PART I
The C# Language

1

Introducing C#

WHAT YOU WILL LEARN IN THIS CHAPTER

- What the .NET Framework is
- What C# is
- Explore Visual Studio 2017

Welcome to the first chapter of the first section of this book. This section provides you with the basic knowledge you need to get up and running with the most recent version of C#. Specifically, this chapter provides an overview of the .NET Framework and C#, including what these technologies are, the motivation for using them, and how they relate to each other.

It begins with a general discussion of the .NET Framework. This technology contains many concepts that are tricky to come to grips with initially. This means that the discussion, by necessity, covers many concepts in a short amount of space. However, a quick look at the basics is essential to understanding how to program in C#. Later in the book, you revisit many of the topics covered here, exploring them in more detail.

After that general introduction, the chapter provides a basic description of C# itself, including its origins and similarities to C++. Finally, you look at the primary tool used throughout this book: Visual Studio (VS). Visual Studio 2017 is the latest in a long line of development environments that Microsoft has produced, and it includes all sorts of features (including full support for Windows Store, Azure, and cross-platform applications) that you will learn about throughout this book.

WHAT IS THE .NET FRAMEWORK?

The .NET Framework (now at version 4.7) is a revolutionary platform created by Microsoft for developing applications. The most interesting thing about this statement is how vague and limited it is—but there are good reasons for this. To begin with, note that it doesn't actually "develop applications only on the Windows operating system." Although the Microsoft release of the .NET Framework runs on the Windows and Windows Mobile operating systems, it is possible to find alternative versions that will work on other systems. One example of this is Mono, an open source version of the .NET Framework (including a C# compiler) that runs on several operating systems, including various -flavors of Linux and Mac OS; you can read more about it at `http://www.mono-project.com`.

Mono is a very important part of the .NET ecosystem, especially for creating client-side applications with Xamarin. Microsoft has also created a cross platform open source library called .NET Core (`https://github.com/dotnet/core`) which they hope will have a positive impact on both the Mono and .NET Core frameworks. Programmers in both ecosystems can use examples from each other's libraries to improve performance, security, and the breadth of language feature offerings—collaboration is a key characteristic in the open source community.

In addition, the definition of the .NET Framework includes no restriction on the type of applications that are possible. The .NET Framework enables the creation of desktop applications, Windows Store (UWP) applications, cloud/web applications, Web APIs, and pretty much anything else you can think of. Also, it's worth noting that web, cloud, and Web API applications are, by definition, multi-platform applications, since any system with a web browser can access them.

The .NET Framework has been designed so that it can be used from any language, including C# (the subject of this book) as well as C++, F#, JScript, Visual Basic, and even older languages such as COBOL. For this to work, .NET-specific versions of these languages have also appeared, and more are being released all the time. For a list of languages, see `https://msdn.microsoft.com/en-us/library/ee822860(v=vs.100).aspx`. Not only do these languages have access to the .NET Framework, but they can also communicate with each other. It is possible for C# developers to make use of code written by Visual Basic programmers, and vice versa.

All of this provides an extremely high level of versatility and is part of what makes using the .NET Framework such an attractive prospect.

What's in the .NET Framework?

The .NET Framework consists primarily of a gigantic library of code that you use from your client- or server-side languages (such as C#) using object-oriented programming (OOP) techniques. This library is categorized into different modules—you use portions of it depending on the results you want to achieve. For example, one module contains the building blocks for Windows applications, another for network programming, and another for web development. Some modules are divided into more specific submodules, such as a module for building web services within the module for web development.

The intention is for different operating systems to support some or all of these modules, depending on their characteristics. A smartphone, for example, includes support for all the base .NET functionality, but is unlikely to require some of the more esoteric modules.

Part of the .NET Framework library defines some basic *types*. A type is a representation of data, and specifying some of the most fundamental of these (such as "a 32-bit signed integer") facilitates interoperability between languages using the .NET Framework. This is called the *Common Type System* (CTS).

As well as supplying this library, the .NET Framework also includes the .NET *Common Language Runtime* (CLR), which is responsible for the execution of all applications developed using the.NET library.

.NET Standard and .NET Core

When the .NET Framework was originally created, although it was designed for running on multiple platforms, there was no industry accepted open-source forking concept. These days, usually on GitHub, a project can be forked and then customized to run on multiple platforms. For example, the .NET Compact Framework and the .NET Micro Framework are forks of the .NET Framework, like .NET Core, which was created as the most optimal solution for cross-platform code development. Each of those .NET Framework "flavors" or "verticals" had a specific set of requirements and objectives that triggered the need to make that fork.

