PART I
The C# Language

1 .NET Application Architectures

WHAT'S IN THIS CHAPTER?

- Reviewing the history of .NET
- Understanding differences between .NET Framework 4.6 and .NET Core 1.0
- Assemblies and NuGet Packages
- The Common Language Runtime
- Features of the Windows Runtime
- Programming Hello, World!
- Universal Windows Platform
- Technologies for creating Windows Apps
- Technologies for creating Web Apps

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/professionalcsharp6` on the Download Code tab. The code for this chapter is divided into the following major examples:

- DotnetHelloWorld
- HelloWorldApp (.NET Core)

CHOOSING YOUR TECHNOLOGIES

In recent years, .NET has become a huge ecosystem for creating any kind of applications on the Windows platform. With .NET you can create Windows apps, web services, web applications, and apps for the Microsoft Phone.

The newest release of .NET is a big change from the last version—maybe the biggest change to .NET since its invention. Much of the .NET code has become open-source code, and you can create applications for other platforms as well. The new version of .NET (.NET Core) and NuGet packages allow Microsoft to provide faster update cycles for delivering new features. It's not easy to decide what technology should be used for creating applications. This chapter helps you with that. It gives you information about the different technologies available for creating Windows and web applications and services, offers guidance on what to choose for database access, and highlights the differences between .NET and .NET Core.

REVIEWING .NET HISTORY

To better understand what is available with .NET and C#, it is best to know something about its history. The following table shows the version of .NET in relation to the Common Language Runtime (CLR), the version of C#, and the Visual Studio edition that gives some idea about the year when the corresponding versions have been released. Besides knowing what technology to use, it's also good to know what technology is not recommended because there's a replacement.

.NET	CLR	C#	VISUAL STUDIO
1.0	1.0	1.0	2002
1.1	1.1	1.2	2003
2.0	2.0	2.0	2005
3.0	2.0	2.0	2005 + Extensions
3.5	2.0	3.0	2008
4.0	4.0	4.0	2010
4.5	4.0	5.0	2012
4.5.1	4.0	5.0	2013
4.6	4.0	6	2015
.NET Core 1.0	CoreCLR	6	2015 + Extensions

The following sections cover the details of this table and the progress of C# and .NET.

C# 1.0—A New Language

C# 1.0 was a completely new programming language designed for the .NET Framework. At the time it was developed, the .NET Framework consisted of about 3,000 classes and the CLR.

After Microsoft was not allowed by a court order (filed by Sun, the company that created Java) to make changes to the Java code, Anders Hejlsberg designed C#. Before working for Microsoft, Hejlsberg had his roots at Borland where he designed the Delphi programming language (an Object Pascal dialect). At Microsoft he was responsible for J++ (Microsoft's version of the Java programming language). Given Hejlsberg's background, the C# programming language was mainly influenced by C++, Java, and Pascal.

Because C# was created later than Java and C++, Microsoft analyzed typical programming errors that happened with the other languages, and did some things differently to avoid these errors. Some differences include the following:

- With `if` statements, Boolean expressions are required (C++ allows an integer value here as well).
- It's permissible to create value and reference types using the `struct` and `class` keywords (Java only allows creating custom reference types; with C++ the distinction between `struct` and `class` is only the default for the access modifier).
- Virtual and non-virtual methods are allowed (this is similar to C++; Java always creates virtual methods).

Of course there are a lot more changes as you'll see reading this book.

At this time, C# was a pure object-oriented programming language with features for inheritance, encapsulation, and polymorphism. C# also offered component-based programming enhancements such as delegates and events.

Before the existence of .NET with the CLR, every programming language had its own runtime. With C++, the C++ Runtime is linked with every C++ program. Visual Basic 6 had its own runtime with VBRun. The runtime of Java is the Java Virtual Machine—which can be compared to the CLR. The CLR is a runtime that is used by every .NET programming language. At the time the CLR appeared on the scene, Microsoft offered JScript.NET, Visual Basic .NET, and Managed C++ in addition to C#. JScript.NET was Microsoft's JavaScript compiler that was to be used with the CLR and .NET classes. Visual Basic.NET was the name for

Visual Basic that offered .NET support. Nowadays it's just called Visual Basic again. Managed C++ was the name for a language that mixed native C++ code with Managed .NET Code. The newer C++ language used today with .NET is C++/CLR.

A compiler for a .NET programming language generates *Intermediate Language* (IL) code. The IL code looks like object-oriented machine code and can be checked by using the tool ildasm.exe to open DLL or EXE files that contain .NET code. The CLR contains a just-in-time (JIT) compiler that generates native code out of the IL code when the program starts to run.

> **NOTE** *IL code is also known as* managed code.

Other parts of the CLR are a garbage collector (GC), which is responsible for cleaning up managed memory that is no longer referenced; a security mechanism that uses code access security to verify what code is allowed to do; an extension for the debugger to allow a debug session between different programming languages (for example, starting a debug session with Visual Basic and continuing to debug within a C# library); and a threading facility that is responsible for creating threads on the underlying platform.

The .NET Framework was already huge with version 1. The classes are organized within namespaces to help facilitate navigating the 3,000 available classes. Namespaces are used to group classes and to solve conflicts by allowing the same class name in different namespaces. Version 1 of the .NET Framework allowed creating Windows desktop applications using Windows Forms (namespace `System.Windows.Forms`), creating web applications with ASP.NET Web Forms (`System.Web`), communicating with applications and web services using ASP.NET Web Services, communicating more quickly between .NET applications using .NET Remoting, and creating COM+ components for running in an application server using Enterprise Services.

ASP.NET Web Forms was the technology for creating web applications with the goal for the developer to not need to know something about HTML and JavaScript. Server-side controls that worked similarly to Windows Forms itself created HTML and JavaScript.

C# 1.2 and .NET 1.1 was mainly a bug fix release with minor enhancements.

> **NOTE** *Inheritance is discussed in Chapter 4, "Inheritance"; delegates and events are covered in Chapter 9, "Delegates, Lambdas, and Events."*

> **NOTE** *Every new release of .NET has been accompanied by a new version of the book* Professional C#. *With .NET 1.0, the book was already in the second edition as the first edition had been published with Beta 2 of .NET 1.0. You're holding the 10th edition of this book in your hands.*

C# 2 and .NET 2 with Generics

C# 2 and .NET 2 was a huge update. With this version, a change to both the C# programming language and the IL code had been made; that's why a new CLR was needed to support the IL code additions. One big change was *generics*. Generics make it possible to create types without needing to know what inner types are used. The inner types used are defined at instantiation time, when an instance is created.

This advance in the C# programming language also resulted in many new types in the Framework—for example, new generic collection classes found in the namespace `System.Collections.Generic`. With this, the older collection classes defined with 1.0 are rarely used with newer applications. Of course, the older classes still work nowadays, even with the new .NET Core version.

> **NOTE** *Generics are used all through the book, but they're explained in detail in Chapter 6, "Generics." Chapter 11, "Collections," covers generic collection classes.*

.NET 3—Windows Presentation Foundation

With the release of .NET 3.0 no new version of C# was needed. 3.0 was only a release offering new libraries, but it was a huge release with many new types and namespaces. Windows Presentation Foundation (WPF) was probably the biggest part of the new Framework for creating Windows desktop applications. Windows Forms wrapped the native Windows controls and was based on pixels, whereas WPF was based on DirectX to draw every control on its own. The vector graphics in WPF allow seamless resizing of every form. The templates in WPF also allow for complete custom looks. For example, an application for the Zurich airport can include a button that looks like a plane. As a result, applications can look very different from the traditional Windows applications that had been developed up to that time. Everything below the namespace `System.Windows` belongs to WPF, with the exception of `System.Windows.Forms`. With WPF the user interface can be designed using an XML syntax: XML for Applications Markup Language (XAML).

Before .NET 3, ASP.NET Web Services and .NET Remoting were used for communicating between applications. Message Queuing was another option for communicating. The various technologies had different advantages and disadvantages, and all had different APIs for programming. A typical enterprise application had to use more than one communication API, and thus it was necessary to learn several of them. This was solved with Windows Communication Foundation (WCF). WCF combined all the options of the other APIs into the one API. However, to support all of the features WCF has to offer, you need to configure WCF.

The third big part of the .NET 3.0 release was Windows Workflow Foundation (WF) with the namespace `System.Workflow`. Instead of creating custom workflow engines for several different applications (and Microsoft itself created several workflow engines for different products), a workflow engine was available as part of .NET.

With .NET 3.0, the class count of the Framework increased from 8,000 types in .NET 2.0 to about 12,000 types.

> **NOTE** *In this book, WPF is covered in Chapters 29, 30, 31, 34, 35, and 36. You can read information about WCF in Chapter 44, "Windows Communication Foundation."*

C# 3 and .NET 3.5—LINQ

.NET 3.5 came together with a new release of C# 3. The major enhancement was a query syntax defined with C# that allows using the same syntax to filter and sort object lists, XML files, and the database. The language enhancements didn't require any change to the IL code as the C# features used here are just syntax sugar. All of the enhancements could have been done with the older syntax as well, just a lot more code would be necessary. The C# language makes it really easy to do these queries. With LINQ and lambda expressions, it's possible to use the same query syntax and access object collections, databases, and XML files.

