Beginning C# 2008: From Novice to Professional, Second Edition

Copyright © 2008 by Christian Gross

ISBN-13 (pbk): 978-1-4302-1033-7

ISBN-13 (electronic): 978-1-4302-1034-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewer: Christian Kenyeres
Editorial Board: Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Sofia Marchant
Developmental Editor: Ron Pronk
Copy Editor: Damon Larson
Associate Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositors: Susan Glinert and Octal Publishing, Inc.
Proofreader: Lisa Hamilton
Indexer: Broccoli Information Management
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The source code for this book is available to readers at http://www.apress.com.

CHAPTER 1

Ready, Steady, Go!

This book is about the C# programming language and helping you become a proficient C# programmer—even if you've never programmed before or you've only worked with procedural languages like Visual Basic. (C# is called an *object-oriented language*, which differs in approach from *procedural languages* like Visual Basic, Pascal, COBOL, and a lot of others that have been just about left for dead.) Object-oriented languages are not only the wave of the future, they're the wave of today. You can't program for the Web if you don't know how to use an object-oriented language like Java, C++, or C#. And if you want to use the .NET platform to program web sites and web data exchanges (an extremely popular approach), then C# is the language you want to learn.

In this chapter, you'll get started by acquiring the tools you need to develop C# applications and taking tools for a test spin. Along the way, you'll create a few C# applications.

Downloading and Installing the Tools

If you're just getting started with C# 3.0, you're probably eager to write some code that actually *does* something. The great part of .NET is that you can start writing code immediately 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 a productive and valuable coding experience.

Note This book covers the C# 3.0 programming language as it's used to write applications for the .NET Framework. With C# 3.0, you'll use the .NET 3.0 and 3.5 Frameworks. .NET 3.0 provides you with all of the programming and coding essentials, and .NET 3.5 gives you a lot of extras and many additional programming options.

For the examples in this book, we'll use Visual C# 2008 Express Edition. Why? Well, it's freely available and has everything you need to get started with C# 3.0. The other Express Edition IDEs available from Microsoft are tailored to different languages (Visual Basic and C++) or, in the case of Visual Web Developer Express, support 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 feature sets 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 installed, you can use that for the examples in this book. The Visual Studio 2008 edition can do everything that Visual C# Express can do, and, in fact, has 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 are available and determining which tools will work best for you.

Installing and downloading Visual C# Express from the Microsoft web site involves the transfer of large files. If you don't have a broadband connection, you might prefer to install the IDE from a CD, which you can order from Microsoft's online site.

Downloading Visual C# Express

Here's the procedure for downloading Visual C# Express from the Microsoft web site. By the time you read this book, the procedure might have changed, but it should be similar enough that you'll be able to find and download the IDE package.

1. Go to `http://msdn.microsoft.com/vstudio/express/`.
2. Select the Visual Studio 2008 Express Editions link.
3. Select Windows Development (because, for the scope of this book, that is what you'll be doing).
4. Click the Visual Studio Express Download link.
5. You'll see a list of Visual Studio Express editions, as shown in Figure 1-1. Click Visual C# 2008 Express Edition.
6. 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 C# Express IDE. Choose to save the file on the desktop.

These steps can be carried out very quickly—probably within a few minutes. If you follow this process, please don't mistake the procedure for downloading the complete Visual C# Express application, because that's not what happened. The installation procedure itself (which you'll perform next) will download the vast majority of the IDE. At this point, you're just downloading the initial setup file.

