CHAPTER 1

Ready, Steady, Go!

This book is about the Visual Basic programming language first and foremost. It is about becoming a proficient Visual Basic programmer. Reading this book from cover to cover will not make you a superstar, but it will make you a programmer who understands what needs to be done when writing robust, stable, and maintainable Visual Basic applications.

In this chapter, you'll get started by acquiring the tools you need to develop Visual Basic applications and taking those tools for a test spin. Along the way, you'll create a couple Visual Basic applications.

Downloading and Installing the Tools

Getting started with Visual Basic 2008, you're probably really excited about writing some code that does something. It's like getting your driver's license and wanting to drive a car without even thinking about where you want to go. You just want to drive. The great part of .NET is that you can start writing some code after you have installed either the .NET software development kit (.NET SDK) or a Visual Studio integrated development environment (IDE). Downloading and installing the right environment is critical to taking your first step toward an enjoyable coding experience.

Note Software version numbers, product descriptions, and technologies can be confusing. Having used Microsoft technologies for over a decade, I can say that naming a technology or product has never been Microsoft's strong point. The technologies have been great (for the most part), but product classification and identification have not been so great. Thus, this book covers the Visual Basic 2008 programming language that is used to write applications for the .NET Framework. With Visual Basic 2008, the .NET 3.0 and 3.5 Frameworks are used. .NET 3.0 gives you all of the essentials, and .NET 3.5 gives you the extras.

For the examples in this book, you'll be using Visual Basic 2008 Express Edition, because it's freely available and has everything you need to get started with Visual Basic 2008. The other Express Edition IDEs available from Microsoft are tailored to different languages (C# and C++) or, in the case of Visual Web Developer Express, specific functionality that is too restrictive for our purposes.

Microsoft also offers full versions of the Visual Studio IDE, such as the Standard, Professional, and Team editions. Each of these editions has different features and different price tags. See the Microsoft Visual Studio web site (`http://msdn2.microsoft.com/en-us/vstudio/default.aspx`) for more information. If you already have Visual Studio 2008 Professional, you can use that for the examples in this book. That edition can do everything that Visual Basic 2008 Express can do, and in fact, has many more options.

■ **Note** I personally use Visual Studio Standard or Professional in combination with other tools such as X-develop and JustCode! from Omnicore (`http://www.omnicore.com`), TestDriven.NET (`http://www.testdriven.net/`), and NUnit (`http://www.nunit.org`). The Visual Studio products are very good, but others are available. Being a good developer means knowing which tools work best for you.

Installing and downloading Visual Basic Express from the Microsoft web site involves the transfer of large files. If you do not have a broadband connection, I suggest that you install the IDE from a CD instead.

Downloading Visual Basic Express

The following is the procedure for downloading Visual Basic Express from the Microsoft web site. By the time you are reading this book, the procedure may be a bit different, but it will be similar enough that you'll be able to find and download the IDE package.

1. Go to `http://www.microsoft.com/express/`.
2. Select the Download Now! link.
3. Scroll down to the Visual Basic 2008 Express Edition section, as shown in Figure 1-1.
4. Click the Download link.
5. A dialog box appears, asking where you want to store the downloaded file. The file that you are downloading is a small bootstrap file, which you'll use to begin the actual installation of the Visual Basic Express IDE. Choose to save the file on the desktop.

These steps can be carried out very quickly—probably within a few minutes. Do not mistake this procedure for downloading the complete Visual Basic Express application, because that's not what happened. The installation procedure will download the vast majority of the IDE.

Figure 1-1. *Selecting Visual Basic 2008 Express Edition*

Installing Visual Basic Express

After you've downloaded the setup file, you can start the Visual Basic Express installation. During this process, all the pieces of the IDE—about 300MB—are downloaded and installed. Follow these steps:

1. On your desktop, double-click the `vbsetup.exe` file. Wait while the setup program loads all the required components.
2. Click Next on the initial setup screen.
3. A series of dialog boxes will appear. Select the defaults and click Next to continue through the setup program. In the final dialog box, click Install.
4. After all the elements have been downloaded and installed, you may need to restart your computer.

After Visual Basic Express is installed, you can start it by selecting it from the Start menu.

Choosing the Application Type

With Visual Basic Express running, you're ready to write your first .NET application. However, first you need to make a choice: what type of application will you write? Broadly speaking, in .NET, you can develop three main types of programs:

- A *console application* is designed to run at the command line with no graphical user interface (GUI).
- A *Windows application* is designed to run on a user's desktop and has a GUI.
- A *class library* holds reusable functionality that can be used by console and Windows applications. It cannot be run by itself.

So that you know what each type of program is about, in this chapter, you will code all three. They are all variations of the Hello, World example, which displays the text "hello, world" on the screen. Hello, World programs have been used for decades to demonstrate programming languages.

Creating Projects and Solutions

Regardless of which program type you are going to code, when using the Visual Studio line of products, you will create projects and solutions:

- A *project* is a classification used to describe a type of .NET application.
- A *solution* is a classification used to describe multiple .NET applications that most likely relate to each other.

Imagine building a car. A project could be the steering wheel, engine, or car body. Putting all of the car projects together creates a complete solution called the car.

A solution contains projects. For the examples in this chapter, our solution will contain three projects representing each of the three different program types.

When using Visual Basic Express, creating a project implies creating a solution, because creating an empty solution without a project does not make sense. It's like building a car with no parts. When I say "project" or "application" in this book, from a workspace organization perspective, it means the same thing. *Solution* is an explicit reference to one or more projects or applications.

Our plan of action in terms of projects and solutions in this chapter is as follows:

- Create the .NET solution by creating a Windows application called `WindowsApplication` (creating this application also creates a solution).
- Add to the created solution a console application called `ConsoleApplication`.
- Add to the created solution a class library project called `ClassLibrary`.

Creating the Windows Application

We'll dive right in and start with the Windows application. With Visual Basic Express running, follow these steps to create the Windows application:

1. Select File ➤ New Project from the menu.
2. Select the Windows Forms Application icon. This represents a project style based on a predefined template called Windows Forms Application.

3. Change the default name to `WindowsApplication`.
4. Click OK.

These steps create a new project and solution at the same time: Visual Basic Express will display only the complete project, as shown in Figure 1-2.

