around key features available in various operating systems, programming languages, components, and distributed environments.

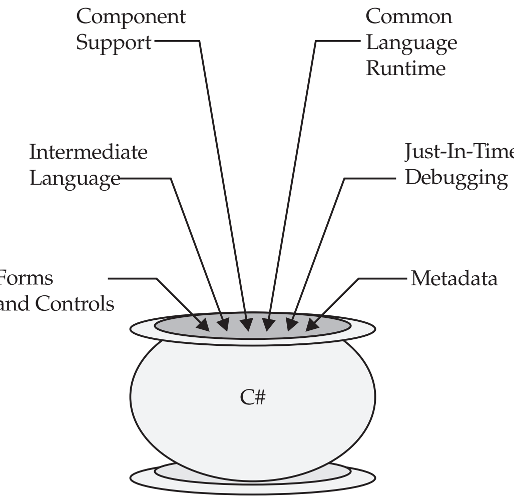

Figure 1.1 C# was developed from a melting pot of programming techniques

Using C# Instead of C++

If you are new to programming, it might be of some comfort to know that C# is much easier to learn than C++ as a first language. The C and C++ languages are powerful languages, but they do let you perform operations that you should not.

C# is a *type-safe* language, and it hides much of the underlying problems that a newcomer to programming might have with C or C++. For example, C# uses pointers, but the programmer does not have to deal with them directly. Instead, the language hides pointers under the name *reference-type* variables.

C# uses much of the same syntax used in C++, and the learning probably will be shorter if you later migrate to C++ after learning C#. Programmers already familiar with C++ will have to surmount their bias toward the language. (C++ programmers have a tendency to look with disdain on Visual Basic programs, and C# seems more like Visual Basic than like C++.) It may be difficult to get used to some of the restrictions C# places on various operators and statements, but these restrictions are intended to reduce bugs and to promote rapid development.

In this book, I will occasionally refer to the way you'd do something in C++ and compare it with the C# method. Generally, changes from the C++ method will produce a slight reduction in the "power" of statements and operations but will be geared toward simplicity and ease of operation. For example, after a long day of programming in C++, you might write something like the following:

```
#include <stdio.h>

void Function (double *);

void main (void)
```

```
{
    int arr [10];
    Function ((double *) arr);
}
void Function (double *arr)
{
    for (int x = 0; x < 10, ++x)
        arr[x] = x * 3.14159;
}
```

Syntactically, this program is 100 percent correct, and the compiler will not so much as issue a warning. However, it also is guaranteed to produce a crash. Writing 10 double values into an array of 10 integers is going to overwrite something on the stack. That is part of the power of programming in C++, but also one of its dangers.

In such an instance, the C# compiler takes the helm. It simply will not let you do something like this. The equivalent code in C# would appear like the following:

```
class clsMain
{
    static public void Main ()
    {
        int [] arr = new int [10];
        Function ((double []) arr);
    }
    static public void Function (double [] arr)
    {
        for (int x = 0; x < 10; ++x)
            arr[x] = x * 3.14159;
    }
}
```

No matter how you cast the variables, you cannot coax the C# compiler into letting you get away with this operation. This is part of the type-safety of C#. The language guarantees that a variable of a certain data type will always contain a value of that data type before you can use the variable.

This does not mean that you will not have bugs in a program that compiles correctly. You still will find yourself dropping into infinite loops or such, but the probability of that happening is greatly reduced. And the fact that C# allow access to most objects only through pointers (the reference-type variable) leads to its own set of bugs.

All of this leads to a shorter debugging cycle, which in turn means shorter development times. In-house developers and those working at software houses should find a higher level of productivity when using C#. Recreational programmers or hobbyists will appreciate seeing results faster.

C# code is also compatible with other languages that support the .NET framework. For example, you will find it much easier to interface with programs and controls written in the .NET version of Visual Basic and other programming languages.

When to Use C# and C++

Back in the 1970s, before most colleges had computer science departments, I was studying programming as part of the Mathematics Department at the local university. A common amusement on the computer (a rather large and lumbering beast compared with the personal computer today) was to toss random points into a square that measured 1 unit per side. We then calculated the distance of a point from the lower left corner of the square (the square'e origin). The distance across a 1×1 square from corner to corner is the square root of 2, so a point could have a distance greater than or less than 1 unit.

Theoretically, if you tossed enough points into the square, the ratio of the number of points with a distance less than 1 unit to the total number of random points you tossed would approximate *pi/4*. Now, we're talking a lot of points to come up with just this reasonable approximation—in fact, millions of points. But then, computers in the 1970s didn't have a lot better to do than entertain math students. The exercise also proved a good test for our random-number generating routines. The closer we could approximate *pi* with the fewest number of points meant that the random-number generator was working better. Most languages today have random-number routines as part of their library code.

USE IT To compare execution times, I wrote a program to throw 10 million such points using both C# and C++. The C# program was a little easier to write and had a few less lines than the C++ program. Then I ran each program 10 times to get a good approximation of the times it took to execute the programs. The programs use several floating-point operations.

These programs, shown next, are both Visual Studio projects, so you do not have to worry about command-line compiling yet. After you copy the projects to your computer, start the Visual Studio, and then select File | Open Solution. Maneuver to the directory where you copied the files and look for the *CalcPi.sln* file in the C# or CPP subdirectory. Double-click this file to open the project.