For accessing the database and creating LINQ queries, LINQ to SQL was released as part of .NET 3.5. With the first update to .NET 3.5, the first version of Entity Framework was released. Both LINQ to SQL and Entity Framework offered mapping of hierarchies to the relations of a database and a LINQ provider. Entity Framework was more powerful, but LINQ to SQL was simpler. Over time, features of LINQ to SQL have been implemented in Entity Framework, and now this one is here to stay. (Nowadays it looks very different from the first version released.)

Another technology introduced as part of .NET 3.5 was the `System.AddIn` namespace, which offers an add-in model. This model offers powerful features that run add-ins even out of process, but it is also complex to use.

NOTE *LINQ is covered in detail in Chapter 13, "Language Integrated Query." The newest version of the Entity Framework is very different from the .NET 3.5 release; it's described in Chapter 38, "Entity Framework Core."*

C# 4 and .NET 4—Dynamic and TPL

The theme of C# 4 was dynamic—integrating scripting languages and making it easier to use COM integration. C# syntax has been extended with the `dynamic` keyword, named and optional parameters, and enhancements to co- and contra-variance with generics.

Other enhancements have been made within the .NET Framework. With multi-core CPUs, parallel programming had become more and more important. The Task Parallel Library (TPL), with abstractions of threads using `Task` and `Parallel` classes, make it easier to create parallel running code.

Because the workflow engine created with .NET 3.0 didn't fulfill its promises, a completely new Windows Workflow Foundation was part of .NET 4.0. To avoid conflicts with the older workflow engine, the newer one is defined in the `System.Activity` namespace.

The enhancements of C# 4 also required a new version of the runtime. The runtime skipped from version 2 to 4.

With the release of Visual Studio 2010, a new technology shipped for creating web applications: ASP.NET MVC 2.0. Unlike ASP.NET Web Forms, this technology required programming HTML and JavaScript, and it used C# and .NET with server-side functionality. As this technology was very new as well as being out of band (OOB) to Visual Studio and .NET, ASP.NET MVC was updated regularly.

NOTE *The dynamic keyword of C# 4 is covered in Chapter 16, "Reflection, Metadata, and Dynamic Programming." The Task Parallel Library is covered in Chapter 21, "Tasks and Parallel Programming."*

Version 5 of ASP.NET and Version 6 of ASP.NET MVC are covered in Chapter 40, "ASP.NET Core," and Chapter 41, "ASP.NET MVC."

C# 5 and Asynchronous Programming

C# 5 had only two new keywords: `async` and `await`. However, they made programming of asynchronous methods a lot easier. As touch became more significant with Windows 8, it also became a lot more important to not block the UI thread. Using the mouse, users are accustomed to scrolling taking some time. However, using fingers on a touch interface that is not responsive is really annoying.

Windows 8 also introduced a new programming interface for Windows Store apps (also known as Modern apps, Metro apps, Universal Windows apps, and, more recently, Windows apps): the Windows Runtime. This is a native runtime that looks like .NET by using language projections. Many of the WPF controls have been redone for the new runtime, and a subset of the .NET Framework can be used with such apps.

As the `System.AddIn` framework was much too complex and slow, a new composition framework was created with .NET 4.5: Managed Extensibility Framework with the namespace `System.Composition`.

A new version of platform-independent communication is offered by the ASP.NET Web API. Unlike WCF, which offers stateful and stateless services as well as many different network protocols, the ASP.NET Web API is a lot simpler and based on the Representational State Transfer (REST) software architecture style.

NOTE *The* `async` *and* `await` *keywords of C# 5 are discussed in detail in Chapter 15, "Asynchronous Programming." This chapter also shows the different asynchronous patterns that have been used over time with .NET.*

Managed Extensibility Framework (MEF) is covered in Chapter 26, "Composition." Windows apps are covered in Chapters 29 to 33, and the ASP.NET Web API is covered in Chapter 42, "ASP.NET Web API."

C# 6 and .NET Core

C# 6 doesn't involve the huge improvements that were made by generics, LINQ, and async, but there are a lot of small and practical enhancements in the language that can reduce the code length in several places. The many improvements have been made possible by a new compiler engine code named Roslyn.

NOTE *Roslyn is covered in Chapter 18, ".NET Compiler Platform."*

The full .NET Framework is not the only .NET Framework that was in use in recent years. Some scenarios required smaller frameworks. In 2007, the first version of Microsoft Silverlight was released (code named WPF/E, WPF Everywhere). Silverlight was a web browser plug-in that allowed dynamic content. The first version of Silverlight supported programming only via JavaScript. The second version included a subset of the .NET Framework. Of course, server-side libraries were not needed because Silverlight was always running on the client, but the Framework shipped with Silverlight also removed classes and methods from the core features to make it lightweight and portable to other platforms. The last version of Silverlight for the desktop (version 5) was released in December 2011. Silverlight had also been used for programming for the Windows Phone. Silverlight 8.1 made it into Windows Phone 8.1, but this version of Silverlight is also different from the version on the desktop.

On the Windows desktop, where there is such a huge framework with .NET and the need for faster and faster development cadences, big changes were also required. In a world of DevOps where developers and operations work together or are even the same people to bring applications and new features continuously to the user, there's a need to have new features available in a fast way. Creating new features or making bug fixes is a not-so-easy task with a huge framework and many dependencies.

With several smaller .NET Frameworks available (e.g. Silverlight, Silverlight for the Windows Phone), it became important to share code between the desktop version of .NET and a smaller version. A technology to share code between different .NET versions is the portable library. Over time, with many different .NET Frameworks and versions, the management of the portable library has become a nightmare.

With all these issues, a new version of .NET is a necessity. (Yes, it's really a requirement to solve these issues.) The new version of the Framework is invented with the name *.NET Core*. .NET Core is smaller with modular NuGet packages, has a runtime that's distributed with every application, is open source, and is available not only for the desktop version of Windows but also for many different Windows devices, as well as for Linux and OS X.

For creating web applications, ASP.NET Core 1.0 is a complete rewrite of ASP.NET. This release is not completely backward compatible to older versions and requires some changes to existing ASP.NET MVC code (with ASP.NET MVC 6). However, it also has a lot of advantages when compared with the older versions, such as a lower overhead with every network request—which results in better performance—and it can also run on Linux. ASP.NET Web Forms is not part of this release because ASP.NET Web Forms was not designed for best performance; it was designed for developer friendliness based on patterns known by Windows Forms application developers.

Of course, not all applications can be changed easily to make use of .NET Core. That's why the huge framework received improvements as well—even if those improvements are not completed in as fast a pace as .NET Core. The new version of the full .NET Framework is 4.6. Small updates for ASP.NET Web Forms are available on the full .NET stack.

NOTE *Roslyn is covered in Chapter 18. The changes to the C# language are covered in all the language chapters in Part I—for example, read-only properties are in Chapter 3, "Objects and Types"; the* `nameof` *operator and null propagation are in Chapter 8, "Operators and Casts"; string interpolation is in Chapter 10, "Strings and Regular Expressions"; and exception filters are in Chapter 14, "Errors and Exceptions."*

Where possible, .NET Core is used in this book. You can read more information about .NET Core and NuGet packages later in this chapter.

Choosing Technologies and Going Forward

When you know the reason for competing technologies within the Framework, it's easier to select a technology to use for programming applications. For example, if you're creating new Windows applications it's not a good idea to bet on Windows Forms. Instead, you should use an XAML-based technology, such as Windows apps or Windows desktop applications using WPF.

If you're creating web applications, a safe bet is to use ASP.NET Core with ASP.NET MVC 6. Making this choice rules out using ASP.NET Web Forms. If you're accessing a database, you should use Entity Framework rather than LINQ to SQL, and you should opt for the Managed Extensibility Framework instead of `System.AddIn`.

Legacy applications still use Windows Forms and ASP.NET Web Forms and some other older technologies. It doesn't make sense to change existing applications just to use new technologies. There must be a huge advantage to making the change—for example, when maintenance of the code is already a nightmare and a lot of refactoring is needed to change to faster release cycles that are being demanded by customers, or when using a new technology allows for reducing the coding time for updates. Depending on the type of legacy application, it might not be worthwhile to switch to a new technology. You can allow the application to still be based on older technologies because Windows Forms and ASP.NET Web Forms will still be supported for many years to come.

The content of this book is based on the newer technologies to show what's best for creating new applications. In case you still need to maintain legacy applications, you can refer to older editions of this book, which cover ASP.NET Web Forms, Windows Forms, `System.AddIn`, and other legacy technologies that are still part of and available with the .NET Framework.

.NET 2015

.NET 2015 is an umbrella term for all the .NET technologies. Figure 1-1 gives an overall picture of these technologies. The left side represents the .NET Framework 4.6 technologies such as WPF and ASP.NET 4. ASP.NET Core 1.0 can run on .NET Framework 4.6 as well, as you can see in this figure. The right side represents the new .NET Core technologies. Both ASP.NET Core 1.0 and the Universal Windows Platform (UWP) run on .NET Core. You can also create console applications that run on .NET Core.

