PART I
The C# Language

- ➤ **CHAPTER 1:** .NET Applications and Tools
- ➤ **CHAPTER 2:** Core C#
- ➤ **CHAPTER 3:** Objects and Types
- ➤ **CHAPTER 4:** Object-Oriented Programming with C#
- ➤ **CHAPTER 5:** Generics
- ➤ **CHAPTER 6:** Operators and Casts
- ➤ **CHAPTER 7:** Arrays
- ➤ **CHAPTER 8:** Delegates, Lambdas, and Events
- ➤ **CHAPTER 9:** Strings and Regular Expressions
- ➤ **CHAPTER 10:** Collections
- ➤ **CHAPTER 11:** Special Collections
- ➤ **CHAPTER 12:** Language Integrated Query
- ➤ **CHAPTER 13:** Functional Programming with C#
- ➤ **CHAPTER 14:** Errors and Exceptions

COPYRIGHTED MATERIAL

- **CHAPTER 15:** Asynchronous Programming
- **CHAPTER 16:** Reflection, Metadata, and Dynamic Programming
- **CHAPTER 17:** Managed and Unmanaged Memory
- **CHAPTER 18:** Visual Studio 2017

1

.NET Applications and Tools

WHAT'S IN THIS CHAPTER?

- Reviewing the history of .NET
- Understanding differences between .NET Framework and .NET Core
- NuGet packages
- The Common Language Runtime
- Features of the Windows Runtime
- Programming Hello World!
- .NET Core Command-Line Interface
- Visual Studio 2017
- 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` on the Download Code tab. The source code is also available at `https://github.com/ProfessionalCSharp/ProfessionalCSharp7` in the directory `HelloWorld`.

The code for this chapter is divided into the following major examples:

- HelloWorld
- WebApp
- SelfContained HelloWorld

CHOOSING YOUR TECHNOLOGIES

.NET has been a great technology for creating applications on the Windows platform. Now .NET is a great technology for creating applications on Windows, Linux, and the Mac.

The creation of .NET Core has been the biggest change for .NET since its invention. Now .NET code is open-source code, you can create apps for other platforms, and .NET uses modern patterns. .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 apps and services, offers guidance on what to choose for database access, and highlights the differences between the .NET Framework 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 the .NET Framework 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 FRAMEWORK	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
4.7	4.0	7	2017

When you create applications with .NET Core, it's important to know the timeframe for the support level. *LTS* (Long Time Support) has a longer support length than *Current*, but Current gets new features faster. LTS is supported for three years after the release or 12 months after the next LTS version, whichever is shorter. So, .NET Core 1.0 is supported until June 27, 2019 if the next LTS version is not released before June 27, 2018. In case the next LTS version is released earlier, .NET Core 1.0 is supported one year after the release of the next LTS.

.NET Core 1.1 originally was a Current release, but it changed to LTS with the same support length as .NET Core 1.0.

.NET Core 2.0 is a release with the support level Current. This means it is supported for 3 years, 12 months after the next LTS, or 3 months after the next Current release—whichever is shorter. It can be assumed that the last option will be the case, and .NET Core 2.0 will be supported 3 months after .NET Core 2.1 is available.

The next table lists .NET Core versions, their release dates, and the support level.

.NET CORE VERSION	RELEASE DATE	SUPPORT LEVEL
1.0	June 27, 2016	LTS
1.1	Nov 16, 2016	LTS*
2.0	Aug 14, 2017	Current

The following sections cover the details of these tables 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 like 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 not only a pure object-oriented programming language with features for inheritance, encapsulation, and polymorphism. Instead, C# also offered component-based programming enhancements such as delegates and events.

Before .NET and 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 were mainly a bug fix release with minor enhancements.

> **NOTE** *Inheritance is discussed in Chapter 4, "Object-Oriented Programming with C#"; delegates and events are covered in Chapter 8, "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 11th edition of this book in your hands.*

C# 2 and .NET 2 with Generics

C# 2 and .NET 2 were 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 .NET Core.

> **NOTE** *Generics are used all through the book, but they're explained in detail in Chapter 5, "Generics." Chapter 10, "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, except for `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 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** *To read about WPF and WCF, you need the previous edition of the book,* Professional C# 6 and .NET Core 1.0.

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 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 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. The new version of Entity Framework, Entity Framework Core (EF Core) 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 12, "Language Integrated Query." The newest version of the Entity Framework is very different from the .NET 3.5.1 release; it's described in Chapter 26, "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 version skipped from 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 has a focus on the Model-View-Controller (MVC) pattern, which is enforced by the project structure. This technology also has a focus on programming HTML and JavaScript. HTML and JavaScript gained a great push in the developer community with the release of HTML 5. 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."*
>
> *The next generation of ASP.NET, ASP.NET Core is covered in Chapter 30, "ASP .NET Core." Chapter 31, "ASP.NET Core MVC," covers the ASP.NET Core version of ASP.NET Core 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 Bonus Chapter 1, "Composition." Windows apps are covered in Chapters 33 to 36, and the Web API with ASP.NET Core MVC is covered in Chapter 32, "Web API."*

C# 6 and .NET Core 1.0

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 or the .NET Compiler Platform.

The full .NET Framework is not the only .NET version 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 versions 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 was 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 was a complete rewrite of ASP.NET. This release is not completely backward compatible with older versions and requires some changes to existing ASP.NET MVC code (with ASP.NET Core MVC). 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 at 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 *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 6, "Operators and Casts"; string interpolation is in Chapter 9, "Strings and Regular Expressions"; and exception filters are in Chapter 14, "Errors and Exceptions."*

C# 7 and .NET Core 2.0

C# has been updated to have a faster pace. Major version 7.0 was released in March 2017, and the minor versions 7.1 and 7.2 soon after in August 2017 and December 2017. With a project setting, you can select the compiler version to use.

C# 7 introduces many new features (these are outlined in the Introduction.) The most significant of these features come from functional programming: *pattern matching* and *tuples*.

NOTE *Pattern matching and tuples are covered in Chapter 13, "Functional Programming with C#."*

.NET Core 2.0 is focused on making it easier to bring existing applications written with the .NET Framework to .NET Core. Types that haven't been available with .NET Core but are still in use with many .NET Framework applications and libraries are now available with .NET Core. More than 20,000 APIs have been added to .NET Core 2.0. For example, binary serialization, and the `DataSet` are back, and you can use these features also on Linux. Another feature that helps bring legacy applications to .NET Core is the Windows Compatibility Pack (`Microsoft.Windows.Compatibility`). This NuGet package defines APIs for WCF, registry access, cryptography, directory services, drawing, and more. See `https://github.com/dotnet/designs/blob/master/accepted/compat-pack/compat-pack.md` for a current state.

The .NET Standard is a spec that defines which APIs should be available on any platform that supports the standard. The higher the standard version, the more APIs are available. .NET Standard 2.0 extended the standard by more than 20,000 APIs and is supported by .NET Framework 4.6.1, .NET Core 2.0, and the Universal Windows Platform (Windows Apps) starting with build 16299 (the Fall Creators Update of Windows 10).