The listing for both programs follows if you want to compare your times. The computer I used was an 866 MHz dual Pentium III-processor with 512 MB of RAM running Windows 2000 Server. First, the C# program, *CalcPi.cs*:

```
using System;

namespace CalcPi
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            const int throws = 10000000;
            DateTime now = DateTime.Now;
            Random rand = new Random ((int) now.Millisecond);
            int Inside = 0;
            for (int i = 0; i < throws; ++i)
```

```
            {
                double cx = rand.NextDouble();
                double cy = rand.NextDouble();
                double distance = Math.Sqrt ((cx * cx) + (cy * cy));
                if (distance < 1.0)
                    ++Inside;
            }
            double pi = 4 * (double) Inside / (double) throws;
            DateTime End = DateTime.Now;
            TimeSpan Diff = End - now;
            Console.WriteLine ("pi = " + pi);
            Console.WriteLine ("Elapsed time = {0} milliseconds" ,
                               Diff.TotalMilliseconds);
        }
    }
}
```

The C++ program required a few extra lines of code, but most of those were the *#include* statements at the beginning of the file. You don't need those with C#. The C++ listing for *CalcPi.cpp* follows:

```
// CalcPi.cpp : Defines the entry point for the console application.
//

#include <stdio.h>
#include <windows.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char* argv[])
{
    const int throws = 10000000;
    SYSTEMTIME now;
    SYSTEMTIME end;
    GetSystemTime (&now);
    srand ((unsigned) time (NULL));
    int Inside = 0;
    for (int i = 0; i < throws; ++i)
    {
        double cx = (double) rand() / (double) RAND_MAX;
        double cy = (double) rand() / (double) RAND_MAX;
        double distance = sqrt ((cx * cx) + (cy * cy));
        if (distance < 1.0)
            ++Inside;
```

```
    }
    double pi = 4 * (double) Inside / (double) throws;
    GetSystemTime (&end);
// Note: the following interval calculation will give an incorrect
// result if the program is run exactly on the hour.
    int msStart = 1000 * 60 * now.wMinute + 1000 * now.wSecond
                  + now.wMilliseconds;
    int msEnd = 1000 * 60 * end.wMinute + 1000 * end.wSecond
                + end.wMilliseconds;
    int milliseconds = msEnd - msStart;
    printf ("pi = %6f\n", pi);
    printf ("Elapsed time = %d milliseconds\n", milliseconds);
    return 0;
}
```

Both programs were tested in the release version. (The release version is compiled without any debugging information. To compile it, select Build | Configuration Manger and then select Release from the Active Solution Configuration box of the Configuration Manager dialog box.) The relative times were not surprising, and both programs came up with a reasonable approximation of *pi* to four decimal places. The closest the C++ program came was 3.14158, and the closest the C# program came was 3.14157—essentially the same number.

The C# program ranged from 3185 to 3285 milliseconds to toss the 10 million points, with an average time of 3222 milliseconds. The longest time differed from the shortest time by 100 milliseconds, or about 3.1 percent of the average time.

The C++ program, on the other had, ranged in time from 1573 to 1592 milliseconds, with an average time of 1583 milliseconds. The difference between the longest and shortest times was only 19 milliseconds, about 1.2 percent of the average time.

Outside the Visual Studio integrated development environment (IDE), the C# program did considerably better, turning in an average time of about 1998 milliseconds. The C++ program times did not change much outside the IDE, but they did turn in times as short as 1542 milliseconds.

In addition, if you do not *optimize* the C++ code when you compile the program, it will turn in considerably worse times than the C# program. Visual Studio normally applies optimization when it compiles the release version of a program. It is reasonable to assume that the CLR code is optimized and would have an edge over the C++ program in this case. However, optimization is part of the power of C++.

Just so there is no argument, these are code execution times. Each program gets the start time at the beginning of the program and the end time just after it calculates the value of *pi*. The times do not include the time it takes the .NET framework to compile the IL code into machine code for the computer, or the time it takes the operating system to load the C++ program.

Even outside the IDE, the C++ program turned in times that were nearly 30 percent faster. Of course, a case could be made that you could design a program that would show that C# is faster. To be sure, this program was not designed specifically for either language. It is a common mathematical exercise, originally written in FORTRAN, that I simply translated into C# and C++. In addition, a personal computer is not the best device to use when you want to resolve a time interval down to the

millisecond. These programs assume both time methods will exhibit the same inaccuracies on the same computer, and repeated executions should average out any errors.

No programming language is suitable for every type of application, and C# is no exception. If your program is processor intensive, you might want to consider writing it in C++. That is a decision you, as the programmer, will have to make. Remember that a graphics program or other mathematics-intensive program may perform *billions* of floating-point operations during the course of its run.

On the other hand, C# certainly is easier if you are writing a program that will run over the Internet, or if you want to write small components that perform specific tasks. Many large applications will benefit from C# as well. In a word processing application, for example, you are not going to type any faster if the application is written in C# or C++. Most programs will fall somewhere between the two extremes.

Despite the self-appointed “keepers of the languages” who would impose a rigorous scientific dogma upon us, programming is an art. You select the language in much the same way a painter selects the medium for a painting—oil, watercolor, charcoal, for example—or the poet selects the meter for the verse. Your selection criteria might include such things as your familiarity and comfort with a language, its efficiency, and its underlying support for the fundamental purpose of the program.

Distinguishing .NET from Other Environments

C# is only one part of a programming architecture designed around a network distributed platform. The platform may be a single machine, a local intranet, or even other networks located across the Internet. That *platform* is what Microsoft calls the *.NET Framework*, which is designed to support a managed code environment in which programs may safely and securely operate on a single computer or across a distributed network.