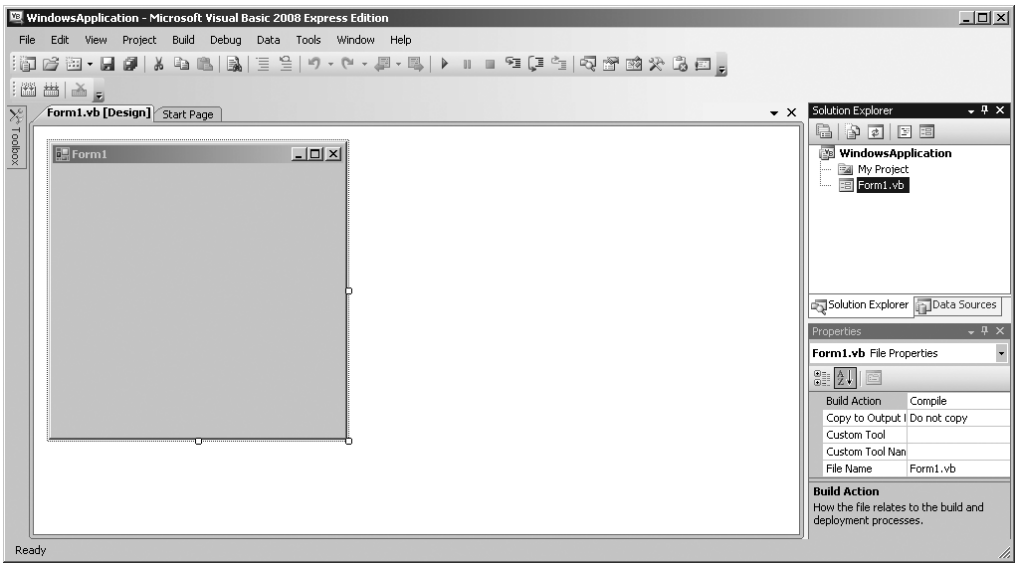

Figure 1-2. *The Visual Basic Express IDE with the new WindowsApplication project*

Viewing the Source Code

When you create a new application, Visual Basic Express automatically generates some source code for it. Right-click the `Form1.vb` item in the Solution Explorer and select View Code from the context menu. The following source code will appear in the area to the left of the Solution Explorer.

■ **Note** To shift between the user interface and generated code, right-click `Form1.vb` in the Solution Explorer. A submenu appears with the options View Code (to see the code) or View Designer (to see the user interface).

```
Public Class Form1

End Class
```

In Visual Basic, the source code is spartan because Visual Basic is what was once called a rapid application development (RAD) environment. The idea at the core of Visual Basic is the ability to develop an application as quickly as possible without the esoteric details of the language

getting in your way. This legacy is both good and bad. Figure 1-2 shows a simple project with a single file, but another file exists at the hard disk level, as shown in Figure 1-3 (you can see this file by clicking the Show All Files icon in the Solution Explorer and expanding the `Form1.vb` node).

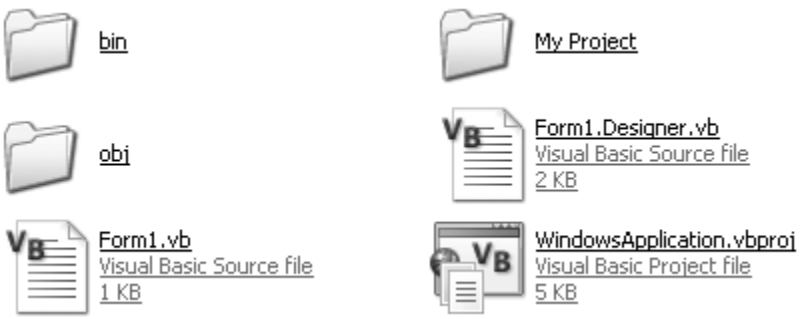

Figure 1-3. *All files that make up the WindowsApplication project*

In previous versions, the `Form1.Designer.vb` file used to be a binary file that you could not edit. Now it is a text file that you can edit, but you should not make any changes to this file because it is maintained by the IDE. Contained within the `Form1.Designer.vb` file are the details of how to construct `Form1`, as is shown in Figure 1-2. At this point, `Form1` does not contain anything noteworthy, and neither does the text file. However, if you were to add a button or text box to the form, those details would be added to the text file `Form1.Designer.vb`.

Visual Basic is a complete programming language that still adheres to the RAD model. For example, the following code creates a user-defined type (which you'll learn about throughout the rest of the book).

```
Public Class Example
    Public Sub Empty()
    End Sub
End Class
```

The main elements to note are as follows:

Class: An organizational unit that groups related code together. This grouping is much more specific than a solution or a project. To use the car analogy again, if a project is a car engine, then a class can be the carburetor. In other words, projects are made up of multiple classes.

Sub: A set of instructions that carry out a task. Also called a *method*, a sub is analogous to a function in many other languages. The `Empty()` method can be called by another piece of code to carry out some type of action.

Saving the Project

After you've renamed the solution, it's good practice to save your changes. To save the project, follow these steps:

1. Highlight the project name in the Solution Explorer.
2. Select File ➤ Save WindowsApplication.
3. Notice that Visual Basic Express wants to save the solution using the `WindowsApplication` name, which isn't ideal. (We're using three projects in this solution, of which one is a Windows Forms application.) To save the solution with a new name, you need to change the `WindowsApplication` solution name to `ThreeExamples` (make sure you leave the `WindowsApplication` project name as it is). Note the path of where Visual Basic Express saves your projects, as you will need to know it from time to time.
4. Click the Save button.

When the solution and project are successfully saved, you'll see the message "Item(s) Saved" in the status bar in the lower-left corner of the window.

In the future, whenever you want to save the solution and project, you can use the keyboard shortcut: Ctrl+S.

■ **Note** If you have not saved your changes and choose to exit Visual Basic Express, you will be asked if you want to save or discard the solution and project.

To open a solution you have previously saved, you can choose File ➤ Open Project at any time and navigate to the solution file. You can also select the solution from the Recent Projects window when you first start Visual Basic Express. The Recent Projects window is always available on the Start Page tab of the main Visual Basic Express window as well (and the File menu contains a list, too).

