CHAPTER 2

Building C# Applications

As a C# programmer, you can choose from among numerous tools to build .NET applications. The tool (or tools) you select will be based primarily on three factors: any associated costs, the OS you are using to develop the software, and the computing platforms you are targeting. The point of this chapter is to provide a survey of the most common integrated development environments (IDEs) that support the C# language. Do understand that this chapter will *not* go over every single detail of each IDE; it will give you enough information to select your programming environment as you work through this text and give you a foundation to build on.

The first part of this chapter will examine a set of IDEs from Microsoft that enable development of .NET applications on a Windows operating system (7, 8.*x*, and 10). As you will see, some of these IDEs can be used to build Windows-centric applications only, while others support the construction of C# apps for alternative operating systems and devices (such as macOS, Linux, or Android). The latter part of this chapter will then examine some IDEs that can *run* on a non-Windows OS. This enables developers to build C# programs using Apple computers as well as Linux distributions.

Note This chapter will give you an overview of a good number of IDEs. However, this book will assume you are using the (completely free) Visual Studio 2017 Community IDE. If you want to build your applications on a different OS (macOS or Linux), this chapter will guide you in the right direction; however, your IDE will differ from the various screenshots in this text.

Building .NET Applications on Windows

As you will see over the course of this chapter, you can choose from a variety of IDEs to build C# applications; some come from Microsoft, and others come from third-party (many of which are open source) vendors. Now, despite what you might be thinking, many Microsoft IDEs are completely free. Thus, if your primary interest is to build .NET software on the Windows operating system (7, 8.*x*, or 10), you will find the following major options:

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

The Express editions have been removed, leaving three versions of Visual Studio 2017. The Community and Professional editions are *essentially* the same, with the main technical difference being that Professional has CodeLens and Community does not. The more significant difference is in the licensing model.

A. Troelsen and P. Japikse, *Pro C# 7*, https://doi.org/10.1007/978-1-4842-3018-3_2

Community is licensed for open source, academic, and small-business uses. Professional is licensed for enterprise development. As one would expect, the Enterprise edition has many additional features above the Professional edition.

■ **Note** For specific licensing details, please go to https://www.visualstudio.com. Licensing Microsoft products can be complex, and this book does not cover the details. For the purposes of writing (and following along with) this book, Community is legal to use.

Each IDE ships with sophisticated code editors, key database designers, integrated visual debuggers, GUI designers for desktop and web applications, and so forth. Since they all share a common core set of features, the good news is that it is easy to move between them and feel quite comfortable with their basic operation.

Installing Visual Studio 2017

Before using Visual Studio 2017 to develop, execute, and debug C# applications, you need to get it installed. The installation experience is dramatically different from previous versions and is worth discussing in more detail.

■ **Note** You can download Visual Studio 2017 Community from https://www.visualstudio.com/downloads.

The Visual Studio 2017 installation process is now broken down into application-type workloads. This allows you to install just the components you need for the work you plan on doing. For example, if you are going to build web applications, you would install the "ASP.NET and web development" workload.

Another (extremely) significant change is that Visual Studio 2017 supports true side-by-side installation. Note that I am not referring to just previous versions of Visual Studio but to Visual Studio 2017 itself! On my main workstation, I have Visual Studio 2017 Enterprise installed. For this book, I will be using Visual Studio Community. With Visual Studio 2015 (and the previous edition of this book), I had to use a different machine than the one I use to service clients. Now, it's all on the same machine. If you have Professional or Enterprise supplied by your employer, you can still install the Community edition to work on the open source projects (or the code in this book).

When you launch the installer for Visual Studio 2017 Community, you are presented with the screen shown in Figure 2-1. This screen has all of the workloads available, the option to select individual components, and a summary on the right side showing what has been selected. Notice the red warning at the bottom stating "A nickname must be provided to disambiguate this install." This is because I have other installs of Visual Studio 2017 on my machine. If this is your first install, you won't see this.

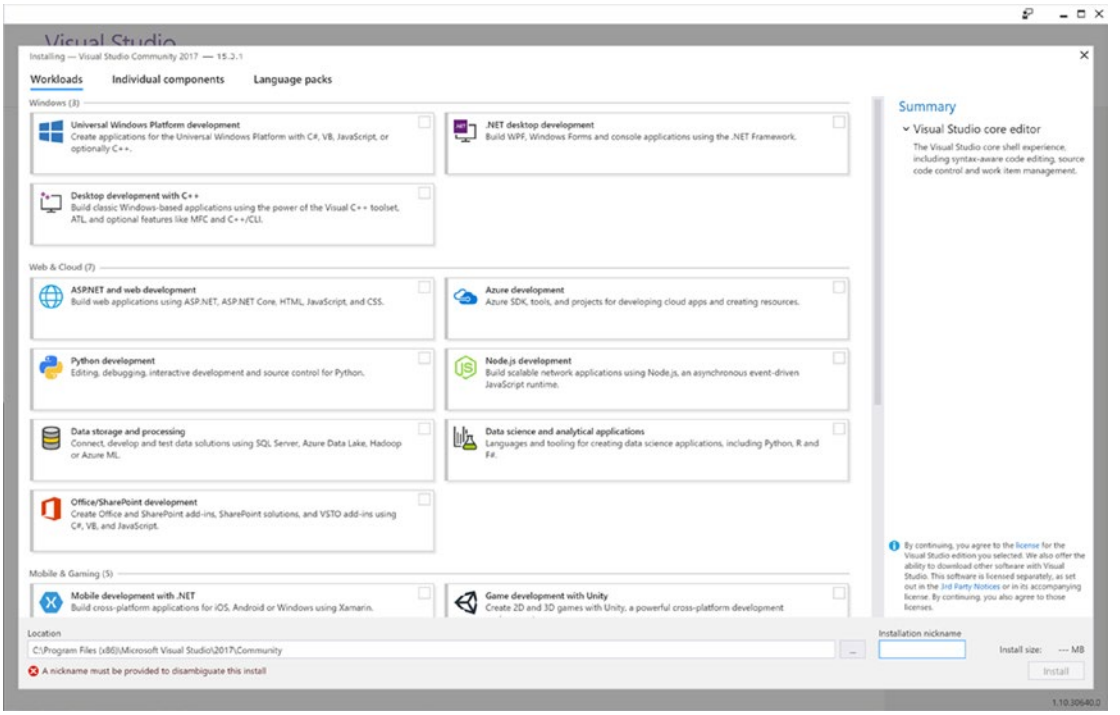

Figure 2-1. *The new Visual Studio installer*

For this book, you will want to install the following workloads:

- .NET desktop development
- ASP.NET and web development
- Data storage and processing
- .NET Core cross-platform development

I also had to add an installation nickname, ProC#. These selections are shown in Figure 2-2.