The .NET environment is also designed to allow programs to operate across different operating systems. The .NET framework comprises two major components:

- The CLR provides core services such as memory and thread management. It manages code written for the .NET environment and enforces type-safety.
- For developing programs that run under the CLR, the .NET Framework offers a library of reusable, object-oriented classes that simplify programming tasks and speed up development time. The library of classes supports development programs that run as command-line programs and as Windows applications.

The CLR is the latest development in COM, which it is designed to replace. The CLR is designed to alleviate many of the problems that plagued COM while retaining COM’s better features. Some of the advantages of the new CLR over the old COM are better interoperability between languages, easier deployment of components, and improved versioning.

Language interoperability is provided by having C# and Visual Basic—along with a managed-code version of C++ and other languages that support .NET—use a common IL. With COM, it was not uncommon to write a C++ component that was totally useless to Visual Basic programs. In addition, interfacing C++ programs to modules such as ActiveX controls written in Visual Basic often required considerable programming gymnastics.

Code management is at the core of the CLR, and the CLR is at the core of the .NET environment. Any programming language that targets the CLR is subject to code management. Programmers are free to create objects in the heap memory without having to worry about freeing the memory before the program exits. In older code, programs that allocated heap memory and failed to free it left behind memory leaks that persisted even after the program ended. Repeated instances of running the program might make the computer run low on memory.

Managed code modules also have varying degrees of security applied to them, depending on whether they originated on the local computer, a local network, or over the Internet. Modules that do not have a high degree of trust, such as those originating on the Internet, might not be able to perform some operations, such as opening and writing to files.

The security and programming model used by the .NET Framework means that many applications that had to be installed on a computer now may be deployed over the Internet. A program may run components from different vendors and from various Internet sites, and each component would have its own security and permissions on the local computer. That could mean that, in many cases, you would not have to wait endless months for a vendor to provide a service pack or program upgrade to fix or improve software. Components on the Internet sites could be updated more quickly, and deployment would be greatly simplified.

The .NET environment supports development of a number of program types such as *console applications.* Such programs have the advantage of being utility command-line programs that have access to the Windows graphical user interface (GUI). In this book, you will explore many of the features of C# using console applications. In fact, the environment provides a set of reusable GUI objects in the Windows forms classes. For projects involving the Web, a similar set of objects are included in the Windows Web forms classes.

You will see references to a "Portable Executable" (PE) format file throughout the course of using the Visual Studio and the .NET Framework Software Development Kit (SDK). A PE file contains a block of information for the program's metadata. The metadata contains information about the object types in your program, security information, and custom attributes you use in your program.

The primary tool used to develop C# applications for the Windows platform is Visual Studio .NET. This integrated development environment includes compilers and debuggers for C#, Visual Basic, and Visual C++, and it supports JScript.

USE IT Visual Studio and the .NET SDK include a number of tools to help in writing programs and migrating existing code to the .NET environment. Among them are the following tools:

- **DbgCLR.exe** The Microsoft CLR Debugger, a Visual Studio–like debugger with a GUI.
- **Cordbg.exe** A command-line based runtime debugger that uses the CLR debugging application programming interface (API).
- **Ilasm.exe** The Microsoft Intermediate Language Assembler, which generates files in the Portable Executable format from an intermediate language file. You can run the code generated by this utility to test whether the intermediate code performs as expected.
- **Ildasm.exe** A disassembler that converts a Portable Executable format file into a text file that can be used by *Ilasm.exe*.

- **Aximp.exe** An ActiveX control importing tool. This program converts the type definitions in a COM type library for an ActiveX control into a Windows Forms control. This tool is handy for converting existing components to the .NET Framework.
- **Sn.exe**, **Al.exe**, and **GacUtil.exe** These three programs provide utilities to create a security key, add the key to an assembly, and then add the assembly to the assembly cache.

You do not need to run Visual Studio to use these tools. You may choose to write many of your programs without the overhead that comes with an integrated development environment. A number of new editors are showing up on the Internet that support the C# syntax, which you can use without the IDE. You can find a particularly good one, written in C#, at **http://www.icsharpcode.net/opensource/sd**.

An update to Windows Explorer also provides a panel for viewing the global assembly cache, which is a hidden directory in the Windows or WinNT directory.

The .NET framework SDK is a part of the Visual Studio .NET installation, but it is available on a separate CD and may be installed on your computer without having to install the complete integrated development environment.

When you install the .NET SDK, you can install code samples, tutorials, and QuickStarts to help acquaint you with the framework and programming mode.

Using the Common Language Runtime

Programmers in the past have linked their code with various libraries and made use of external components for their applications. In addition to the runtime library, a number of specialized code libraries are available. Dynamic link libraries (DLLs) have provided another source of code at runtime. The code for the Windows Application Programming Interface, for example, resides in several DLLs that a program can call as needed.

For C#, the .NET environment defines more than 1000 classes that a programmer may access directly or use as a base for deriving new classes to extend their functionality. The .NET Framework is a library of components along with the supporting classes and structures.

The CLR is the environment used by the .NET environment. The CLR manages code at runtime and provides services that make program development easier. The code that you write for the CLR is called *managed code*.