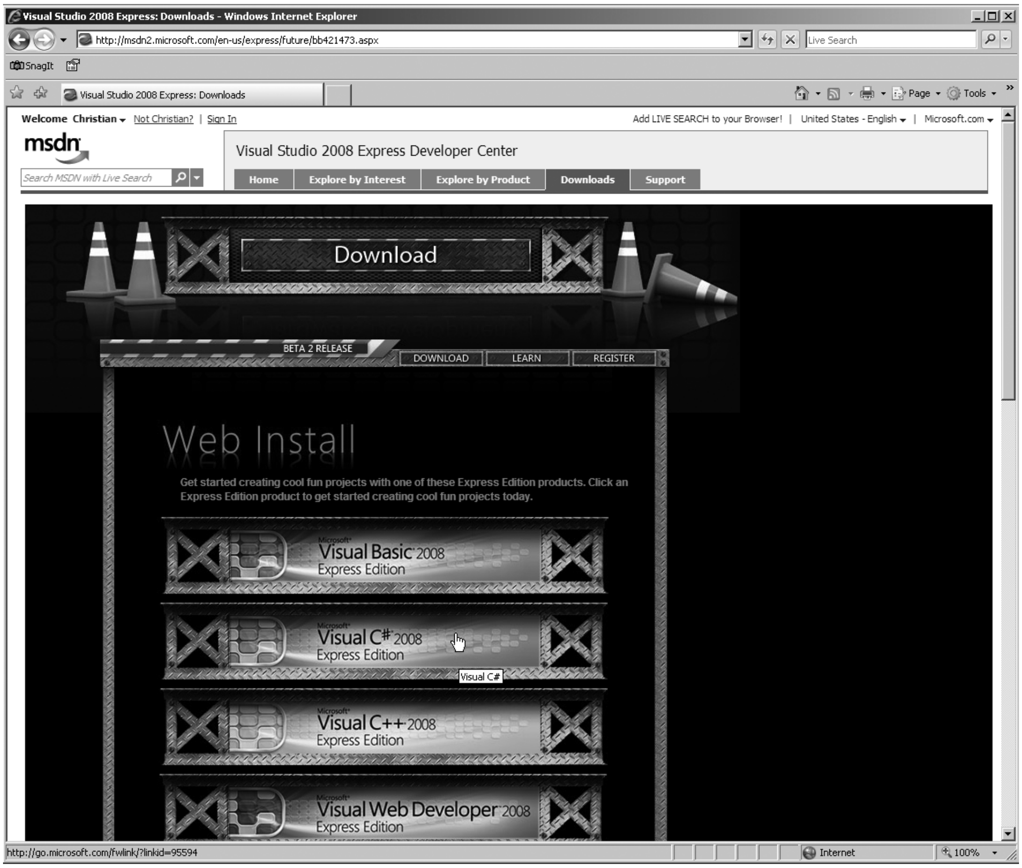

Visual Studio 2008 Express: Downloads - Windows Internet Explorer
http://msdn2.microsoft.com/en-us/express/future/bb421473.aspx
Visual Studio 2008 Express Developer Center
Home
Explore by Interest
Explore by Product
Downloads
Support
Download
BETA 2 RELEASE
DOWNLOAD
LEARN
REGISTER
Web Install
Get started creating cool fun projects with one of these Express Edition products. Click an Express Edition product to get started creating cool fun projects today.
Visual Basic 2008 Express Edition
Visual C# 2008 Express Edition
Visual C++ 2008 Express Edition
Visual Web Developer 2008 Express Edition
http://go.microsoft.com/fwlink/?linkid=95594

Figure 1-1. *Selecting Visual C# 2008 Express Edition*

After you've downloaded the setup file, you can start the Visual C# 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 `vcssetup.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 C# Express has been installed, you can start it by selecting it from the Start menu.

Choosing the Application Type

With Visual C# 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,* which is designed to run at the command line with no user interface.
- A *Windows application,* which is designed to run on a user's desktop and has a user interface.
- A *class library,* which holds reusable functionality that can be used by console and Windows applications. This library cannot be run by itself.

So now that you know what each type of program is about, in this chapter, you'll 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 what a programming language can do.

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. One project could be the steering system, another could be the exhaust system, and still another could be the starting system. Putting all of the car projects together creates a complete solution called (surprise) "the car."

The bottom line: A solution contains multiple projects that are related. For the examples in this chapter, our solution will contain three projects representing each of the three different program types.

When you use Visual C# 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 a .NET solution by creating the Windows application called `Example1` (creating this application also creates a solution).
- Add to the created solution a console application called `Example2`.
- Add to the created solution a class library project called `Example3`.

