BUILD WINDOWS STORE APPS WITH VB

Join the discussion @ p2p.wrox.com

Professional

Visual Basic® 2012 and .NET 4.5 Programming

Bill Sheldon, Billy Hollis, Rob Windsor, David McCarter, Gastón C. Hillar, Todd Herman

Table of Contents

Introduction

Part I

Language Constructs and Environment

Chapter 1

Visual Studio 2012

What's in this chapter?

Versions of Visual Studio
An introduction to key Visual Basic terms
Targeting a runtime environment
Creating a baseline Visual Basic Windows Form
Project templates
Project properties—application, compilation, debug
Setting properties
IntelliSense, code expansion, and code snippets
Debugging
The Class Designer

Wrox.com Code Downloads for this Chapter

The wrox.com code downloads for this chapter are found at www.wrox.com/remtitle.cgi?isbn=9781118314456 on the Download Code tab. The code is in the chapter 1 download and individually named according to the code filenames listed in the chapter.

You can work with Visual Basic without Visual Studio. In practice, however, most Visual Basic developers treat the two as almost inseparable; without a version of Visual

Studio, you're forced to work from the command line to create project files by hand, to make calls to the associated compilers, and to manually address the tools necessary to build your application. While Visual Basic supports this at the same level as C#, F#, C++, and other .NET languages, this isn't the typical focus of a Visual Basic professional.

Visual Basic's success rose from its increased productivity in comparison to other languages when building business applications. Visual Studio 2012 increases your productivity and provides assistance in debugging your applications and is the natural tool for Visual Basic developers.

Accordingly this book starts off by introducing you to Visual Studio 2012 and how to build and manage Visual Basic applications. The focus of this chapter is on ensuring that everyone has a core set of knowledge related to tasks like creating and debugging applications in Visual Studio 2012. Visual Studio 2012 is used throughout the book for building solutions. Note while this is the start, don't think of it as an “intro” chapter. This chapter will intro key elements of working with Visual Studio, but will also go beyond that. You may find yourself referencing back to it later for advanced topics that you glossed over your first time through. Visual Studio is a powerful and, at times, complex tool, and you aren't expected to master it on your first read through this chapter.

This chapter provides an overview of many of the capabilities of Visual Studio 2012. The goal is to demonstrate how Visual Studio makes you, as a developer, more productive and successful.

Visual Studio 2012

For those who aren't familiar with the main elements of .NET development there is the common language runtime (CLR), the .NET Framework, the various language compilers and

Visual Studio. Each of these plays a role; for example, the CLR—covered in Chapter 2—manages the execution of code on the .NET platform. Thus code can be targeted to run on a specific version of this runtime environment.

The .NET Framework provides a series of classes that developers leverage across implementation languages. This framework or Class Library is versioned and targeted to run on a specific minimum version of the CLR. It is this library along with the language compilers that are referenced by Visual Studio. Visual Studio allows you to build applications that target one or more of the versions of what is generically called .NET.

In some cases the CLR and the .NET Framework will be the same; for example, .NET Framework version 1.0 ran on CLR version 1.0. In other cases just as Visual Basic's compiler is on version 10, the .NET Framework might have a newer version targeting an older version of the CLR.

The same concepts carry into Visual Studio. Visual Studio 2003 was focused on .NET 1.1, while the earlier Visual Studio .NET (2002) was focused on .NET 1.0. Originally, each version of Visual Studio was optimized for a particular version of .NET. Similarly, Visual Studio 2005 was optimized for .NET 2.0, but then along came the exception of the .NET Framework version 3.0. This introduced a new Framework, which was supported by the same version 2.0 of the CLR, but which didn't ship with a new version of Visual Studio.

Fortunately, Microsoft chose to keep Visual Basic and ASP.NET unchanged for the .NET 3.0 Framework release. However, when you looked at the .NET 3.0 Framework elements, such as Windows Presentation Foundation, Windows Communication Foundation, and Windows Workflow Foundation, you found that those items needed to be addressed outside of Visual Studio. Thus, while Visual Studio is separate from Visual Basic, the CLR, and .NET development, in practical terms Visual Studio was tightly coupled to each of these items.