NOTE *The .NET Standard is covered in detail in Chapter 19, "Libraries, Assemblies, Packages, and NuGet."*

To check whether your application can easily be ported to .NET Core, you can use the .NET Portability Analyzer. You can install this tool as an extension to Visual Studio. It analyzes your binaries. You can configure the portability information for what versions and frameworks you would like to get, and you can select portability information for .NET Core, .NET Framework, .NET Standard, Mono, Silverlight, Windows, Xamarin, and more. The result can be JSON, HTML, and Excel.

Figure 1-1 shows the summary report after selecting a .NET Framework binary that is 100% compatible with the .NET Framework, 96.67% with .NET Core, and just 69.7% with Windows Apps. Figure 1-2 shows detail information about the problematic APIs.

FIGURE 1-1

FIGURE 1-2

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 a XAML-based technology, such as the Universal Windows Platform (UWP). Of course, there are still good reasons to use other technologies. Do you need to support Windows 7 clients? In that case, UWP is not an option, but WPF is. You still can create your WPF applications in a way that make it easy to switch to other technologies, such as UWP and Xamarin.

> **NOTE** *Read Chapter 34, "Patterns with XAML Apps," for information about how to design your app to share as much code as possible between WPF, UWP, and Xamarin.*

If you're creating web applications, a safe bet is to use ASP.NET Core with ASP.NET Core MVC. Making this choice rules out using ASP.NET Web Forms. If you're accessing a database, you should use Entity Framework Core, 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, WCF, Windows Forms, `System.AddIn`, Workflow Foundation, and other legacy technologies that are still part of and available with the .NET Framework.

.NET TERMS

What are the current .NET technologies? Figure 1-3 gives an overall picture of how the .NET Framework, .NET Core, and Mono relate to each other. All .NET Framework apps, .NET Core apps, and Xamarin apps can use the same libraries if they are built with the .NET Standard. These technologies share the same compiler platform, programming languages, and runtime components. They do not share the same runtime, but they do share components within their runtime. For example, the just-in-time (JIT) compiler RyuJIT is used by the .NET Framework and .NET Core.

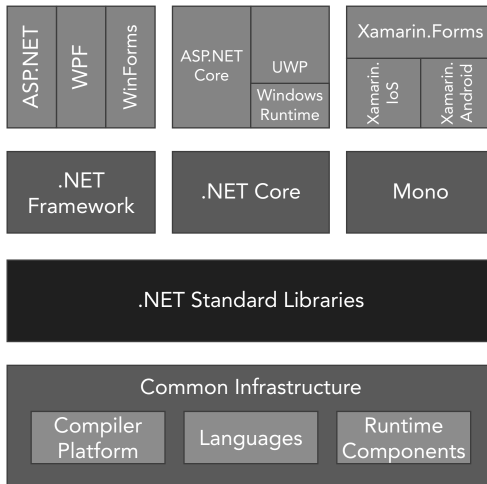

FIGURE 1-3

With the .NET Framework, you can create Windows Forms, WPF, and legacy ASP.NET applications that run on Windows.

Using .NET Core, you can create ASP.NET Core and console apps that run on different platforms. .NET Core is also used by the Universal Windows Platform (UWP), but this doesn't make UWP available on Linux. UWP also makes use of the Windows Runtime, which is available only on Windows.

Xamarin offers Xamarin.IoS and Xamarin.Android, libraries that enable you to develop C# apps for the iPhone and for Android. With Xamarin.Forms, you have a library to share the user interface between the two mobile platforms. Xamarin is currently still based on the Mono framework, a .NET variant developed by Xamarin. At some point, this might change to .NET Core. However, what's important is that all these technologies can use the same libraries created for the .NET Standard.

In the lower part of Figure 1-3, you can see there's also some sharing going on between .NET Framework, .NET Core, and Mono. *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. The .NET Compiler Platform (also known as Roslyn) and the programming languages are used by all these platforms.

.NET Framework

NET Framework 4.7 is the .NET Framework that has been continuously enhanced in the past 15 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. .NET Framework 4.7 still offers enhancements for Windows Forms, such as support for High DPI.

If you want to continue working with ASP.NET Web Forms, ASP.NET 4.7 with .NET Framework 4.7 is the way to go. Otherwise, you need to rewrite some code to move to .NET Core. Depending on the quality of the source code and the need to add new features, rewriting the code might be worthwhile.

.NET Core

.NET Core 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 is designed in a modular approach. The framework splits up into a large list of NuGet packages. So that you don't have to deal with all the packages, metapackages are used that reference the smaller packages that work together. Metapackages even improved with .NET Core 2.0 and ASP.NET Core 2.0. With ASP.NET Core 2.0, you just need to reference `Microsoft.AspNetCore.All` to get all the packages you typically need with ASP.NET Core web applications.

.NET Core can be updated at a fast pace. Even updating the runtime doesn't influence existing applications because the runtime can be 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 "Using the .NET Core CLI."*

.NET Standard

The .NET Standard is not an implementation; it's a contract. This contract specifies what APIs need to be implemented. .NET Framework, .NET Core, and Xamarin implement this standard.

The standard is versioned. With every version additional APIs are added. Depending on the APIs you need, you can choose the standard version for a library. You need to check whether your platform of choice supports the standard of the needed version.

You can find a detailed table for the platform support for the .NET Standard at `https://docs.microsoft.com/en-us/dotnet/standard/net-standard`. The following are the most important parts you need to know:

- .NET Core 1.1 supports .NET Standard 1.6; .NET Core 2.0 supports .NET Standard 2.0.
- .NET Framework 4.6.1 supports .NET Standard 2.0.
- UWP build 16299 and later supports .NET Standard 2.0; older versions support only .NET Standard 1.4.
- With Xamarin to use .NET Standard 2.0 you need Xamarin.iOS 10.14 and Xamarin.Android 8.0.

> **NOTE** *Read detailed information on the .NET Standard in Chapter 19.*

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 1-4). 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. One place to search for packages is your own shared directory where your internal used packages are placed.

> **NOTE** *When you use third-party packages from the NuGet server, you're always at risk if a package is available later. 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.*