Included in the .NET Framework is a set of Base Class Libraries (BCL) that contain APIs for basic actions most developers need a program to do. These actions include, for example, file access, string manipulation, managing streams, storing data in collections, security attributes, and many others. These fundamental capabilities are often implemented differently within each of the .NET Framework flavors. This requires a developer to learn, develop, and manage multiple BCLs for each fork or flavor of their application based on the platform it runs. This is the problem that the .NET Standard has solved.

The expectation is that shortly, this forking concept will result in many more flavors of the .NET Framework. This increase will necessitate a standard set of basic programming APIs that works with each fork and flavor. Without this cross platform base library, the development and support complexities would prevent the speedy adoption of the forked version. In short, .NET Standard is a class library that exposes APIs that support any fork or flavor of application using the .NET Platform.

Writing Applications Using the .NET Framework and .NET Core

Writing an application using either the .NET Framework or .NET Core means writing code (using any of the languages that support the Framework) using the .NET code library. In this book you use Visual Studio for your development. Visual Studio is a powerful, integrated development environment that supports C# (as well as managed and unmanaged C++, Visual Basic, and some others).

The advantage of this environment is the ease with which .NET features can be integrated into your code. The code that you create will be entirely C# but use the .NET Framework throughout, and you'll make use of the additional tools in Visual Studio where necessary.

In order for C# code to execute, it must be converted into a language that the target operating -system understands, known as *native code*. This conversion is called *compiling* code, an act that is performed by a *compiler*. Under the .NET Framework and .NET Core, this is a two-stage process.

CIL and JIT

When you compile code that uses either the .NET Framework or .NET Core library, you don't immediately create operating system–specific native code. Instead, you compile your code into *Common Intermediate Language* (CIL) code. This code isn't specific to any operating system (OS) and isn't specific to C#. Other .NET languages—Visual Basic .NET or F#, for example—also compile to this language as a first stage. This compilation step is carried out by Visual Studio when you develop C# applications.

Obviously, more work is necessary to execute an application. That is the job of a *just-in-time* (JIT) compiler, which compiles CIL into native code that is specific to the OS and machine architecture being targeted. Only at this point can the OS execute the application. The *just-in-time* part of the name reflects the fact that CIL code is compiled only when it is needed. This compilation can happen on the fly while your application is running, although luckily this isn't something that you normally need to worry about as a developer. Unless you are writing extremely advanced code where performance is critical, it's enough to know that this compilation process will churn along merrily in the background, without interfering.

In the past, it was often necessary to compile your code into several applications, each of which targeted a specific operating system and CPU architecture. Typically, this was a form of optimization (to get code to run faster on an AMD chipset, for example), but at times it was critical (for applications to work in both Win9*x* and WinNT/2000 environments, for example). This is now unnecessary because JIT compilers (as their name suggests) use CIL code, which is independent of the machine, operating system, and CPU. Several JIT compilers exist, each targeting a different architecture, and the CLR/CoreCLR uses the appropriate one to create the native code required.

The beauty of all this is that it requires a lot less work on your part—in fact, you can forget about system-dependent details and concentrate on the more interesting functionality of your code.

> ***NOTE*** *As you learn about .NET you might come across references to Microsoft Intermediate Language (MSIL). MSIL was the original name for CIL, and many developers still use this terminology today. See* `https://en.wikipedia.org/wiki/Common_Intermediate_Language` *for more information about CIL.*

Assemblies

When you compile an application, the CIL code is stored in an *assembly*. Assemblies include both executable application files that you can run directly from Windows without the need for any other programs (these have an `.exe` file extension) and libraries (which have a `.dll` extension) for use by other applications.

In addition to containing CIL, assemblies also include *meta* information (that is, information about the information contained in the assembly, also known as *metadata*) and optional *resources* (additional data used by the CIL, such as sound files and pictures). The meta information enables assemblies to be fully self-descriptive. You need no other information to use an assembly, meaning you avoid situations such as failing to add required data to the system registry and so on, which was often a problem when developing with other platforms.

This means that deploying applications is often as simple as copying the files into a directory on a remote computer. Because no additional information is required on the target systems, you can just run an executable file from this directory and, assuming the .NET CLR is installed for .NET Framework targeted applications, you're good to go. For .NET Core targeted applications, all modules required to run the program are included in the deployment package and therefore no additional configurations are required.