Microsoft describes the process of running code under the CLR as running with a "contract of cooperation" with the runtime environment. Managed code undergoes a process of "verification" before the CLR will let it run on the computer. The CLR provides a type-safe environment in which no application will cause a memory fault, an attempt to access memory that it does not own. This is a common problem among C++ programmers who forget to initialize a pointer before attempting to access memory. Such bugs can go unnoticed through the debugging process, and they might only show up when the code is released to users.

Usually problems such as unitialized pointers will never get beyond the C# compiler. C# does use pointers, per se—it hides them from the programmer in the form of reference-type variables. The compiler

simply will not allow your code to use a pointer variable—or any variable for that matter—until you have assigned it a value. There are some exceptions, such as an *out* parameter for a method, but these generally result in type-safe assignments to the variables.

The C# coding process is not perfect, though. It is still possible for the compiler to let a variable past an initialization test, and you wind up with a null value in a reference-type variable. For example, if a font or brush initialization fails at runtime and you attempt to use the variable, you are going to get a program fault.

The CLR watches the code and catches these situations. When this happens, the program, through the CLR, throws an exception. The programmer may include code in the program to watch for exceptions and handle them in the program. If not handled, the exception will cause the program to end. The CLR, however, is designed so that the demise of a single program will not affect other programs running on the computer.

Programmers will find the CLR much more flexible when it comes to language interoperability. It used to take considerable programming gymnastics to make C++ and Visual Basic programs and modules work together, for example. Sometimes it was virtually impossible.

Another feature of the CLR is "garbage collection." Most C++ programmers are familiar with the concept of a *memory leak*: You create an object in the heap memory but forget to free up the memory before your program ends. The result is memory that has been assigned but is unusable. *Garbage collection* watches for unused objects in the heap and automatically frees them when it determines that the object no longer can be referenced by any code in a program. In fact, C# does not provide any statements through which you can directly free objects. You cannot control when garbage collection occurs. However, C# does provide ways, such as the *System.GC* class, by which you can nudge the garbage collection or keep an object in memory past its normal lifetime.

USE IT The following short C# program, *gc.cs*, shows how you can force garbage collection using methods from the *System.*GC class. The program first gets the amount of memory that has been allocated, creates 10,000 instances of a nonsense class, and then asks the garbage collector to fix it up:

```
//
// gc.cs -- Demonstrates forced garbage collection
//
//          Compile this program with the following command line:
//              C:>csc gc.cs
//
namespace nsGarbage
{
    using System;
    class clsMain
    {
        static public void Main ()
        {
            long Mem = GC.GetTotalMemory (false);
            Console.WriteLine ("Beginning allocated memory is " + Mem);
            for (int x = 0; x < 10000; ++x)
```

```
            {
                clsClass howdy = new clsClass();
            }
            Mem = GC.GetTotalMemory (false);
            Console.WriteLine ("Allocated memory before " +
                               "garbage collection is " + Mem);
            GC.Collect ();
            Mem = GC.GetTotalMemory (true);
            Console.WriteLine ("Allocated memory after " +
                               "garbage collection is " + Mem);
        }
    }
    class clsClass
    {
        public clsClass () { }
        public int x = 42;
        public float f = 2E10f;
        public double d = 3.14159;
        public string str = "This here's a string";
    }
}
```

The *GetTotalMemory()* method returns the CLR's estimate of how much memory has been allocated for your program. You will probably see larger numbers at the beginning than at the end because of the number of objects that are needed for startup.

Visual Studio–supported languages that use the CLR include C#, Visual Basic, and Managed C++. Programs written in these languages will share a common runtime environment, and modules written in one language are supposed to appear transparent to a program written in another language.

Visual Basic programmers will have to give up some niceties to code in .NET. For example, the Option Base statement is gone in Visual Basic .NET, and you will no longer be able to use a variable without declaring it. This is necessary, of course, in the interest of type-safety. The CLR guarantees that a variable of a given data type will always contain a value of that data type. Undeclared variables in Visual Basic assumed the *Variant* data type, which could hold values of virtually any type. In Visual Basic .NET, you will have to declare all variables before you use them.

Language interoperability is made possible through the Common Language Specification (CLS) and the Common Type System (CTS). Supported languages adhere to the CLS, which defines how programs and modules may expose objects to other programs. If your program adheres to the CLS, it is "guaranteed" to be accessible from a program or module written in any other language that supports the CLS. The Common Type System determines how data types are declared, used, and managed in the CLR, and it is a key element in language interoperability.

The .NET framework allows developers and computer users (or administrators in the case of a business environment) to configure how programs and modules may access and use local resources. Developers can put the configuration information into a file (hark! remember the old .INI file) and thus not need to recompile the program when a setting changes. The configuration file is an XML

(eXtensible Markup Language) format file, and the .NET framework provides classes in the *System .Configuration* namespace to manipulate the information in these files.

The .NET environment also supports the concept of *rich data*. At one time, it was fairly easy to break down the data that a program used into things like strings, numbers, dates, and currency values. Now that data may include e-mail, hypertext links, sound files, video, and the like. The Web tools in C# allow you to include and manipulate these new data types.

Viewing Intermediate Language Code

The C# compiler converts source code files to an IL, which theoretically may be used to run the same program on different operating systems and computer platforms. When you run a program, the runtime sends this IL (Microsoft Intermediate Language, or MSIL) to a *just-in-time* (JIT) compiler, which converts the code into native code for the computer on which the program is running. Microsoft uses the term *just-in-time* to indicate an action that is taken only when it is necessary.