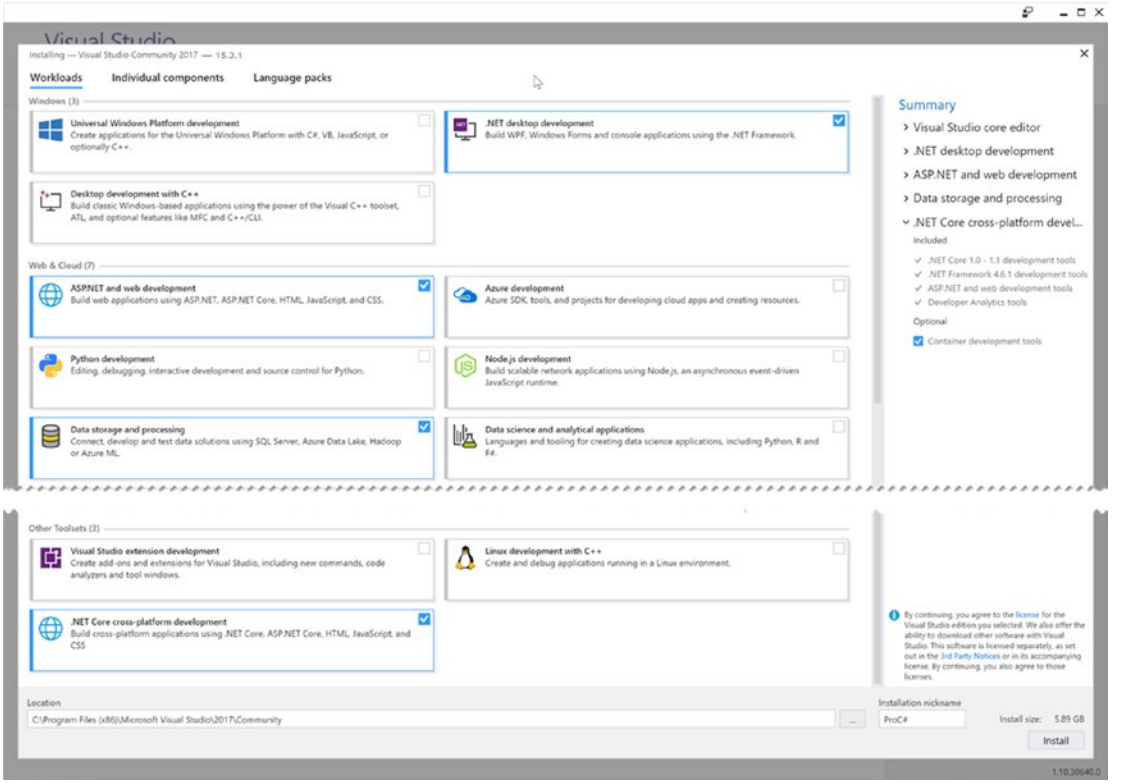

Figure 2-2. *The selected workloads*

Select Individual Components from the top of the installer, and select the following additional items:

- .NET Core runtime
- .NET Framework 4.6.2 SDK
- .NET Framework 4.6.2 targeting pack
- .NET Framework 4.7 SDK
- .NET Framework 4.7 targeting pack
- Class Designer
- Testing tools core features
- Visual Studio Tools for Office (VSTO)

Once you have all of them selected, click Install. This will provide you with everything you need to work through the examples in this book, including the new section on .NET Core.

Taking Visual Studio 2017 for a Test-Drive

Visual Studio 2017 is a one-stop shop for software development with the .NET platform and C#. Let's take a quick look at Visual Studio by building a simple Windows console application.

Building .NET Applications

To get your feet wet, let's take some time to build a simple C# application and keep in mind that the topics illustrated here will be useful for all editions of Visual Studio.

The New Project Dialog Box and C# Code Editor

Now that you have installed Visual Studio, activate the File ➤ New Project menu option. As you can see in Figure 2-3, this IDE has support for console apps, WPF/Windows Forms apps, Windows services, and many more. To start, create a new C# Console Application project named SimpleCSharpConsoleApp, making sure to change the Framework version to 4.7.

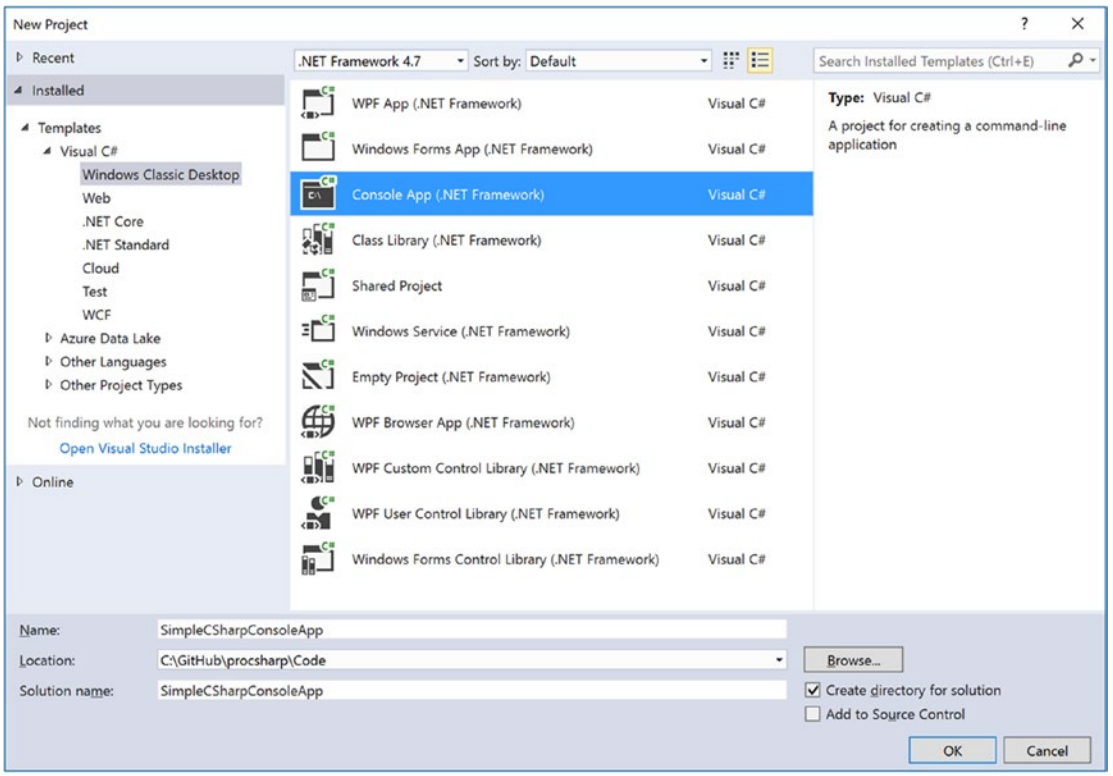

Figure 2-3. *The New Project dialog box*

As you can see from Figure 2-3, Visual Studio is capable of creating a variety of application types, including Windows Desktop, Web, .NET Core, and many more. These will be covered throughout this book.