A part of .NET Core is a new runtime: the CoreCLR. This runtime is used from ASP.NET Core 1.0. Instead of using the CoreCLR runtime, .NET can also be compiled to native code. The UWP automatically makes use of this feature; these .NET applications are compiled to native code before being offered from the Windows Store. You can also compile other .NET Core applications—and the applications running on Linux—to native code.

In the lower part of Figure 1-1, you can see there's also some sharing going on between .NET Framework 4.6 and .NET Core. *Runtime components*, such as the code for the garbage collector and the RyuJIT (this is a new JIT compiler to compile IL code to native code) are shared. The garbage collector is used by CLR, CoreCLR, and .NET Native. The RyuJIT just-in-time compiler is used by CLR and CoreCLR. Libraries can be shared between applications based on the .NET Framework 4.6 and .NET Core 1.0. The concept of NuGet packages helps put these libraries in a common package that is available on all .NET platforms. And, of course, the new .NET compiler platform is used by all these technologies.

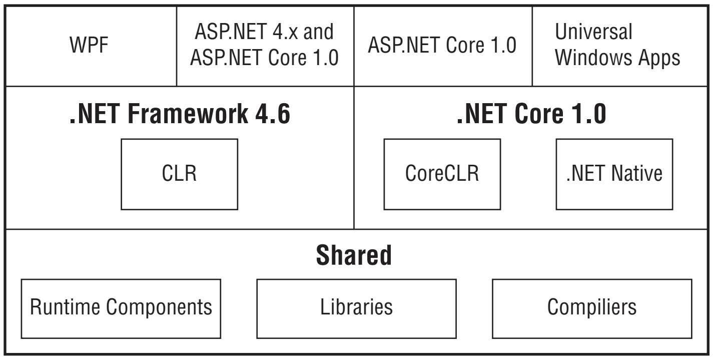


FIGURE 1-1

.NET Framework 4.6

NET Framework 4.6 is the .NET Framework that has been continuously enhanced in the past 10 years. Many of the technologies that have been discussed in the history section are based on this framework. This framework is used for creating Windows Forms and WPF applications. Also, although ASP.NET 5 can run on .NET Core, it can also run on .NET Framework 4.6.

If you want to continue working with ASP.NET Web Forms, ASP.NET 4.6 with .NET Framework 4.6 is the way to go. ASP.NET 4.6 also has new features compared to version 4.5, such as support for HTTP2 (a new version of the HTTP protocol that is discussed in Chapter 25, "Networking"), compilation on the fly with the Roslyn compiler, and asynchronous model binding. However, you can't switch to .NET Core with ASP.NET Web Forms.

You can find the libraries of the framework as well as the CLR in the directory `%windows%\Microsoft .NET\Framework\v4.0.30319`.

The classes available with the .NET Framework are organized in namespaces starting with the name `System`. The following table describes a few of the namespaces to give you an idea about the hierarchy.

NAMESPACE	DESCRIPTION
`System.Collections`	This is the root namespace for collections. Collections are also found within sub-namespaces such as `System.Collections.Concurrent` and `System.Collections.Generic`.
`System.Data`	This is the namespace for accessing databases. `System.Data.SqlClient` contains classes to access the SQL Server,
`System.Diagnostics`	This is the root namespace for diagnostics information, such as event logging and tracing (in the namespace `System.Diagnostics.Tracing`).
`System.Globalization`	This is the namespace that contains classes for globalization and localization of applications.
`System.IO`	This is the namespace for File IO, which are classes to access files and directories. Readers, writers, and streams are here.
`System.Net`	This is the namespace for core networking, such as accessing DNS servers and creating sockets with `System.Net.Sockets`.
`System.Threading`	This is the root namespace for threads and tasks. Tasks are defined within `System.Threading.Tasks`.
`System.Web`	This is the root namespace for ASP.NET. Below this namespace, many sub-namespaces are defined, such as `System.Web.UI`, `System.Web.UI.WebControls`, and `System.Web.Hosting`.
`System.Windows`	This is the root namespace for Windows desktop applications with WPF. Example subnamespaces are `System.Windows.Shapes`, `System.Windows.Data`, and `System.Windows.Documents`.

NOTE *Some of the new .NET classes use namespaces that start with the name* `Microsoft` *instead of* `System`*, like* `Microsoft.Data.Entity` *for the Entity Framework and* `Microsoft.Extensions.DependencyInjection` *for the new dependency injection framework.*

.NET Core 1.0

.NET Core 1.0 is the new .NET that is used by all new technologies and has a big focus in this book. This framework is *open source*—you can find it at `http://www.github.com/dotnet`. The runtime is the *CoreCLR* repository; the framework containing collection classes, file system access, console, XML, and a lot more is in the *CoreFX* repository.

Unlike the .NET Framework, where the specific version you needed for the application had to be installed on the system, with .NET Core 1.0 the framework, including the runtime, is delivered with the application. Previously there were times when you might have had problems deploying an ASP.NET web application to a shared server because the provider had older versions of .NET installed; those times are gone. Now you can deliver the runtime with the application and are not dependent on the version installed on the server.

.NET Core 1.0 is designed in a modular approach. The framework splits up into a large list of NuGet packages. With the application you decide what packages you need. The .NET Framework was growing larger and larger when new functionality was added. It was not possible to remove old functionality that's no longer needed, such as the old collection classes that are unnecessary because of the generic collection classes that were added, .NET Remoting that has been replaced by the new communication technology, or LINQ to SQL that has been updated to Entity Framework. Applications can break when something is removed. This does not apply to .NET Core, as the application distributes the parts of the framework that it needs.

The framework of .NET Core is currently as huge as .NET Framework 4.6 is. However, this can change, and it can grow even bigger, but because of the modularity that growth potential is not an issue. .NET Core is already so huge that we can't cover every type in this book. Just have a look at `http://www.github.com/dotnet/corefx` to see all the sources. For example, old nongeneric collection classes are already covered with .NET Core to make it easier to bring legacy code to the new platform.

.NET Core can be updated at a fast pace. Even updating the runtime doesn't influence existing applications because the runtime is installed with the applications. Now Microsoft can improve .NET Core, including the runtime, with faster release cycles.

NOTE *For developing apps using .NET Core, Microsoft created new command-line utilities named .NET Core Command line (CLI). These tools are introduced later in this chapter through a "Hello, World!" application in the section "Compiling with CLI."*

Assemblies

Libraries and executables of .NET programs are known by the term *assembly*. An assembly is the logical unit that contains compiled IL code targeted at the .NET Framework.

An assembly is completely self-describing and is a logical rather than a physical unit, which means that it can be stored across more than one file. (Indeed, dynamic assemblies are stored in memory, not on file.) If an assembly is stored in more than one file, there will be one main file that contains the entry point and describes the other files in the assembly.

The same assembly structure is used for both executable code and library code. The only difference is that an executable assembly contains a main program entry point, whereas a library assembly does not.

An important characteristic of assemblies is that they contain metadata that describes the types and methods defined in the corresponding code. An assembly, however, also contains assembly metadata that describes the assembly. This assembly metadata, contained in an area known as the *manifest*, enables checks to be made on the version of the assembly and on its integrity.

Because an assembly contains program metadata, applications or other assemblies that call up code in a given assembly do not need to refer to the registry, or to any other data source, to find out how to use that assembly.

With the .NET Framework 4.6, assemblies come in two types: *private* and *shared* assemblies. Shared assemblies don't apply to the Universal Windows Platform because all the code is compiled to one native image.

Private Assemblies

Private assemblies normally ship with software and are intended to be used only with that software. The usual scenario in which you ship private assemblies is when you supply an application in the form of an executable and a number of libraries, where the libraries contain code that should be used only with that application.

The system guarantees that private assemblies will not be used by other software because an application may load only private assemblies located in the same folder that the main executable is loaded in, or in a subfolder of it.

Because you would normally expect that commercial software would always be installed in its own directory, there is no risk of one software package overwriting, modifying, or accidentally loading private assemblies intended for another package. And, because private assemblies can be used only by the software package that they are intended for, you have much more control over what software uses them. There is, therefore, less need to take security precautions because there is no risk, for example, of some other commercial software overwriting one of your assemblies with some new version of it (apart from software designed specifically to perform malicious damage). There are also no problems with name collisions. If classes in your private assembly happen to have the same name as classes in someone else's private assembly, that does not matter because any given application can see only the one set of private assemblies.

Because a private assembly is entirely self-contained, the process to deploy it is simple. You simply place the appropriate file(s) in the appropriate folder in the file system. (No registry entries need to be made.) This process is known as *zero impact (xcopy) installation*.

Shared Assemblies

Shared assemblies are intended to be common libraries that any other application can use. Because any other software can access a shared assembly, more precautions need to be taken against the following risks:

- Name collisions, where another company's shared assembly implements types that have the same names as those in your shared assembly. Because client code can theoretically have access to both assemblies simultaneously, this could be a serious problem.
- The risk of an assembly being overwritten by a different version of the same assembly; the new version is incompatible with some existing client code.

The solution to these problems is placing shared assemblies in a special directory subtree in the file system, known as the *global assembly cache* (*GAC*). With private assemblies, this can be done by simply copying the assembly into the appropriate folder, but with shared assemblies it must be specifically installed into the cache. This process can be performed by a number of .NET utilities and requires certain checks on the assembly, as well as setting up of a small folder hierarchy within the assembly cache used to ensure assembly integrity.