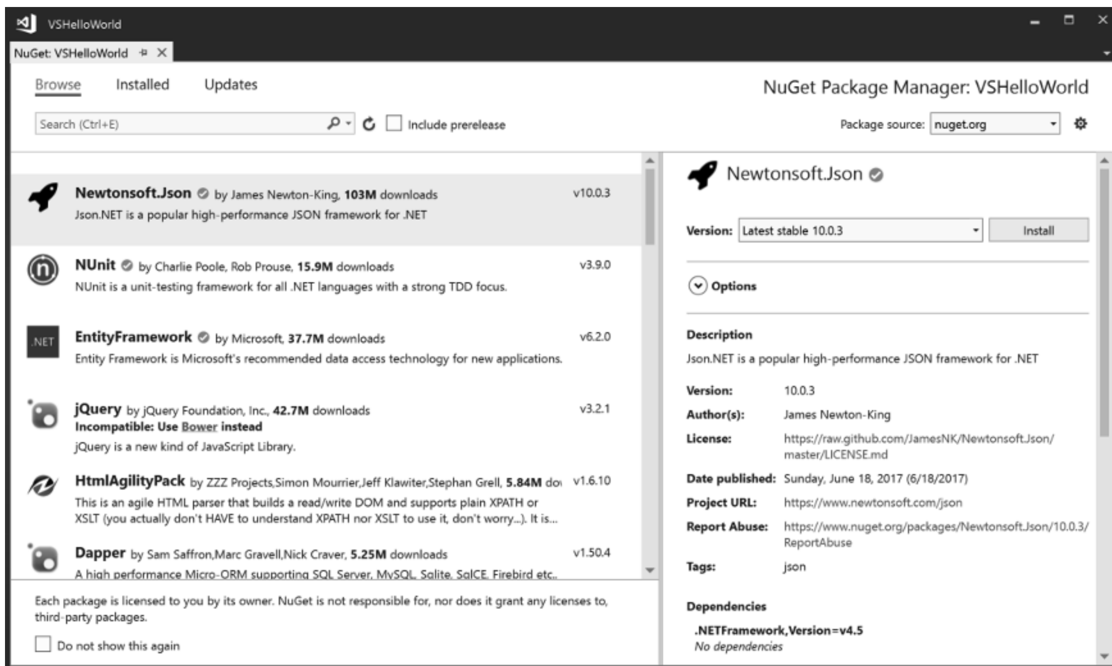

FIGURE 1-4

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

Namespaces

The classes available with .NET are organized in namespaces whose names start with the `System`. To give you an idea about the hierarchy, the following table describes a few of the namespaces.

NAMESPACE	DESCRIPTION
`System.Collections`	This is the root namespace for collections. Collections are also found within subnamespaces, 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.

continues

(continued)

NAMESPACE	DESCRIPTION
`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`.

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

Common Language Runtime

The Universal Windows Platform makes use of Native .NET to compile IL to native code with an AOT Compiler. This is like Xamarin.iOS. With all other scenarios, with both applications using the .NET Framework and applications using .NET Core 1.0, a *Common Language Runtime* (CLR) is needed. .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 the 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.

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." Chapter 17, "Managed and Unmanaged Memory," gives information about the garbage collector and how to clean up memory.*

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 apps, 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 regarding 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 types; 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.

continues

(continued)

NAMESPACE	DESCRIPTION
`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.

USING THE .NET CORE CLI

For many chapters in this book you don't need Visual Studio; you can use any editor and a command line. For creating and compiling your applications, you can use the .NET Core Command Line Interface (CLI). Let's have a look how to set up your system and how you can use this tool.

Setting Up the Environment

In case you have Visual Studio 2017 with the latest updates installed, you can immediately start with the CLI tools. As previously mentioned, you can set up a system without Visual Studio 2017. You also can use most of the samples on Linux and OS X. To download the applications for your environment, just go to `https://dot.net` and click the Get Started button. From there, you can download the .NET SDK for Windows, Linux, and macOS.

For Windows, you can download an executable that installs the SDK. With Linux, you need to select the Linux distribution to get the corresponding command:

- With Red Hat and CentOS, install the .NET SDK using `yum`.
- With Ubuntu and Debian, use `apt-get`.
- With Fedora, use `dnf install`.
- With SLES/openSUSE, use `zipper install`.
- To install the .NET SDK on the Mac, you can download a `.pkg` file.

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 --help
```

to see all the different options of the dotnet tools available. Many of the options have a shorthand notation. For help, you can type

```
> dotnet -h
```

Creating the Application

The dotnet tools offer an easy way to create a “Hello World!” application. Just enter this command:

```
> dotnet new console --output HelloWorld
```

This command creates a new `HelloWorld` directory and adds the source code file `Program.cs` and the project file `HelloWorld.csproj`. Starting with .NET Core 2.0, this command also includes a `dotnet restore` where all NuGet packages are downloaded. To see a list of dependencies and versions of libraries used by the application, you can check the file `project.assets.json` in the `obj` subdirectory. Without using the option `--output` (or `-o` as shorthand), the files would be generated in the current directory.

The generated source code looks like the following code snippet (code file `HelloWorld/Program.cs`):

```
using System;

namespace HelloWorld
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");
    }
  }
}
```

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. With the .NET Core CLI, this program is automatically generated.

Let’s get into the syntax of this program. 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.

`Console.WriteLine` invokes the `WriteLine` method of the `Console` class. You can find the `Console` class in the `System` namespace. You don’t need to write `System.Console.WriteLine` to invoke this method; the `System` namespace is opened with the `using` declaration on top of the source file.

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

The created project configuration file is named `HelloWorld.csproj`. Compared to older `csproj` files, the new project file is reduced to a few lines with several defaults:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
</Project>
```

With the project file, the `OutputType` defines the type of the output. With a console application, this is `Exe`. The `TargetFramework` specifies the framework and the version that is used to build the application. With the sample project, the application is built using .NET Core 2.0. You can change this element to `TargetFrameworks` and specify multiple frameworks, such as `netcoreapp2.0;net47` to build applications both for .NET Framework 4.7 and .NET Core 2.0 (project file `HelloWorld/HelloWorld.csproj`):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFrameworks>netcoreapp2.0;net47</TargetFrameworks>
  </PropertyGroup>
</Project>
```

The `Sdk` attribute specifies the SDK that is used by the project. Microsoft ships two main SDKs: `Microsoft.NET.Sdk` for console applications, and `Microsoft.NET.Sdk.Web` for ASP.NET Core web applications.

You don't need to add source files to the project. Files with the `.cs` extension in the same directory and subdirectories are automatically added for compilation. Resource files with the `.resx` extension are automatically added for embedding the resource. You can change the default behavior and exclude/include files explicitly.

You also don't need to add the .NET Core package. By specifying the target framework `netcoreapp2.0`, the metapackage `Microsoft.NetCore.App` that references many other packages is automatically included.

Building the Application

To build the application, you need to change the current directory to the directory of the application and start `dotnet build`. When you compile for .NET Core 2.0 and .NET Framework 4.7, you see output like the following:

```
> dotnet build
Microsoft (R) Build Engine version 15.5.179.9764 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Restore completed in 19.8 ms for
    C:\procsharp\Intro\HelloWorld\HelloWorld.csproj.
  HelloWorld -> C:\procsharp\Intro\HelloWorld\bin\Debug\net47\HelloWorld.exe
  HelloWorld ->
    C:\procsharp\Intro\HelloWorld\bin\Debug\netcoreapp2.0\HelloWorld.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.58
```

> **NOTE** *The commands* `dotnet new` *and* `dotnet build` *now include restoring NuGet packages. You can also explicitly restore NuGet packages with* `dotnet restore`.

Because of the compilation process, you find the assembly containing the IL code of the `Program` class within the `bin/debug/[netcoreapp2.0|net47]` folders. If you compare the build of .NET Core with .NET 4.7, you will find a DLL containing the IL code with .NET Core, and an EXE containing the IL code with .NET 4.7. 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.

To build release code, you need to specify the option `--Configuration Release` (shorthand `-c Release`):

```
> dotnet build --configuration Release
```

Some of the code samples in the following chapters make use of features offered by C# 7.1 or C# 7.2. By default, the latest major version of the compiler is used, which is C# 7.0. To enable newer versions of C#, you need to specify this in the project file as shown with the following project file section. Here, the latest version of the C# compiler is configured.

