Introducing C#

Welcome to the first chapter of the first section of this book. Over the course of this section we'll be taking a look at the basic knowledge required to get up and running. In this first chapter we'll be looking at an overview of C# and the .NET Framework, and we'll consider what these technologies are, the motivation behind using them, and how they relate to each other.

We'll start with a general discussion of the .NET Framework. This is a new technology, and contains many concepts that are tricky to get to grips with at first (mainly because the Framework introduces a "new way of doing things" to application development). This means that the discussion will, by necessity, cover many new concepts in a short space of time. However, a quick look at the basics is essential to understand how to program in C#, so this is a necessary evil. Later on in the book we'll revisit many of the topics covered here in more detail.

After this discussion, we'll move on to a simple description of C# itself, including its origins and similarities to C++.

Finally, we'll look at the main tool that will be used throughout this book: Visual Studio .NET (VS).

What is the .NET Framework?

The .NET Framework is a new and revolutionary platform created by Microsoft for developing applications.

The most interesting thing about this statement is how vague I've been – but there are good reasons for this. For a start, note that I didn't say "developing applications on the Windows operating system". Although the first release of the .NET Framework runs on the Windows operating system, future plans include versions that will work on others, such as FreeBSD, Linux, Macintosh, and even personal digital assistant (PDA) class devices. One of the key motivational forces behind this technology is its intention as a means of integrating disparate operating systems.

In addition, the definition of the .NET Framework given above includes no restriction on the type of applications that are possible. This is because there is no restriction – the .NET Framework allows the creation of Windows applications, web applications, web services, and pretty much anything else you can think of.

The .NET Framework has been designed such that it can be used from any language. This includes the subject of this book, C#, as well as C++, Visual Basic, JScript, and even older languages such as COBOL. In order for this to work, .NET-specific versions of these languages have also appeared: Managed C++, Visual Basic .NET, JScript .NET, J#, and so on – and more are being released all the time. Not only do all of these have access to the .NET Framework, they can also communicate with each other. It is perfectly possible for C# developers to make use of code written by Visual Basic .NET programmers, and vice versa.

All of this provides a hitherto unthinkable 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 we use from our client languages (such as C#) using object-oriented programming (OOP) techniques. This library is categorized into different modules – we use portions of it depending on the results we 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 sub-modules, such as a module for building web services within the module for web development.

The intention here is that different operating systems may support some or all of these modules, depending on their characteristics. A PDA, for example, would include support for all the core .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 framework also includes the .NET **Common Language Runtime** (**CLR**), which is responsible for maintaining the execution of all applications developed using the .NET library.

How do I Write Applications using the .NET Framework?

Writing an application using the .NET Framework means writing code (using any of the languages that support the framework) using the .NET code library. In this book we'll be using VS for our development, which is a powerful integrated development environment that supports C# (as well as managed and unmanaged C++, Visual Basic .NET, and some others). The advantage of this environment is the ease with which .NET features may be integrated into our code. The code that we will create will be entirely C#, but will use the .NET Framework throughout, and we'll make use of the additional tools in VS 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, however, this is a two-stage process.

MSIL and JIT

When we compile code that uses the .NET Framework library, we don't immediately create operating system-specific native code. Instead, we compile our code into **Microsoft Intermediate Language** (**MSIL**) code. This code isn't specific to any operating system, and isn't specific to C#. Other .NET languages – for example, Visual Basic .NET – also compile to this language as a first stage. This compilation step is carried out by VS when we use it to develop C# applications.

Obviously, in order to execute an application more work is necessary. This is the job of a **Just-In-Time** (**JIT**) compiler, which compiles MSIL 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 here reflects the fact that MSIL code is only compiled as and when it is needed.

In the past it has often been necessary to compile your code into several applications, each of which targets a specific operating system and CPU architecture. Often, this was a form of optimization (in order to get code to run faster on an AMD chipset, for example), but at times it was critical (for applications to work on both Win9x and WinNT/2000 environments, for example). This is now unnecessary, as JIT compilers (as their name suggests) use MSIL code, which is independent of the machine, operating system, and CPU. Several JIT compilers exist, each targeting a different architecture, and the appropriate one will be used to create the native code required.

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

Assemblies

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

As well as containing MSIL, assemblies also contain **meta** information (that is, information about the information contained in the assembly, also known as **metadata**) and optional **resources** (additional data used by the MSIL, such as sound files and pictures). The meta information allows assemblies to be fully self-descriptive. We need no other information in order to use an assembly, meaning that we avoid situations such as failing to add required data to the system registry and so on, which was often a problem when developing using other platforms.

This means that deploying applications is often as simple as copying the files into a directory on a remote computer. Since no additional information is required on the target systems, we can just run an executable file from this directory and (assuming the .NET CLR is installed) away we go.