When Visual Studio 2005 was released, Microsoft expanded on the different versions of Visual Studio available for use. Earlier editions of this book actually went into some of the differences between these versions. This edition focuses on using Visual Studio's core features. While some of the project types require Visual Studio Professional, the core features are available in all versions of Visual Studio.

In Visual Studio 2008, Microsoft loosened the framework coupling by providing robust support that allowed the developer to target any of three different versions of the .NET Framework. Visual Studio 2010 continued this, enabling you to target an application to run on .NET 2.0, .NET 3.0,.NET 3.5, or .NET 4.

However, that support didn't mean that Visual Studio 2010 wasn't still tightly coupled to a specific version of each compiler. In fact, the new support for targeting frameworks is designed to support a runtime environment, not a compile-time environment. This is important, because when projects from previous versions of Visual Studio are converted to the Visual Studio 2010 format, they cannot be reopened by a previous version.

The reason for this was that the underlying build engine used by Visual Studio 2010 accepts syntax changes and even language feature changes, but previous versions of Visual Studio do not recognize these new elements of the language. Thus, if you move source code written in Visual Studio 2010 to a previous version of Visual Studio, you face a strong possibility that it would fail to compile. However, Visual Studio 2012 changed this, and it is now possible to open projects associated with older versions of Visual Studio in Visual Studio 2012, work on them, and have someone else continue to work in an older version of Visual Studio.

Multitargeting support continues to ensure that your application will run on a specific version of the framework. Thus, if your organization is not supporting .NET 3.0, .NET 3.5, or .NET 4, you can still use Visual Studio 2012. The compiler generates byte code based on the language syntax, and at its core that byte code is version agnostic. Where you can get in trouble is if you reference one or more classes that aren't part of a given version of the CLR. Visual Studio therefore manages your references when targeting an older version of .NET, allowing you to be reasonably certain that your application will not reference files from one of those other framework versions. Multitargeting is what enables you to safely deploy without requiring your customers to download additional framework components they don't need.

Complete coverage of all of Visual Studio's features warrants a book of its own, especially when you take into account all of the collaborative and Application Lifecycle Management features introduced by Team Foundation Server and its tight integration with both Team Build and SharePoint Server.

Visual Basic Keywords and Syntax

Those with previous experience with Visual Basic are already familiar with many of the language keywords and syntax. However, not all readers will fall into this category, so this introductory section is for those new to Visual Basic. A glossary of keywords is provided, after which this section will use many of these keywords in context.

Although they're not the focus of the chapter, with so many keywords, a glossary follows. Table 1.1 briefly summarizes most of the keywords discussed in the preceding section, and provides a short description of their meaning in Visual Basic. Keep in mind there are two commonly used terms that aren't Visual Basic keywords that you will read repeatedly, including in the glossary:

1. **Method**—A generic name for a named set of commands. In Visual Basic, both `subs` and `functions` are types of methods.
2. **Instance**—When a class is created, the resulting object is an instance of the class's definition.

Table 1.1 Commonly Used Keywords in Visual Basic

Keyword	Description
`Namespace`	A collection of classes that provide related capabilities. For example, the `System.Drawing` namespace contains classes associated with graphics.
`Class`	A definition of an object. Includes properties (variables) and methods, which can be `Subs` or `Functions`.
`Sub`	A method that contains a set of commands, allows data to be transferred as parameters, and provides scope around local variables and commands, but does not return a value.
`Function`	A method that contains a set of commands, returns a value, allows data to be transferred as parameters, and provides scope around local variables and commands.
`Return`	Ends the currently executing `Sub` or `Function`. Combined with a return value for functions.
`Dim`	Declares and defines a new variable.
`New`	Creates an instance of an object.
`Nothing`	Used to indicate that a variable has no value. Equivalent to null in other languages and databases.
`Me`	A reference to the instance of the object within which a method is executing.
`Console`	A type of application that relies on a command-line interface. Console applications are commonly used for simple test frames. Also refers to a .NET Framework Class that manages access of the command window to and from which applications can read and write text data.
`Module`	A code block that isn't a class but which can contain `Sub` and `Function` methods. Used when only a single copy of code or data is needed in memory.