From a .NET Framework perspective, you won't necessarily want to include everything required to run an application in one place. You might write some code that performs tasks required by multiple applications. In situations like that, it is often useful to place the reusable code in a place accessible to all applications. In the .NET Framework, this is the *global assembly cache* (GAC). Placing code in the GAC is simple—you just place the assembly containing the code in the directory containing this cache.

Managed Code

The role of the CLR/CoreCLR doesn't end after you have compiled your code to CIL and a JIT compiler has compiled that to native code. Code written using the .NET Framework and .NET Core are *managed* when it is executed (a stage usually referred to as *runtime*). This means that the CLR/CoreCLR looks after your applications by managing memory, handling security, allowing cross-language debugging, and so on. By contrast, applications that do not run under the control of the CLR/CoreCLR are said to be *unmanaged*, and certain languages such as C++ can be used to write such applications, which, for example, access low-level functions of the operating system. However, in C# you can write only code that runs in a managed environment. You will make use of the managed features of the CLR/CoreCLR and allow .NET itself to handle any interaction with the operating system.

Garbage Collection

One of the most important features of managed code is the concept of *garbage collection*. This is the .NET method of making sure that the memory used by an application is freed up completely when

the application is no longer in use. Prior to .NET this was mostly the responsibility of programmers, and a few simple errors in code could result in large blocks of memory mysteriously disappearing as a result of being allocated to the wrong place in memory. That usually meant a progressive slowdown of your computer, followed by a system crash.

.NET garbage collection works by periodically inspecting the memory of your computer and removing anything from it that is no longer needed. There is no set time frame for this; it might happen thousands of times a second, once every few seconds, or whenever, but you can rest assured that it will happen.

There are some implications for programmers here. Because this work is done for you at an unpredictable time, applications have to be designed with this in mind. Code that requires a lot of memory to run should tidy itself up, rather than wait for garbage collection to happen, but that isn't as tricky as it sounds.

Fitting It Together

Before moving on, let's summarize the steps required to create a .NET application as discussed previously:

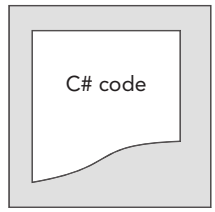

FIGURE 1-1

1. Application code is written using a .NET-compatible language such as C# (see Figure 1-1).
2. That code is compiled into CIL, which is stored in an assembly (see Figure 1-2).

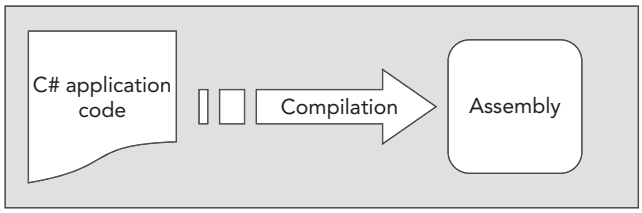

FIGURE 1-2

3. When this code is executed (either in its own right if it is an executable or when it is used from other code), it must first be compiled into native code using a JIT compiler (see Figure 1-3).

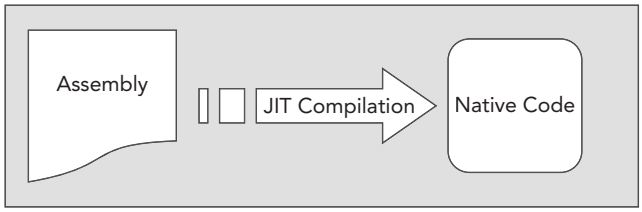

FIGURE 1-3

4. The native code is executed in the context of the managed CLR/CoreCLR, along with any other running applications or processes, as shown in Figure 1-4.

