1

Introducing C#

WHAT YOU WILL LEARN IN THIS CHAPTER

- What .NET is
- What C# is
- Explore Visual Studio

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

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

After this general .NET 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 is an Integrated Development Environment (IDE) that Microsoft has produced since the late 1990s and gets updated regularly with new features. Visual Studio includes all sorts of capabilities including full support for desktop, cloud, web, mobile, database, machine learning, AI, and cross-platform programming that you will learn about throughout this book.

WHAT IS .NET?

.NET is a revolutionary software framework created by Microsoft for developing computer programs. To begin with, note that .NET provides more than the means for creating programs that target the Windows operating system. .NET is fully open source and fully supports running

cross platform. *Cross platform* means that the code you write using .NET will run on Linux and MacOS operating systems as well. The source code for .NET is open source and you can find it at `github.com/dotnet/core`.

The .NET software framework is made of prewritten computer code, which provides simple access to basic computing resources like the hard drive and computer memory. One aspect of this framework is referred to as the *Base Class Library* (BCL), which contains the `System` class. You will become very familiar with it as you progress through this book. Taking a deeper look into the source code inside the `System` class, you will find that it includes the definitions of data *types* like `strings`, `integers`, `Boolean`, and `characters`. If you need one of these data types in your program to store information, you can use the already written .NET code to achieve that. If such code did not already exist, you would need to use low-level programming languages like assembly or machine code to allocate and manage the required memory yourself. The basic types found in the `System` class also facilitate interoperability between .NET programming languages, a concept referred to as the *Common Type System* (CTS). Interoperability means that a `string` in C# has the same attributes and behaviors as a `string` in Visual Basic or F#. Besides supplying this source code library, .NET also includes the *Common Language Runtime* (CLR), which is responsible for the execution of all applications developed using the .NET library; more on that later.

In addition to the `System` class, .NET contains many, many other classes, often called modules. Some would say it is a gigantic library of object-oriented programming (OOP) code categorized into different modules—you use portions of it depending on the results you want to achieve. For example, `System.IO` and `System.Text` are the classes you would use to read and write to files located on a computer hard drive. A programmer can manipulate the contents of a file simply by using the code that already exists in the `System.IO` class without needing to manage handles or load the file from the hard drive into memory. There exist many classes in .NET that help programmers write programs at a fast pace, because all of the low-level code required to achieve their tasks has already been written. The programmer only needs the knowledge of which classes they require to achieve the program objectives.

Not only does .NET speed up application development, but it also can be utilized by numerous other programming languages, not just C# (which is the subject of this book). Programs written in C++, F#, Visual Basic, and even older languages such as COBOL can use the classes that exist in .NET. These languages have access to the code in the .NET library, but the code written in one programming language can communicate with code from another. For example, a program written in C# can make use of code written in Visual Basic or F# and vice versa. All of these examples are what makes .NET such an attractive prospect for building customized software.

.NET Framework, .NET Standard, and .NET Core

When the .NET Framework was originally created, it targeted the Windows operating system platform. Through the years, the .NET Framework code was forked to support numerous other platforms like IoT devices, desktops, mobile devices, and other operating systems. You may recognize some of the branches going by the names of .NET Compact Framework, .NET Portable, or .NET Micro Framework. Each of these forks contained its own, slightly modified BCL. Take note that a BCL is more than just `strings`, `Booleans`, and `integers`. It includes capabilities like file access, string manipulation, managing streams, storing data in collections, security attributes, and many others.

Having even a slightly different BCL required a programmer to learn, develop, and manage the subtle difference between the BCLs for each .NET fork. Each fork of the .NET Framework that targeted desktops, the Internet, or mobile platforms could have significant implementation differences, even though each program used .NET. It was (and still is) very common for a company to have desktop, website, and phone applications that ran the same program logic but did so on those different platforms. In that scenario, using .NET required a version of the company's application for each platform. That was not efficient. This is the problem that the .NET Standard solved. The .NET Standard provided a place for programmers to create application logic that could be used across any of the .NET Framework forks. The .NET Standard made the different platforms, like desktop, mobile, and web, BCL agnostic by decoupling a company's program logic from platform-specific dependencies.

.NET Core was the open source, cross-platform version of the .NET library. This fork of the code could be used to create programs that targeted numerous different platforms and operating systems like Linux, MacOS, and of course Windows. It was also the fork that would eventually become the one and only maintained branch of the .NET source code library. As of 2020, knowing about the .NET Framework, .NET Standard, and .NET Core is no longer as relevant as it once was. These three branches of .NET must be mentioned here because you will still likely see them, read about them, and be confronted with them for some years to come. It is important that you know what they are and their purpose, in case you need to work on a project that implements them. As of 2020, there is a new version of .NET simply named ".NET." .NET is fully open source, is fully cross platform, and can be used on many platforms without having to support multiple versions, forks, and branches of your program.

Writing Programs Using .NET

Creating a computer program with .NET means writing code that uses existing code found within the .NET library. In this book you use Visual Studio for developing your programs. Visual Studio is a powerful, integrated development environment that supports C# (as well as C++, Visual Basic, F#, 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 .NET throughout, and also some additional tools in Visual Studio, where necessary. 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*, an act that is performed by a *compiler*, which is a two-stage process.

CIL and JIT

When you compile code that uses .NET, 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 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* `en.wikipedia.org/wiki/Common_Intermediate_Language` *for more information about CIL (also known as Intermediate Language [IL]).*

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 CLR is installed for .NET targeted applications, you're good to go. Depending on the deployment scenario, the modules required to run the program are included in the deployment package which means no additional configurations.