Running the Windows Application

The source code generated by Visual Basic Express is a basic application that contains an empty window with no functionality. The source code gives you a starting point where you can add more source code, debug the source code, and run the application.

To run the application, select Debug ➤ Start Debugging. Alternatively, use the keyboard shortcut F5. You'll see a window representing the `WindowsApplication` application. You can exit the application by clicking the window's close button. Figure 1-4 illustrates the process. (Debugging is covered in Chapter 5.)

Running the application enables you to see what it does. When you run an application though the IDE, it is identical to a user clicking to start the application from the desktop. In this example, `WindowsApplication` displays an empty window without any controls or functionality. The source code's functionality is to display an empty window when started and provide a button to end the application (and buttons to maximize and minimize the window). Close the application now.

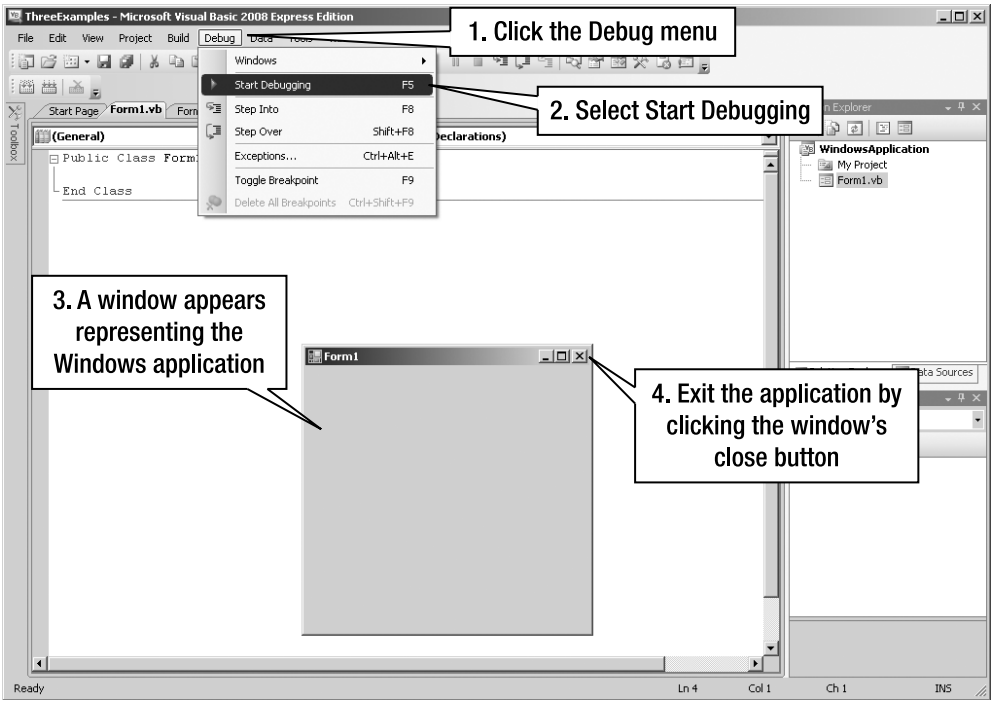

Figure 1-4. *Running an application*

You have not written a single line of code, yet you have created an application and something actually happened, and all because Visual Basic Express generates some boilerplate Visual Basic code that works straight out of the box.

You have created an application, seen its source code, and run it. You did all of this in the context of a comfortable, do-it-all-for-you development environment called Visual Basic Express. Visual Basic Express is both a good thing and a bad thing. Visual Basic Express is good because it hides the messy details, but it is bad because the messy details are hidden. Imagine being a car mechanic. It is good that car manufacturers produce dashboards that have little lights that go on when something is wrong. But it would be bad if the mechanic had to rely on the little lights to fix problems in a car.

Making the Windows Application Say Hello

The Windows application does nothing other than appear with a blank window that you can close. To make the application do something, you need to add user interface elements or add some code. Adding code without adding user interface elements will make the program do something, but it's not as exciting. So, we'll add a button that, when clicked, will display "hello, world" in a text box.

First, you need to add the Button control to the form. Double-click `Form1.vb` in the Solution Explorer to display a blank form. Then click the Toolbox tab to access the controls and open the Common Controls tab (click the pin icon on the Toolbox to leave the Toolbox tab open if you

like). Click Button, and then click the form to place the button on the form. These steps are illustrated in Figure 1-5.

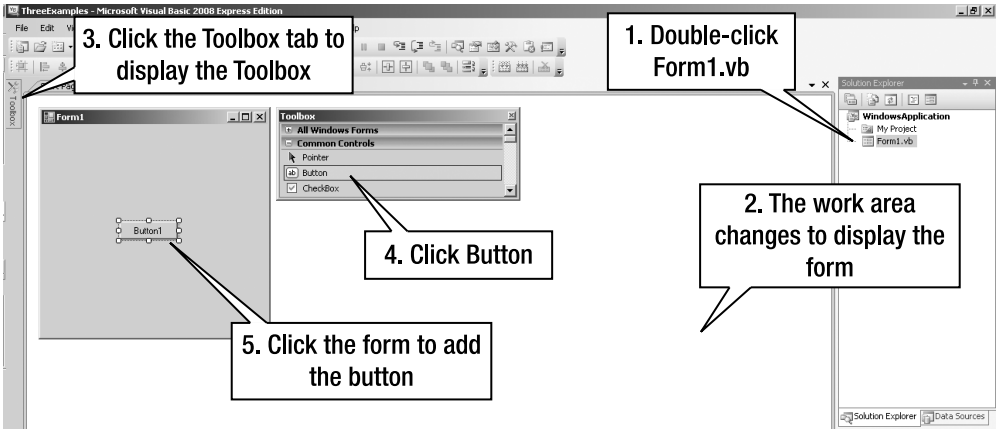

Figure 1-5. *Adding a button to the form*

Next, add a TextBox control using the same basic procedure. Finally, align the button and text box as shown in Figure 1-6. To move a control, use the handles that appear when you highlight the control. Visual Basic Express will align the edge of a control to nearby edges as you drag it, so that you can align controls accurately.