USE IT You can view the MSIL code by using the IL Disassembler, too, *ildasm.exe*:

1. Run a Visual Studio .NET command line, and then type **ildasm** at the command prompt.
2. When the Disassembler window appears, choose File | Open.
3. Maneuver to a directory in which a program compiled from a C# program is located. The *CalcPi.exe* program will serve well—the program is short enough that there isn't a lot of intermediate code.
4. When you display the program in the Disassembler, you should see three lines. The first is a line containing the full path to the program file. Below that is the Manifest, and directly below that is a line that should read *CalcPi*. This last line is the namespace you used in the program. If you did not use a namespace, it will contain the name of the first class in the program. Double click the plus sign (+) symbol next to the *CalcPi* item. This will reveal the classes in the namespace.
5. Click on the plus sign (+) symbol next to the *Class1* item—which is the only class in the namespace to display the members of the class. The first line shows the class declaration as seen by the IL compiler. The *ctor* item is the class constructor. The *CalcPi.cs* program does not have a constructor, so the C# compiler provided it with a default.
6. The last line shows the *Main()* method, the only method in *Class1*. Double-click this item to display the IL code for the method. The Disassembler with the IL code should look like Figure 1-2.

7. To write the program code to a text file, return to the Disassembler window and press CTRL-D, or choose File | Dump.
8. Save the intermediate code in a file with an extension *.il*, such as *dump.il*. Examine the files in the directory where you wrote the file. The Disassembler should have written a resource file, *dump.res*, as well.

Now that you have disassembled the code, you can assemble it again to provide debug information. Type the following command:

```
C:>ilasm /debug /resource=dump.res dump.il
```

This command re-creates the original program as *dump.exe*, but with debugging information, and is ready to run through one of the .NET debuggers.

```
Class1::Main : void(string[])
            int32 V_2,
            int32 V_3,
            float64 V_4,
            float64 V_5,
            float64 V_6,
            float64 V_7,
            valuetype [mscorlib]System.DateTime V_8,
            valuetype [mscorlib]System.TimeSpan V_9)
  IL_0000:  call       valuetype [mscorlib]System.DateTime [mscorlib]System.D
  IL_0005:  stloc.0
  IL_0006:  ldloca.s   V_0
  IL_0008:  call       instance int32 [mscorlib]System.DateTime::get_Millisec
  IL_000d:  newobj     instance void [mscorlib]System.Random::.ctor(int32)
  IL_0012:  stloc.1
  IL_0013:  ldc.i4.0
  IL_0014:  stloc.2
  IL_0015:  ldc.i4.0
  IL_0016:  stloc.3
  IL_0017:  br.s       IL_0050
  IL_0019:  ldloc.1
  IL_001a:  callvirt   instance float64 [mscorlib]System.Random::NextDouble()
  IL_001f:  stloc.s    V_4
  IL_0021:  ldloc.1
```

Figure 1.2 The Intermediate Language Disassembler gives you a platform to view the intermediate code used for C# programs

Taking Advantage of Just-In-Time Debugging

One of the better features of an integrated development environment such as Visual Studio is JIT debugging. Whether your program is a Windows application or a command-line program, even the best of planning will not guarantee that your program will not throw an exception at some time. In the case of a Windows application that you develop in Visual Studio, eventually you are going to have to test it outside the IDE.

If your program *does* throw an exception and you do not handle it, your program is going to exit. On a computer with Visual Studio .NET or the .NET framework SDK installed, the exception pops up a dialog box that gives you the option of entering one of the debugging programs to find and fix the error.

USE IT To demonstrate exception throwing, the following program, *Throw.cs*, intentionally causes an exception that it does not handle. Use a text editor such as Notepad to enter the program, and then compile and run it outside the IDE. Remember that to compile C# from the command line, you need to run the Visual Studio .NET command line.

After you install the IDE, choose Start | Programs | Visual Studio .NET 7.0 | Visual Studio .NET Tools | Visual Studio .NET Command Line. You will see a window that looks much like a Windows command window, but the environment variables will be set to compile and test .NET programs.

```
//  Throw.cs -- Intentionally throws an exception to demonstrate
//              Just-In-Time debugging.
//
//              Compile this program with the following command line:
//                  C:>csc /debug:full Throw.cs
//
namespace nsThrow
{
    using System;
    class clsMain
    {
        static public void Main ()
        {
           Console.WriteLine ("This program intentionally causes an error.");
            int x = 42;
            int y = 0;
            int z = x / y;
        }
    }
}
```

Compiling the program will create two new files. The *Throw.pdb* is the "program database" file that contains line and symbol information for the debuggers. The other file is the executable file, *Throw.exe*, which is the program you will run. When you run this program, you will get an exception almost immediately. The error will cause the Just-In-Time Debugging dialog box to appear, as shown in Figure 1-3.

Assuming you have Visual Studio. NET installed, the Just-In-Time Debugging dialog box will offer two choices. The first option is to run the CLR debugger, the graphical debugger supplied with the .NET framework. The second option is to run the Microsoft Development Environment, which is Visual Studio itself. Figure 1-4 shows the program file open in the Microsoft Development Environment.

With either debugger, the source code for the program will open automatically, and the line that caused the exception will be highlighted in yellow.