Creating the Windows Application

We'll dive right in and start with the Windows application. If you've got Visual C# Express running, you can follow these steps to create the Windows application:

1. Select File ➤ New Project from the menu.
2. Select the Windows Application icon. This represents a project style based on a predefined template called Windows Application.
3. Change the default name to `Example1`.
4. Click OK.

These steps create a new project and solution at the same time: the `Example1` solution and `Example1` project. Visual C# Express will display a complete project and solution, as shown in Figure 1-2.

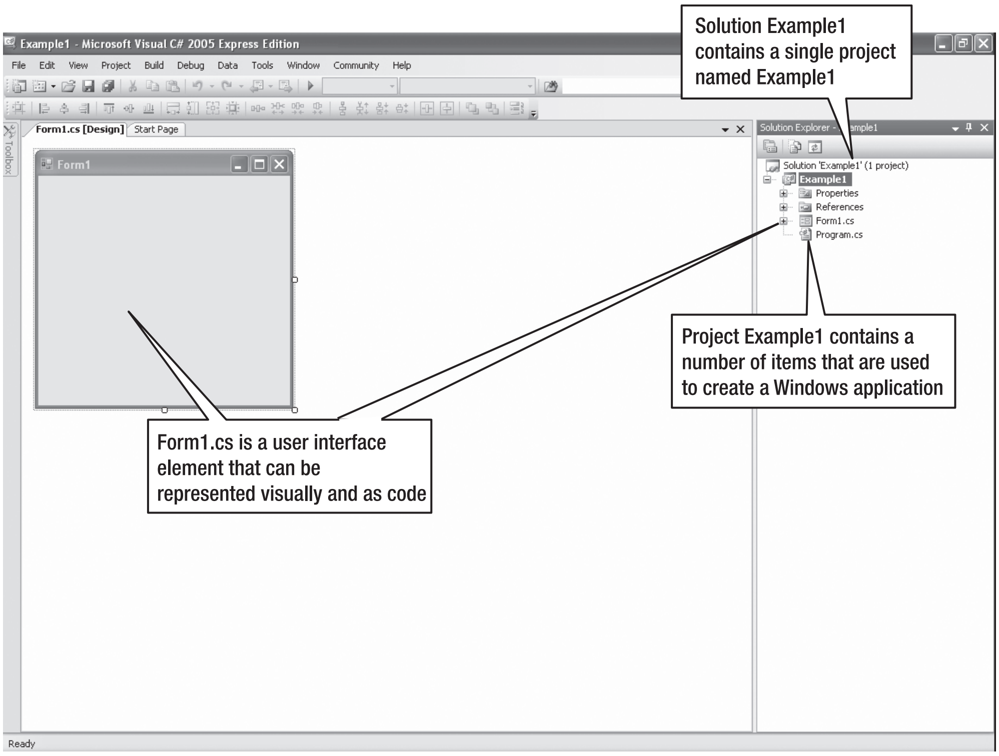
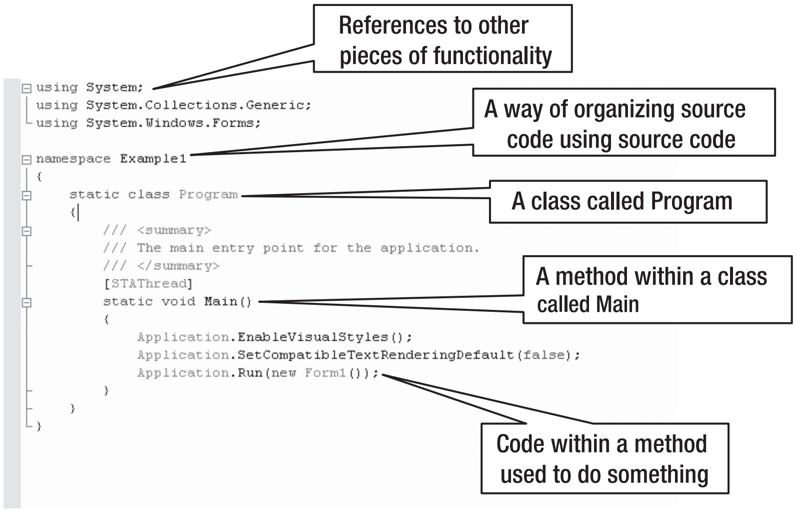