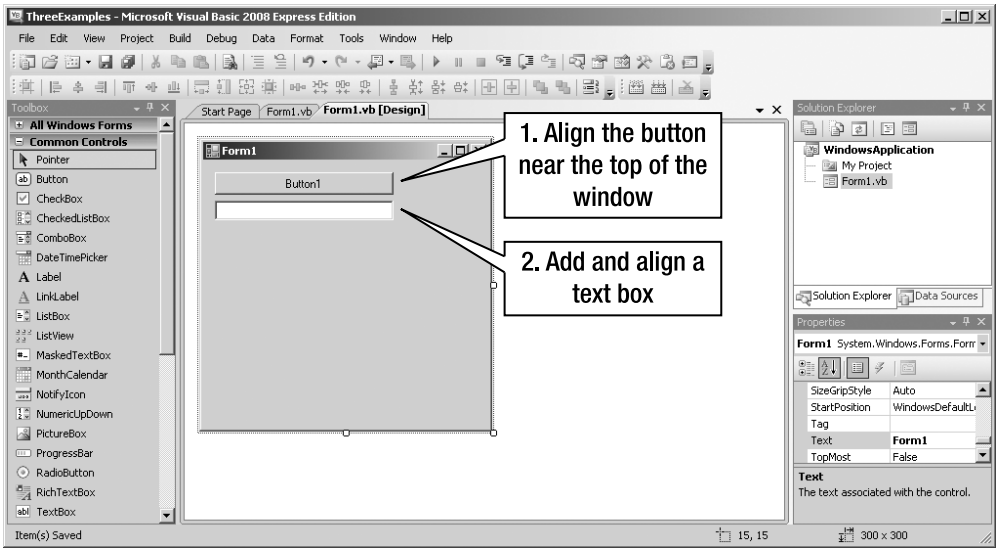

Figure 1-6. *Aligned button and text box*

If you now executed `WindowsApplication` by pressing Ctrl+F5 (Ctrl+F5 starts the application without debugging), you would see a window with a button and a text box. You can click the button and add or delete text from the text box. But whatever you do has no effect, because neither control has been associated with any code.

To make the application do something, you need to think in terms of *events*. For example, if you have a garage with an automatic door opener, you would expect that pressing the remote control button would open the garage door when it's closed and close the door when it's open. The automatic garage door manufacturer associated the event of pushing the remote control button with the action of either opening or closing the garage door. In `WindowsApplication`, we'll associate the clicking of the button with the action of showing text in the text box.

Select the button on the form in Visual Basic Express and double-click it. The work area changes to source code, with the cursor in the `Button1_Click()` function. Add this source code to the function:

```
TextBox1.text = "hello, world"
```

Figure 1-7 illustrates the procedure for associating an event with an action.

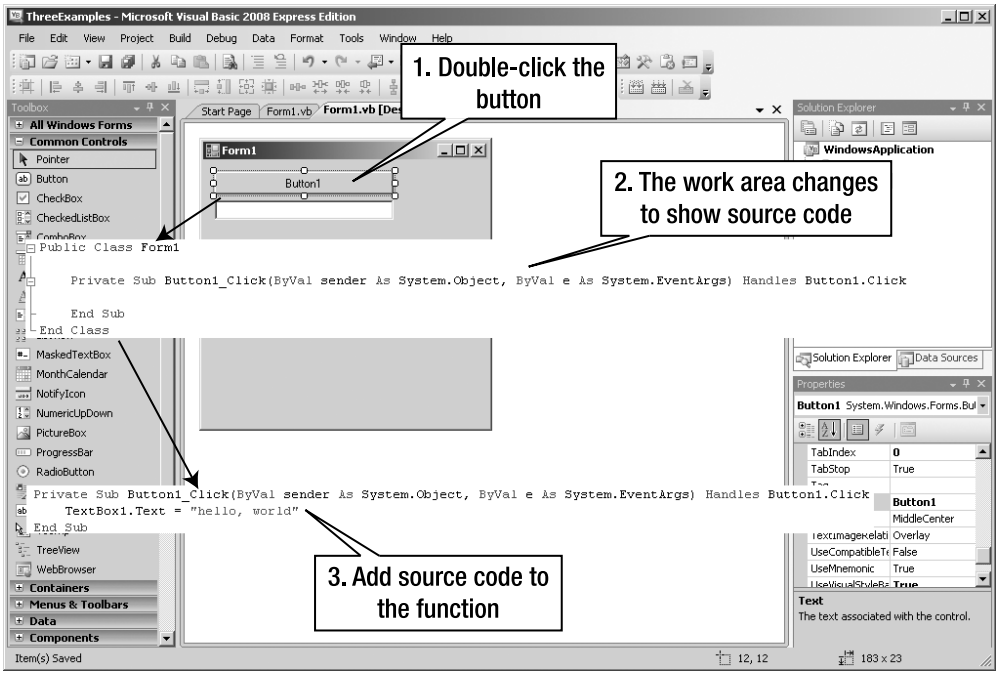

Figure 1-7. *Associating the button click event with the action of adding text to the text box*

Note that `TextBox1` is the name of the text box you added to the form. This name is generated by Visual Basic Express, just as it generated a default name for the button. You can change the default names (through each control's Properties window), but we've left the default for this example.

Adding an action to an event is very simple when following the instructions shown in Figure 1-7. The simplicity is due to Visual Basic Express, and not because the event or action is

simple. Visual Basic Express makes the assumption that when you double-click a control, you want to modify the *default event* of the control, and as such, automatically generates the code in step 3 of Figure 1-7. In the case of a button, the default event is the click event; that is, the event that corresponds to a user clicking the button. The assumption of the click event being the default event for a button is logical. Other controls have different default events. For example, double-clicking a TextBox control will generate the code for the text-changed event.

Run the application by pressing Ctrl+F5, and then click the button. The text box fills with the text "hello, world." Congratulations, you've just finished your first Visual Basic application.

You have associated an event with an action: the button click with the text display. Associating events with actions is the basis of all Windows applications.