Even though the focus of this chapter is on Visual Studio, during this introduction a few basic elements of Visual Basic will be referenced and need to be spelled out. This way, as you read, you can understand the examples. Chapter 2, for instance, covers working with namespaces, but some examples and other code are introduced in this chapter that will mention the term, so it is defined here.

Let's begin with `namespace`. When .NET was being created, the developers realized that attempting to organize all of these classes required a system. A namespace is an arbitrary system that the .NET developers used to group classes containing common functionality. A namespace can have multiple levels of grouping, each separated by a period (.). Thus, the `System` namespace is the basis for classes that are used throughout .NET, while the `Microsoft.VisualBasic` namespace is used for classes in the underlying .NET Framework but specific to Visual Basic. At its most basic level, a namespace does not imply or indicate anything regarding the relationships between the class implementations in that namespace; it is just a way of managing the complexity of both your custom application's classes, whether it be a small or large collection, and that of the .NET Framework's thousands of classes. As noted earlier, namespaces are covered in detail in Chapter 2.

Next is the keyword `Class`. Chapters 3 and 4 provide details on object-oriented syntax and the related keywords for objects and types, but a basic definition of this keyword is needed here. The `Class` keyword designates a common set of data and behavior within your application. The class is the definition of an object, in the same way that your source code, when compiled, is the definition of an application. When someone runs your code, it is considered to be an instance of your application. Similarly, when your code creates or instantiates an object from your class definition, it is considered to be an instance of that class, or an instance of that object.

Creating an instance of an object has two parts. The first part is the `New` command, which tells the compiler to create an instance of that class. This command instructs code to call your object definition and instantiate it. In some cases you might need to run a method and get a return value, but in most cases you use the `New` command to assign that instance of an object to a variable. A variable is quite literally something which can hold a reference to that class's instance.

To declare a variable in Visual Basic, you use the `Dim` statement. *Dim* is short for "dimension" and comes from the ancient past of Basic, which preceded Visual Basic as a language. The idea is that you are telling the system to allocate or dimension a section of memory to hold data. As discussed in subsequent chapters on objects, the `Dim` statement may be replaced by another keyword such as `Public` or `Private` that not only dimensions the new value, but also limits the accessibility of that value. Each variable declaration uses a `Dim` statement similar to the example that follows, which declares a new variable, `winForm`:

```
Dim winForm As System.Windows.Forms.Form = New
System.Windows.Forms.Form()
```

In the preceding example, the code declares a new variable (`winForm`) of the type `Form`. This variable is then set to an instance of a `Form` object. It might also be assigned to an existing instance of a `Form` object or alternatively to `Nothing`. The `Nothing` keyword is a way of telling the system that the variable does not currently have any value, and as such is not actually using any memory on the heap. Later in this chapter, in the discussion of value and reference types, keep in mind that only reference types can be set to `Nothing`.

A class consists of both state and behavior. State is a fancy way of referring to the fact that the class has one or more values also known as properties associated with it. Embedded in the class definition are zero or more `Dim` statements that create variables used to store the properties of the class. When you create an instance of this class, you create these variables; and in most cases the class contains logic to populate them. The logic used for this, and to carry out other actions, is the *behavior*. This behavior is encapsulated in what, in the object-oriented world, are known as *methods*.

However, Visual Basic doesn't have a “method” keyword. Instead, it has two other keywords that are brought forward from Visual Basic's days as a procedural language. The first is `Sub`. `Sub`, short for “subroutine,” and it defines a block of code that carries out some action. When this block of code completes, it returns control to the code that called it without returning a value. The following snippet shows the declaration of a `Sub`:

```
Private Sub Load(ByVal object As System.Object)

End Sub
```

The preceding example shows the start of a `Sub` called `Load`. For now you can ignore the word `Private` at the start of this declaration; this is related to the object and is further explained in the next chapter. This method is implemented as a `Sub` because it doesn't return a value and accepts one parameter when it is called. Thus, in other languages this might be considered and written explicitly as a function that returns `Nothing`.

The preceding method declaration for `Sub Load` also includes a single parameter, `object`, which is declared as being of type `System.Object`. The meaning of the `ByVal` qualifier is explained in chapter 2, but is related to how that value is passed to this method. The code that actually loads the object would be written between the line declaring this method and the `End Sub` line.