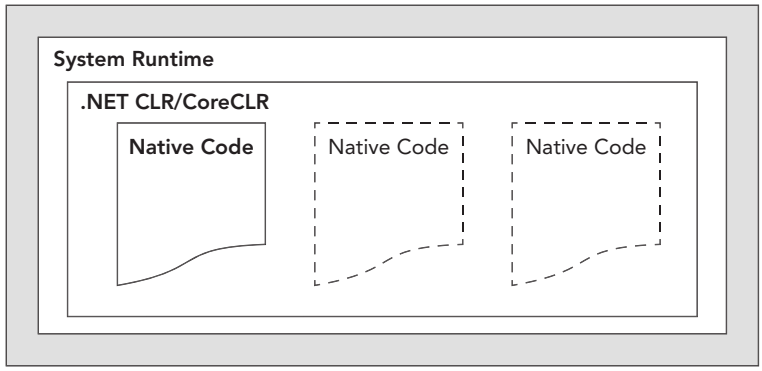

FIGURE 1-4

Linking

Note one additional point concerning this process. The C# code that compiles into CIL in step 2 needn't be contained in a single file. It's possible to split application code across multiple source-code files, which are then compiled together into a single assembly. This extremely useful process is known as *linking*. It is required because it is far easier to work with several smaller files than one enormous one. You can separate logically related code into an individual file so that it can be worked on independently and then practically forgotten about when completed. This also makes it easy to locate specific pieces of code when you need them and enables teams of developers to divide the programming burden into manageable chunks, whereby individuals can "check out" pieces of code to work on without risking damage to otherwise satisfactory sections or sections other people are working on.

WHAT IS C#?

C#, as mentioned earlier, is one of the languages you can use to create applications that will run in the .NET CLR/CoreCLR. It is an evolution of the C and C++ languages and has been created by Microsoft specifically to work with the .NET platform. The C# language has been designed to incorporate many of the best features from other languages, while clearing up their problems.

Developing applications using C# is simpler than using C++ because the language syntax is simpler. Still, C# is a powerful language, and there is little you might want to do in C++ that you can't do in C#. Having said that, those features of C# that parallel the more advanced features of C++, such as directly accessing and manipulating system memory, can be carried out only by using code marked as *unsafe*. This advanced programmatic technique is potentially dangerous (hence its name) because it is possible to overwrite system-critical blocks of memory with potentially catastrophic results. For this reason, and others, this book does not cover that topic.

At times, C# code is slightly more verbose than C++. This is a consequence of C# being a *typesafe* language (unlike C++). In layperson's terms, this means that once some data has been assigned to a type, it cannot subsequently transform itself into another unrelated type. Consequently, strict rules must be adhered to when converting between types, which means you will often need to write more code to carry out the same task in C# than you might write in C++. However, there are benefits to this—the code is more robust, debugging is simpler, and .NET can always track the type of a piece of data at any time. In C#, you therefore might not be able to do things such as "take the region of memory 4 bytes into this data and 10 bytes long and interpret it as X," but that's not necessarily a bad thing.

C# is just one of the languages available for .NET development, but it is certainly the best. It has the advantage of being the only language designed from the ground up for the .NET Framework and is the principal language used in versions of .NET that are ported to other operating systems. To keep languages such as the .NET version of Visual Basic as similar as possible to their predecessors yet compliant with the CLR/CoreCLR, certain features of the .NET code library are not fully supported, or at least require unusual syntax.

By contrast, C# can make use of every feature that the .NET Framework code library has to offer, but not all features have been ported to .NET Core. Also, each new version of .NET has included additions to the C# language, partly in response to requests from developers, making it even more powerful.

Applications You Can Write with C#

The .NET Framework has no restrictions on the types of applications that are possible, as discussed earlier. C# uses the framework and therefore has no restrictions on possible applications. (However, currently it is possible to write only Console and ASP.NET applications using .NET Core.) However, here are a few of the more common application types:

- **Desktop applications**—Applications, such as Microsoft Office, that have a familiar Windows look and feel about them. This is made simple by using the Windows Presentation Foundation (WPF) module of the .NET Framework, which is a library of *controls* (such as buttons, toolbars, menus, and so on) that you can use to build a Windows user interface (UI).
- **Windows Store applications**—Windows 8 introduced a new type of application, known as a Windows Store application. This type of application is designed primarily for touch devices, and it is usually run full-screen, with a minimum of clutter, and an emphasis on simplicity. You can create these applications in several ways, including using WPF.
- **Cloud/Web applications**—The .NET Framework and .NET Core include a powerful system named ASP.NET, for generating web content dynamically, enabling personalization, security, and much more. Additionally, these applications can be hosted and accessed in the Cloud, for example on the Microsoft Azure platform.
- **Web APIs**—An ideal framework for building RESTful HTTP services that support a broad variety of clients, including mobile devices and browsers.

- **WCF services**—A way to create versatile distributed applications. Using WCF you can exchange virtually any data over local networks or the Internet, using the same simple syntax regardless of the language used to create a service or the system on which it resides.