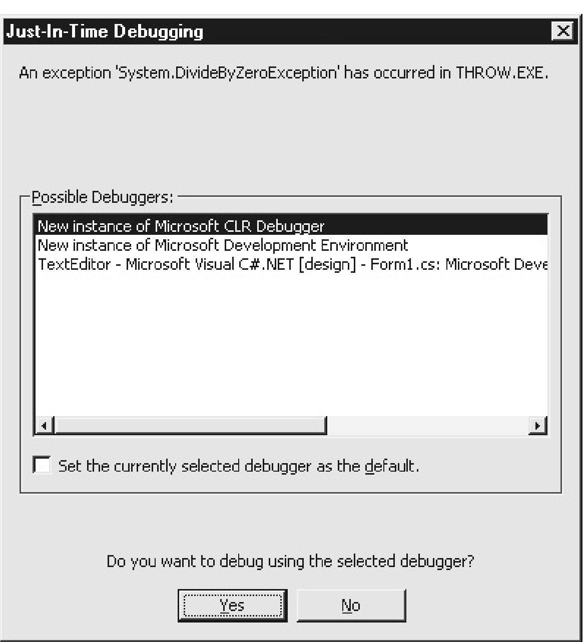

Figure 1.3 Using the Just-InTime Debugging dialog box, you can go directly from running a .NET program to debugging it

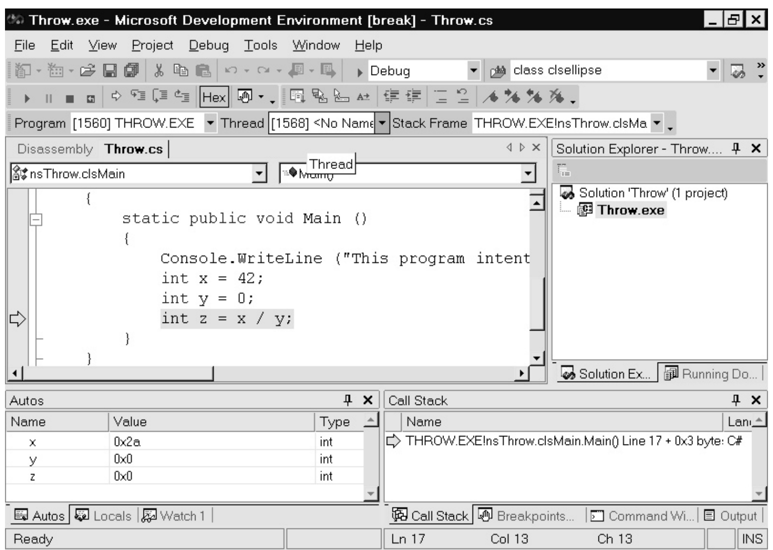

Figure 1.4 Opening the program in the Microsoft Development Environment highlights the line where the exception occurred

Exploiting the .NET Interoperability with COM

Microsoft has put a lot of thought into the design of components that will be used with the .NET framework. The goal is to eliminate many of the problems that plagued the systems that we used in the past to create components for programs. Over the years, as programs got more complex and exceeded the memory capacity of the machines on which they were to run, it became convenient to break them into smaller blocks that could be loaded on demand. The early form of this was the *overlay* file, which, when loaded into memory, actually overlaid part of the code for the program.

When dynamic link libraries (the DLL on Windows) came along, they brought the possibility of programs sharing code. This saved considerable memory, especially with multitasking or multi-user operating systems such as UNIX, and later Windows.

Next came COM, which uses the DLL. COM has two advantages. First, it is language independent. A COM component can be shared between programs written in different languages as long as those languages are aware of the COM specification. COM also brings the possibility of being able to run the components across machine boundaries.

The .NET environment is meant to operate with existing COM components. A .NET utility, *tlbimp.exe*, is provided to read the type library used with the COM object and to create a *proxy* library. Once you create the proxy library, you can use the COM component just as you would use any other .NET class.

USE IT The .NET provides tools to help you perform these tasks. Once a COM component, such as a custom Windows control, has been registered, you may view it with the OLE/COM Object Viewer and examine the interface with the TypeLib Viewer.

To use the TypeLib Viewer:

1. Start Visual Studio .NET, and then select Tools | OLE/COM Object Viewer. The viewer window should appear, as shown in Figure 1-5.
2. To view the type library for a COM object, click on the plus (+) sign next to the Type Libraries item.
3. Locate the name of the COM object in the list and double-click it to display the TypeLib Viewer, as shown in Figure 1-6.