Alternatively, a method can return a value; Visual Basic uses the keyword `Function` to describe this behavior. In Visual

Basic, the only difference between a `Sub` and the method type `Function` is the return type.

The `Function` declaration shown in the following sample code specifies the return type of the function as a `Long` value. A `Function` works just like a `Sub` with the exception that a `Function` returns a value, which can be `Nothing`. This is an important distinction, because when you declare a function the compiler expects it to include a `Return` statement. The `Return` statement is used to indicate that even though additional lines of code may remain within a `Function` or `Sub`, those lines of code should not be executed. Instead, the `Function` or `Sub` should end processing at the current line, and if it is in a function, the return value should be returned. To declare a `Function`, you write code similar to the following:

```
Public Function Add(ByVal ParamArray values() As Integer) As
Long
    Dim result As Long = 0
    'TODO: Implement this function
    Return result
    'What if there is more code
    Return result
End Function
```

In the preceding example, note that after the function initializes the second line of code, there is a `Return` statement. There are *two* `Return` statements in the code. However, as soon as the first `Return` statement is reached, none of the remaining code in this function is executed. The `Return` statement immediately halts execution of a method, even from within a loop.

As shown in the preceding example, the function's return value is assigned to a local variable until returned as part of the `Return` statement. For a `Sub`, there would be no value on the line with the `Return` statement, as a `Sub` does not return a value when it completes. When returned, the return value is usually assigned to something else. This is shown in the next example line of code, which calls a function:

```
Dim ctrl = Me.Add(1, 2)
```

The preceding example demonstrates a call to a function. The value returned by the function `Add` is a `Long`, and the code assigns this to the variable `ctrl`. It also demonstrates another keyword that you should be aware of: `Me`. The `Me` keyword is how, within an object, you can reference the current instance of that object.

You may have noticed that in all the sample code presented thus far, each line is a complete command. If you're familiar with another programming language, then you may be used to seeing a specific character that indicates the end of a complete set of commands. Several popular languages use a semicolon to indicate the end of a command line.

Visual Basic doesn't use visible punctuation to end each line. Traditionally, the BASIC family of languages viewed source files more like a list, whereby each item on the list is placed on its own line. At one point the term was *source listing*. By default, Visual Basic ends each source list item with the carriage-return line feed, and treats it as a command line. In some languages, a command such as `X = Y` can span several lines in the source file until a semicolon or other terminating character is reached. Thus previously, in Visual Basic, that entire statement would be found on a single line unless the user explicitly indicates that it is to continue onto another line.

To explicitly indicate that a command line spans more than one physical line, you'll see the use of the underscore at the end of the line to be continued. However, one of the features of Visual Basic, originally introduced in version 10 with Visual Studio 2010, is support for an implicit underscore when extending a line past the carriage-return line feed. However, this feature is limited, as there are still places where underscores are needed.

When a line ends with the underscore character, this explicitly tells Visual Basic that the code on that line does not constitute a completed set of commands. The compiler will then continue to the next line to find the continuation of the command, and will end when a carriage-return line feed is found without an accompanying underscore.

In other words, Visual Basic enables you to use exceptionally long lines and indicate that the code has been spread across multiple lines to improve readability. The following line demonstrates the use of the underscore to extend a line of code:

```
MessageBox.Show("Hello World", "A Message Box Title",  _
     MessageBoxButtons.OK, MessageBoxIcon.Information)
```

Prior to Visual Basic 10 the preceding example illustrated the only way to extend a single command line beyond one physical line in your source code. The preceding line of code can now be written as follows:

```
MessageBox.Show("Hello World", "A Message Box Title",
     MessageBoxButtons.OK, MessageBoxIcon.Information)
```

The compiler now recognizes certain key characters like the “,” or the “=” as the type of statement where a line isn't going to end. The compiler doesn't account for every situation and won't just look for a line extension anytime a line doesn't compile. That would be a performance nightmare; however, there are several logical places where you, as a developer, can choose to break a command across lines and do so without needing to insert an underscore to give the compiler a hint about the extended line.

Finally, note that in Visual Basic it is also possible to place multiple different statements on a single line, by separating the statements with colons. However, this is generally considered a poor coding practice because it reduces readability.