Example1 - Microsoft Visual C# 2005 Express Edition
File Edit View Project Build Debug Data Tools Window Community Help
Form1.cs [Design]
Start Page
Toolbox
Form1
Solution Explorer - Example1
Solution 'Example1' (1 project)
Example1
Properties
References
Form1.cs
Program.cs
Ready
Solution Example1 contains a single project named Example1
Project Example1 contains a number of items that are used to create a Windows application
Form1.cs is a user interface element that can be represented visually and as code

Figure 1-2. *The Visual C# Express IDE with the new Example1 project and solution*

Viewing the Source Code

When you create a new application, Visual C# Express automatically generates some source code for it. Double-click `Program.cs` in the Solution Explorer to see the generated code. The source code shown in Figure 1-3 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.cs` in the Solution Explorer. A submenu appears with the options View Code (to see the code) or View Designer (to see the user interface).



```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace Example1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

References to other pieces of functionality
A way of organizing source code using source code
A class called Program
A method within a class called Main
Code within a method used to do something

Figure 1-3. *Source code pieces in a C# file*

The elements labeled in Figure 1-3 represent the essence of the C# source code that you'll be writing. You'll learn about them throughout this book. For now, 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 might be the starter system. Another class might be the exhaust system. Still another class might be the steering system. In other words, projects are made up of multiple classes.

Method: A set of instructions that carry out a task. A method is analogous to a function in many other languages. The `Main()` method runs when an application starts; therefore, it contains the code you want to use at the beginning of any program.

Renaming the Solution

Visual C# Express names both the solution and project `Example1` automatically, which isn't ideal. Fortunately, it's easy to rename the solution. Follow these steps:

1. Right-click the solution name in the Solution Explorer and select Rename from the context menu.
2. The solution name will become editable. Change it to `ThreeExamples`.
3. Press Enter to apply the change.

You can use this same technique to rename projects or any other items shown in the Solution Explorer.

Saving the Solution

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

1. Highlight the solution name in the Solution Explorer.
2. Select File ➤ Save ThreeExamples.sln.
3. Notice that Visual C# Express wants to save the solution using the old `Example1` name, not the new solution name (`ThreeExamples`). To save the new solution name to the hard disk, you need to yet again change `Example1` to `ThreeExamples`. Note the path where Visual C# Express saves your projects—you will need to return to this path 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 Ctrl+S keyboard shortcut.

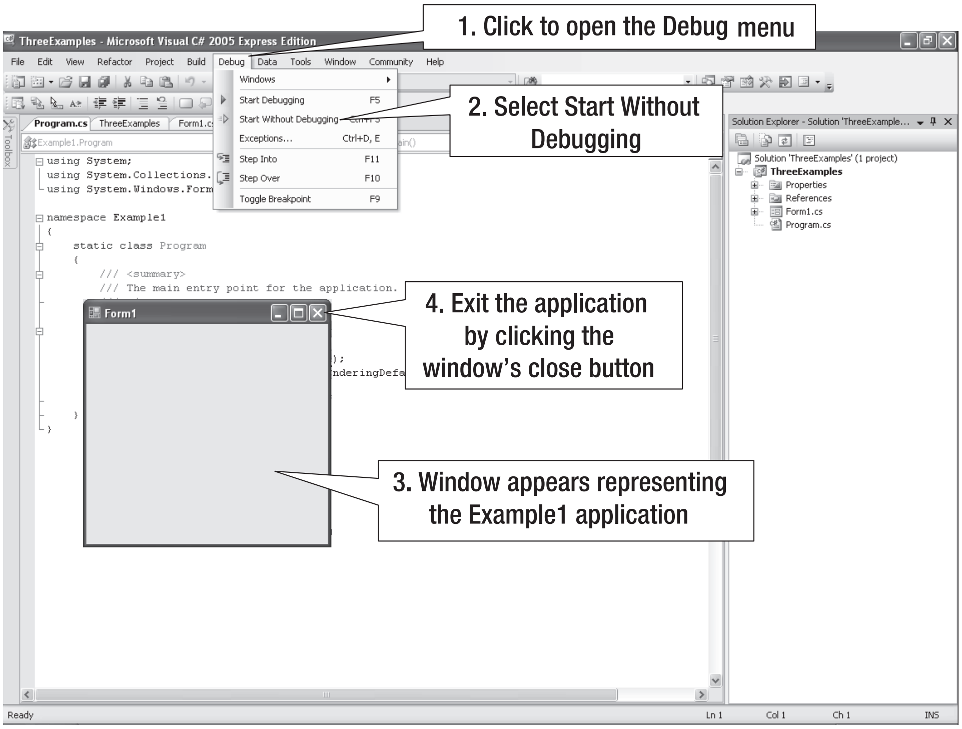
■ **Note** If you have not saved your changes and choose to exit Visual C# Express, you will be asked if you want to save or discard the solution and project.

To open a solution that 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 C# Express. The Recent Projects window is always available on the Start Page tab of the main Visual C# Express window as well.

Running the Windows Application

The source code generated by Visual C# Express is a basic application that contains an empty window with no functionality. In other words, the source code just gives you a starting point. With the source code in place, you can add more source code as desired to create your solution, debug the source code you've written, and of course use the source to run and test the application.

To run the application, select Debug ➤ Start Without Debugging. You can also use the keyboard shortcut Ctrl+F5 to accomplish the same task. When the application starts, you'll see a window that displays the application's code—in this case, the `Example1` application. You can exit the application by clicking the window's close button. Figure 1-4 illustrates the process.


1. Click to open the Debug menu
2. Select Start Without Debugging
3. Window appears representing the Example1 application
4. Exit the application by clicking the window's close button
ThreeExamples - Microsoft Visual C# 2005 Express Edition
File Edit View Refactor Project Build Debug Data Tools Window Community Help
Windows
Start Debugging F5
Start Without Debugging Ctrl+F5
Exceptions... Ctrl+D, E
Step Into F11
Step Over F10
Toggle Breakpoint F9
Program.cs ThreeExamples Form1.cs
Example1.Program
Form1
Solution Explorer - Solution 'ThreeExamples'
Solution 'ThreeExamples' (1 project)
ThreeExamples
Properties
References
Form1.cs
Program.cs
Ready Ln 1 Col 1 Ch 1 INS