```
<PropertyGroup>
  <LangVersion>latest</LangVersion>
</PropertyGroup>
```

Running the Application

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

```
> dotnet run
```

In case the project file targets multiple frameworks, you need to tell the `dotnet run` command which framework to use to run the app by using the option `--framework`. This framework must be configured with the `csproj` file. With the sample application, you can see output like the following after the restore information:

```
> dotnet run --framework netcooreapp2.0
Microsoft (R) Build Engine version 15.5.179.9764 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Restore completed in 20.65 ms for
    C:\procsharp\Intro\HelloWorld\HelloWorld.csproj.
Hello World!
```

On a production system, you don't use `dotnet run` to run the application. Instead, you use `dotnet` with the name of the library:

```
> dotnet bin/debug/netcoreapp2.0/HelloWorld.dll
```

You can also create an executable, but executables are platform specific. How this is done is shown later in this chapter in the section "Packaging and Publishing the Application."

> **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.*
>
> *The focus of this book is on Windows, as Visual Studio 2017 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.*

Creating a Web Application

You also can use the .NET Core CLI to create a web application. When you start `dotnet new`, you can see a list of templates available (see Figure 1-5).

```
Developer Command Prompt for VS 2017
Templates                                         Short Name       Language          Tags
---------------------------------------------------------------------------------------------------
Console Application                               console          [C#], F#, VB      Common/Console
Class library                                     classlib         [C#], F#, VB      Common/Library
Unit Test Project                                 mstest           [C#], F#, VB      Test/MSTest
xUnit Test Project                                xunit            [C#], F#, VB      Test/xUnit
ASP.NET Core Empty                                web              [C#], F#          Web/Empty
ASP.NET Core Web App (Model-View-Controller)      mvc              [C#], F#          Web/MVC
ASP.NET Core Web App                              razor            [C#]              Web/MVC/Razor Pages
ASP.NET Core with Angular                         angular          [C#]              Web/MVC/SPA
ASP.NET Core with React.js                        react            [C#]              Web/MVC/SPA
ASP.NET Core with React.js and Redux              reactredux       [C#]              Web/MVC/SPA
ASP.NET Core Web API                              webapi           [C#], F#          Web/WebAPI
global.json file                                  globaljson                         Config
NuGet Config                                      nugetconfig                        Config
Web Config                                        webconfig                          Config
Solution File                                     sln                                Solution
Razor Page                                        page                               Web/ASP.NET
MVC ViewImports                                   viewimports                        Web/ASP.NET
MVC ViewStart                                     viewstart                          Web/ASP.NET

Examples:
    dotnet new mvc --auth Individual
    dotnet new mstest
    dotnet new --help
```

FIGURE 1-5

The command

```
> dotnet new mvc -o WebApp
```

creates a new ASP.NET Core web application using ASP.NET Core MVC. After changing to the `WebApp` folder, build and run the program using

```
> dotnet build
> dotnet run
```

starts the Kestrel server of ASP.NET Core to listen on port 5000. You can open a browser to access the pages returned from this server, as shown in Figure 1-6.

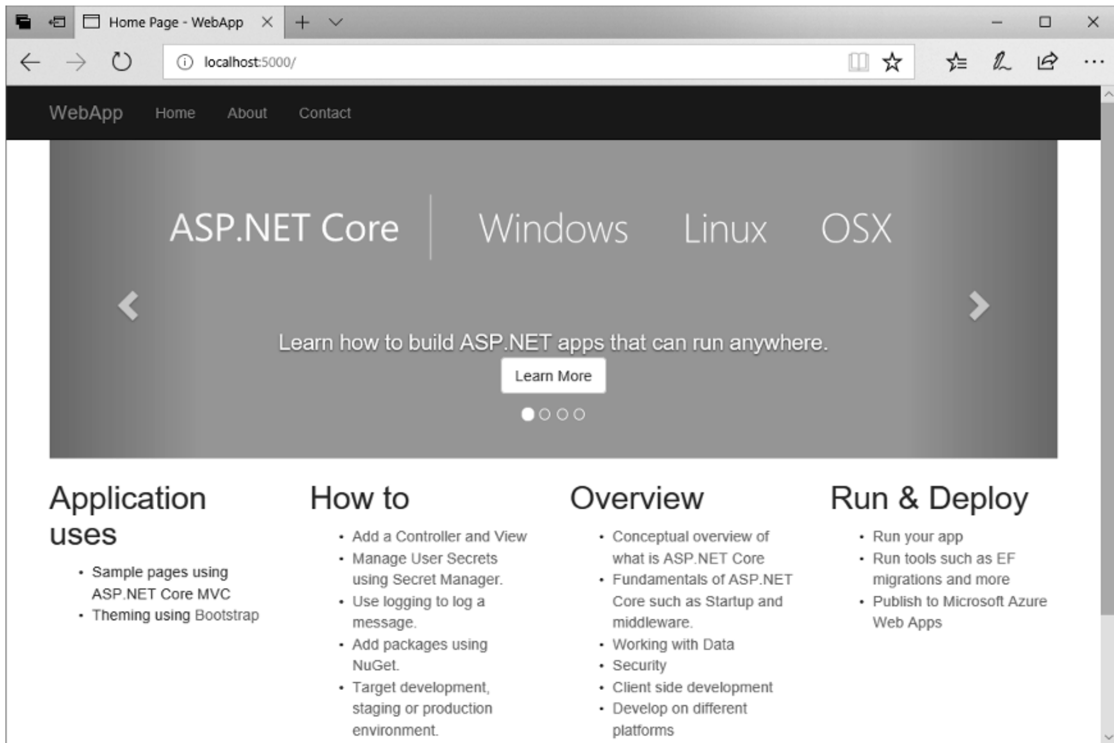

FIGURE 1-6

Publishing the Application

With the dotnet tool you can create a NuGet package and publish the application for deployment. Let's first create a framework-dependent deployment of the application. This reduces the files needed with publishing.

Using the previously created console application, you just need the following command to create the files needed for publishing. The framework is selected by using `-f`, and the release configuration is selected by using `-c`:

```
> dotnet publish -f netcoreapp2.0 -c Release
```

The files needed for publishing are put into the `bin/Release/netcoreapp2.0/publish` directory.

Using these files for publishing on the target system, the runtime is needed as well. You can find the runtime downloads and installation instructions at `https://www.microsoft.com/net/download/`.

Contrary to the .NET Framework where the same installed runtime can be used by different .NET Framework versions (for example, the .NET Framework 4.0 runtime with updates can be used from .NET Framework 4.7, 4.6, 4.5, 4.0… applications), with .NET Core, to run the application, you need the same runtime version.

> **NOTE** *In case your application uses additional NuGet packages, these need to be referenced in the* `csproj` *file, and the libraries need to be delivered with the application. Read Chapter 19 for more information.*

Self-Contained Deployments

Instead of needing to have the runtime installed on the target system, the application can deliver the runtime with it. This is known as *self-contained deployment*.

Depending on the platform, the runtime differs. Thus, with self-contained deployment you need to specify the platforms supported by specifying `RuntimeIdentifiers` in the project file, as shown in the following project file. Here, the runtime identifiers for Windows 10, MacOS, and Ubuntu Linux are specified (project file `SelfContainedHelloWorld/SelfContainedHelloWorld.csproj`):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <PropertyGroup>
    <RuntimeIdentifiers>
      win10-x64;ubuntu-x64;osx.10.11-x64;
    </RuntimeIdentifiers>
  </PropertyGroup>
</Project>
```

> **NOTE** *Get all the runtime identifiers for different platforms and versions from the .NET Core Runtime Identifier (RID) catalog at* `https://docs.microsoft.com/en-us/dotnet/core/rid-catalog`.