■ **Note** If you don't see an option for .NET Framework 4.7, you will need to install the 4.7 Developer Pack, available at `https://www.microsoft.com/en-us/download/details.aspx?id=55168`. You can also click the "Install other frameworks…" option in drop-down list to get the Developer Pack.

Once the project has been created, you will see that the initial C# code file (named `Program.cs`) has been opened in the code editor. Add the following C# code to your `Main()` method. You'll notice as you type that IntelliSense will kick in as you apply the dot operator.

```
static void Main(string[] args)
{
  // Set up Console UI (CUI)
  Console.Title = "My Rocking App";
  Console.ForegroundColor = ConsoleColor.Yellow;
  Console.BackgroundColor = ConsoleColor.Blue;
  Console.WriteLine("************************************");
  Console.WriteLine("***** Welcome to My Rocking App *****");
  Console.WriteLine("************************************");
  Console.BackgroundColor = ConsoleColor.Black;

  // Wait for Enter key to be pressed.
  Console.ReadLine();
}
```

Here, you are using the `Console` class defined in the `System` namespace. Because the `System` namespace has been automatically included at the top of your file via a `using` statement, you have no need to qualify the namespace before the class name (e.g., `System.Console.WriteLine()`). This program does not do anything too interesting; however, note the final call to `Console.ReadLine()`. This is in place simply to ensure the user must press a key to terminate the application. If you did not do this, the program would disappear almost instantly when debugging the program!

Using C# 7.1 Features

At the time of this writing, Visual Studio doesn't have direct support for creating C# 7.1 projects. To use the new features, there are two options. The first is to update the project file manually, and the second is to let the Visual Studio help (in the form of the quick-fix light bulb) update your project file for you. While the latter sounds easiest, it's not currently reliable, but I am certain it will get better in upcoming Visual Studio releases.

To update the project file, open the `SimpleCSharpConsoleApp.csproj` file in any text editor (except Visual Studio), and update the Debug and Release property groups to the following:

```
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
    <PlatformTarget>AnyCPU</PlatformTarget>
    <DebugSymbols>true</DebugSymbols>
    <DebugType>full</DebugType>
    <Optimize>false</Optimize>
    <OutputPath>bin\Debug\</OutputPath>
    <DefineConstants>DEBUG;TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
    <LangVersion>7.1</LangVersion>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
    <PlatformTarget>AnyCPU</PlatformTarget>
    <DebugType>pdbonly</DebugType>
    <Optimize>true</Optimize>
```

```
    <OutputPath>bin\Release\</OutputPath>
    <DefineConstants>TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
    <LangVersion>7.1</LangVersion>
  </PropertyGroup>
```

Running and Debugging Your Project

Now, to run your program and see the output, you can simply press the Ctrl+F5 keyboard command (which is also accessed from the Debug ➤ Start Without Debugging menu option). Once you do, you will see a Windows console window pop on the screen with your custom (and colorful) message. Be aware that when you "run" your program, you bypass the integrated debugger.

If you need to debug your code (which will certainly be important when building larger programs), your first step is to set breakpoints at the code statement you want to examine. Although there isn't much code in this example, set a breakpoint by clicking the leftmost gray bar of the code editor (note that breakpoints are marked with a red dot icon; see Figure 2-4).

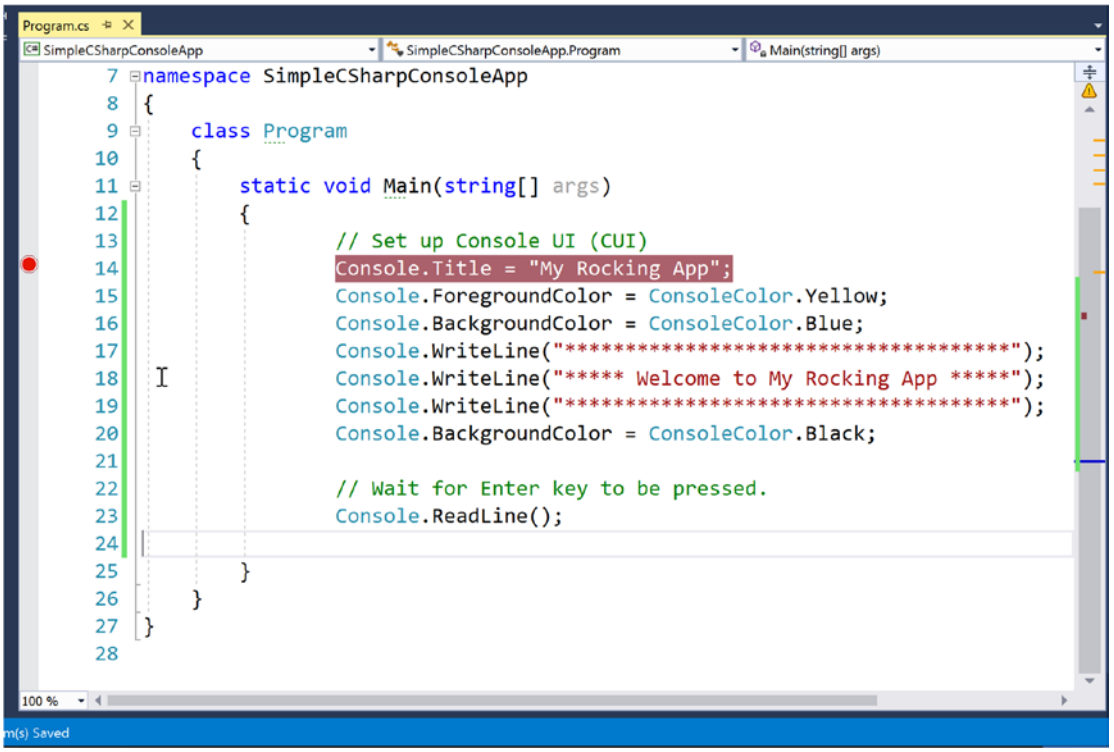

Figure 2-4. *Setting breakpoints*

If you now press the F5 key (or use the Debug ➤ Start Debugging menu option or click the green arrow with Start next to it in the toolbar), your program will halt at each breakpoint. As you would expect, you can interact with the debugger using the various toolbar buttons and menu options of the IDE. Once you have evaluated all breakpoints, the application will eventually terminate once `Main()` has completed.

■ **Note** Microsoft IDEs have sophisticated debuggers, and you will learn about various techniques over the chapters to come. For now, be aware that when you are in a debugging session, a large number of useful options will appear under the Debug menu. Take a moment to verify this for yourself.