Adding Comments to the Application

Now that you have a working program, it would be good to document what it does, right there in the source code. Then if you come back to the application in the future, you won't be puzzled by your previous work. In fact, you may not even be the person who maintains your code, so leaving comments in the code to help explain it is definitely good practice. Even if you know you will be maintaining the code forever, treat your future self as a stranger. You may be surprised how long it takes to decipher code you have written when revisited months or years later.

To add a single-line comment, use the following syntax:

```
' A single-line comment
```

Anything after the ' on the same line is ignored by the compiler and is not included in the final application. Let's document our Windows application:

```
' When the user clicks the button, we display text in the text box
    Private Sub Button1_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Button1.Click
        TextBox1.Text = "hello, world"
    End Sub
```

The Visual Basic language is a single-line language. This means that a statement must be part of a single line. Let's look at a single statement:

```
        TextBox1.Text = "hello, world"
```

This line of code is a single statement because it is considered an assignment of one variable by another piece of source code. You could not write the statement as follows:

```
        TextBox1.Text =
                 "hello, world"
```

When the statement is broken into two lines of source code, the Visual Basic compiler sees it as two statements. Since those two statements are not complete, a compilation error will result. If you need to break a single statement over two lines, you must let the compiler know by adding the line-continuation character—an underscore (_)—at the end of the continued code, as follows:

```
        TextBox1.Text = _
                 "hello, world"
```

Navigating the User Controls of the Solution

When you are writing your code, your most important form of navigation is the Solution Explorer. The Solution Explorer is the tree control that contains the references to your solutions and projects. Consider the Solution Explorer as your developer dashboard, which you can use to fine-tune how your .NET application is assembled and executed.

I suggest that you take a moment to click around the Solution Explorer. Try some right-clicks on various elements. The context-sensitive click is a fast way of fine-tuning particular aspects of your solution and project. However, when clicking, please do not click OK in any dialog box; for now, click Cancel so that any changes you may have made are not saved.

To the right of the Solution Explorer is your work area. The work area is where you write your code or edit your user interface. The work area will display only a single piece of information, which could be some code, a user interface, or a project. As you saw earlier, when you double-click `Form1.vb` in the Solution Explorer, the work area displays the form related to the `Form1.vb` file.

Now that you have an idea of how the IDE works, let's continue with our examples. Next up is the console application.

Creating the Console Application

A console application is a text-based application. This means that rather than displaying a GUI, it uses a command-line interface.

The console has a very long history because the console was the first way to interact with a computer. Consoles are not very user-friendly and become very tedious for any complex operations, yet some people claim that a console is all you need. (See `http://en.wikipedia.org/wiki/Command_line_interface` for more information about the console.)

Writing to the console works only if the currently running application has a console. To open the console in Windows, select Start ➤ Run and type `cmd` in the dialog box. When you test a console application, Visual Basic Express opens a console for you.

Visual Basic Express can create, build, and manage console applications.

Adding a Console Application Project to the Solution

We will now create an application that outputs the text "hello, world" to the console. Follow these steps to add the new project to the `ThreeExamples` solution:

1. Choose File ➤ Add ➤ New Project.
2. Make sure the location is the same as that of `WindowsApplication`.
3. Select Console Application and change the name to `ConsoleApplication`.

The Solution Explorer changes to show the additional project and now also shows the solution. The work area displays the source code.

Notice the simplicity of the console application. It contains a single, plain-vanilla source code file, called `Module1.vb`. Console applications typically do not have any specialized groupings and do not have any events.

Making the Console Application Say Hello

To make the console application do something, you need to add some source code to the `Main()` method, as follows:

```
Module Module1

    Sub Main()
        Console.WriteLine("hello, world")
        Console.ReadKey()
    End Sub

End Module
```

The bolded line writes the text "hello, world" to the console.

If you tried to run the console application by pressing Ctrl+F5, you would instead cause the Windows application, `WindowsApplication`, to run. Let's change that next.

Setting the Startup Project

To execute the console application, you need to set the console application as the *startup project*. Did you notice how the `WindowsApplication` project is in bold in the Solution Explorer? That means `WindowsApplication` is the startup project. Whenever you run or debug an application, the startup project is executed or debugged.

To switch the startup project to `ConsoleApplication`, right-click the `ConsoleApplication` project and select Set As StartUp Project. `ConsoleApplication` will now be in bold, meaning it is the startup project of the `ThreeExamples` solution.

Running the Console Project

With `ConsoleApplication` set as the startup project, you can now press Ctrl+F5 to run the console application. The output is as follows:

```
hello, world
```

Executing the console application does not generate a window, as you saw with the Windows application. Instead, a command prompt is started with `ConsoleApplication` as the application to execute. Executing that application generates the text "hello, world." You will also see that you can press any key to close the command prompt window. Visual Basic Express automatically generated the code to show this output and execute this action.

In general, the console application is limited, but it's an easy way to run specific tasks. Now let's move on to the next example.

Creating the Class Library

The third example in this chapter is not a .NET application; rather, it is a shareable piece of functionality, typically called a *class library*. Windows applications and console applications are programs that you can execute from a command prompt or Windows Explorer. A class library cannot be executed by the user, but needs to be accessed by a Windows application or console application. It is a convenient place to put code that can be used by more than one application.

Adding a Class Library Project to the Solution

We will now create a class library for the Windows application and console application to share. Follow these steps to add the new project to the `ThreeExamples` solution:

1. Right-click the solution name, `ThreeExamples`, in the Solution Explorer.
2. Select Add ➤ New Project.
3. Select Class Library and change the name to `ClassLibrary`.

The resulting solution project should look like Figure 1-8.

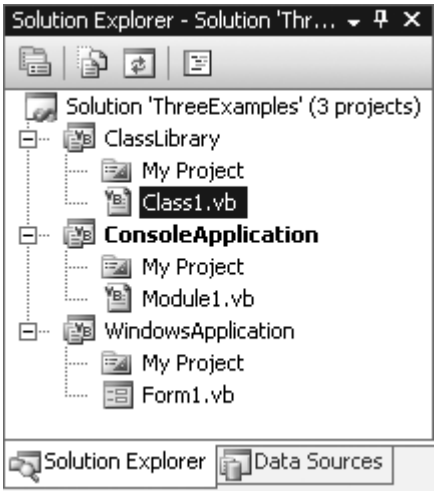

Figure 1-8. *Updated solution structure that contains three projects*

The added `ClassLibrary` project has a single file called `Class1.vb`, which is a plain-vanilla source code file.