Now you can create publish files for all the different platforms:

```
> dotnet publish -c Release -r win10-x64
> dotnet publish -c Release -r osx.10.11-x64
> dotnet publish -c Release -r ubuntu-x64
```

After running these commands, you can find the files needed for publishing in the Release/[win10-x64|osx.10.11-x64|ubuntu-x64]/publish directories. As .NET Core 2.0 is a lot larger, the size needed for publishing was growing. In these directories, you can find platform-specific executables that you can start directly without using the dotnet command.

> **NOTE** *Chapter 19 gives more details on working with the .NET Core CLI and adding NuGet packages, adding projects, creating libraries, working with solution files, and more.*

USING VISUAL STUDIO 2017

Next, let's get into using Visual Studio 2017 instead of the command line. In this section, the most important parts of Visual Studio are covered to get you started. More features of Visual Studio are covered in Chapter 18, "Visual Studio 2017."

Installing Visual Studio 2017

Visual Studio 2017 offers a new installer that should make it easier to install the products you need. With the installer, you can select the Workloads you need for developing applications (see Figure 1-7). To cover all the chapters of the book, install these workloads:

- Universal Windows Platform development
- .NET Desktop development
- ASP.NET and web development
- Azure development
- Mobile development with .NET
- .NET Core cross-platform development

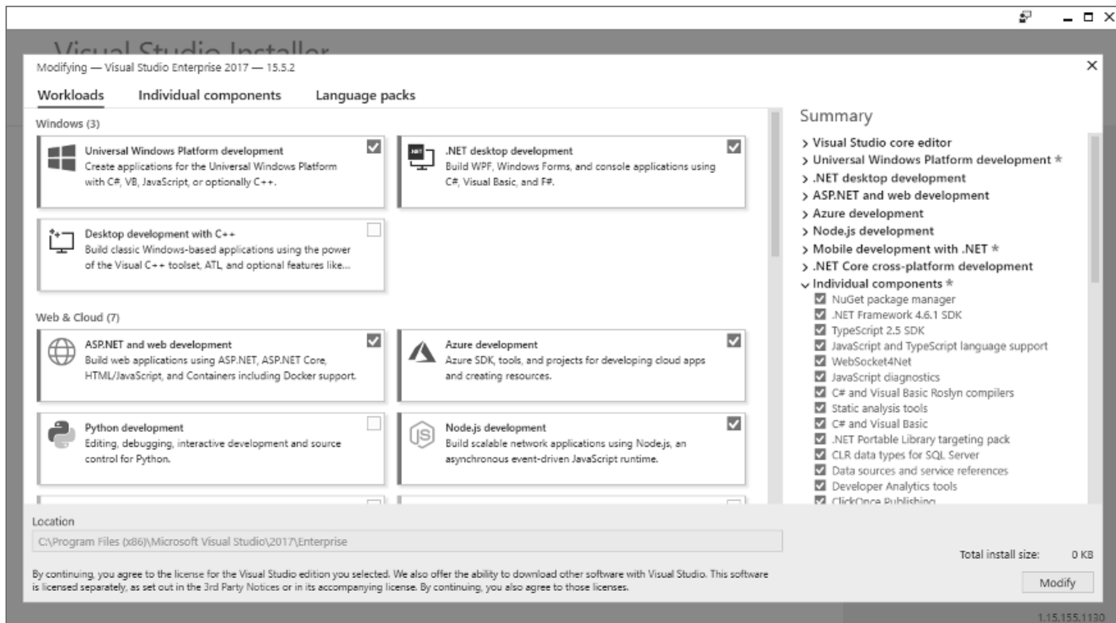

FIGURE 1-7

Creating a Project

You might be overwhelmed by the huge number of menu items and the many options in Visual Studio. To create simple apps in the first chapters of this book, you need only a small subset of the features of Visual Studio. Also, this complete book covers only a part of all the things you can do with Visual Studio. Many features within Visual Studio are offered for legacy applications, as well as for other programming languages.

The first thing you do after starting Visual Studio is create a new project. Select the menu File ➪ New ➪ Project. The dialog shown in Figure 1-8 opens. You see a list of project items that you can use to create new projects.