Any of these types might also require some form of database access, which can be achieved using the ADO.NET (Active Data Objects .NET) section of the .NET Framework, through the Entity Framework, or through the LINQ (Language Integrated Query) capabilities of C#. For .NET Core applications requiring database access you would use the Entity Framework Core library. Many other resources can be drawn on, such as tools for creating networking components, outputting graphics, performing complex mathematical tasks, and so on.

C# in this Book

The first part of this book deals with the syntax and usage of the C# language without too much emphasis on the .NET Framework or .NET Core. This is necessary because you can't use either the .NET Framework or .NET Core at all without a firm grounding in C# programming. You'll start off even simpler, in fact, and leave the more involved topic of OOP until you've covered the basics. These are taught from first principles, assuming no programming knowledge at all.

After that, you'll be ready to move on to developing more complex (but more useful) applications. Part II tackles Windows programming, Part III explores cloud and cross-platform programming, and Part IV examines data access (for ORM database concepts, filesystem, and XML data) and LINQ. Part V of this book looks at WCF and Windows Store application programming.

VISUAL STUDIO 2017

In this book, you use the Visual Studio 2017 development tool for all of your C# programming, from simple command-line applications to more complex project types. A development tool, or integrated development environment (IDE), such as Visual Studio isn't essential for developing C# applications, but it makes things much easier. You can (if you want to) manipulate C# source code files in a basic text editor, such as the ubiquitous Notepad application, and compile code into assemblies using the command-line compiler that is part of the .NET Framework and .NET Core. However, why do this when you have the power of an IDE to help you?

Visual Studio 2017 Products

Microsoft supplies several versions of Visual Studio, for example:

- Visual Studio Code
- Visual Studio Community
- Visual Studio Professional
- Visual Studio Enterprise

Visual Studio Code and Community are freely available at `https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs`. The Professional and Enterprise version have additional capabilities, which carry a cost.

The various Visual Studio products enable you to create almost any C# application you might need. Visual Studio Code is a simple yet robust code editor that runs on Windows, Linux, and iOS. Visual Studio Community, unlike Visual Studio Code, retains the same look and feel as Visual Studio Professional and Enterprise. Microsoft offers many of the same features in Visual Studio Community as exist in the Professional and Enterprise version; however, some notable features are absent, like deep debugging capabilities and code optimization tools. However, not so many features are absent that you can't use Community to work through the chapters of this book. Visual Studio Community 2017 is the version of the IDE used to work the examples in this book.

Solutions

When you use Visual Studio to develop applications, you do so by creating *solutions*. A solution, in Visual Studio terms, is more than just an application. Solutions contain *projects,* which might be WPF projects, Cloud/Web Application projects, ASP.NET Core projects, and so on. Because solutions can contain multiple projects, you can group together related code in one place, even if it will eventually compile to multiple assemblies in various places on your hard disk.

This is very useful because it enables you to work on shared code (which might be placed in the GAC) at the same time as applications that use this code. Debugging code is a lot easier when only one development environment is used because you can step through instructions in multiple code modules.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
.NET Framework and .NET Core fundamentals	The .NET Framework is Microsoft's latest development platform, and is currently in version 4.7. It includes a common type system (CTS) and common language runtime (CLR/CoreCLR). Both .NET Framework and .NET Core applications are written using object-oriented programming (OOP) methodology, and usually contain managed code. Memory management of managed code is handled by the .NET runtime; this includes garbage collection.
.NET Framework applications	Applications written using the .NET Framework are first compiled into CIL. When an application is executed, the CLR uses a JIT to compile this CIL into native code as required. Applications are compiled, and different parts are linked together into assemblies that contain the CIL.
.NET Core applications	.NET Core applications work similar to .NET Framework applications; however, instead of using the CLR it uses CoreCLR.
.NET Standard	.NET Standard provides a unified class library which can be targeted from multiple .NET platforms like the .NET Framework, .NET Core, and Xamarin.
C# basics	C# is one of the languages included in the .NET Framework. It is an evolution of previous languages such as C++, and can be used to write any number of applications, including web, cross-platform, and desktop applications.
Integrated Development Environments (IDEs)	You can use Visual Studio 2017 to write any type of .NET application using C#. You can also use the free, but less powerful, Community product to create .NET applications in C#. This IDE works with solutions, which can consist of multiple projects.