Moving Functionality

Now we will move the code used to say "hello, world" from `ConsoleApplication` to `ClassLibrary`. Add the code to `Class1.vb` as follows (the bolded code):

```
Public Class Class1
    Public Shared Sub HelloWorld()
        Console.WriteLine("hello, world")
    End Sub
End Class
```

The modified code contains a method called `HelloWorld()`. When called, this method will output the text "hello, world." As mentioned earlier in the chapter, a method is a set of instructions that carry out a task. Methods are discussed in more detail in Chapter 2.

In order for applications to actually share the code that's in a class library, you must make the projects aware of each other's existence. You do that through references.

Defining References

To make one project aware of definitions in another project, you need to define a *reference*. The idea behind a reference is to indicate that a project knows about another piece of functionality.

■ **Note** The project only knows about the functionality that has been declared as being public. Public functionality, or what Visual Basic programmers call *public scope*, is when you declare a type with the `Public` keyword. You will learn about public and other scopes throughout this book.

To make `ConsoleApplication` aware of the functionality in the `ClassLibrary` project, you need to set a physical reference, as follows:

1. In the Solution Explorer, click `ConsoleApplication`.
2. Right-click and select Add Reference.
3. Click the Projects tab.
4. Select `ClassLibrary`, and then click OK. `ClassLibrary` will be added to `ConsoleApplication`'s references.

Once the reference has been assigned, `ConsoleApplication` can call the functionality in `ClassLibrary`.

To know which references your application or class library has, you need to look in the project settings. To do so, right-click the project name, `ConsoleApplication`, in the Solution Explorer and select Properties. In the Properties window, select the References tab, as shown in Figure 1-9.

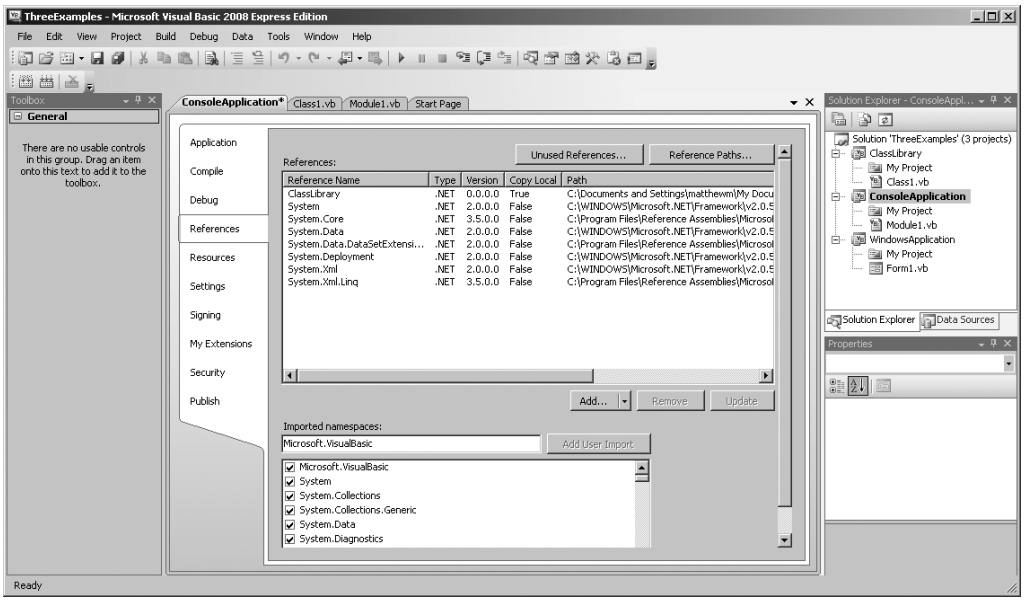

Figure 1-9. *References used by the Visual Basic project*

Calling Class Library Functionality

Now we need to change `ConsoleApplication` so that it calls the function in `ClassLibrary`. Modify the `Module1.vb` file in `ConsoleApplication` as follows:

```
Module Module1

    Sub Main()
        Console.WriteLine("hello, world")
        ClassLibrary.Class1.HelloWorld()
        Console.ReadKey()
    End Sub

End Module
```

Run `ConsoleApplication` by pressing Ctrl+F5. A command prompt window should appear and generate the "hello, world" text twice. The first "hello, world" is generated by the code `Console.WriteLine()`. Calling the function `ClassLibrary.Class1.HelloWorld()` generates the second "hello, world."

USING REFERENCE SHORTHAND

`ClassLibrary.Class1.HelloWorld()` is the longhand way to use a reference. If we were to use longhand for the `Console.WriteLine()` call, we would write `System.Console.WriteLine()`, because the `Console.WriteLine()` method is defined in the `System` reference. However, Visual Basic Express includes the `System` reference by default, so we don't need to do it this way.

To use shorthand for the `ClassLibrary` call, we would include an `Imports` line at the beginning of `Module1.vb` in `ConsoleApplication` and change the call to `Class1`'s `HelloWorld()` method:

```
Imports ClassLibrary
Module Module1

    Sub Main()
        Console.WriteLine("hello, world")
        Class1.HelloWorld()
        Console.ReadKey()
    End Sub

End Module
```

But shorthand like this has a downside. What if we had many references, each containing a class called `Class1`? In this case, Visual Basic Express wouldn't know which `Class1` to use without the help of longhand. Granted, you are not likely to name multiple classes `Class1`, but even sensible names can be duplicated in a collection of references. And if you are using someone else's code as a reference, the possibility of duplicate names becomes higher. Therefore, you're better off using longhand in this case.

Using Variables and Constants

One of the core concepts in a Visual Basic program is to use variables. Think of a variable as a block of memory where you can store data for later use. This allows you to pass data around within your program very easily.