Console Applications

The simplest type of application is a *console application*. This application doesn't have much of a user interface; in fact, for those old enough to remember the MS-DOS operating system, a console application looks just like an MS-DOS application. It works in a command window without support for graphics or input devices such as a mouse. A console application is a text-based user interface that displays text characters and reads input from the keyboard.

The easiest way to create a console application is to use Visual Studio. For the current discussion let's just look at a sample source file for a Console application, as shown in the following example. Notice that the console application contains a single method, a `Sub` called `Main`. By default, if you create a console application in Visual Studio, the code located in the `Sub Main` is the code which is by default started. However, the `Sub Main` isn't contained in a class; instead, the `Sub Main` that follows is contained in a `Module`:

```
Module Module1
    Sub Main()
         Console.WriteLine("Hello World")
         Dim line = Console.ReadLine()
    End Sub
End Module
```

A `Module` isn't truly a class, but rather a block of code that can contain methods, which are then referenced by code in classes or other modules—or, as in this case, it can represent the execution start for a program. A `Module` is similar to having a `Shared` class. The `Shared` keyword indicates that only a single instance of a given item exists.

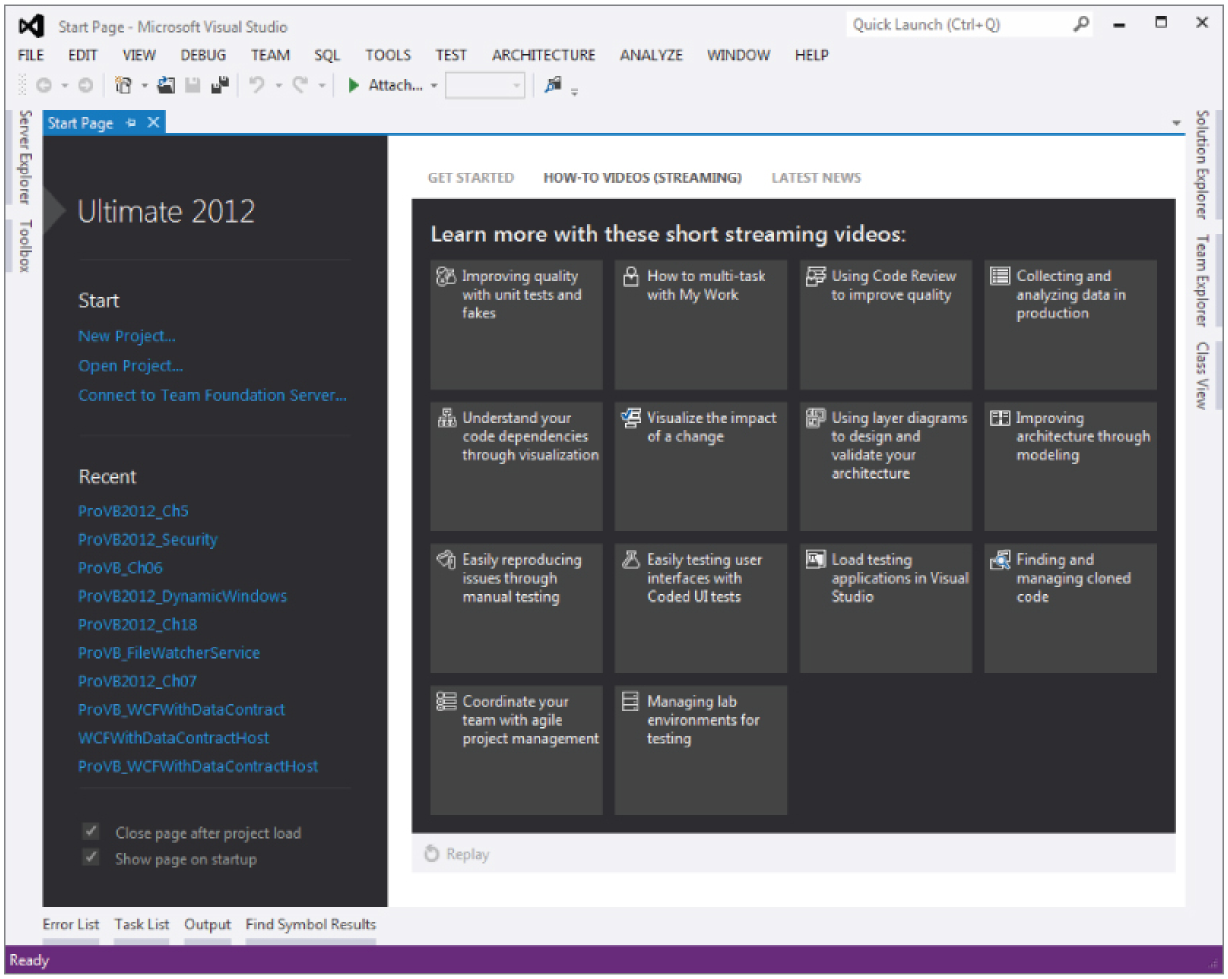
For example, in C# the `Static` keyword is used for this purpose, and can be used to indicate that only a single instance of a given class exists. Visual Basic doesn't support the use of the `Shared` keyword with a `Class` declaration; instead, Visual Basic developers create modules that provide the same capability. The `Module` represents a valid construct to group methods that don't have state-related or instance-specific data.