Solution Explorer

If you look at the right of the IDE, you will see a window named Solution Explorer, which shows you a few important things. First, notice that the IDE has created a solution with a single project (see Figure 2-5). This can be confusing at first, as they both have been given the same name (SimpleCSharpConsoleApp). The idea here is that a "solution" can contain multiple projects that all work together. For example, your solution might include three class libraries, one WPF application, and one WCF web service. The earlier chapters of this book will always have a single project; however, when you build some more complex examples, you'll see how to add new projects to your initial solution space.

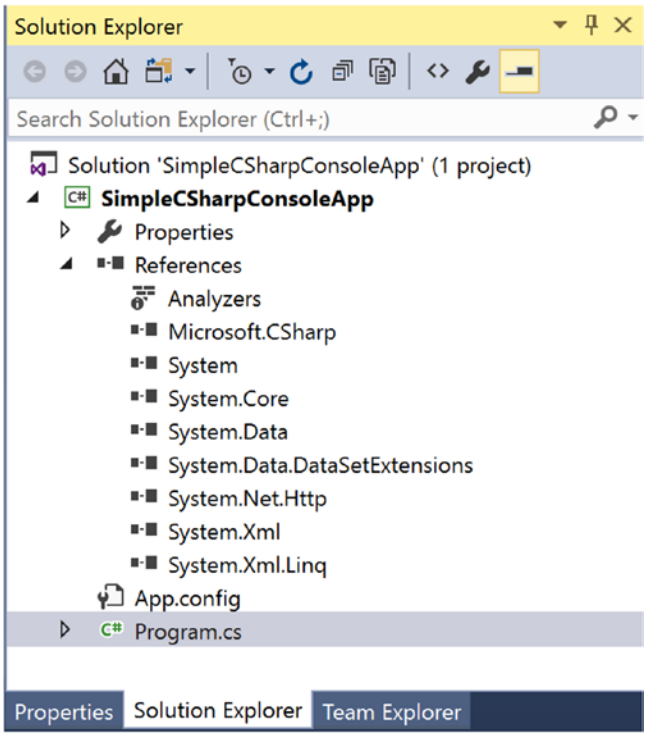

Figure 2-5. *Solution Explorer*

■ **Note** Be aware that when you select the topmost solution in the Solution Explorer window, the IDE's menu system will show you a different set of choices than when you select a project. If you ever find yourself wondering where a certain menu item has disappeared to, double-check you did not accidentally select the wrong node.

You will also notice a References icon. You can use this node when your application needs to reference additional .NET libraries beyond what are included for a project type by default. Because you have created a C# Console Application project, you will notice a number of libraries have been automatically added such as `System.dll`, `System.Core.dll`, `System.Data.dll`, and so forth (note the items listed under the References node don't show the `.dll` file extension). You will see how to add libraries to a project shortly.

■ **Note** Recall from Chapter 1 that all .NET projects have access to a foundational library named `mscorlib.dll`. This library is so necessary that it is not even listed explicitly in Solution Explorer.

The Object Browser

If you were to right-click any library under the References node and select View in Object Browser, you will open the integrated Object Browser (you can also open this using the View menu). Using this tool, you can see the various namespaces in an assembly, the types in a namespace, and the members of each type. Figure 2-6 shows some namespaces of the always-present `mscorlib.dll` assembly.

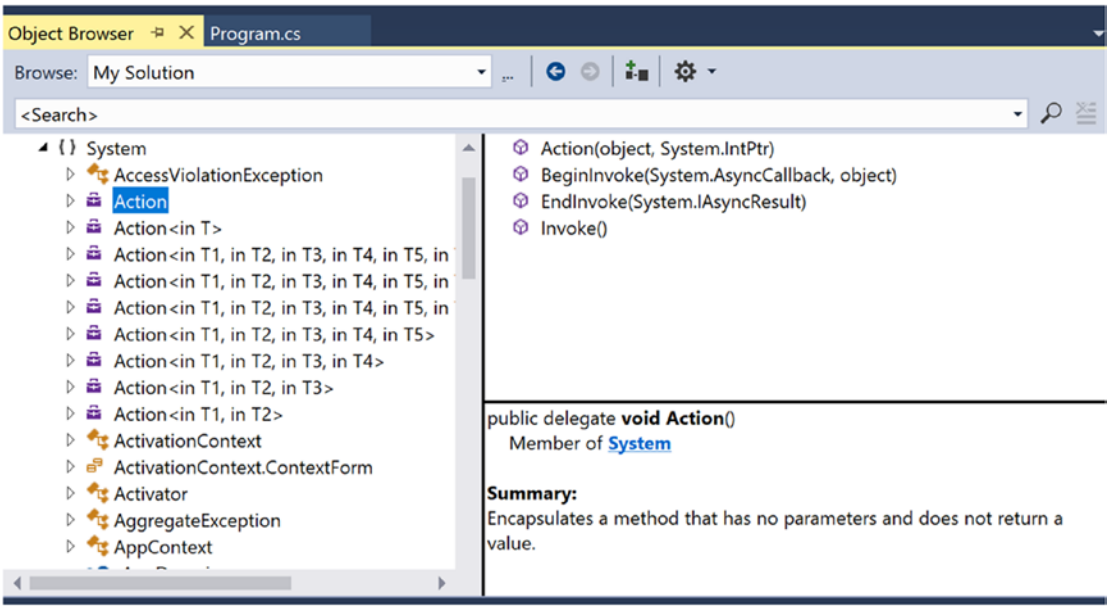

Figure 2-6. *The Object Browser*

This tool can be useful when you want to see the internal organization of a .NET library as well as when you want to get a brief description of a given item. Also notice the <Search> bar at the top of the window. This can be helpful when you know the name of a type you want to use but have no idea where it might be located. On a related note, keep in mind that the search feature will search only the libraries used in your current solution by default (you can search the entire .NET Framework by changing the selection in the Browse drop-down box).

Referencing Additional Assemblies

To continue your test, let's add an assembly (aka code library) not automatically included in a Console Application project. To do so, right-click the References tab of Solution Explorer and select Add Reference (or select the Project ➤ Add Reference menu option). From the resulting dialog box, find a library named `System.Windows.Forms.dll` (again, you won't see the file extension here) and check it off (Figure 2-7).