In our `ClassLibrary` project, it would make life easier if we could define the message to display at the beginning of the method. That way, if we decide to change the message, we can get at it much more easily. As it stands, if we were to add more code before the `Console.WriteLine()` call, we would need to scroll through the text to find the message to change. A variable is perfect for this, as we can define some data (the message to print), and then use it later in our program.

```
Public Class Class1
    Public Shared Sub HelloWorld()
        Dim message As String = "hello, world"
        Console.WriteLine(message)
    End Sub
End Class
```

Here, we've defined a variable called `message` of type `String` (a `String` is a length of text). We can then refer to the `message` variable later when we want to place its contents into the code. In the example, we place its contents into the `Console.WriteLine()` call, which works as before. This time, however, we have set the message to display in a separate statement.

This is very useful for us, but there is more to variables. They have something that is called *scope*. The `message` variable has method-level scope, which means it is available only in the method in which it is defined. Consider this code:

```
    Public Shared Sub HelloWorld()
        Dim message As String = "hello, world"
        Console.WriteLine(message)
    End Sub

    Public Shared Sub DisplayeMessageText()
        Console.WriteLine("hello, world")
        Console.WriteLine(message)
    End Sub
```

The `DisplayMessageText()` method prints two lines of text to tell us what the message text should be. However, this doesn't compile, because the compiler knows that the variable `message` is not available to the `DisplayMessageText()` method because of its method-level scope.

To fix this, we need to give `message` class-level scope by moving it to the beginning of the class definition (as it is used by methods marked `Shared`, it must also be `Shared`):

```
Public Class Class1
    Shared Dim message As String = "hello, world"
    Public Shared Sub HelloWorld()
        Console.WriteLine(message)
    End Sub

    Public Shared Sub DisplayeMessageText()
        Console.WriteLine("hello, world")
        Console.WriteLine(message)
    End Sub
End Class
```

Now the variable `message` is shared by all the methods of `Class1`. You'll learn much more about method-level and class-level scopes, as well as the `Public` and `Shared` keywords, throughout this book.

Sharing a variable among methods of a class can be useful, but it's sometimes not wise to do this. That's because methods can change variables as they carry out their tasks, which can produce unpredictable results further down the line. We can lock the value by using a *constant* instead of a variable. The `Const` keyword denotes the constant:

```
Public Class Class1
    Const MESSAGE As String = "hello, world"
    Public Shared Sub HelloWorld()
        Console.WriteLine(MESSAGE)
    End Sub

    Public Shared Sub DisplayeMessageText()
        Console.WriteLine("hello, world")
        Console.WriteLine(MESSAGE)
    End Sub
End Class
```

Constant names are usually all uppercase. The contents of a constant cannot be changed at any point. The following would not compile, for instance.

```
Public Class Class1
    Const MESSAGE As String = "hello, world"

    Public Shared Sub DisplayeMessageText()
        MESSAGE = "another text that cannot be assigned"
        Console.WriteLine(MESSAGE)
    End Sub
End Class
```

Now that you've worked through this chapter's examples, let's talk a bit about how your Visual Basic code in Visual Basic Express actually turns into a program that can run on an operating system like Windows.

Understanding How the .NET Framework Works

When you write Visual Basic source code, you are creating instructions for the program to follow. The instructions are defined using the Visual Basic programming language, which is useful for you, but not useful for the computer. The computer does not understand pieces of text; it understands ones and zeros. To feed instructions to the computer, developers have created a higher-level instruction mechanism that converts your instructions into something that the computer can understand. The conversion utility is called a *compiler*.

The twist with .NET, in contrast to traditional programming languages such as C++ and C, is that the compiler generates a binary-based intermediate language called Common Intermediate Language (CIL). The .NET Framework then converts the CIL into the binary instructions required by the computer's processor.

You might think converting the source code into an intermediate language is inefficient, but it is a good approach. Let's use an analogy. There are dogs that learn quickly and those that take a while to learn. For example, German shepherds tend to learn quickly and don't require much repetition of lessons. On the other hand, bullmastiffs need quite a bit of patience, as they tend to be stubborn. Now imagine being a trainer who has created instructions on how to teach things specifically geared toward the bullmastiff. If those same instructions are used on the German shepherd, you end up boring the German shepherd and possibly failing to teach that dog what you wanted him to learn.

The problem with the instructions is that they are specifically tuned for a single dog. If you want to teach both dogs, you need two sets of instructions. To solve this problem, the instructions should be general, with added interpretation notes saying things like, "If dog is stubborn, repeat."

Converting this into computer-speak, the two sets of instructions are for two different processors or processors used in specific situations. For example, there are server computers and client computers. Each type of computer has different requirements. A server computer needs to process data as quickly as possible, whereas a client computer needs to show data on the screen as quickly as possible. There are compilers for each context, but to have the developer create multiple distributions with different compiler(s) or setting(s) is inefficient. The solution

is to create a set of instructions that are general, but have associated interpretation notes. The .NET Framework then applies these instructions using the interpretation notes.

The .NET Framework compiles to instructions (CIL) that are then converted into processor-specific instructions using notes embedded in the .NET Framework. The .NET architecture is illustrated in Figure 1-10.