Of course, we won't necessarily want to include everything required to run an application in one place. We might write some code that performs tasks required by multiple applications. In situations like this, it is often useful to place this reusable code in a place accessible to all applications. In the .NET Framework, this is the **Global Assembly Cache** (**GAC**). Placing code in this cache is simple – we just place the assembly containing the code in the directory containing this cache.

Managed Code

The role of the CLR doesn't end once we have compiled our code to MSIL and a JIT compiler has compiled this to native code. Code written using the .NET Framework is **managed** when it is executed (this stage is usually referred to as being at "**runtime**"). This means that the CLR looks after our 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, that, for example, access low-level functions of the operating system. However, in C# we can only write code that runs in a managed environment. We 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 in use. Prior to .NET this has mostly been 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. This usually meant a progressive slowdown of your computer followed by a system crash.

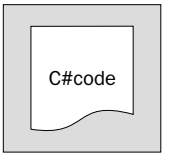
.NET garbage collection works by inspecting the memory of your computer every so often, and removing anything from it that is no longer needed. There is no set timeframe for this, it might happen thousands of times a second, once every few seconds, or whenever, but you can rest assured that it will happen.

There are some implications for programmers here. Since 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 waiting for garbage collection to happen, but this isn't anything like as tricky as it sounds.

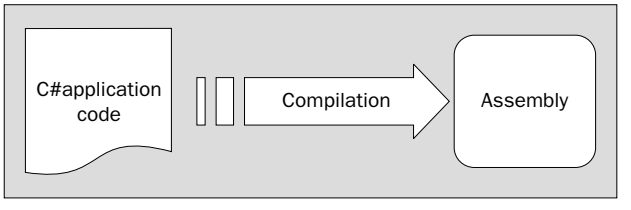
Fitting it Together

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

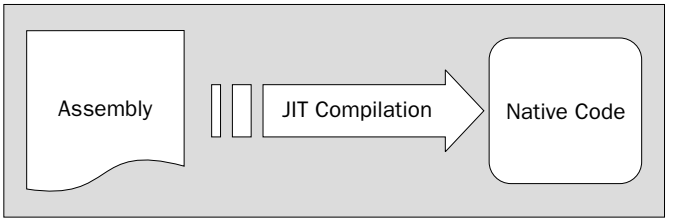
1. Application code is written using a .NET-compatible language such as C#:



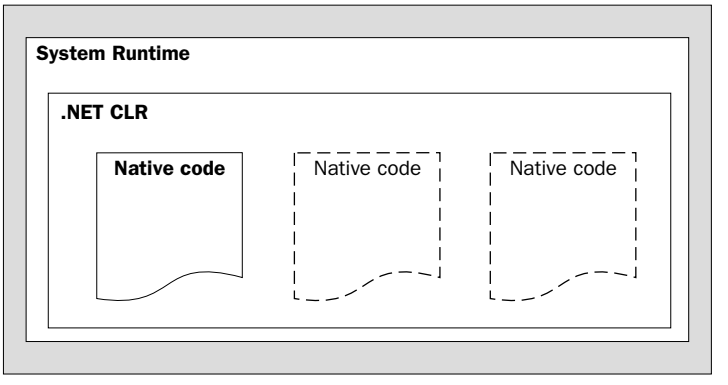
2. This code is compiled into MSIL, which is stored in an assembly:



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:



4. The native code is executed in the context of the managed CLR, along with any other running applications or processes:



Linking

There is one additional point to note concerning the above process. The C# code that compiles into MSIL in step 2 needn't 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 process is known as **linking**, and is extremely useful. The reason for this is that it is far easier to work with several smaller files that one enormous one. You can separate out 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 much easier to locate specific pieces of code when you need them, and enables teams of developers to divide up the programming burden into manageable chunks, where individuals can "check out" pieces of code to work on without risking damage to otherwise satisfactory sections, or sections that other people are working on.

What is C#?

C#, as mentioned above, is one of the languages that can be used 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. As it is a recent development, the C# language has been designed with hindsight, taking into account many of the best features from other languages while clearing up their problems.

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

At times, C# code is slightly more verbose than C++. This is a consequence of C# being a **type-safe** language (unlike C++). In layman's terms, this means that once some data has been assigned to a type, it cannot subsequently transform itself into another unrelated type. Consequently, there are strict rules that must be adhered to when converting between types, which means that we will often need to write more code to carry out the same task in C# as we might do in C++, but we get the benefit that code is more robust and debugging is simpler – .NET can always track what type a piece of data is at any time. In C# we therefore may 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 in my opinion it is certainly the best. It has the advantage of being the only language designed from the ground-up for the .NET Framework and may be the principal language used in versions of .NET that are ported to other operating systems. To keep languages such as Visual Basic .NET as similar as possible to their predecessors yet compliant with the CLR, certain features of the .NET code library are not fully supported. By contrast C# is able to make use of every feature that the .NET Framework code library has to offer.

What Kind of Applications Can I Write with C#?