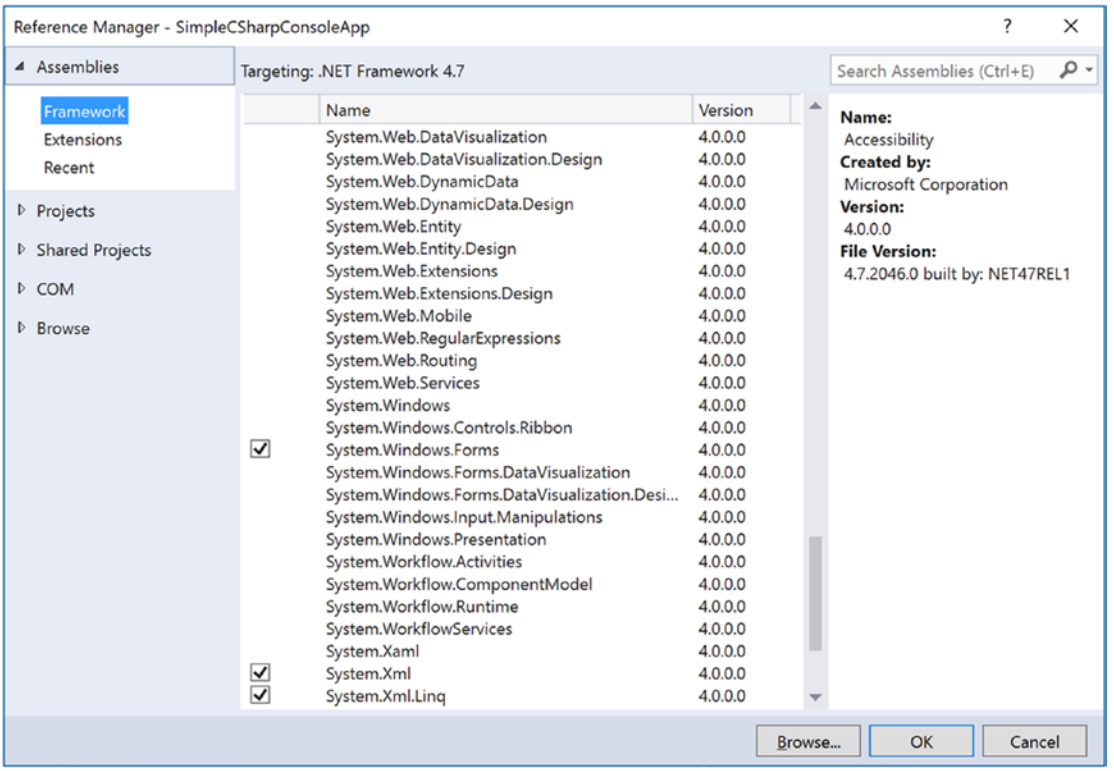

Figure 2-7. *Adding references*

Once you click the OK button, this new library is added to your reference set (you'll see it listed under the References node). As explained in Chapter 1, however, referencing a library is only the first step. To use the types in a given C# code file, you need to define a `using` statement. Add the following line to the `using` directives in your code file:

```
using System.Windows.Forms;
```

Then add the following line of code directly after the call to `Console.ReadLine()` in your `Main()` method:

```
MessageBox.Show("All done!");
```

When you run or debug your program once again, you will find a simple message box appears before the program terminates.

Viewing Project Properties

Next, notice an icon named Properties within Solution Explorer. When you double-click this item, you are presented with a sophisticated project configuration editor. For example, in Figure 2-8, notice how you can change the version of the .NET Framework you are targeting for the solution.

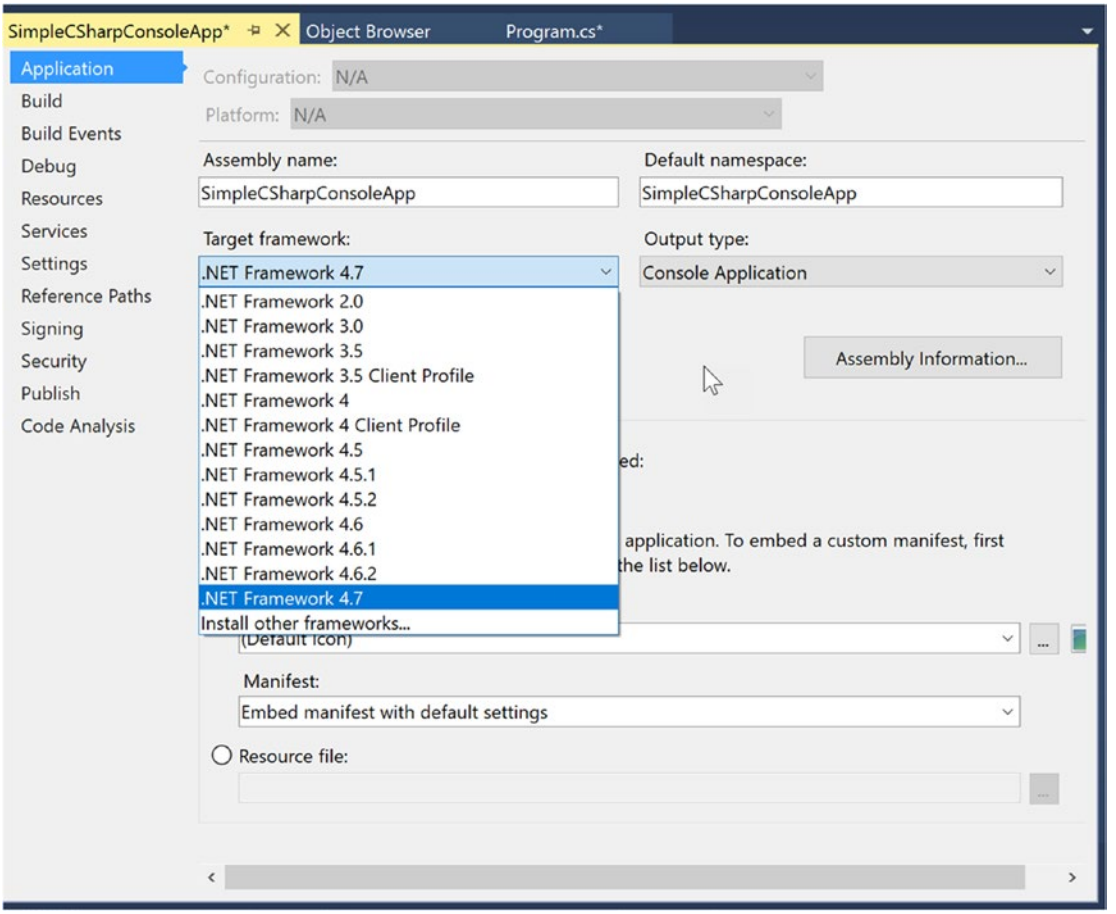

Figure 2-8. *The Project Properties window*

You will see various aspects of the Project Properties window as you progress through this book. However, if you take some time to poke around, you will see that you can establish various security settings, strongly name your assembly (see Chapter 14), deploy your application, insert application resources, and configure pre- and post-build events.