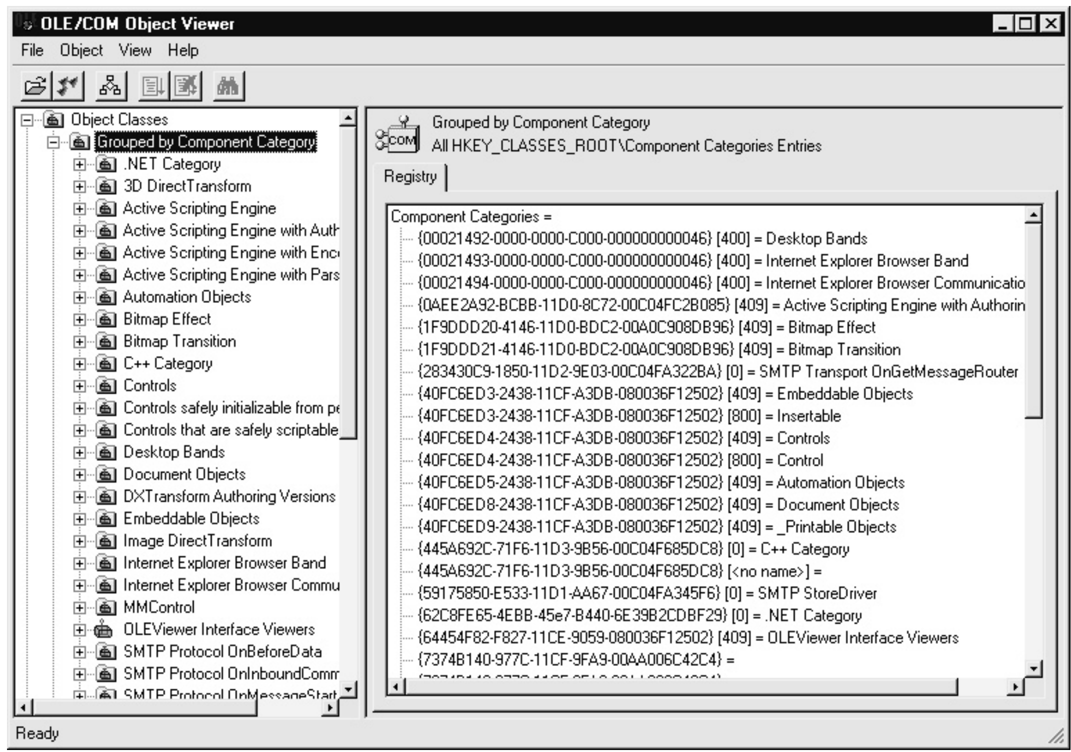

Figure 1.5 The OLE/COM Object View is the starting point for examining COM modules

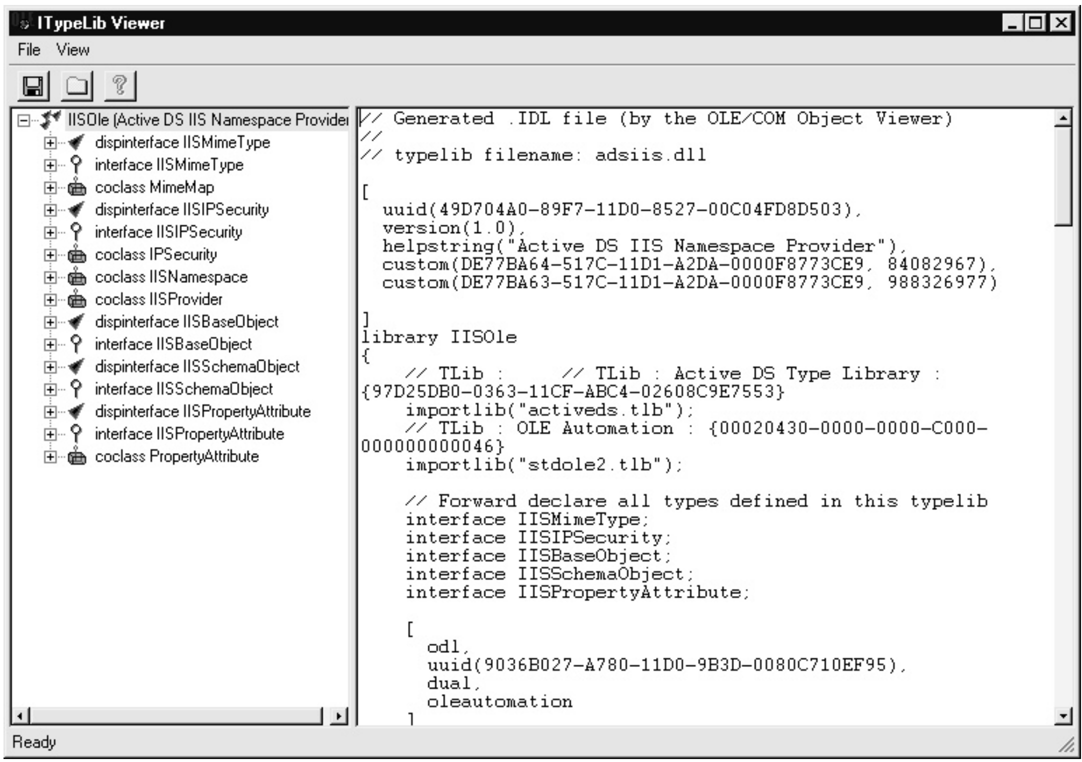

Figure 1.6 The TypeLib Viewer shows the interface listing for a COM object; COM and ActiveX programmers will recognize this as the IDL file generated by Visual Studio

But COM is another topic, and this book is about C#. However, at times you will have to interrogate the interface for a COM module. If you have done some COM programming, you will recognize the text in the viewer as the interface definition file (IDF) generated when you create a COM project in Visual Studio.

Using .NET Versioning to Handle Software Updates

C# and .NET provide better versioning capabilities than the resource files permitted with C++. When you place an assembly in the global assembly cache, its name is made up of the assembly name and

its version number. It is possible, then, for several different versions of an assembly to be stored in the assembly cache. A version of an assembly consists of the four parts that are familiar to C++ programmers: First are the major and minor numbers, followed by the build number and the revision number. The version of an assembly is written with periods between the parts in the form *Major .Minor.Build.Revision*.

The major and minor numbers are intended to indicate new releases of an assembly, but assemblies with different major and minor numbers may be compatible with one another. The build and revision numbers are intended to indicate bug fixes and Quick Fix Engineering changes.

