PART I
The C# Ecosystem

1

The C# Environment

WHAT'S IN THIS CHAPTER

- IL and the CLR
- JIT compiling
- Programs and assemblies
- The .NET Framework

WROX.COM DOWNLOADS FOR THIS CHAPTER

Please note that all the code examples for this chapter are available as a part of this chapter's code download on the book's website at `www.wrox.com/go/csharp5programmersref` on the Download Code tab.

A C# program cannot exist in isolation. You can't write C# programs without using other tools. You can't even run a compiled C# program without libraries that provide runtime support.

This chapter describes the tools that you need in the Windows environment to write, compile, and execute C# programs. Most of the time those tools work smoothly behind the scenes, so you don't need to be aware of their presence. It's still worth knowing what they are, however, so you know how all the pieces of the C# environment fit together.

VISUAL STUDIO

You can write a C# program in a text editor and then use a command-line interface to compile the program. (For example, see "Working with the C# 2.0 Command Line Compiler" at `http://msdn.microsoft.com/library/ms379563.aspx` for more information.)

That approach is a lot of work, however, so most C# programmers use Visual Studio.

Visual Studio is a powerful *integrated development environment* (*IDE*) that includes code editors, form and window designers, and flexible debugging tools. Some versions also include testing, profiling, team programming, and other tools.

The Visual Studio code editors provide IntelliSense help, which displays prompts and descriptions of items you need to enter into the code. The code editor's features such as IntelliSense make writing correct C# programs much easier than it is with a simple text editor.

If you haven't already installed Visual Studio, you should probably do it now. It takes a while, so be sure you have a fast Internet connection.

To learn about and download one of the Visual Studio Express Editions, go to `www.visualstudio.com/products/visual-studio-express-vs`.

To learn about the other Visual Studio editions, go to `www.microsoft.com/visualstudio/eng/products/compare`.

While Visual Studio is downloading and installing, you can read further.

The most important tool integrated into Visual Studio is the compiler, which turns C# code into a compiled executable program—well, sort of.

THE C# COMPILER

The C# compiler doesn't actually compile code into a truly executable program. Instead it translates your C# code into an assembly-like language called *Intermediate Language* (*IL*).

WHAT'S IN A NAME?

While under development, the intermediate language was called Microsoft Intermediate Language (MSIL). When .NET was released, the name was changed to IL.

The international standards organization Ecma created the Common Language Infrastructure (CLI) standard that defines a Common Intermediate Language (CIL).

To summarize the alphabet soup, MSIL is the old name for Microsoft's intermediate language; IL is the current name; and CIL is the name for the non-Microsoft standard. There are some differences between IL and CIL but many .NET developers use MSIL, IL, and CIL interchangeably.

PROGRAMS AND ASSEMBLIES

The C# compiler doesn't compile only programs; it can also compile other kinds of assemblies. An *assembly* is the smallest possible piece of compiled code. Assemblies include programs, code libraries, control libraries, and anything else you can compile. An executable program consists of one or more assemblies.

Consider the following C# code.

```
static void Main(string[] args)
{
    foreach (string arg in args) Console.WriteLine(arg);
    Console.WriteLine("Press Enter to continue");
    Console.ReadLine();
}
```

The C# compiler translates this into the following IL code.

```
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       51 (0x33)
  .maxstack  2
  .locals init ([0] string arg,
           [1] string[] CS$6$0000,
           [2] int32 CS$7$0001,
           [3] bool CS$4$0002)
  IL_0000:  nop
  IL_0001:  nop
  IL_0002:  ldarg.0
  IL_0003:  stloc.1
  IL_0004:  ldc.i4.0
  IL_0005:  stloc.2
  IL_0006:  br.s       IL_0017
  IL_0008:  ldloc.1
  IL_0009:  ldloc.2
  IL_000a:  ldelem.ref
  IL_000b:  stloc.0
  IL_000c:  ldloc.0
  IL_000d:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0012:  nop
  IL_0013:  ldloc.2
  IL_0014:  ldc.i4.1
  IL_0015:  add
  IL_0016:  stloc.2
  IL_0017:  ldloc.2
  IL_0018:  ldloc.1
  IL_0019:  ldlen
  IL_001a:  conv.i4
  IL_001b:  clt
  IL_001d:  stloc.3
  IL_001e:  ldloc.3
  IL_001f:  brtrue.s   IL_0008
  IL_0021:  ldstr      "Press Enter to continue"
  IL_0026:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_002b:  nop
  IL_002c:  call       string [mscorlib]System.Console::ReadLine()
  IL_0031:  pop
  IL_0032:  ret
} // end of method Program::Main
```

DISPLAYING IL

You can use the ildasm program to view a compiled program's IL code. (Ildasm is pronounced "eye-ell-dazm" so it rhymes with "chasm." The name stands for "IL disassembler.") For information about ildasm, see `http://msdn.microsoft.com/library/f7dy01k1.aspx`.