The Visual Class Designer

Visual Studio also gives you the ability to design classes and other types (such as interfaces or delegates) in a visual manner. The Class Designer utility allows you to view and modify the relationships of the types (classes, interfaces, structures, enumerations, and delegates) in your project. Using this tool, you are able to visually add (or remove) members to (or from) a type and have your modifications reflected in the corresponding C# file. Also, as you modify a given C# file, changes are reflected in the class diagram.

To access the visual type designer tools, the first step is to insert a new class diagram file. To do so, activate the Project ➤ Add New Item menu option and locate the Class Diagram type (Figure 2-9).

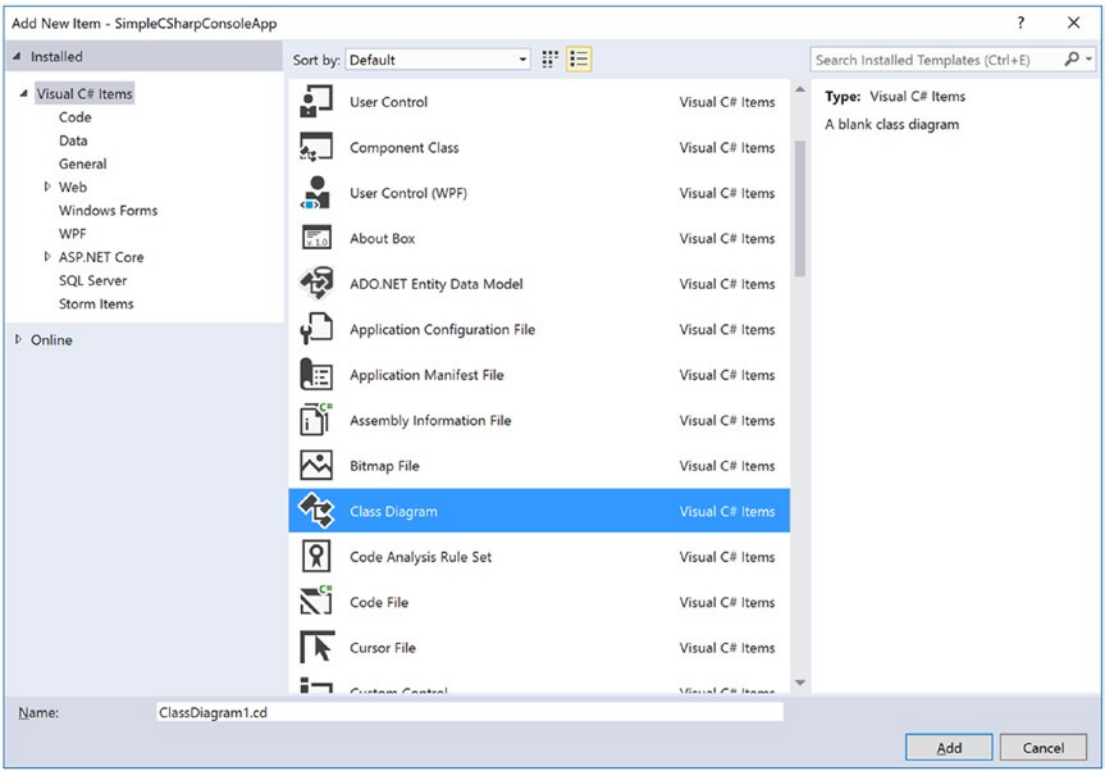

Figure 2-9. *Inserting a class diagram file into the current project*

Initially, the designer will be empty; however, you can drag and drop files from your Solution Explorer window on the surface. For example, once you drag `Program.cs` onto the designer, you will find a visual representation of the `Program` class. If you click the arrow icon for a given type, you can show or hide the type's members (see Figure 2-10).

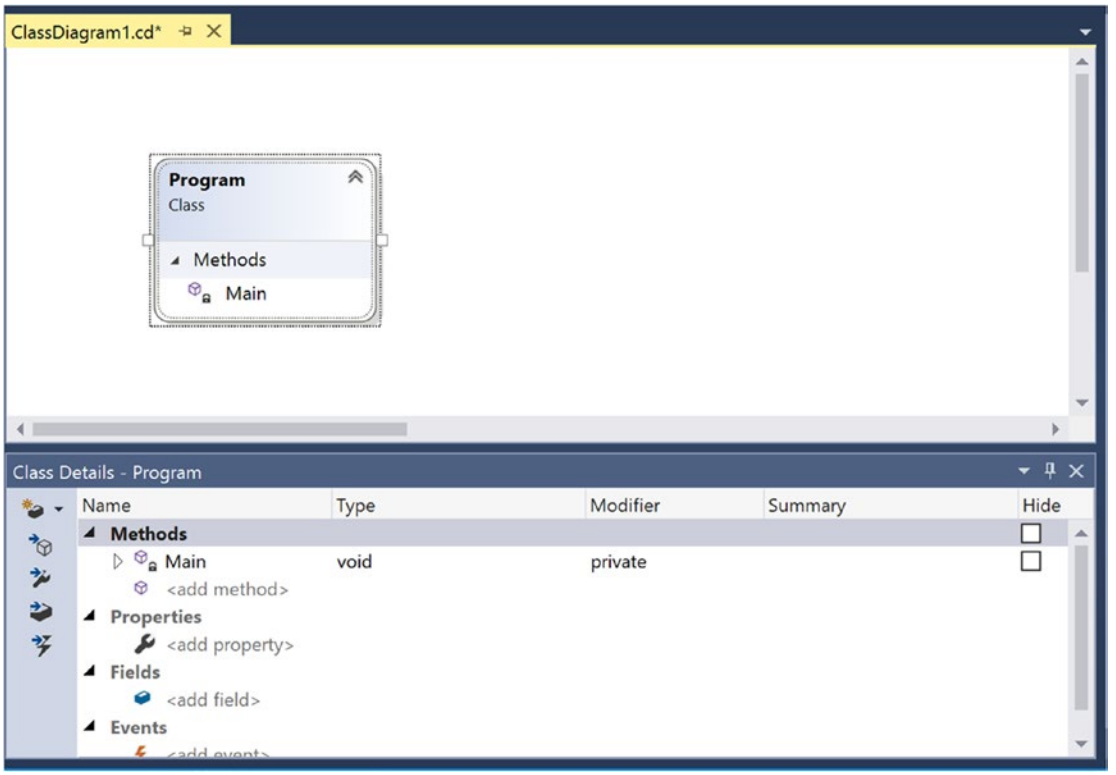

Figure 2-10. *The Class Diagram viewer*

■ **Note** Using the Class Designer toolbar, you can fine-tune the display options of the designer surface.