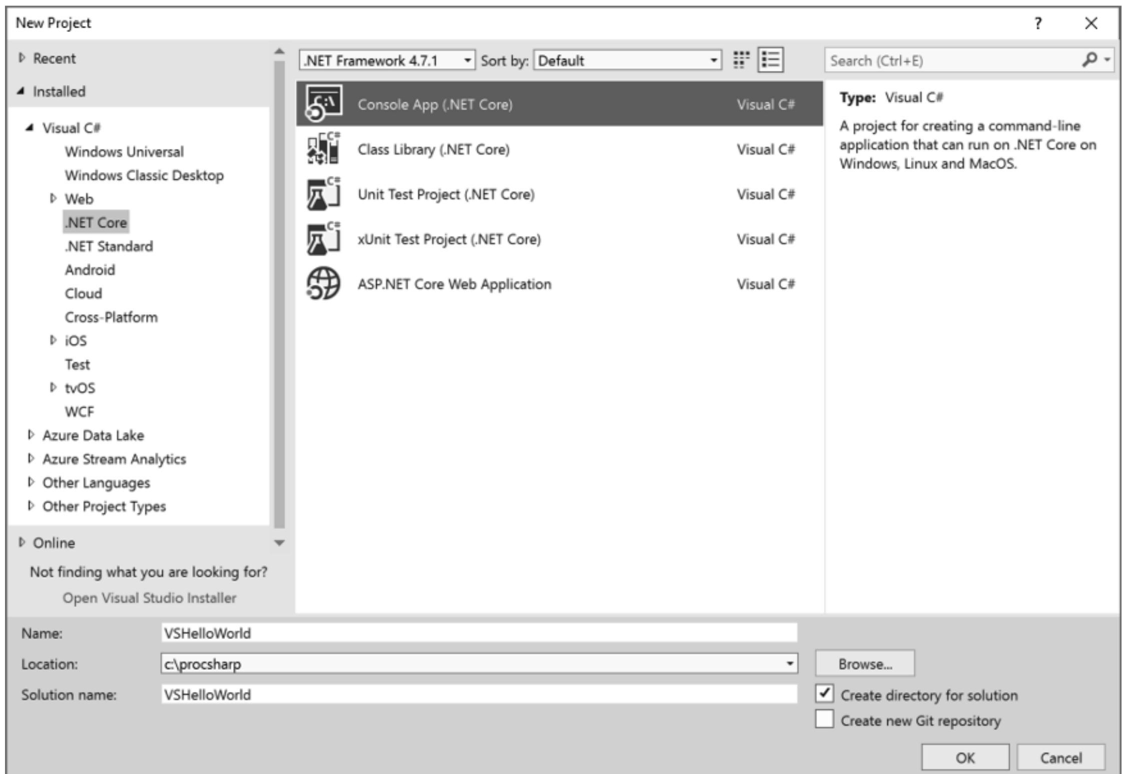

FIGURE 1-8

For this book, you're interested in a subset of the Visual C# project items. With the first chapters of this book, you select the .NET Core category and the project template Console App (.NET Core). On top of the dialog shown in Figure 1-8 you can see where .NET Framework version is selected. Don't be confused, this selection does not apply to .NET Core projects.

In the lower part of this dialog, you can enter the name of the application, chose the folder where to store the project, and enter a name for the solution. Solutions can contain multiple projects.

Clicking the OK button creates a "Hello World!" application.

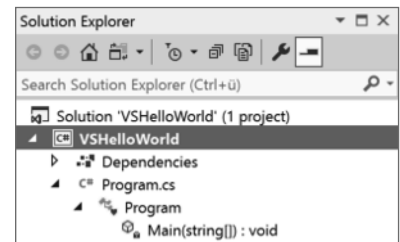

FIGURE 1-9

Working with Solution Explorer

In the Solution Explorer (see Figure 1-9), you can see the solution, the projects belonging to the solution, and the files in the project. You can select a source code file you can get into the classes and class members.

When you select an item in the Solution Explorer and click the right mouse key or press the application key on the keyboard, you open the context menu for the item, as shown in Figure 1-10. The available menus depend on the item you selected and on the features installed with Visual Studio.

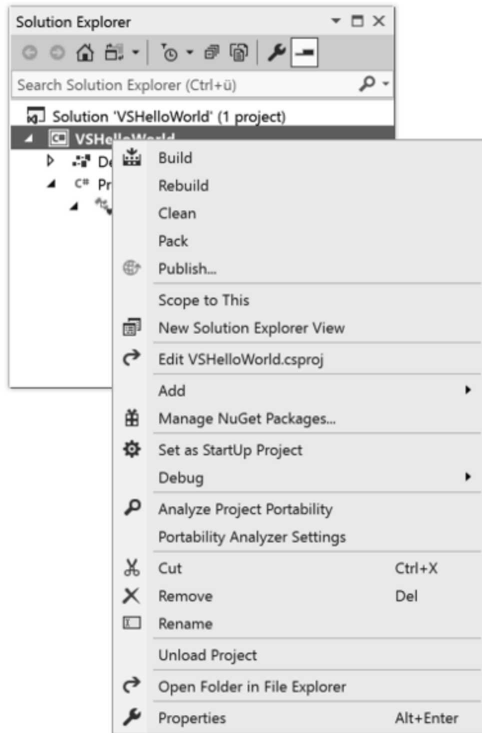

FIGURE 1-10

When you open the context menu for the project, one menu item is to edit the project file. This option opens the project file `VSHelloWorld.csproj` with the same content you've already seen earlier when using the .NET Core CLI.

Configuring Project Properties

You can configure the project properties by selecting the context menu of the project in the Solution Explorer and clicking Properties, or by selecting Project ➪ VSHelloWorld Properties. This opens the view shown in Figure 1-11. Here, you can configure different settings of the project, such as the .NET Core version to use (if you have multiple frameworks installed), build settings, commands that should be invoked during the build process, package configuration, and arguments and environmental variables used while debugging the application. As previously mentioned, with some code samples, C# 7.0 is not enough. You can configure a different version of the C# compiler with the Build category. Clicking the Advanced button opens the Advanced Build Settings dialog (see Figure 1-12). Here, you can configure the version of the C# compiler. This selection goes into the `csproj` project configuration file.

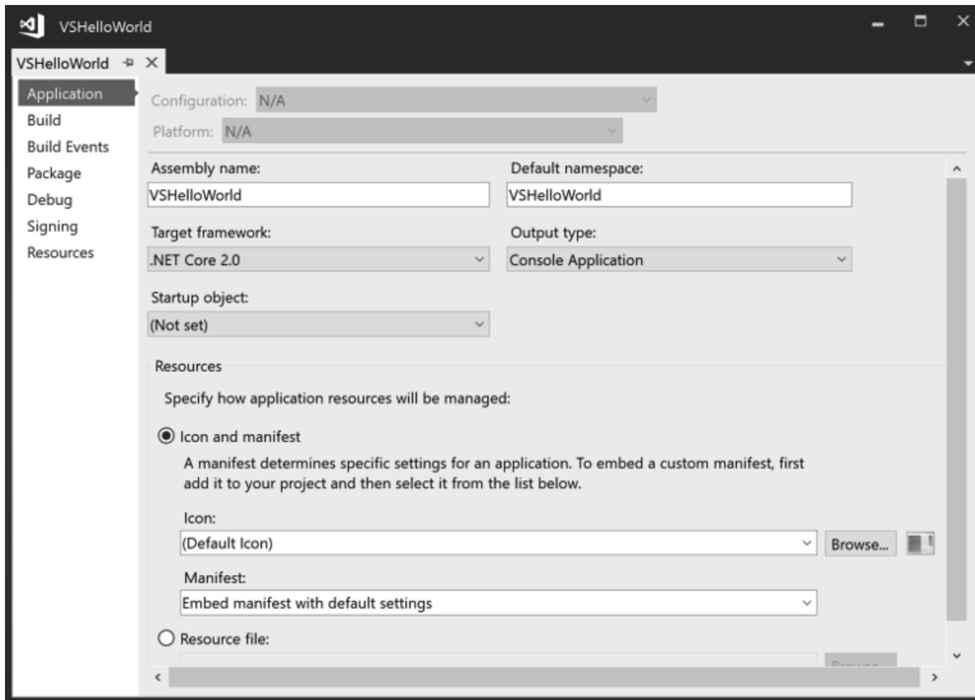

FIGURE 1-11

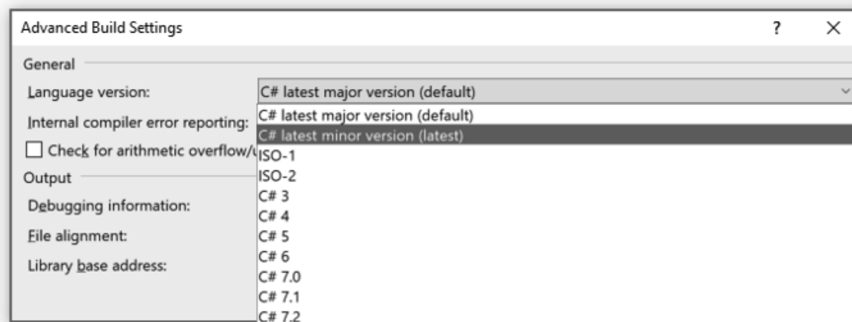

FIGURE 1-12

NOTE *When making a change with the project properties, you need to make sure to select the correct Configuration at the top of the dialog. If you change the version of the C# compiler only with the Debug configuration, building release code will fail when you use newer C# language features. For settings you would like to have with all configurations, select the configuration All Configurations.*

Getting to Know the Editor

The Visual Studio editor is extremely powerful. It offers IntelliSense to offer you available options to invoke methods and properties and completes your typing as you press the Tab button. Compilation takes place while you type, so you can immediately see syntax errors with underlined code. Hovering the mouse pointer over the underlined text brings up a small box that contains the description of the error.