The IL code is fairly cryptic; although, if you look closely you can see the method's declaration and calls to `Console.WriteLine` and `Console.ReadLine`.

IL code looks a lot like assembly language but it's not. Assembly language is a (barely) human-readable version of machine code that can run on a specific kind of computer. If the program were translated into assembly or machine code, it could run only on one kind of computer. That would make sharing the program on different computers difficult.

To make sharing programs on multiple computers easier, IL provides another layer between C# code and machine code. It's like a virtual assembly language that still needs to be compiled into executable machine code. You can copy the IL code onto different computers and then use another compiler to convert it into machine code at run time. In .NET, the *Common Language Runtime* (*CLR*) performs that compilation.

THE CLR

CLR is a virtual machine component of the .NET Framework that translates IL into native machine code when you run a C# program. When you double-click a C# program's compiled executable program, the CLR translates the IL code into machine code that can be executed on the computer.

The CLR uses a *just-in-time compiler* (*JIT compiler*) to compile pieces of the IL code only when they are needed. When the program is loaded, the loader creates a stub for each method. Initially, that stub points to the method's IL code.

When the program invokes the method, the JIT compiler translates its IL code into machine code, makes the stub point to it, and then runs the machine code. If the program calls the method again later, its stub already points to the machine code, so the method doesn't need to be compiled again.

Figure 1-1 shows the process graphically.

Usually, the time needed to compile a method is small, so you don't notice the tiny bits of extra time used as each method is called for the first time. After a method is compiled, it runs a tiny bit faster when it is called later.

If a method is never called by the program, it is never compiled by the JIT compiler, so the compiler saves some time.

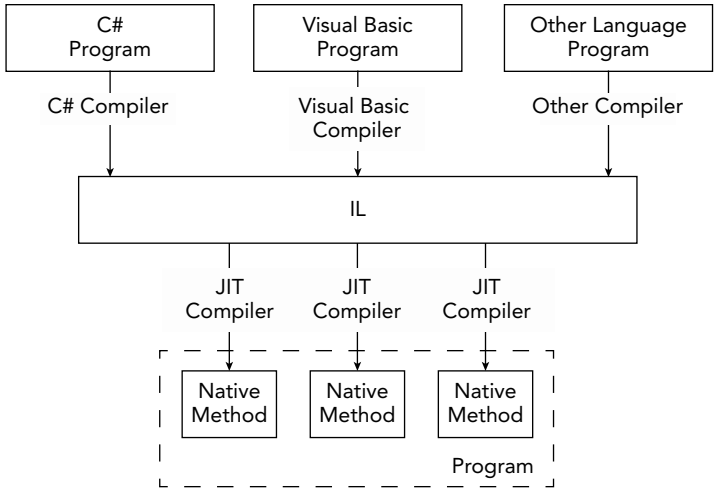

FIGURE 1-1: Compilers translate C# code (and code in other languages) into IL code. At run time, the JIT compiler translates methods from IL code into machine code as needed.

AHEAD-OF-TIME COMPILING

Normally .NET programs use the JIT compiler, but if you want to, you can use the NGen.exe program to precompile a program into native code. Then when you run the program, it is already compiled, so each method doesn't need to be compiled just in time.

This might not save you as much time as you would think. Hard drives are slow compared to operations performed in memory, so loading the compiled methods from disk may take much longer than the time saved precompiling the program.

You can speed things up if you give an assembly a strong name and install it in the system's Global Assembly Cache (GAC, pronounced "gack"). In that case, multiple programs that share the assembly may not need to load the compiled assembly from disk. The whole process is rather involved and only useful under specialized conditions, so it's not described in any greater detail here. For more information about NGen and the JIT compiler, see "Compiling MSIL to Native Code" at `http://msdn.microsoft.com/library/ht8ecch6.aspx`. For more information about using NGen and the GAC to improve performance, see "The Performance Benefits of NGen" at `http://msdn.microsoft.com/magazine/cc163610.aspx`.

In addition to providing JIT compilation, the CLR also provides some low-level services used by programs such as memory management, thread management, and exception handling. (Chapter 12, "Classes and Structures," describes the .NET memory management model. Chapter 6, "Methods,"

and Chapter 22, "Parallel Programming," discuss using multiple threads. Chapter 9, "Error Handling," describes exception handling in C#.)