The Class Designer utility works in conjunction with two other aspects of Visual Studio: the Class Details window (activated using the View ➤ Other Windows menu) and the Class Designer Toolbox (activated using the View ➤ Toolbox menu item). The Class Details window not only shows you the details of the currently selected item in the diagram but also allows you to modify existing members and insert new members on the fly (see Figure 2-11).

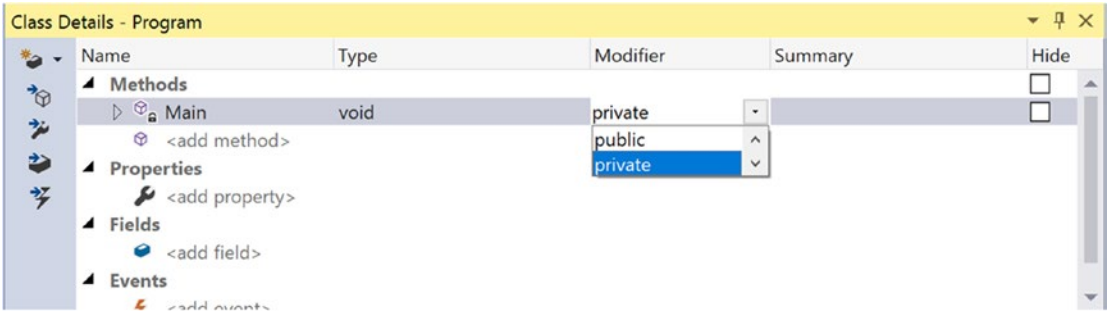

Figure 2-11. *The Class Details window*

The Class Designer Toolbox, which can also be activated using the View menu, allows you to insert new types (and create relationships between these types) into your project visually (see Figure 2-12). (Be aware you must have a class diagram as the active window to view this toolbox.) As you do so, the IDE automatically creates new C# type definitions in the background.

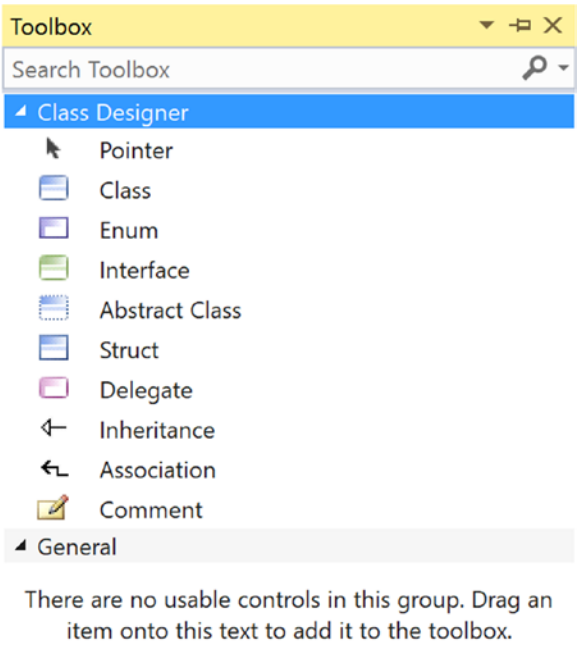

Figure 2-12. *The Class Designer Toolbox*

By way of example, drag a new class from the Class Designer Toolbox onto your Class Designer. Name this class `Car` in the resulting dialog box. This will result in the creation of a new C# file named `Car.cs` that is automatically added to your project. Now, using the Class Details window, add a public `string` field named `PetName` (see Figure 2-13).

Figure 2-13. *Adding a field with the Class Details window*

If you now look at the C# definition of the `Car` class, you will see it has been updated accordingly (minus the additional code comments shown here).

```
public class Car
{
   // Public data is typically a bad idea; however,
   // it keeps this example simple.
   public string PetName;
}
```

Now, activate the designer file once again and drag another new class onto the designer and name it `SportsCar`. Select the Inheritance icon from the Class Designer Toolbox and click the top of the `SportsCar` icon. Next, click the mouse on top of the `Car` class icon. If you performed these steps correctly, you have just derived the `SportsCar` class from `Car` (see Figure 2-14).

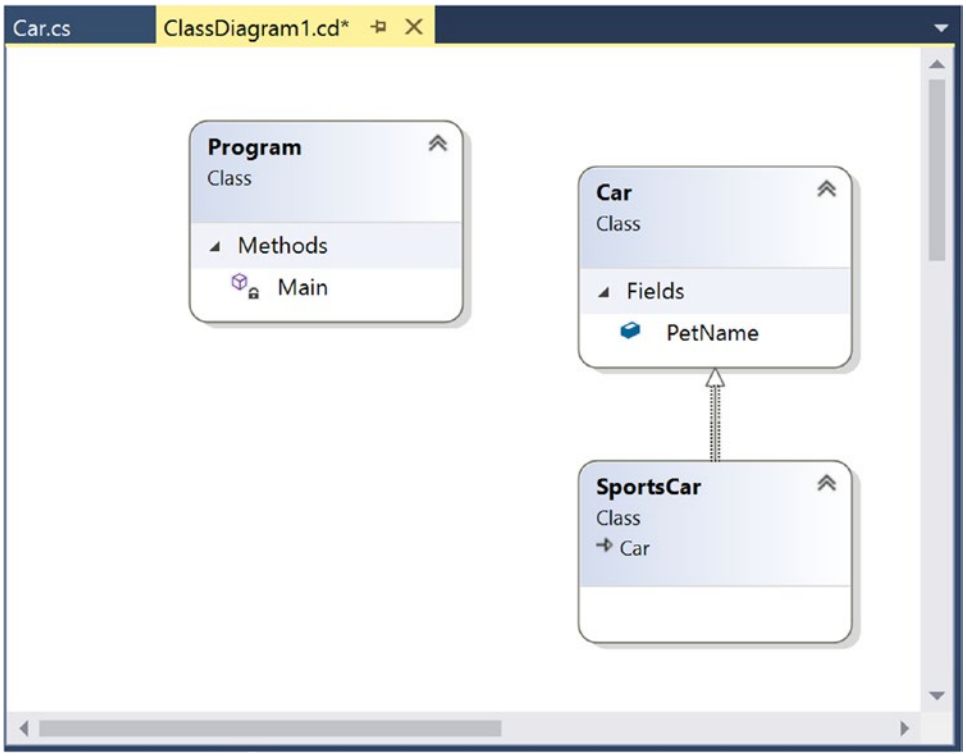

Figure 2-14. *Visually deriving from an existing class*

■ **Note** The concept of inheritance will be fully examined in Chapter 6.

To complete this example, update the generated `SportsCar` class with a public method named `GetPetName()`, authored as follows:

```
public class SportsCar : Car
{
   public string GetPetName()
   {
     PetName = "Fred";
     return PetName;
   }
}
```

As you would expect, the visual type designer is one of the many features of Visual Studio Community. As mentioned earlier, this edition of the book will assume you are using Visual Studio Community as your IDE of choice. Over the chapters to come, you will learn many more features of this tool.