To prevent name collisions, shared assemblies are given a name based on private key cryptography. (Private assemblies are simply given the same name as their main filename.) This name is known as a *strong name*; it is guaranteed to be unique and must be quoted by applications that reference a shared assembly.

Problems associated with the risk of overwriting an assembly are addressed by specifying version information in the assembly manifest and by allowing side-by-side installations.

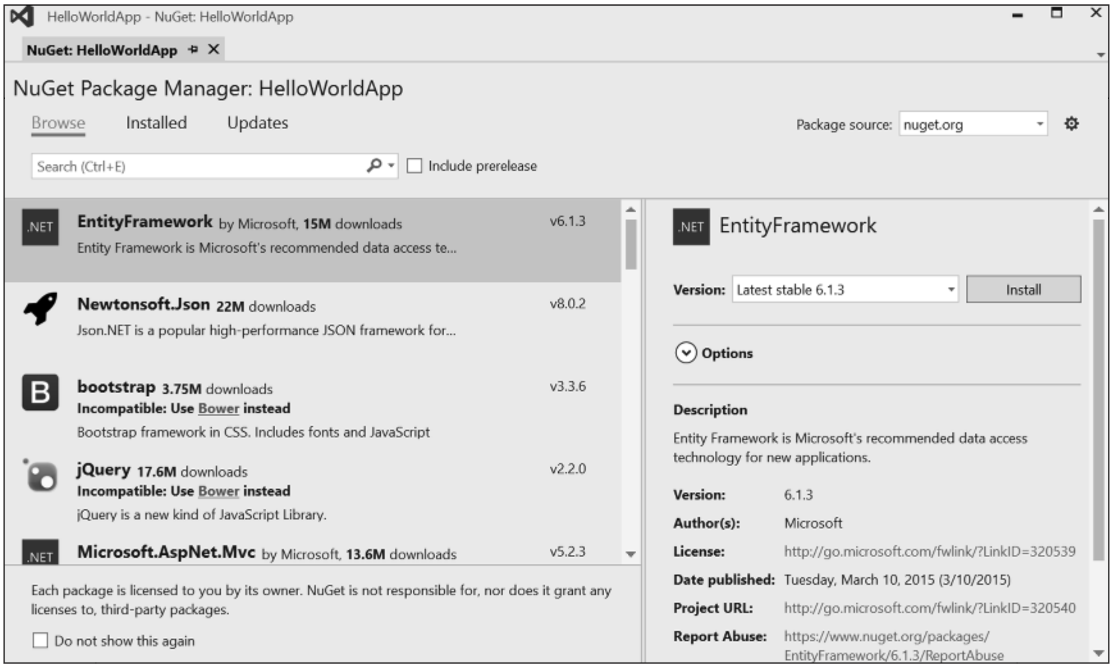
NuGet Packages

In the early days, assemblies were reusable units with applications. That use is still possible (and necessary with some assemblies) when you're adding a reference to an assembly for using the public types and methods from your own code. However, using libraries can mean a lot more than just adding a reference and using it. Using libraries can also mean some configuration changes, or scripts that can be used to take advantage of some features. This is one of the reasons to package assemblies within NuGet packages.

A NuGet package is a zip file that contains the assembly (or multiple assemblies) as well as configuration information and PowerShell scripts.

Another reason for using NuGet packages is that they can be found easily; they're available not only from Microsoft but also from third parties. NuGet packages are easily accessible on the NuGet server at `http://www.nuget.org`.

From the references within a Visual Studio project, you can open the NuGet Package Manager (see Figure 1-2). There you can search for packages and add them to the application. This tool enables you to search for packages that are not yet released (include prerelease option) and define the NuGet server where the packages should be searched.



HelloWorldApp - NuGet: HelloWorldApp

NuGet: HelloWorldApp

NuGet Package Manager: HelloWorldApp

Browse Installed Updates

Package source: nuget.org

Search (Ctrl+E)

Include prerelease

EntityFramework by Microsoft, **15M** downloads v6.1.3
Entity Framework is Microsoft's recommended data access te...

Newtonsoft.Json **22M** downloads v8.0.2
Json.NET is a popular high-performance JSON framework for...

bootstrap **3.75M** downloads v3.3.6
Incompatible: Use Bower instead
Bootstrap framework in CSS. Includes fonts and JavaScript

jQuery **17.6M** downloads v2.2.0
Incompatible: Use Bower instead
jQuery is a new kind of JavaScript Library.

Microsoft.AspNet.Mvc by Microsoft, **13.6M** downloads v5.2.3

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

Do not show this again

EntityFramework

Version: Latest stable 6.1.3 Install

Options

Description

Entity Framework is Microsoft's recommended data access technology for new applications.

Version:	6.1.3
Author(s):	Microsoft
License:	http://go.microsoft.com/fwlink/?LinkID=320539
Date published:	Tuesday, March 10, 2015 (3/10/2015)
Project URL:	http://go.microsoft.com/fwlink/?LinkID=320540
Report Abuse:	https://www.nuget.org/packages/EntityFramework/6.1.3/ReportAbuse

FIGURE 1-2

> **NOTE** *When you use third-party packages from the NuGet server, you're always at risk if a package is available at a later time. You also need to check about the support availability of the package. Always check for project links with information about the package before using it. With the package source, you can select Microsoft and .NET to only get packages supported by Microsoft. Third-party packages are also included in the Microsoft and .NET section, but they are third-party packages that are supported by Microsoft.*
>
> *You can also use your own NuGet server with your development team. You can define to only allow packages from your own server to be used by the development team.*

Because .NET Core is so modular, all applications—other than the simplest ones—need additional NuGet packages. To make it easier for you to find the package, with every sample application that's built with .NET Core this book shows a table that lists packages and namespaces that need to be added.

> **NOTE** *More information about the NuGet Package Manager is covered in Chapter 17, "Visual Studio 2015."*

Common Language Runtime

The Universal Windows Platform makes use of Native .NET to compile IL to native code. With all other scenarios, with both applications using the .NET Framework 4.6 and applications using .NET Core 1.0, a *Common Language Runtime* (CLR) is needed. However, .NET Core uses the CoreCLR whereas the .NET Framework uses the CLR. So, what's done by a CLR?

Before an application can be executed by the CLR, any source code that you develop (in C# or some other language) needs to be compiled. Compilation occurs in two steps in .NET:

1. Compilation of source code to Microsoft Intermediate Language (IL)
2. Compilation of IL to platform-specific native code by the CLR

The IL code is available within a .NET assembly. During runtime, a Just-In-Time (JIT) compiler compiles IL code and creates the platform-specific native code.

The new CLR and the CoreCLR include a new JIT compiler named *RyuJIT*. The new JIT compiler is not only faster than the previous one; it also has better support for the Edit & Continue feature while debugging with Visual Studio. The Edit & Continue feature enables you to edit the code while debugging, and you can continue the debug session without the need to stop and restart the process.

The runtime also includes a type system with a type loader that is responsible for loading types from assemblies. Security infrastructure with the type system verifies whether certain type system structures are permitted—for example, with inheritance.

After creating instances of types, the instances also need to be destroyed and memory needs to be recycled. Another feature of the runtime is the garbage collector. The garbage collector cleans up memory from the managed heap that isn't referenced anymore. Chapter 5, "Managed and Unmanaged Resources," explains how this is done and when it happens.

The runtime is also responsible for threading. Creating a managed thread from C# is not necessarily a thread from the underlying operating system. Threads are virtualized and managed by the runtime.

NOTE *How threads can be created and managed from C# is covered in Chapter 21, "Tasks and Parallel Programming," and in Chapter 22, "Task Synchronization."*

.NET Native

A new feature of .NET 2015 is to compile a managed program to native code, *.NET Native*. With Windows apps this generates optimized code that can have a startup time that's up to 60 percent faster and uses 15 to 20 percent less memory.

.NET Native started with compiling UWP apps to native code for apps deployed to the Windows Store. .NET Native will also be available with other .NET Core applications. However, this feature did not make it into version 1 of .NET Core and will be available with a future version. You can compile .NET Core applications running on both Windows and Linux to native code. Of course, you need different native images on each of these platforms. Behind the scenes, .NET Native shares the C++ optimizer for generating the native code.

Windows Runtime

Starting with Windows 8, the Windows operating system offers another framework: the Windows Runtime. This runtime is used by the Windows Universal Platform and was version 1 with Windows 8, version 2 with Windows 8.1, and version 3 with Windows 10.

Unlike the .NET Framework, this framework was created using native code. When it's used with .NET applications, the types and methods contained just look like .NET. With the help of language projection, the Windows Runtime can be used with the JavaScript, C++, and .NET languages, and it looks like it's native to the programming environment. Methods are not only behaving differently in regard to case sensitivity; the methods and types can also have different names depending on where they are used.

The Windows Runtime offers an object hierarchy organized in namespaces that start with `Windows`. Looking at these classes, there's not a lot with duplicate functionality to the .NET Framework; instead, extra functionality is offered that is available for apps running on the Universal Windows Platform.