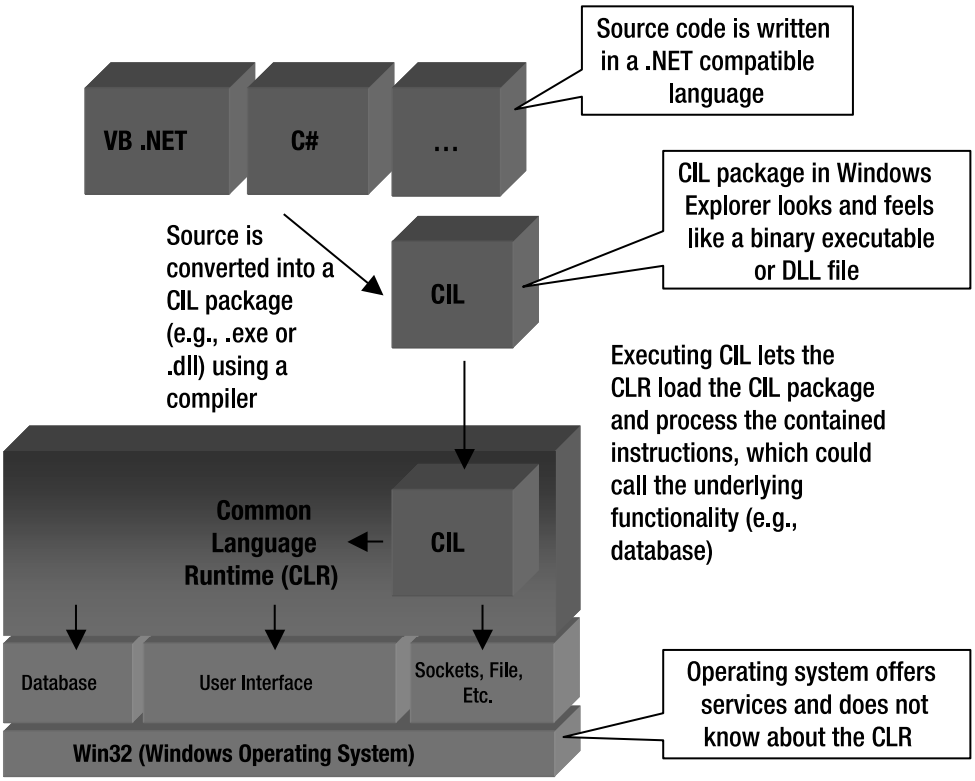

Figure 1-10. *.NET architecture*

In Figure 1-10, Visual Basic Express is responsible for converting the Visual Basic source code into a CIL package. The converted CIL package is a binary file that, when executed, requires a common language runtime (CLR). Without a CLR installed on your computer, you cannot run the CIL package. When you installed Visual Basic Express, you installed the CLR in the background as a separate package. Visual Basic Express is an application that allows you to develop for the CLR, but it also uses the CLR.

The CLR has the ability to transform your instructions in the CIL package into something that the operating system and processor can understand. If you look at the syntax of .NET-compatible languages, such as Visual Basic, C#, and Eiffel.NET, you will see that they are not similar. Yet the CLR can process the CIL package generated by one of those languages because a .NET compiler, regardless of programming language, generates a set of instructions common to the CLR.

When using the .NET Framework, you are writing for the CLR, and everything you do must be understood by the CLR. Generally speaking, this is not a problem if you are writing code in Visual Basic. The following are some advantages of writing code targeted to the CLR:

Memory and garbage collection: Programs use resources such as memory, files, and so on. In traditional programming languages, such as C and C++, you are expected to open and close a file, and allocate and free memory. With .NET, you don't need to worry about closing files or freeing memory. The CLR knows when a file or memory is not in use and will automatically close the file or free the memory.

■ **Note** Some programmers may think that the CLR promotes sloppy programming behavior, as you don't need to clean up after yourself. However, practice has shown that for any complex application, you will waste time and resources figuring out where memory has not been freed.

Custom optimization: Some programs need to process large amounts of data, such as that from a database, or display a complex user interface. The performance focus for each is on a different piece of code. The CLR has the ability to optimize the CIL package and decide how to run it as quickly and efficiently as possible.

Common Type System (CTS): A string in Visual Basic is a string in C#. This ensures that when a CIL package generated by Visual Basic talks to a CIL package generated by C#, there will be no data type misrepresentations.

Safe code: When writing programs that interact with files or memory, there is a possibility that a program error can cause security problems. Hackers will make use of that security error to run their own programs and potentially cause financial disaster. The CLR cannot stop application-defined errors, but it can stop and rein in a program that generates an error due to incorrect file or memory access.

The benefit of the CLR is allowing developers to focus on application-related problems, because they do not need to worry about infrastructure-related problems. With the CLR, you focus on the application code that reads and processes the content of a file. Without the CLR, you would need to also come up with the code that uses the content in the file and the code that is responsible for opening, reading, and closing the file.

The Important Stuff to Remember

This chapter got you started working with Visual Basic in an IDE. Here are the key points to remember:

- There are three major types of Visual Basic programs: Windows applications, console applications, and class libraries.
- A Windows application has a user interface and behaves like other Windows applications (such as Notepad and Calculator). For Windows applications, you associate events with actions.

- A console application is simpler than a Windows application and has no events. It is used to process data. Console applications generate and accept data from the command line.
- You will want to use an IDE to manage your development cycle of coding, debugging, and application execution.
- Among other things, IDEs manage the organization of your source code using projects and solutions.
- In an IDE, keyboard shortcuts make it easier for you to perform operations that you will do repeatedly. For example, in Visual Basic Express, use Ctrl+S to save your work and Ctrl+F5 to run your application without debugging.
- Visual Basic Express projects contain plain-vanilla files and specialized groupings. When dealing with specialized groupings, make sure that you understand how the groupings function and modify only those files that you are meant to modify.

Some Things for You to Do

The following are some questions related to what you've learned in this chapter. Answering them will help you to get started developing your projects in the IDE.

■ **Note** The answers/solutions to the questions/exercises included at the end of each chapter are available with this book's downloadable code, found in the Source Code/Download section of the Apress web site (`http://www.apress.com`). Additionally, you can send me an e-mail message at `christianhgross@gmail.com`.

---

1. In an IDE, solutions and projects are used to classify related pieces of functionality. The analogy I used talked about cars and car pieces. Would you ever create a solution that contained unrelated pieces of functionality? For example, would you create an airplane solution that contained car pieces?
2. Projects are based on templates created by Microsoft. Can you think of a situation where you would create your own template and add it to Visual Basic Express?
3. In the Solution Explorer, each item in the tree control represents a single item (such as a file, user interface control, and so on). If you were to double-click a `.vb` file, you would be manipulating a Visual Basic file that would contain Visual Basic code. Should a single Visual Basic file reference a single Visual Basic class or namespace? And if not, how would you organize your Visual Basic code with respect to Visual Basic files?
4. You have learned about how a .NET application can generate an executable file. Let's say that you take the generated application and execute it on another Windows computer. Will the generated application run? Let's say that you take the executable file to a Macintosh OS X or Linux computer, will the application run? Why will it run or not run?

5. You are not happy with the naming of the element `TextBox1`, and want to rename it to `TxtOutput`. How do you go about renaming `TextBox1`?

6. `ClassLibrary` has embedded logic that assumes the caller of the method is a console application. Is it good to assume a specific application type or logic of the caller in a library? If yes, why? If no, why not?