The ability to have different versions in the assembly cache overcomes a major problem with COM objects and DLLs. If you have only one program using the DLL files, installing a new version is no problem. You simply update the component files at the same time. However, if the files are shared between programs, updating the files for one may *break* the other applications. A program should be compatible with the assembly if the major and minor revision numbers are the same as in the application's manifest.

The CLR performs version checking only on shared assemblies. Private assemblies—those that you place in the same directory as your executable file—do not get checked before they are linked or loaded to run with your program. This is so you can have control over the contents of the directory in which you install your files.

Assemblies in the global assembly cache are intended to be shared among applications. For these assemblies, the CLR will compare the version number for a shared assembly with the one that was recorded when the application was compiled. Assemblies in the cache must be prepared for the cache and given a security key. An assembly contains two keys, one private and the other public. Only programs that "know" the private key will be able to use a shared assembly.

In Chapter 10, you will create the security keys that a shared assembly needs, add them to the assembly, set the version number, and place the assembly in the global assembly cache.

USE IT When you compile your program and add a reference, the compiler imbeds the version of the shared assembly into the application's manifest. To view a manifest:

1. Return to the IL Disassember program, *ildasm.exe*. Change to the directory where you have stored a program such as the *gc.cs* program earlier in this chapter.
2. Open the executable file in the disassembler using the following command line:

```
C:> ildasm gc.exe
```

3. Double-click the line that reads *MANIFEST* to open the MANIFEST window, as shown in Figure 1-7.

MANIFEST

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\
  .ver 1:0:2411:0
}
.assembly gc
{
  // --- The following custom attribute is added automatically, do not uncomm
  //  .custom instance void [mscorlib]System.Diagnostics.DebuggableAttribute:
  //
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module gc.exe
// MVID: {067C24E8-AE4A-4AB7-B5B0-FF45A3AC1FD9}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x03080000
```

Figure 1.7 The fourth line of the manifest for *gc.exe* shows that it needs Version 1.0 of the *mscorlib* assembly. The third number in the version is the build, and the fourth is the revision number.

The first item in the manifest shown here is the *mscorlib* assembly. This is akin to linking a C++ application with the runtime library when you compile the program. The C# compiler automatically includes *mscorlib* when you compile the program, regardless of whether you use any of the .NET code in your program.

Querying Class Capabilities Through .NET Reflection

The assembly's manifest contains a collection of information about the assembly called *metadata*. The metadata tells about the external assemblies it needs to execute and security information. Your program can use the information in the metadata through *reflection*, the ability to determine information about an object's members at runtime. Java programmers probably will be familiar with reflection, and reflection in C# is similar.

USE IT A common use for reflection is to discover information about the members of an object. In C#, all classes inherit from the *object* class. To see what members a class you define inherits from *object*, run the following program, *Reflect.cs*:

```
// Reflect.cs -- Uses reflection to show the inherited members of a class
//
//               Compile this program with the following command line:
//                   C:>csc Reflect.cs
//
using System;
using System.Reflection;

namespace nsReflect
{
    class clsReflection
    {
    }
    class clsMain
    {
        static public void Main ()
        {
            clsReflection refl = new clsReflection ();
            Type t = refl.GetType();
            Console.WriteLine ("The type of t is " + t.ToString());
            MemberInfo [] members = t.GetMembers();
            Console.WriteLine ("The members of t are:");
            foreach (MemberInfo m in members)
                Console.WriteLine ("   " + m);
        }
    }
}
```

The *clsReflection* class is empty, but when you run the program you see that it contains five member methods, four of which are inherited from *object* (the *.ctor* item is the default constructor for the class):

```
The type of t is nsReflect.clsReflection
The members of t are:
   Int32 GetHashCode()
   Boolean Equals(System.Object)
   System.String ToString()
   System.Type GetType()
   Void .ctor()
```

Add a field, a property, and a method to the *clsReflection* class, as shown in the following listing:

```
class clsReflection
{
```

```
    private double pi = 3.14159;
    public double Pi
    {
        get {return (pi);}
    }
    public string ShowPi ()
    {
        return ("Pi = " + pi);
    }
}
```

Recompile and run the program, and you will see the members you just added to the list:

```
The type of t is nsReflect.clsReflection
The members of t are:
   Int32 GetHashCode()
   Boolean Equals(System.Object)
   System.String ToString()
   Double get_Pi()
   System.String ShowPi()
   System.Type GetType()
   Void .ctor()
   Double Pi
```

You can use reflection to execute a class member directly. The following program, *Reflect2.cs*, uses reflection to get the *ShowPi()* method and then executes it using the *Invoke()* method:

```
// Reflect2.cs -- Uses reflection to execute a class method indirectly
//
//                Compile this program with the following command line:
//                    C:>csc Reflect2.cs
//
using System;
using System.Reflection;

namespace nsReflect
{
    class clsReflection
    {
        private double pi = 3.14159;
        public double Pi
        {
            get {return (pi);}
        }
        public string ShowPi ()
```

```
        {
            return ("Pi = " + pi);
        }
    }
    class clsMain
    {
        static public void Main ()
        {
            clsReflection refl = new clsReflection ();
            Type t = refl.GetType();
            MethodInfo GetPi = t.GetMethod ("ShowPi");
            Console.WriteLine (GetPi.Invoke (refl, null));
        }
    }
}
```

Reflection is similar to the C++ Run Time Type Information (RTTI), which allows a programmer to obtain information about a class at runtime. Once the type of an object is obtained, an object can be cast to that type. However, C# carries this much further than the C++ RTTI, and you can discover details about a class, such as its member, at runtime.