NAMESPACE	DESCRIPTION
`Windows.ApplicationModel`	This namespace and its subnamespaces, such as `Windows.ApplicationModel.Contracts`, define classes to manage the app lifecycle and communication with other apps.
`Windows.Data`	`Windows.Data` defines subnamespaces to work with Text, JSON, PDF, and XML data.
`Windows.Devices`	Geolocation, smartcards, point of service devices, printers, scanners, and other devices can be accessed with subnamespaces of `Windows.Devices`.
`Windows.Foundation`	`Windows.Foundation` defines core functionality. Interfaces for collections are defined with the namespace `Windows.Foundation.Collections`. You will not find concrete collection classes here. Instead, interfaces of .NET collection types map to the Windows Runtime types.
`Windows.Media`	`Windows.Media` is the root namespace for playing and capturing video and audio, accessing playlists, and doing speech output.
`Windows.Networking`	This is the root namespace for socket programming, background transfer of data, and push notifications.
`Windows.Security`	Classes from `Windows.Security.Credentials` offer a safe store for passwords; `Windows.Security.Credentials.UI` offers a picker to get credentials from the user.
`Windows.Services.Maps`	This namespace contains classes for location services and routing.
`Windows.Storage`	With `Windows.Storage` and its subnamespaces, it is possible to access files and directories as well as use streams and compression.
`Windows.System`	The `Windows.System` namespace and its subnamespaces give information about the system and the user, but they also offer a `Launcher` to launch other apps.
`Windows.UI.Xaml`	In this namespace, you can find a ton of types for the user interface.

HELLO, WORLD

Let's get into coding and create a *Hello, World* application. Since the 1970s, when Brian Kernighan and Dennis Ritchie wrote the book *The C Programming Language*, it's been a tradition to start learning programming languages using a Hello, World application. Interestingly, the syntax for Hello, World changed with C# 6; it's the first time this simple program has looked different since the invention of C#.

The first samples will be created without the help of Visual Studio so you can see what happens behind the scenes by creating the application with command-line tools and a simple text editor (such as Notepad). Later, you'll switch to using Visual Studio because it makes programming life easier.

Type the following source code into a text editor, and save it with a `.cs` extension (for example, `HelloWorld.cs`). The `Main` method is the entry point for a .NET application. The CLR invokes a static `Main` method on startup. The `Main` method needs to be put into a class. Here, the class is named `Program`, but you could call it by any name. `WriteLine` is a `static` method of the `Console` class. All the static members of the `Console` class are opened with the `using` declaration in the first line. `using static System.Console` opens the static members of the `Console` class with the result that you don't need to type the class name calling the method `WriteLine` (code file `Dotnet/HelloWorld.cs`):

```
using static System.Console;

class Program
{
  static void Main()
  {
        WriteLine("Hello, World!");
      }
    }
```

As previously mentioned, the syntax of Hello, World changed slightly with C# 6. Previous to C# 6, `using static` was not available, and only a namespace could be opened with the `using` declaration. Of course, the following code still works with C# 6 (code file `Dotnet/HelloWorld2.cs`):

```
using System;

class Program
{
  static void Main()
  {
    Console.WriteLine("Hello, World!");
  }
}
```

The `using` declaration is there to reduce the code with opening a namespace. Another way to write the Hello, World program is to remove the `using` declaration and add the `System` namespace to the `Console` class with the invocation of the `WriteLine` method (code file `Dotnet/HelloWorld3.cs`):

```
class Program
{
  static void Main()
  {
    System.Console.WriteLine("Hello, World!");
  }
}
```

After writing the source code, you need to compile the code to run it.

COMPILING WITH .NET 4.6

You can compile this program by simply running the C# command-line compiler (`csc.exe`) against the source file, like this:

```
csc HelloWorld.cs
```

If you want to compile code from the command line using the `csc` command, you should be aware that the .NET command-line tools, including `csc`, are available only if certain environment variables have been set up. Depending on how you installed .NET (and Visual Studio), this may or may not be the case on your machine.

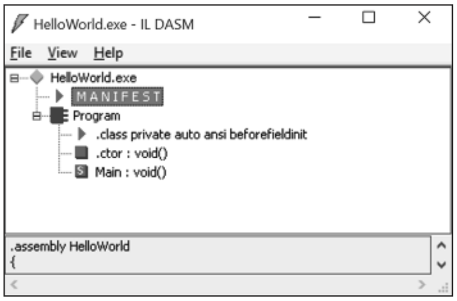
NOTE *If you do not have the environment variables set up, you have three options: The first is to add the path to the call of the* `csc` *executable. It is located at* `%Program Files%\MsBuild\14.0\Bin\csc.exe` *With the dotnet tools installed, you can also find the* `csc` *at* `%ProgramFiles%\dot.net\bin\csc.exe`*. The second option is to run the batch file* `%Microsoft Visual Studio 2015%\Common7\Tools\vsvars32.bat` *from the command prompt before running* `csc`*, where* `%Microsoft Visual Studio 2015%` *is the folder to which Visual Studio 2015 has been installed. The third, and easiest, way is to use the Visual Studio 2015 command prompt instead of the Windows command prompt. To find the Visual Studio 2015 command prompt from the Start menu, select Programs ➪ Microsoft Visual Studio 2015 ➪ Visual Studio Tools. The Visual Studio 2015 command prompt is simply a command prompt window that automatically runs* `vsvars32.bat` *when it opens.*

Compiling the code produces an executable file named `HelloWorld.exe`, which you can run from the command line. You can also run it from Windows Explorer as you would run any other executable. Give it a try:

```
> csc HelloWorld.cs
Microsoft (R) Visual C# Compiler version 1.1.0.51109
Copyright (C) Microsoft Corporation. All rights reserved.
> HelloWorld
Hello World!
```

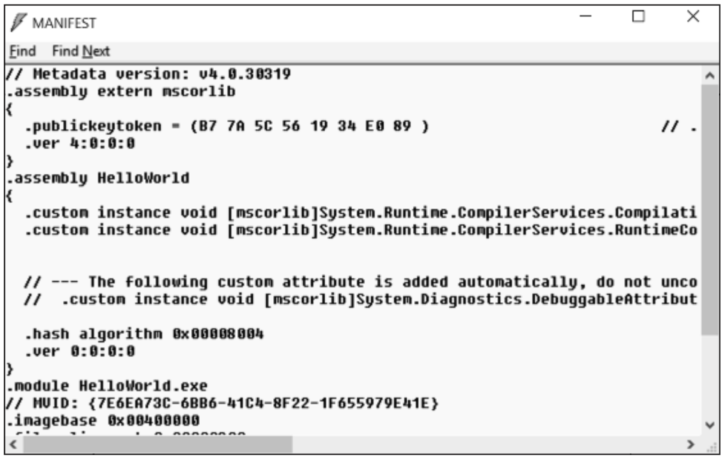
Compiling an executable this way produces an assembly that contains Intermediate Language (IL) code. The assembly can be read using the Intermediate Language Disassembler (IL DASM) tool. If you run `ildasm.exe` and open `HelloWorld.exe`, you see that the assembly contains a `Program` type and a `Main` method as shown in Figure 1-3.

Double-click the MANIFEST node in the tree view to reveal metadata information about the assembly (see Figure 1-4). This assembly makes use of the `mscorlib` assembly (because the `Console` class is located there), and some configuration and version of the HelloWorld assembly.



```
.assembly HelloWorld
{
```

FIGURE 1-3



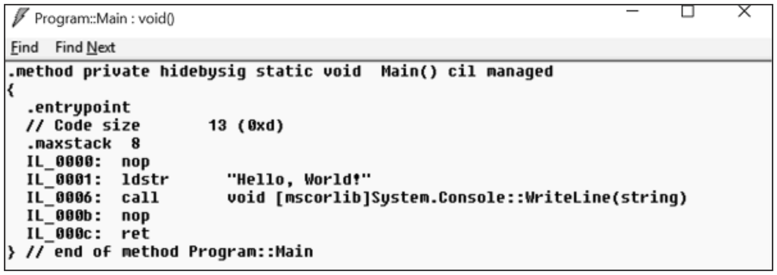
```
// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .
  .ver 4:0:0:0
}
.assembly HelloWorld
{
  .custom instance void [mscorlib]System.Runtime.CompilerServices.Compilati
  .custom instance void [mscorlib]System.Runtime.CompilerServices.RuntimeCo


  // --- The following custom attribute is added automatically, do not unco
  //  .custom instance void [mscorlib]System.Diagnostics.DebuggableAttribut

  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module HelloWorld.exe
// MVID: {7E6EA73C-6BB6-41C4-8F22-1F655979E41E}
.imagebase 0x00400000
```

FIGURE 1-4

Double-click the `Main` method to reveal the IL code of this method (see Figure 1-5). No matter what version of the Hello, World code you compiled, the result is the same. The string `Hello, World!` is loaded before calling the method `System.Console.WriteLine` that is defined within the `mscorlib` assembly passing the string. One feature of the CLR is the JIT compiler. The JIT compiler compiles IL code to native code when running the application.



```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       13 (0xd)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr      "Hello, World!"
  IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  ret
} // end of method Program::Main
```

FIGURE 1-5

COMPILING WITH .NET CORE CLI

Using the new .NET Core Command line (CLI), some preparations need to be done to compile the application without the help of Visual Studio. Let's have a look at the new tools next to compile the Hello, World sample application.