■ **Source Code** You can find the SimpleCSharpConsoleApp project in the `Chapter 2` subdirectory.

Visual Studio 2017 Professional

As mentioned earlier, the main difference between Community and Professional editions is the allowable usage scenarios. If you are currently employed as a software engineer, the chances are good your company has purchased a copy of this edition for you as your tool of choice.

Visual Studio 2017 Enterprise

To wrap up your examination of the Visual Studio editions that run on Windows, let's take a quick look at Visual Studio 2017 Enterprise. Visual Studio 2017 Enterprise has all the same features found in Visual Studio Professional, as well as additional features geared toward corporate-level collaborative development and full support for cross-platform mobile development with Xamarin.

I won't be saying much more about Visual Studio 2017 Enterprise edition. For the purposes of this book, any of the three versions (Community, Professional, and Enterprise) will work.

■ **Note** You can find a side-by-side comparison of Community vs. Professional vs. Enterprise at `https://www.visualstudio.com/vs/compare/`.

The .NET Framework Documentation System

The final aspect of Visual Studio you *must* be comfortable with from the outset is the fully integrated help system. The .NET Framework documentation is extremely good, very readable, and full of useful information. Given the huge number of predefined .NET types (which number well into the thousands), you must be willing to roll up your sleeves and dig into the provided documentation. If you resist, you are doomed to a long, frustrating, and painful existence as a .NET developer.

You can view the .NET Framework SDK documentation at the following web address:

```
https://docs.microsoft.com/en-us/dotnet/
```

■ **Note** It would not be surprising if Microsoft someday changes the location of the online .NET Framework Class Library documentation. If this is the case, a web search for the same topic (*.NET Framework Class Library documentation*) should quickly help you find the current location.

Once you are on this main page, click "Switch to the Library TOC view." This will change the page to a view that is easier to navigate. Locate the .NET Development node in the TOC, and click the arrow to expand the TOC. Next, click the arrow next to the .NET Framework 4.7, 4.6, and 4.5 node. Finally, click the ".NET Framework class library" entry. At this point, you can use the tree navigation window to view each namespace, type, and member of the platform. See Figure 2-15 for an example of viewing the types of the `System` namespace.

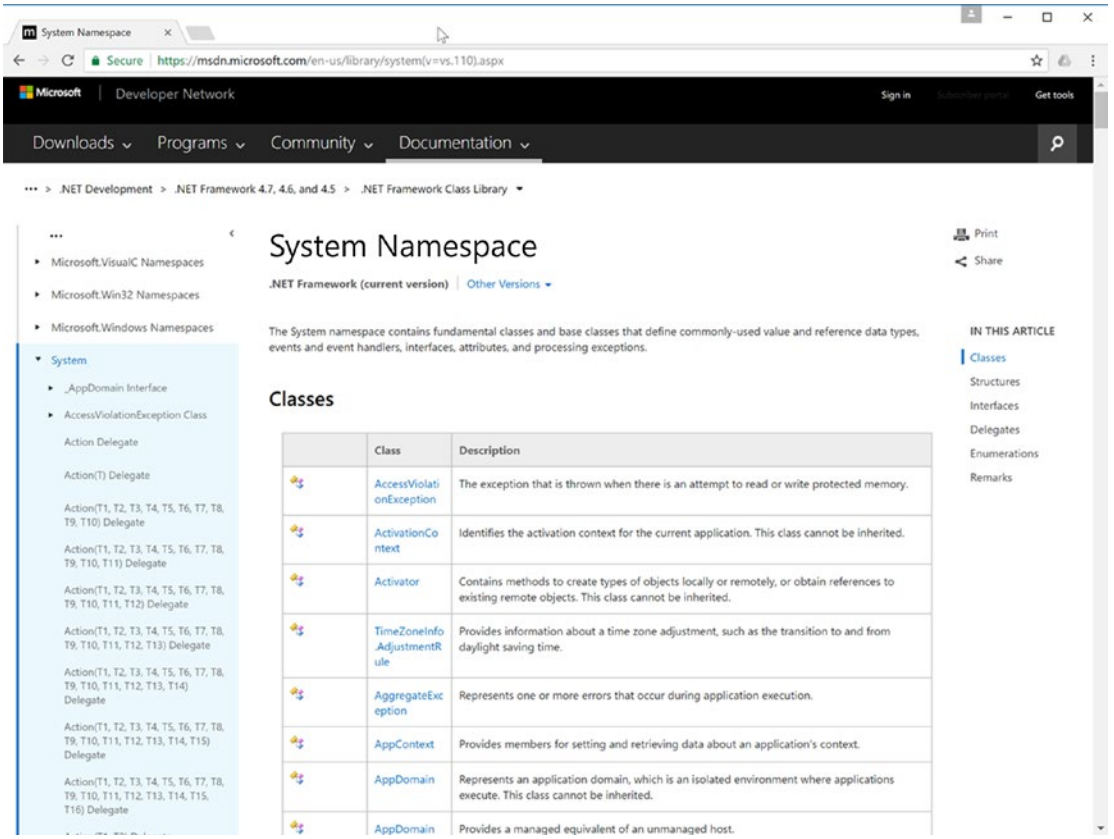

Figure 2-15. *Viewing the .NET Framework documentation online*

■ **Note** At the risk of sounding like a broken record, I can't emphasize enough how important it is that you learn to use the .NET Framework SDK documentation. No book, no matter how lengthy, can cover every aspect of the .NET platform. Make sure you take some time to get comfortable using the help system—you'll thank yourself later.

Building.NET Applications on a Non-Windows OS

There are several options for building .NET applications on non-Windows operating systems. In addition to Xamarin Studio, there are also Visual Studio for the Mac and Visual Studio Code (which also runs on Linux). The types of applications that can be built with these development environments are limited to applications that are being developed either using .NET Core (Visual Studio Code and Visual Studio for the Mac) or for mobile (Visual Studio for the Mac, Xamarin Studio).

That is all I will mention about the non-Windows development tools in this book. But rest assured that Microsoft is embracing all developers, not just developers who own Windows-based computers.

Summary

As you can see, you have many new toys at your disposal! The point of this chapter was to provide you with a tour of the major programming tools a C# programmer may leverage during the development process. As mentioned, if you are interested only in building .NET applications on a Windows development machine, your best bet is to download Visual Studio Community. As also mentioned, this edition of the book will use this particular IDE going forward. Thus, the forthcoming screenshots, menu options, and visual designers will all assume you are using Visual Studio Community.

If you want to build .NET Core applications or cross-platform mobile application on a non-Windows OS, then Visual Studio for the Mac, Visual Studio Code, or Xamarin Studio will be your best choice.