THE .NET FRAMEWORK

The .NET Framework includes the CLR and a large library of powerful tools that make C# programming simpler. Those tools include just about everything you normally use in a C# program that isn't part of the C# language itself. Some of the tools included in the .NET Framework enable you to

- Add attributes to classes and their members to give extra information to runtime tools.
- Use collections such as lists, dictionaries, and hash tables.
- Work with databases.
- Find the computer's physical location using methods such as GPS, Wi-Fi triangulation, and cell tower triangulation.
- Interact with system processes, event logs, and performance data.
- Create sophisticated two-dimensional drawings.
- Interact with Active Directory.
- Provide globalization so that programs use appropriate text and images in different locales.
- Use LINQ (described in Chapter 8, "LINQ").
- Create and use message queues.
- Work with the filesystem.
- Play audio and video.
- Get information about and manage devices.
- Interact with networks and the Internet.
- Print documents.
- Examine the code entities defined by the program or another compiled assembly.
- Serialize and deserialize objects.
- Control program security.
- Support speech recognition.
- Run multiple threads of execution simultaneously.
- Process XML and JSON files.
- Encrypt and decrypt files.
- Much more.

By using all of these tools, you can build standalone programs to run on desktop systems, phone or tablet applications, websites, networked applications, and all sorts of other programs.

A compiled C# program needs the CLR to execute, and most programs also need the .NET Framework. That means to run a program, a computer must have the .NET Framework installed.

If Visual Studio is installed on the computer, the .NET Framework is also installed. That means you can usually copy a compiled C# application onto the computer and it will run. (Of course, you should never copy a compiled program from a source you don't trust! This method works if you want to share a program with your friends, but don't just grab any old compiled C# program off the Internet.)

Most installers also install the .NET Framework if needed, so if you use an installer, the .NET Framework will be installed if it's not already on the machine. For example, ClickOnce deployment installs the .NET Framework if necessary.

You can also install the .NET Framework manually by downloading an installer or by using a web installer. To find the latest installers, go to Microsoft's Download Center at `www.microsoft.com/download/default.aspx` and search for **.NET Framework**.

CLICKONCE

To use ClickOnce deployment, select Build ➪ Publish, and let the Publish Wizard guide you through the process. The wizard enables you to determine

- The location where the distribution package should be built
- Whether the user will install from a website, a file, or a CD or DVD
- Whether the application should check for updates when it runs

For more information on ClickOnce deployment, see `http://msdn.microsoft.com/library/142dbbz4.aspx`.

Most of the .NET Framework features are backward compatible, so usually you can install the most recent version, and programs built with older versions will still run. If you do need a particular version of the .NET Framework, search the Download Center for the version you need.

SUMMARY

A C# program cannot stand completely alone. To create, compile, and run a C# program, you need several tools. You can create a C# program in a text editor, but it's much easier to use Visual Studio to write and debug programs. After you write a program, the C# compiler translates the C# code into IL code. At run time, the CLR (which is part of the .NET Framework) uses JIT compilation to translate the IL code into native machine code for execution.

You can use NGen to precompile assemblies and install them in the GAC, so they don't need to be compiled at run time by the JIT compiler. In many cases, however, that won't save much time. If the

program's methods are called when the user performs actions such as clicking buttons and invoking menu items, the small additional overhead probably won't be noticeable.

All that happens behind the scenes when you build and execute a C# program. The next chapter explains how you can start to build C# programs. It explains the most common types of C# projects and explains how you can use several of them to test C# code as you work through the rest of this book.

EXERCISES

1. Draw a diagram showing the major steps that Visual Studio performs when you write a C# program and press F5 to run it.
2. Suppose you have two applications. The first uses 50 methods to perform tasks as the user selects them. The second uses the same 50 methods to perform all the tasks when it starts. How will the performance of the two applications differ? How do NGen and the GAC apply to this situation?
3. For which of the following scenarios would NGen and the GAC be most useful?
    a. An interactive application that displays forms on the screen
    b. A code library that defines methods for performing tasks that your other programs use
    c. A control library that defines custom buttons, scroll bars, and other controls
    d. A console application that writes output to a console window
4. Suppose your program uses 100 methods as the user performs various actions. You add a parameter to each method, so you can tell it to return without doing anything. Then when the program starts (while the splash screen displays), the code calls every method telling it to return without doing anything. How would the CLR handle those methods?