Setting Up the Environment

In case you have Visual Studio 2015 with the latest updates installed, you can immediately start with the CLI tools. Otherwise, you need to install .NET Core and the CLI tools. You can find instructions for the download at `http://dotnet.github.io` for Windows, Linux, and OS X.

With Windows, different versions of .NET Core runtimes as well as NuGet packages are installed in the user profile. As you work with .NET, this folder increases in size. Over time as you create multiple projects, NuGet packages are no longer stored in the project itself; they're stored in this user-specific folder. This has the advantage that you do not need to download NuGet packages for every different project. After you have this NuGet package downloaded, it's on your system. Just as different versions of the NuGet packages as well as the runtime are available, all the different versions are stored in this folder. From time to time it might be interesting to check this folder and delete old versions you no longer need.

Installing .NET Core CLI tools, you have the dotnet tools as an entry point to start all these tools. Just start

```
> dotnet
```

to see all the different options of the dotnet tools available.

The `repl (read, eval, print, loop')` command is good to learn and test simple features of C# without the need to create a program. Start `repl` with the `dotnet` tool:

```
> dotnet repl
```

This starts an interactive repl session. You can enter the following statements for a Hello, World using a variable:

```
> using static System.Console;
> var hello = "Hello, World!";
> WriteLine(hello);
```

The output you'll see as you enter the last statement is the `Hello, World!` string.

The dotnet repl command is not available with Preview 2 of the tools, but it will be available at a later time as an extension.

Building the Application

The dotnet tools offer an easy way to create a Hello, World application. You create a new directory `HelloWorldApp`, and change to this directory with the command prompt. Then enter this command:

```
> dotnet new
```

This command creates a `Program.cs` file that includes the code for the Hello, World program, a `NuGet.config` file that defines the NuGet server where NuGet packages should be loaded, and `project.json`, the new project configuration file.

> **NOTE** *With* `dotnet new` *you can also create the initial files needed for libraries and ASP .NET web applications (the option* `--template` *will be available with RTM). You can also select other programming languages, such as F# and Visual Basic (with the option* `--lang`*).*

The created project configuration file is named `project.json`. This file is in JavaScript Object Notation (JSON) format and defines the framework application information such as version, description, authors, tags, dependencies to libraries, and the frameworks that are supported by the application. The generated project configuration file is shown in the following code snippet (code file HelloWorldApp/project.json):

```
{
  "version": "1.0.0-*",
  "buildOptions": {
    "emitEntryPoint": true
  },

  "dependencies": {
    "NETStandard.Library": "1.0.0-*"
  },

  "frameworks" : {
    "netstandardapp1.5": {
      "imports": "dnxcore50"
    }
  },
  "runtimes" : {
    "ubuntu.14.04-x64": { },
    "win7-x64": { },
    "win10-x64": { },
    "osx.10.10-x64": { },
    "osx.10.11-x64": { }
  }
}
```

With the buildOptions settings, the emitEntryPoint is set. This is necessary if you create a Main method as a program entry point. This Main method is invoked when you run the application. This setting is not needed with libraries.

With the dependencies section, you can add all dependencies of the program, such as additional NuGet packages needed to compile the program. By default, NetStandard.Library is added as a dependency. NetStandard.Library is a reference NuGet package—a package that references other NuGet packages. With this you can avoid adding a lot of other packages, such as System.Console for the Console class, System.Collections for generic collection classes, and many more. NetStandard.Library 1.0 is a standard that defines a list of assemblies that all .NET platforms must support. At the website https://github.com/dotnet/corefx/blob/master/Documentation/project-docs/standard-platform.md you can find a long list of assemblies and their version numbers that are part of 1.0 and the assemblies that are added with 1.1, 1.2, 1.3, and 1.4 of the .NET standard.

Having a dependency on NetStandard.Library 1.0, you can support the .NET Framework 4.5.2 and up (support for .NET 4, 4.5, 4.5.1 ended in January 2016), .NET Core 1.0, the UWP 10.0, and other .NET Frameworks such as Windows Phone Silverlight 8.0, Mono, and Mono/Xamarin. Changing to version 1.3 restricts the support to .NET 4.6, .NET Core 1.0, UWP 10.0, and Mono/Xamarin platforms. Version 1.4 restricts support to .NET 4.6.1, .NET Core 1.0, and Mono/Xamarin platforms, but you get newer versions and a larger list of assemblies available.

The frameworks section in project.json lists the .NET Frameworks that are supported by your application. By default, the application is only built for .NET Core 1.0 as specified by the netstandardapp1.5 moniker. netstandardapp1.5 is used with applications built for .NET Core. With libraries, you can use the moniker netstandard1.0. This allows using the library both from .NET Core applications and applications using the .NET Framework. The imports section within netstandardapp1.5 references the older name dnxcore50, which maps the old moniker to the new one. This allows packages that still use the old name to be used.

.NET Core is the new open source version of the framework that is available on Windows, Linux, and OS X. The runtime that should be supported needs to be added to the runtimes section. The previous code snippet shows support for the Ubuntu Linux distribution, Windows 7 (which also allows running the app on Windows 8), Windows 10, and OS X.

Adding the string `net46`, the program is built for the .NET Framework, version 4.6, as well:

```
  "frameworks" : {
    "netstandardapp1.5" : { }
    "net46" : { }
  }
```

Adding `net46` to the `frameworks` section also results in no more support for non-Windows runtimes, and thus you need to remove these runtimes.

You can also add additional metadata, such as a description, author information, tags, project, and license URL:

```
"version": "1.0.0-*",
"description": "HelloWorld Sample App for Professional C#",
"authors": [ "Christian Nagel" ],
"tags": [ "Sample", "Hello", "Wrox"  ],
"projectUrl": "http://github.com/professionalCSharp/",
"licenseUrl": "",
```

As you add multiple frameworks to the `project.json` file, you can specify dependencies that are specific to every framework in a `dependencies` section below the `framework`. The dependencies specified in the `dependencies` section that is at the same hierarchical level as the `frameworks` section specify the dependencies common to all frameworks.

After having the project structure in place, you can download all dependencies of the application using the command

```
> dotnet restore
```

while your command prompt is positioned in the same directory where the `project.json` file resides. This command downloads all dependencies needed for the application, as defined in the `project.json` file. Specifying the version `1.0.0-*` gets version 1.0.0 and the latest available version for the *. In the file `project.lock.json` you can see what NuGet packages with which version were retrieved, including dependencies of dependencies. Remember, the packages are stored in a user-specific folder.

To compile the application, start the command `dotnet build` and you can see output like this—compiling for .NET Core 1.0 and .NET Framework 4.6:

```
> dotnet build
Compiling HelloWorldApp for .NETStandardApp, Version=1.5"
Compilation succeeded.
  0 Warning(s)
  0 Error(s)
Time elapsed 00:00:02.6911660

Compiling HelloWorldApp for .NETFramework,Version=v4.6
Compilation succeeded.
  0 Warning(s)
  0 Error(s)
Time elapsed 00:00:03.3735370
```

As a result of the compilation process, you find the assembly containing the IL code of the `Program` class within the `bin/debug/[netstandardapp1.5|net46]` folder. If you compare the build of .NET Core with .NET 4.6, you will find a DLL containing the IL code with .NET Core, and an EXE containing the IL code with .NET 4.6. The assembly generated for .NET Core has a dependency to the `System.Console` assembly, whereas the .NET 4.6 assembly finds the `Console` class in the `mscorlib` assembly.

You can also compile the program to native code using this command line:

```
> dotnet build --native
```

Compiling to native code results in a faster startup of the application as well as less memory consumption. The native compilation process compiles the IL code of the application as well as all dependencies to a single native image. Don't expect that all functionality of .NET Core will be available to compile to native code, but as time continues and development from Microsoft proceeds, more and more applications can be compiled to native code.

To run the application, you can use the `dotnet` command

```
> dotnet run
```

To start the application using a specific version of the framework, you can use the option `–framework`. This framework must be configured with the `project.json` file:

```
> dotnet run --framework net46
```

You can also run the application starting the executable that you can find in the `bin/debug` directory.

NOTE *As you've seen building and running the Hello, World app on Windows, the dotnet tools work the same on Linux and OS X. You can use the same dotnet commands on either platform. Before using the dotnet commands, you just need to prepare the infrastructure using the sudo utility for Ubuntu Linux and install a PKG package on OS X as described at* `http://dotnet.github.io`*. After installing the .NET Core CLI, you can use the dotnet tools in the same way as you've seen in this section—with the exception that the .NET Framework 4.6 is not available. Other than that, you can restore NuGet packages and compile and run the application with* `dotnet restore, dotnet build,` *and* `dotnet run.`

The focus of this book is on Windows, as Visual Studio 2015 offers a more powerful development platform than is available on the other platforms, but many code samples from this book are based on .NET Core, and you will be able to run them on other platforms as well. You can also use Visual Studio Code, a free development environment, to develop applications directly on Linux and OS X. See the section "Developer Tools" later in this chapter for more information about different editions of Visual Studio.

Packaging and Publishing the Application

With the dotnet tool you can also create a NuGet package and publish the application for deployment.