Note a console application focuses on the `Console Class`. The `Console Class` encapsulates Visual Basic's interface with the text-based window that hosts a command prompt from which a command-line program is run. The console window is best thought of as a window encapsulating the older nongraphical style user interface, whereby literally everything was driven from the command prompt. A `Shared` instance of the `Console` class is automatically created when you start your application, and it supports a variety of `Read` and `Write` methods. In the preceding example, if you were to run the code from within Visual Studio's debugger, then the console window would open and close immediately. To prevent that, you include a final line in the `Main Sub`, which executes a `Read` statement so that the program continues to run while waiting for user input.

Creating a Project from a Project Template

While it is possible to create a Visual Basic application working entirely outside of Visual Studio, it is much easier to start from Visual Studio. After you install Visual Studio, you are presented with a screen similar to the one shown in Figure 1.1. Different versions of Visual Studio may have a different overall look, but typically the start page lists your most recent projects on the left, some tips for getting started, and a headline section for topics on MSDN that might be of interest. You may or may not immediately recognize that this content is HTML text; more important, the content is based on an RSS feed that retrieves and caches articles appropriate for your version of Visual Studio.

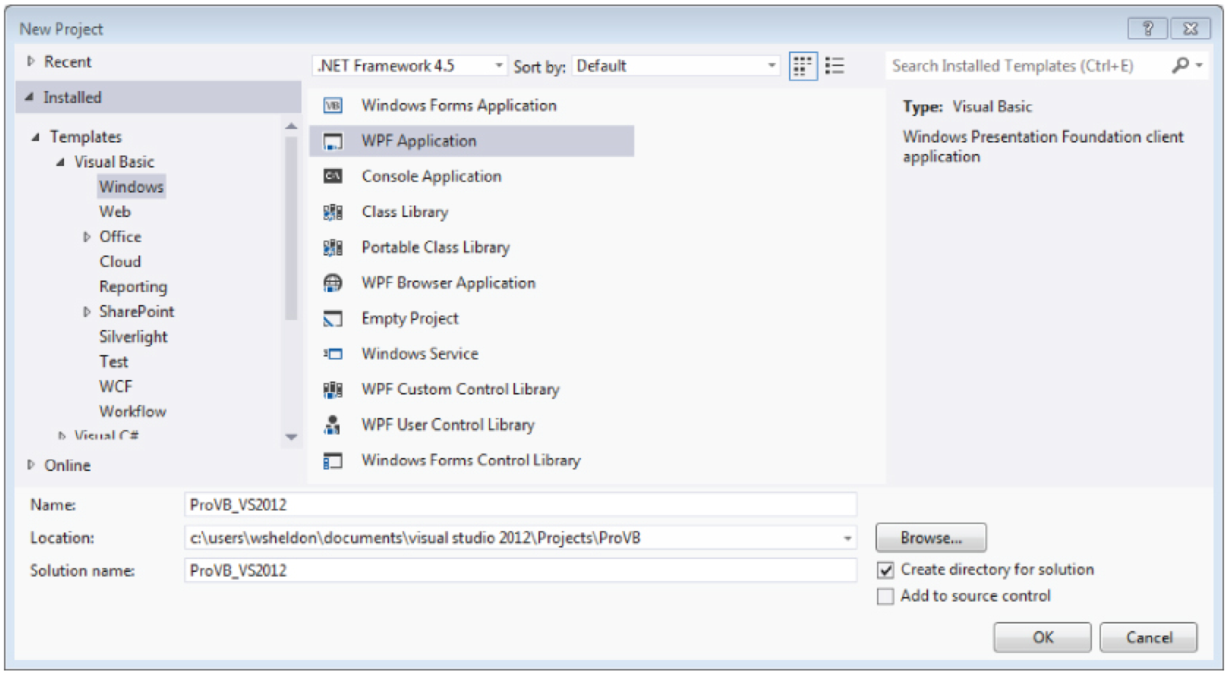
Figure 1.1 Visual Studio 2012 Start screen