One great productivity feature from the code editor is code snippets. They reduce how much you need to type. Just by typing `cw` and pressing Tab twice in the editor, the editor creates `Console.WriteLine();`. Visual Studio comes with many code snippets that you can see when you select Tools ➪ Code Snippets Manager to open the Code Snippets Manager (see Figure 1-13), where you can select CSharp in the Language field for the code snippets defined with the C# language; select the group Visual C# to see all predefined code snippets for C#.

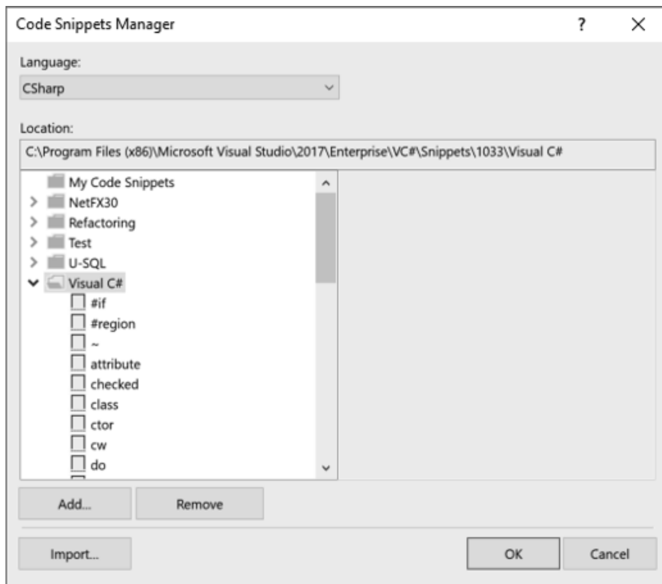

FIGURE 1-13

Building a Project

You compile the project from the menu Build ➪ Build Solution. In case of errors, the Error List window shows errors and warnings. However, the Output window (see Figure 1-14) is more reliable than the Error List. Sometimes the Error List contains older cached information, or it is not that easy to find the error when the list is large. The Output window usually gives great information for many different tools. You open the Output window by selecting View ➪ Output.

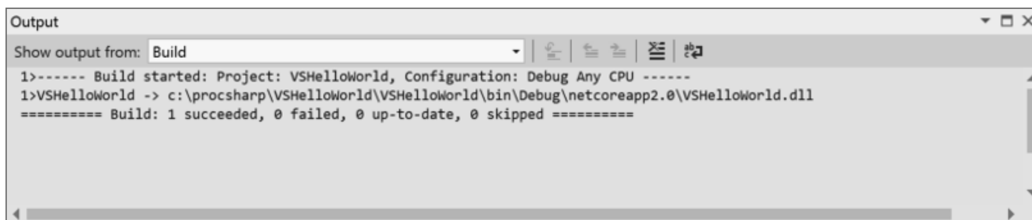

FIGURE 1-14

Running an Application

To run the application, select Debug ➪ Start Without Debugging. This starts the application and keeps the console window opened until you close it.

Remember, you can configure application arguments in the Project Properties selecting the Debug category.

Debugging

To debug an application, you can click the left gray area in the editor to create breakpoints (see Figure 1-15). With breakpoints in place, you can start the debugger by selecting Debug ➪ Start Debugging. When you hit a breakpoint, you can use the Debug toolbar (see Figure 1-16) to step into, over, or out of methods, or you can show the next statement. Hover over variables to see the current values. You also can check the Locals and Watch windows for variables set, and you can change values while the application runs.

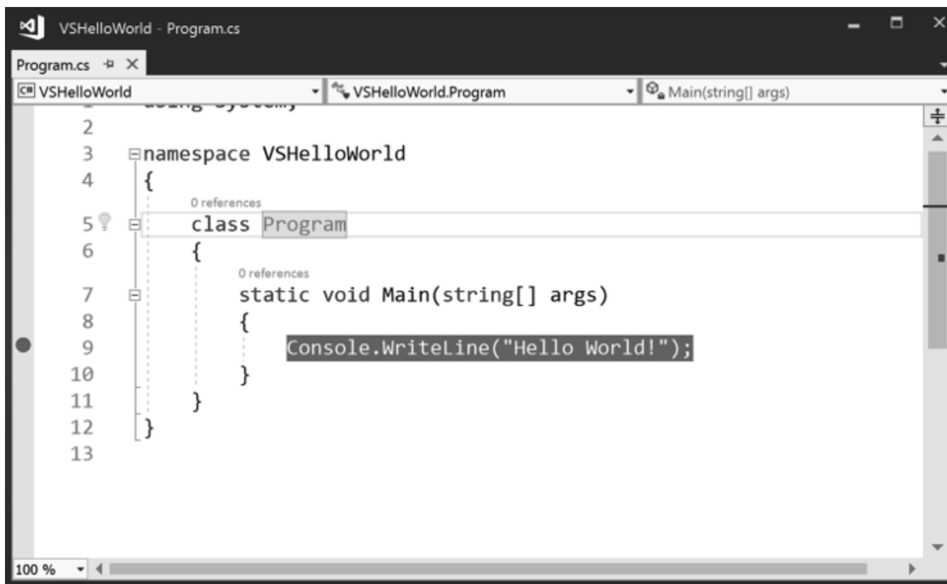

FIGURE 1-15

Now you've seen the parts of Visual Studio that are most important for helping you to survive the first chapters in this book. Chapter 18 takes a deeper look at Visual Studio 2017.

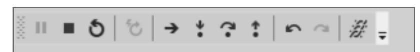

FIGURE 1-16

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 many 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, 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 the different options available here are covered in Chapter 22, "Files and Streams." It's also possible to serialize complete objects in XML or JSON format. Bonus Chapter 2, "XML and JSON," (which you can find online) discusses these options.

To read and write to databases, you can use ADO.NET directly (see Chapter 25, "ADO.NET and Transactions"), or you can use an abstraction layer, Entity Framework Core (Chapter 26, "Entity Framework Core"). Entity Framework Core offers a mapping of object hierarchies to the relations of a database.

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. Entity Framework Core 2.0 has a long list of new features, which are covered in this book.