The command `dotnet pack` creates a NuGet package that you can put on a NuGet server. Developers can now reference the package using this command:

```
> dotnet pack
```

Running this command with the `HelloWorldApp` creates the file `HelloWorldApp.1.0.0.nupkg` that contains the assemblies for all supported frameworks. A NuGet package is a ZIP file. If you rename this file with a `.zip` extension, you can easily look into it to see the content. With the sample app, two folders are created named `dnxcore500` and `net46` that contain the respective assemblies. The file `HelloWorldApp.nuspec` is an XML file that describes the NuGet package, lists the content for supported frameworks, and lists assembly dependencies that are required before the NuGet package can be installed.

To publish the application, on the target system the runtime is needed as well. The files that are needed for publishing can be created with the `dotnet publish` command:

```
> dotnet publish
```

Using optional arguments, you can specify only a specific runtime to publish for (option `-r`) or a different output directory (option `-o`). After running this command on a Windows system you can find a `win7-x64` folder with all the files needed on the target system. Be aware that with .NET Core the runtime is included; thus it doesn't matter what runtime version is installed.

APPLICATION TYPES AND TECHNOLOGIES

You can use C# to create console applications; with most samples in the first chapters of this book you'll do that exact thing. For real programs, console applications are not used that often. You can use C# to create applications that use many of the technologies associated with .NET. This section gives you an overview of the different types of applications that you can write in C#.

Data Access

Before having a look at the application types themselves, let's look at technologies that are used by all application types: access to data.

Files and directories can be accessed by using simple API calls; however, the simple API calls are not flexible enough for some scenarios. With the stream API you have a lot of flexibility, and the streams offer many more features such as encryption or compression. Readers and writers make using streams easier. All of the different options available here are covered in Chapter 23, "Files and Streams." It's also possible to serialize complete objects in XML or JSON format. Chapter 27, "XML and JSON," discusses these options.

To read and write to databases, you can use ADO.NET directly (see Chapter 37, "ADO.NET"), or you can use an abstraction layer, the ADO.NET Entity Framework (Chapter 38, "Entity Framework Core"). Entity Framework offers a mapping of object hierarchies to the relations of a database.

The ADO.NET Entity Framework made it through several iterations. The different versions of the Entity Framework are worth discussing; this gives you good information about why NuGet packages are a good idea. You'll also learn what parts of the Entity Framework shouldn't be used going forward.

The following table describes the different versions of the Entity Framework and each version's new features.

ENTITY FRAMEWORK	DESCRIPTION
1.0	Available with .NET 3.5 SP1. This version offered a mapping through an XML file to map tables to objects.
4.0	With .NET 4, Entity Framework made a jump from version 1 to 4.
4.1	Code First Support.
4.2	Bug fixes.
4.3	Migrations added.
5.0	Released together with .NET 4.5 and offering performance improvements, supporting new SQL Server features.
6.0	Moved to a NuGet package.
7.0	A complete rewrite, also supporting NoSQL, running on Windows apps as well.

Let's get into some details. Entity Framework was originally released as part of the .NET Framework classes that come preinstalled with the .NET Framework. Entity Framework 1 was part of the first service pack of .NET 3.5, which was a feature update: .NET 3.5 Update 1.

The second version had so many new features that the decision was made to move to version 4 together with .NET 4. After that, Entity Framework was released at a faster cadence than the .NET Framework. To get a newer version of Entity Framework, a NuGet package had to be added to the application (versions 4.1, 4.2, 4.3). There was a problem with this approach. Classes that have already been delivered with the .NET

Framework had to be used as is. Just additional features, such as Code First, have been added with NuGet packages.

With .NET 4.5, Entity Framework 5.0 was released. Again some of the classes come with the preinstalled .NET Framework, and additional features are part of NuGet packages. The NuGet package also made it possible to allow installing the NuGet package for Entity Framework 5.0 with .NET 4.0 applications. However, in reality the package decided (via a script) in a case when Entity Framework 5.0 is added to a .NET 4.0 project that the result would be Entity Framework 4.4 because some of the types required belong to .NET 4.5 and are not part of .NET 4.

The next version of Entity Framework solved this problem by moving all the Entity Framework types to a NuGet package; the types that come with the Framework itself are ignored. This allows using version 6.0 with older versions of the Framework; you aren't restricted to 4.5. To not conflict with classes of the Framework, some types moved to a different namespace. Some features of ASP.NET Web Forms had an issue with that because original classes of the Entity Framework have been used, and these do not map that easily to the new classes.

During the different releases, Entity Framework gives different options for mapping the database tables to classes. The first two options were Database First and Model First. With both of these options, the mapping was done via XML files. The XML file is presented via a graphical designer, and it's possible to drag entities from the toolbox to the designer for doing the mapping.

With version 4.1, mapping via code was added: Code First. Code First doesn't mean that the database can't exist beforehand. Both are possible: A database can be created dynamically, but also the database can exist before you write the code. Using Code First, you don't do the mapping via XML files. Instead, attributes or a fluent API can define the mapping programmatically.

Entity Framework Core 1.0 is a complete redesign of Entity Framework, as is reflected with the new name. Code needs to be changed to migrate applications from older versions of Entity Framework to the new version. Older mapping variants, such as Database First and Model First, have been dropped, as Code First is a better alternative. The complete redesign was also done to support not only relational databases but also NoSQL. Azure Table Storage is one of the options where Entity Framework can now be used.

Windows Desktop Applications

For creating Windows desktop applications, two technologies are available: Windows Forms and Windows Presentation Foundation. *Windows Forms* consists of classes that wrap native Windows controls; it's based on pixel graphics. *Windows Presentation Foundation* (*WPF*) is the newer technology and is based on vector graphics.

WPF makes use of XAML in building applications. XAML stands for eXtensible Application Markup Language. This way to create applications within a Microsoft environment was introduced in 2006 and is part of the .NET Framework 3.0. .NET 4.5 introduced new features to WPF, such as ribbon controls and live shaping.

XAML is the XML declaration used to create a form that represents all the visual aspects and behaviors of the WPF application. Though you can work with a WPF application programmatically, WPF is a step in the direction of declarative programming, which the industry is moving to. *Declarative programming* means that instead of creating objects through programming in a compiled language such as C#, Visual Basic, or Java, you declare everything through XML-type programming. Chapter 29, "Core XAML," introduces XAML (which is also used with XML Paper Specification, Windows Workflow Foundation, and Windows Communication Foundation). Chapter 30 covers XAML styles and resources. Chapter 34, "Windows Desktop Applications with WPF," gives details on controls, layout, and data binding. Printing and creating documents is another important aspect of WPF that's covered in Chapter 35, "Creating Documents with WPF."

What's the future of WPF? Isn't the UWP the UI platform to use for new applications going forward? UWP has advantages in supporting mobile devices as well. As long as some of your users have not upgraded to

Windows 10, you need to support older operating systems such as Windows 7. UWP apps don't run on Windows 7 or Windows 8. You can use WPF. In case you also would like to support mobile devices, it's best to do as much code sharing as possible. You can create apps with both WPF and UWP by using as much common code as possible by supporting the MVVM pattern. This pattern is covered in Chapter 31, "Patterns with XAML Apps."

Universal Windows Platform

The Universal Windows Platform (UWP) is a strategic platform from Microsoft. When you use the UWP to create Windows apps, you're limited to Windows 10 and newer versions of Windows. But you're not bound to the desktop version of Windows. With Windows 10 you have a lot of different options, such as Phone, Xbox, Surface Hub, HoloLens, and IoT. There's one API that works on all these devices!

One API for all these devices? Yes! Each device family can add its own Software Development Kit (SDK) to add features that are not part of the API that's available for all devices. Adding these SDKs does not break the application, but you need to programmatically check whether an API from such an SDK is available on the platform the app is running. Depending on how many API calls you need to differentiate, the code might grow into a mess; dependency injection might be a better option.

NOTE *Dependency injection is discussed in Chapter 31, along with other patterns useful with XAML-based applications.*

You can decide what device families to support with your applications. Not all device families will be useful for every app.

Will there be newer versions of Windows after Windows 10? Windows 11 is not planned. With Windows apps (which are also known as Metro apps, Windows Store apps, Modern apps, and Universal Windows apps) you've targeted either Windows 8 or Windows 8.1. Windows 8 apps typically were also running on Windows 8.1, but not the other way around. Now this is very different. When you create an app for the Universal Windows Platform, you target a version such as 10.0.10130.0 and define what minimum version is available and what latest version was tested, and the assumption is that it runs on future versions as well. Depending on the features you can use for your app and what version you're expecting the user to have, you can decide what minimum version to support. Personal users will typically automatically update to newer versions; Enterprise users might stick to older versions.

Windows Apps running on the Universal Windows Platform make use of the Windows Runtime and .NET Core. The most important chapters for these app types are Chapter 32, "Windows Apps: User Interfaces," and Chapter 33, "Advanced Windows Apps." These apps are also covered in many other chapters, such as Chapter 23 and Chapters 29 through 31.

SOAP Services with WCF

Windows Communication Foundation (WCF) is a feature-rich technology that was meant to replace all communication technologies that were available before WCF by offering SOAP-based communication with all the features used by standards-based web services such as security, transactions, duplex and one-way communication, routing, discovery, and so on. WCF provides you with the ability to build your service one time and then expose this service in many ways (even under different protocols) by making changes within a configuration file. WCF is a powerful but complex way to connect disparate systems. Chapter 44, "Windows Communication Foundation," covers this in detail.