Start Page - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW DEBUG TEAM SQL TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Attach...
Server Explorer
Toolbox
Start Page
Solution Explorer
Team Explorer
Class View
Ultimate 2012
Start
New Project...
Open Project...
Connect to Team Foundation Server...
Recent
ProVB2012_Ch5
ProVB2012_Security
ProVB_Ch06
ProVB2012_DynamicWindows
ProVB2012_Ch18
ProVB_FileWatcherService
ProVB2012_Ch07
ProVB_WCFWithDataContract
WCFWithDataContractHost
ProVB_WCFWithDataContractHost
Close page after project load
Show page on startup
GET STARTED
HOW-TO VIDEOS (STREAMING)
LATEST NEWS
Learn more with these short streaming videos:
Improving quality with unit tests and fakes
How to multi-task with My Work
Using Code Review to improve quality
Collecting and analyzing data in production
Understand your code dependencies through visualization
Visualize the impact of a change
Using layer diagrams to design and validate your architecture
Improving architecture through modeling
Easily reproducing issues through manual testing
Easily testing user interfaces with Coded UI tests
Load testing applications in Visual Studio
Finding and managing cloned code
Coordinate your team with agile project management
Managing lab environments for testing
Replay
Error List
Task List
Output
Find Symbol Results
Ready

The start page provides a generic starting point either to select the application you intend to work on, to quickly receive vital news related to offers, as shown in the figure, or to connect with external resources via the community links.

Once here, the next step is to create your first project. Selecting File ⇒ New ⇒ Project opens the New Project dialog, shown in Figure 1.2. This dialog provides a selection of templates customized by application type. One option is to create a Class Library project. Such a project doesn't include a user interface; and instead of creating an assembly with an `.exe` file, it creates an assembly with a `.dll` file. The difference, of course, is that an `.exe` file indicates an executable that can be started by the operating system, whereas a `.dll` file represents a library referenced by an application.

Figure 1.2 New Project dialogue



.NET Framework 4.5 Sort by: Default

Type: Visual Basic

Windows Presentation Foundation client application

Name: ProVB_VS2012

Location: c:\users\wsheldon\documents\visual studio 2012\Projects\ProVB

Solution name: ProVB_VS2012

[✓] Create directory for solution

[ ] Add to source control

Figure 1.2 includes the capability to target a specific .NET version in the drop-down box located above the list of project types. If you change this to .NET 2.0, you'll see the dialog change to show only six project types below the selection listed. For the purposes of this chapter, however, you'll want .NET 4.5 selected, and the template list should resemble what is shown in Figure 1.2. Note this chapter is going to create a Windows .NET application, not a Windows Store application. Targeting keeps you from attempting to create a project for WPF without recognizing that you also need at least .NET 3.0 available on the client. Although you can change your target after you create your project, be very careful when trying to reduce the version number, as the controls to prevent you from selecting dependencies don't check your existing code base for violations. Changing your targeted framework version for an existing project is covered in more detail later in this chapter.

Not only can you choose to target a specific version of the framework when creating a new project, but this window has a new feature that you'll find all over the place in Visual Studio. In the upper-right corner, there is a control that enables you to search for a specific template. As you work through more of the windows associated with Visual Studio, you'll find that a context-specific search capability has often been added to the new user interface.