Windows Apps

For creating Windows apps, the technology of choice should be the Universal Windows Platform. Of course, there are restrictions when this option is not available—for example, if you still need to support older O/S versions like Windows 7. In this case you can use Windows Presentation Foundation (WPF). WPF is not covered in this book, but you can read the previous edition, *Professional C# 6 and .NET Core 1.0,* which has five chapters dedicated to WPF, plus some additional WPF coverage in other chapters.

This book has one focus: developing apps with the Universal Windows Platform (UWP). Compared to WPF, UWP offers a more modern XAML to create the user interface. For example, data binding offers a compiled binding variant where you get errors at compile time instead of not showing the bound data. The application is compiled to native code before it's run on the client systems. And it offers a modern design, which is now called Fluent Design from Microsoft.

> **NOTE** *Creating UWP apps is covered in Chapter 33, "Windows Apps," along with an introduction to XAML, the different XAML controls, and the lifetime of apps. You can create apps with WPF, UWP, and Xamarin by using as much common code as possible by supporting the MVVM pattern. This pattern is covered in Chapter 34, "Patterns with XAML Apps." To create cool looks and style the app, be sure to read Chapter 35, "Styling Windows Apps." Chapter 36, "Advanced Windows Apps," dives into some advanced features of UWP.*

Xamarin

It would have been great if Windows had been a bigger player in the mobile phone market. Then Universal Windows Apps would run on the mobile phones as well. Reality turned out differently, and Windows on the phone is (currently) a thing of the past. However, with Xamarin you can use C# and XAML to create apps on the iPhone and Android. Xamarin offers APIs to create apps on Android and libraries to create apps on iPhone—using the C# code you are used to.

With Android, a mapping layer using Android Callable Wrappers (ACW) and Managed Callable Wrappers (MCW) are used to interop between .NET code and Android's Java runtime. With iOS, an Ahead of Time (AOT) compiler compiles the managed code to native code.

Xamarin.Forms offers XAML code to create the user interface and share as much of the user interface as possible between Android, iOS, Windows, and Linux. XAML only offers UI controls that can be mapped to all platforms. For using specific controls from a platform, you can create platform-specific renderers.

> **NOTE** *Developing with Xamarin and Xamarin.Forms is covered in Chapter 37, "Xamarin.Forms."*

Web Applications

The original introduction of ASP.NET fundamentally changed the web programming model. ASP.NET Core changed it again. ASP.NET Core allows the use of .NET Core for high performance and scalability, and it not only runs on Windows but also on Linux systems.

With ASP.NET Core, ASP.NET Web Forms is no longer covered (ASP.NET Web Forms can still be used and is updated with .NET 4.7).

ASP.NET Core 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 uses C# on the backend.

> **NOTE** *Chapter 30 covers the foundation of ASP.NET Core. Chapter 31 continues building on the foundation and adds using the ASP.NET Core MVC framework.*

Web API

SOAP and WCF fulfilled their duty in the past, and they're not needed anymore. Modern apps make use of REST (Representational State Transfer) and the Web API. Using ASP.NET Core to create a Web API is an option that is a lot easier for communication and fulfills more than 90 percent of requirements by distributed applications. This technology is based on REST, 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, the Universal Windows Platform, and Xamarin.

Creating a 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.

To describe the services, a new standard was defined: the OpenAPI (`https://www.openapis.org`). This standard has its roots with Swagger (`https://swagger.io/`).

> **NOTE** *The ASP.NET Core Web API, Swagger, and more information on microservices are covered in Chapter 32.*

WebHooks and SignalR

For real-time web functionality and bidirectional communication between the client and the server, WebHooks and SignalR are ASP.NET Core technologies available with .NET Core 2.1.

SignalR allows pushing information to connected clients as soon as information is available. SignalR makes use of the WebSocket technology to push information.

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

> **NOTE** *The foundation of SignalR connection management, grouping of connections, and authorization and integration of WebHooks are discussed in Bonus Chapter 3, "WebHooks and SignalR," which you can find online.*

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), Platform as a Service (PaaS), and Functions as a Service (FaaS), 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 Team Services*. Visual Studio Team Services 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. Chapter 18 explains DevOps features that can be used from Visual Studio.

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 128 cores (at the time of this writing, but things change quickly). 128 cores, 2 TB RAM, and 4 TB local SSD belong to the "M-Series" of machines.

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, and many other products.

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 Cosmos DB with different store options like JSON data, relationships, or table storage, and Azure Storage that stores blobs (for example, for images or videos).

App Services can be used to host your web apps and API apps that you are creating with ASP.NET Core.

Microsoft also offers Developer Services in Microsoft Azure. Part of the Developer Services is Visual Studio Team Services. Visual Studio Team Services allows you to manage the source code, automatic builds, tests, and deployments—continuous integration (CI).

Part of the Developer Services is 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 taking to fulfill his or her tasks? With 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.

You also can use *Cognitive Services* that offer functionality to process images, use Bing Search APIs, understand what users say with language services, and more.

Functions as a Service

FaaS is a new concept for cloud service, also known as a *serverless computing* technology. Of course, behind the scenes there's always a server. You just don't pay for reserved CPU and memory as you do with App Services that are used from web apps. Instead the amount you pay is based on consumption—on the number of calls done with some limitations on the memory and time needed for the activity. Azure Functions is one technology that can be deployed using FaaS.

> **NOTE** *In Chapter 29, "Tracing, Logging, and Analytics," you can read about tracing features and learn how to use the Application Insights offering of Microsoft Azure. Chapter 32, "Web API," not only covers creating Web APIs with ASP.NET Core MVC but also shows how the same service functionality can be used from an Azure Function. The Microsoft Bot service as well as Cognitive Services are explained in Bonus Chapter 4, "Bot Framework and Cognitive Services," which you can find online.*

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 2017.

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 to academic and small professional teams. Unlike the Express editions of Visual Studio that previously have been the free editions, this product allows using extensions with Visual Studio.

Visual Studio Professional

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

Unlike 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 a Visual Studio subscription you're entitled to free use of Microsoft Azure up to a specific monthly amount that is contingent on the type of Visual Studio subscription you have.*

> **NOTE** *Chapter 18 includes details on using several features of Visual Studio 2017. Chapter 28, "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 for Mac

Visual Studio for Mac originates in the Xamarin Studio, but now it offers a lot more than the earlier product. For example, the editor shares code with Visual Studio, so you're soon familiar with it. With Visual Studio for Mac you can not only create Xamarin apps, but you also can create ASP.NET Core apps that run on Windows, Linux, and the Mac. With many chapters of this book, you can use Visual Studio for Mac. Exceptions are the chapters covering the Universal Windows Platform, which requires Windows to run the app and also to develop the app.

Visual Studio Code

Visual Studio Code is a completely different development tool compared to the other Visual Studio editions. While Visual Studio 2017 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 UWP and Xamarin applications, and you also don't have access to the features covered in Chapter 18, "Visual Studio 2017." 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 and .NET Core, and you saw how to create and run a Hello World application with all these environments with and 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.

Chapter 2 dives fast into the syntax of C#. You learn variables, implement program flows, organize your code into namespaces, and more.