Web Services with the ASP.NET Web API

An option that is a lot easier for communication and fulfills more than 90 percent of requirements by distributed applications is the ASP.NET Web API. This technology is based on REST (Representational State Transfer), which defines guidelines and best practices for stateless and scalable web services.

The client can receive JSON or XML data. JSON and XML can also be formatted in a way to make use of the Open Data specification (OData).

The features of this new API make it easy to consume from web clients using JavaScript and also by using the Universal Windows Platform.

The ASP.NET Web API is a good approach for creating microservices. The approach to build microservices defines smaller services that can run and be deployed independently, having their own control of a data store.

With ASP.NET 5, the older version of ASP.NET Web API that was separated from ASP.NET MVC now merged with ASP.NET MVC 6 and uses the same types and features.

NOTE *The ASP.NET Web API and more information on microservices are covered in Chapter 42.*

WebHooks and SignalR

For real-time web functionality and bidirectional communication between the client and the server, WebHooks and SignalR are ASP.NET technology that can be used.

SignalR allows pushing information to connected clients as soon as information is available. SignalR makes use of the WebSocket technology, and it has a fallback to a pull-based mechanism of communication in case WebSockets are not available.

WebHooks allows you to integrate with public services, and these services can call into your public ASP.NET Web API service. WebHooks is a technology to receive push notification from services such as GitHub or Dropbox and many other services.

The foundation of SignalR connection management, grouping of connections, and authorization and integration of WebHooks are discussed in Chapter 43, “WebHooks and SignalR.”

Windows Services

A web service, whether it’s done with WCF or ASP.NET Web Services, needs a host to run. Internet Information Server is usually a good option because of all the services it offers, but it can also be a custom program. With the custom option, creating a background process that runs with the startup of Windows is a Windows Service. This is a program designed to run in the background in Windows NT kernel–based operating systems. Services are useful when you want a program to run continuously and be ready to respond to events without having been explicitly started by the user. A good example is the World Wide Web Service on web servers, which listens for web requests from clients.

It is easy to write services in C#. .NET Framework base classes are available in the `System.ServiceProcess` namespace that handles many of the boilerplate tasks associated with services. In addition, Visual Studio .NET enables you to create a C# Windows Service project, which uses C# source code for a basic Windows Service. Chapter 39, “Windows Services,” explores how to write C# Windows Services.

Web Applications

The original introduction of ASP.NET 1 fundamentally changed the web programming model. ASP.NET 5 is the new major release, which allows the use of .NET Core for high performance and scalability. This new release can also run on Linux systems, which was a high demand.

With ASP.NET 5, ASP.NET Web Forms is no longer covered (this can still be used and is updated with .NET 4.6), so this book has a focus on the modern technology ASP.NET MVC 6, which is part of ASP.NET 5.

ASP.NET MVC is based on the well-known Model View Controller (MVC) pattern for easier unit testing. It also allows a clear separation for writing user interface code with HTML, CSS, and JavaScript, and it only uses C# on the backend.

NOTE *Chapter 41, "ASP.NET MVC," covers ASP.NET MVC 6.*

Microsoft Azure

Nowadays you can't ignore the cloud when considering the development picture. Although there's not a dedicated chapter on cloud technologies, Microsoft Azure is referenced in several chapters in this book.

Microsoft Azure offers Software as a Service (SaaS), Infrastructure as a Service (IaaS), and Platform as a Service (PaaS), and sometimes offerings are in between these categories. Let's have a look at some Microsoft Azure offerings.

Software as a Service

SaaS offers complete software; you don't have to deal with management of servers, updates, and so on. Office 365 is one of the SaaS offerings for using e-mail and other services via a cloud offering. A SaaS offering that's relevant for developers is *Visual Studio Online*, which is not Visual Studio running in the browser. Visual Studio Online is the Team Foundation Server in the cloud that can be used as a private code repository, for tracking bugs and work items, and for build and testing services.

Infrastructure as a Service

Another service offering is IaaS. Virtual machines are offered by this service offering. You are responsible for managing the operating system and maintaining updates. When you create virtual machines, you can decide between different hardware offerings starting with shared Cores up to 32 cores (at the time of this writing, but things change quickly). 32 cores, 448 GB RAM, and 6,144 GB local SSD belong to the "G-Series" of machines, which is named after Godzilla.

With preinstalled operating systems you can decide between Windows, Windows Server, Linux, and operating systems that come preinstalled with SQL Server, BizTalk Server, SharePoint, and Oracle.

I use virtual machines often for environments that I need only for several hours a week, as the virtual machines are paid on an hourly basis. In case you want to try compiling and running .NET Core programs on Linux but don't have a Linux machine, installing such an environment on Microsoft Azure is an easy task.

Platform as a Service

For developers, the most relevant part of Microsoft Azure is PaaS. You can access services for storing and reading data, use computing and networking capabilities of app services, and integrate developer services within the application.

For storing data in the cloud, you can use a relational data store SQL Database. SQL Database is nearly the same as the on-premise version of SQL Server. There are also some NoSQL solutions such as DocumentDB that stores JSON data, and Storage that stores blobs (for example, for images or videos) and tabular data (which is really fast and offers huge amounts of data).

Web apps can be used to host your ASP.NET MVC solution, and API Apps can be used to host your ASP.NET Web API services.

Visual Studio Online is part of the Developer Services offerings. Here you also can find Visual Studio Application Insights. With faster release cycles, it's becoming more and more important to get information about how the user uses the app. What menus are never used because the users probably don't find them? What paths in the app is the user is taking to fulfill his or her tasks? With Visual Studio Application Insights, you can get good anonymous user information to find out the issues users have with the application, and with DevOps in place you can do quick fixes.

NOTE *In Chapter 20, "Diagnostics and Application Insights," you can read about tracing features and also how to use the Visual Studio Application Insights offering of Microsoft Azure. Chapter 45, "Deployment of Websites and Services," not only shows deployment to the local Internet Information Server (IIS) but also describes deployment to Microsoft Azure Web Apps.*

DEVELOPER TOOLS

This final part of the chapter, before we switch to a lot of C# code in the next chapter, covers developer tools and editions of Visual Studio 2015.

Visual Studio Community

This edition of Visual Studio is a free edition with features that the Professional edition previously had. There's a license restriction for when it can be used. It's free for open-source projects and training, and also free to academic and small professional teams. Unlike the Express editions of Visual Studio that previously have been the free editions, this product allows using add-ins with Visual Studio.

Visual Studio Professional with MSDN

This edition includes more features than the Community edition, such as the CodeLens and Team Foundation Server for source code management and team collaboration. With this edition, you also get an MSDN subscription that includes several server products from Microsoft for development and testing.

Visual Studio Enterprise with MSDN

Visual Studio 2013 had Premium and Ultimate editions. Visual Studio 2015 instead has the Enterprise edition. This edition offers Ultimate features with a Premium price model. Like the Professional edition, this edition contains a lot of tools for testing, such as Web Load & Performance Testing, Unit Test Isolation with Microsoft Fakes, and Coded UI Testing. (Unit testing is part of all Visual Studio editions.) With Code Clone you can find code clones in your solution. Visual Studio Enterprise also contains architecture and modeling tools to analyze and validate the solution architecture.

NOTE *Be aware that with an MSDN subscription you're entitled to free use of Microsoft Azure up to a specific monthly amount that is contingent on the type of the MSDN subscription you have.*

NOTE *Chapter 17, "Visual Studio 2015," includes details on using several features of Visual Studio 2015. Chapter 19, "Testing," gets into details of unit testing, web testing, and creating Coded UI tests.*

NOTE *For some of the features in the book—for example, the Coded UI Tests —you need Visual Studio Enterprise. You can work through most parts of the book with the Visual Studio Community edition.*

Visual Studio Code

Visual Studio Code is a completely different development tool compared to the other Visual Studio editions. While Visual Studio 2015 offers project-based features with a rich set of templates and tools, Visual Studio is a code editor with little project management support. However, Visual Studio Code runs not only on Windows, but also on Linux and OS X.

With many chapters of this book, you can use Visual Studio Code as your development editor. What you can't do is create WPF, UWP, or WCF applications, and you also don't have access to the features covered in Chapter 17, "Visual Studio 2015." You can use Visual Studio Code for .NET Core console applications, and ASP.NET Core 1.0 web applications using .NET Core.

You can download Visual Studio Code from `http://code.visualstudio.com`.

SUMMARY

This chapter covered a lot of ground to review important technologies and changes with technologies. Knowing about the history of some technologies helps you decide which technology should be used with new applications and what you should do with existing applications.

You read about the differences between .NET Framework 4.6 and .NET Core 1.0, and you saw how to create and run a Hello, World application with all these environments without using Visual Studio.

You've seen the functions of the Common Language Runtime (CLR) and looked at technologies for accessing the database and creating Windows apps. You also reviewed the advantages of ASP.NET Core 1.0.

Chapter 2 steps into using Visual Studio to create the Hello, World application and goes on to discuss the syntax of C#.