Reviewing the top level of the Visual Basic tree in Figure 1.2 shows that a project type can be further separated into a series of categories:

- **Windows**—These are projects used to create applications that run on the local computer within the CLR. Because such projects can run on any operating system (OS) hosting the framework, the category “Windows” is something of a misnomer when compared to, for example, “Desktop.”
- **Web**—You can create these projects, including Web services, from this section of the New Project dialog.
- **Office**—Visual Studio Tools for Office (VSTO). These are .NET applications that are hosted under Office. Visual Studio 2010 includes a set of templates you can use to target Office 2010, as well as a separate section for templates that target Office 2007.
- **Cloud Services**—These are projects that target the Azure online environment model. These projects are deployed to the cloud and as such have special implementation and deployment considerations.
- **Reporting**—This project type enables you to create a Reports application.
- **SharePoint**—This category provides a selection of SharePoint projects, including Web Part projects, SharePoint Workflow projects, and Business Data Catalog projects, as well as things like site definitions and

content type projects. Visual Studio 2010 includes significant new support for SharePoint.

- **Silverlight**—With Visual Studio 2010, Microsoft has finally provided full support for working with Silverlight projects. Whereas in the past you've had to add the Silverlight SDK and tools to your existing development environment, with Visual Studio 2010 you get support for both Silverlight projects and user interface design within Visual Studio.
- **Test**—This section is available only to those using Visual Studio Team Suite. It contains the template for a Visual Basic Unit Test project.
- **WCF**—This is the section where you can create Windows Communication Foundation projects.
- **Workflow**—This is the section where you can create Windows Workflow Foundation (WF) projects. The templates in this section also include templates for connecting with the SharePoint workflow engine.

Not shown in that list is a Windows Store project group. That option is available only if you are running Visual Studio 2012 on Windows 8. The project group has five different project types under Visual Basic, but they are available only if you aren't just targeting Windows 8, but are actually using a Windows 8 computer.

This chapter assumes you are working on a Windows 7 computer. The reason for this is that it is expected the majority of developers will continue to work outside of Windows RT. If you are working in a Windows 8 or Windows RT environment, then what you'll look for in the list of Visual Basic templates is a Windows Store application. Keep in mind, however, that those projects will only run on Windows 8 computers. Details of working with Windows Store applications are the focus of Chapters 14 and 15.

Visual Studio has other categories for projects, and you have access to other development languages and far more project types than this chapter has room for. When looking to create an application you will choose from one or more of the available project templates. To use more than a single project to create an application you'll leverage what is known as a solution. A solution is created by default whenever you create a new project and contains one or more projects.

When you save your project you will typically create a folder for the solution, then later if you add another project to the same solution, it will be contained in the solution folder. A project is always part of a solution, and a solution can contain multiple projects, each of which creates a different assembly. Typically, for example, you will have one or more Class Libraries that are part of the same solution as your Windows Form or ASP.NET project. For now, you can select a WPF Application project template to use as an example project for this chapter.

For this example, use ProVB_VS2012 as the project name to match the name of the project in the sample code download and then click OK. Visual Studio takes over and uses the Windows Application template to create a new WPF Application project. The project contains a blank form that can be customized, and a variety of other elements that you can explore. Before customizing any code, let's first look at the elements of this new project.

The Solution Explorer

The Solution Explorer is a window that is by default located on the right-hand side of your display when you create a project. It is there to display the contents of your solution and includes the actual source file(s) for each of the projects in your solution. While the Solution Explorer window is available and applicable for Express Edition users, it will never contain more than a single project. Visual Studio provides the ability to leverage multiple projects in a single solution. A .NET solution can contain projects of any .NET language and can include the database, testing, and installation projects as part of the overall solution. The advantage of combining these projects is that it is easier to debug projects that reside in a common solution.

Before discussing these files in depth, let's take a look at the next step, which is to reveal a few additional details about your project. Hover over the small icons at the top of the Solution Explorer until you find the one with the hint “Show All Files.” Click that button in the Solution Explorer to display all of the project files, as shown in Figure 1.3. As this image shows, many other files make up your project. Some of these, such as those under the My Project grouping, don't require you to edit them directly. Instead, you can double-click the My Project entry in the Solution Explorer and open the pages to edit your project settings. You do not need to change any of the default settings for this project, but the next section of this chapter walks you through the various property screens.

Figure 1.3 Visual Studio Solution Explorer