The .NET Framework has no restrictions on the types of application possible, as we discussed above. C# uses the framework, and so also has no restrictions on possible applications. However, let's look at a few of the more common application types:

- **Windows Applications** – These are applications such as Microsoft Office, which have a familiar Windows look and feel about them. This is made simple using the **Windows Forms** module of the .NET Framework, which is a library of **controls** (such as buttons, toolbars, menus, and so on) that we can use to build a Windows user interface (UI).
- **Web Applications** – These are web pages such as might be viewed through any web browser. The .NET Framework includes a powerful system of generating web content dynamically, allowing personalization, security, and much more. This system is called **Active Server Pages.NET** (**ASP.NET**), and we can use C# to create ASP.NET applications using **Web Forms**.
- **Web Services** – These are a new and exciting way of creating versatile distributed applications. Using web services we can exchange virtually any data over the Internet, using the same simple syntax regardless of the language used to create a web service, or the system that it resides on.

Any of these types may also require some form of database access, which can be achieved using the **Active Data Objects.NET (ADO.NET)** section of the .NET Framework. 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 second and third sections of this book deal with the syntax and usage of the C# language without too much emphasis on the .NET Framework. This is necessary, as we won't be able to use the .NET Framework at all without a firm grounding in C# programming. We'll start off even simpler, in fact, and leave the more involved topic of Object-Oriented Programming (OOP) until we've covered the basics. These will be taught from first principles, assuming no programming knowledge at all.

Once we have done this, we will be ready to move on to developing the types of application listed in the last section. Section four of this book will look at Windows Forms programming, section five will look at some other .NET topics of interest (such as accessing databases), and section six will look at web application and web service programming. Finally, we'll have a look at some more involved case studies that make use of what we have learned in the earlier parts of the book.

Visual Studio .NET

In this book we'll use Visual Studio .NET (VS) for all of our C# development, from simple command line applications, to the more complex project types considered. VS isn't essential for developing C# applications, but it makes things much easier for us. We can (if we wish 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. However, why do this when we have the full power of VS to help us?

The following is a quick list of some of the features of VS that make it an appealing choice for .NET development:

- VS automates the steps required to compile source code, but at the same time gives us complete control over any options used should we wish to override them.
- The VS text editor is tailored to the languages VS supports (including C#), such that it can intelligently detect errors and suggest code where appropriate as we are typing.
- VS includes designers for Windows Forms and Web Forms applications, allowing simple drag-and-drop design of UI elements.
- Many of the types of project possible in C# may be created with "boilerplate" code already in place. Instead of starting from scratch, we will often find that various code files are started off for us, reducing the amount of time spent getting started on a project.
- VS includes several wizards that automate common tasks, many of which can add appropriate code to existing files without us having to worry about (or even, in some cases, remember) the correct syntax.
- VS contains many powerful tools for visualizing and navigating through elements of our projects, whether they are C# source code files, or other resources such as bitmap images or sound files.
- As well as simply writing applications in VS, it is possible to create deployment projects, making it easy to supply code to clients and for them to install it without much trouble.
- VS enables us to use advanced debugging techniques when developing projects, such as the ability to step through code one instruction at a time while keeping an eye on the state of our application.

There is much more than this, but hopefully you've got the idea!

Visual C# .NET Standard Edition

Visual C# .NET Standard Edition is a cut-down version of Visual Studio .NET Professional, and at a cut-down price too. While it offers many of the same features as Visual Studio .NET Professional, there are some notable feature absences, although not so many that they will prevent you from using the Standard Edition to work through this book.

Throughout the book, unless stated otherwise, the term "Visual Studio .NET" (or simply "VS") will refer to either version – Visual Studio .NET or the Visual C# .NET Standard Edition. There will be some occasions where we mean one version or the other in particular, and we shall mark these carefully, so that if you are an owner of the version not being discussed, you will not get confused!

VS Solutions

When we use VS to develop applications, we do so by creating **solutions**. A solution, in VS terms, is more than just an application. Solutions contain **projects**, which might be "Windows Forms projects", "Web Form projects", and so on. However, solutions can contain *multiple* projects, so that we can group together related code in one place, even if it will eventually compile to multiple assemblies in various places on our hard disk.

This is very useful, as it allows us to work on "shared" code (which might be placed in the Global Assembly Cache) at the same time as applications that use this code. Debugging code is a lot easier when only one development environment is used, as we can step through instructions in multiple code modules.

Summary

In this chapter we've looked at the .NET Framework in general terms, and discussed how it makes it easy for us to create powerful and versatile applications. We've seen what is necessary to turn code in languages such as C# into working applications, and what benefits we gain from using managed code running in the .NET Common Language Runtime.

We've also seen what C# actually is, and how it relates to the .NET Framework, and described the tool that we'll be using for C# development – Visual Studio .NET.

In the next chapter we'll get some C# code running using VS, which will give us enough knowledge to sit back and concentrate on the C# language itself, rather than worrying too much about how VS works.