From a .NET perspective, you won't necessarily want to include everything required to run an application in a single directory. You might write some code that performs tasks required by multiple applications. In situations like these, it is often useful to place the reusable code in a place accessible

to all applications. In .NET, this place 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 doesn't end after you have compiled your code to CIL and a JIT compiler has compiled that to native code. Code written using .NET is *managed* when it is executed (a stage usually referred to as *runtime*). This means that the CLR 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 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 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 using it. 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:

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).

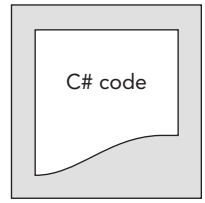

FIGURE 1-1

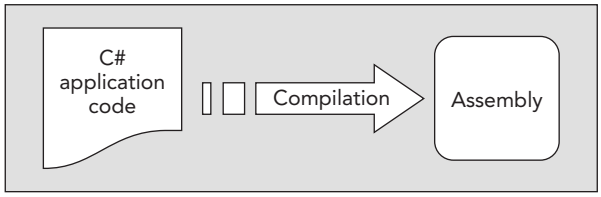

FIGURE 1-2

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).

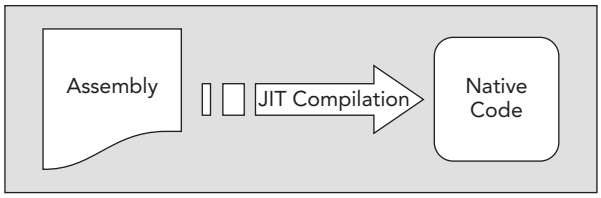

FIGURE 1-3

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

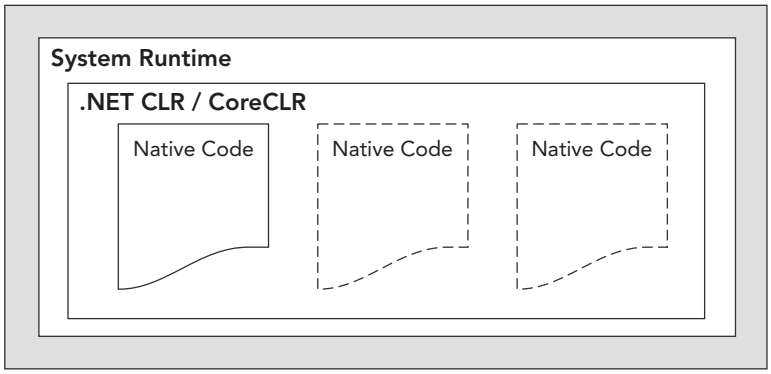

FIGURE 1-4

Linking

Note one additional point concerning this process. The C# code that compiles into CIL in step 2 need not be contained in a single file. It is 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*. This 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. This allows individuals to "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. 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 .NET 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, 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 the .NET code library has to offer. 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#

.NET has no restrictions on the types of applications that are possible. C# uses the framework and therefore has no restrictions on possible applications. 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 .NET, 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).
- **Cloud/web applications—**.NET includes a powerful system named ASP.NET Core, 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.
- **Mobile applications**—Using both C# and Xamarin mobile UI framework you can target mobile applications that target the Android operating system.
- **Web APIs**—An ideal framework for building RESTful HTTP services that support a broad variety of clients, including mobile devices and browsers. These are also referred to as REST APIs.

- **WCF services**—A way to create versatile distributed applications. With 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. This is an older technology that would require an older version of the .NET Framework to create.

Any of these types of applications might also require some form of database access, which can be achieved using the ADO.NET (Active Data Objects .NET) feature set of .NET, through the Entity Framework, or through the LINQ (Language Integrated Query) capabilities of C#. Many other resource assemblies can be utilized that are helpful with creating networking components, outputting graphics, performing complex mathematical tasks, and so on.

C# in This Book

Part I of this book deals with the syntax and usage of the C# language without too much emphasis on .NET. This is necessary because you cannot use .NET at all without a firm grounding in C# programming. You will start off even simpler, in fact, and leave the more involved topic of OOP until you have covered the basics. These are taught from first principles, assuming no programming knowledge at all.

After that, you will be ready to move on to developing more complex (but more useful) applications. Part II examines data access (for ORM database concepts, filesystem, and XML data) and LINQ. Part III explores additional techniques like REST API, the cloud, and Windows Desktop.

VISUAL STUDIO

In this book, you use the most recent version of the Visual Studio 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 is not 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 .NET. However, why do this when you have the power of an IDE to help you?

Visual Studio Products

Microsoft supplies several versions of Visual Studio. For example:

- Visual Studio Community
- Visual Studio Professional
- Visual Studio Enterprise
- Visual Studio Code
- Visual Studio for Mac

Visual Studio Code, Mac, and Community are freely available at `visualstudio.microsoft.com/downloads`. The Professional and Enterprise versions 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 versions; however, some notable features are absent, like deep debugging capabilities and code optimization tools. However, not so many features are absent that you cannot use Community to work through the chapters of this book. Visual Studio Community 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 Console Applications, 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 especially 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 fundamentals	.NET is Microsoft's code development library. It includes a Common Type System (CTS) and Common Language Runtime (CLR). .NET 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 applications	Applications written using .NET 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	.NET Core works similarly to the .NET Framework; however, instead of using the CLR it uses CoreCLR. .NET Core is a branch of the original .NET Framework, which can be run cross platform.
.NET Standard	.NET Standard provides a unified class library that can be targeted from multiple .NET platforms like the .NET Micro Framework, .NET Core, and Xamarin.
C# basics	C# is one of the languages included in .NET. 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 to write any type of .NET application using C#. You can also use the free, but powerful, Community product to create .NET applications in C#. This IDE works with solutions, which can consist of multiple projects.