```
using System;
using System.Collections.
using System.Windows.Form

namespace Example1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.

                                    );
                                    nderingDefa

    }
}
```

Figure 1-4. *Running an application*

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, `Example1` 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.

At this point, you have not written a single line of code, yet you have created an application and something actually happened, and all because Visual C# generates some boilerplate C# 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 C# Express.

Visual C# Express is both a good thing and a bad thing. Visual C# Express is good because it hides the messy details, but it's bad because the messy details are hidden. Imagine being a car mechanic, and imagine that you have very little diagnostic information to do your job. Car manufacturers produce dashboards that have lights that go on when something is wrong. For the driver, that's good. He knows to bring the car in to you. But if you're a mechanic, that's not good enough. A flashing light signals a problem, but doesn't indicate what the problem actually is. That's bad. So let's figure out how to make the good and the bad work for you.

Making the Windows Application Say Hello

The Windows application that we've created so far does nothing other than appear with a blank window that you can close. To make the application do something, we need to add user interface elements, or we need to add some code. Adding code without adding user interface elements will make the program do something, but the result won't be very friendly to the user. So, we'll add a button that, when clicked, will display "hello, world" in a text box.

To begin, you need to add a Button control to the form. Double-click `Form1.cs` in the Solution Explorer to display a blank form. Then click the Toolbox tab to access the controls. Click Button, and then click the form to place the button on the form. These steps are illustrated in Figure 1-5.

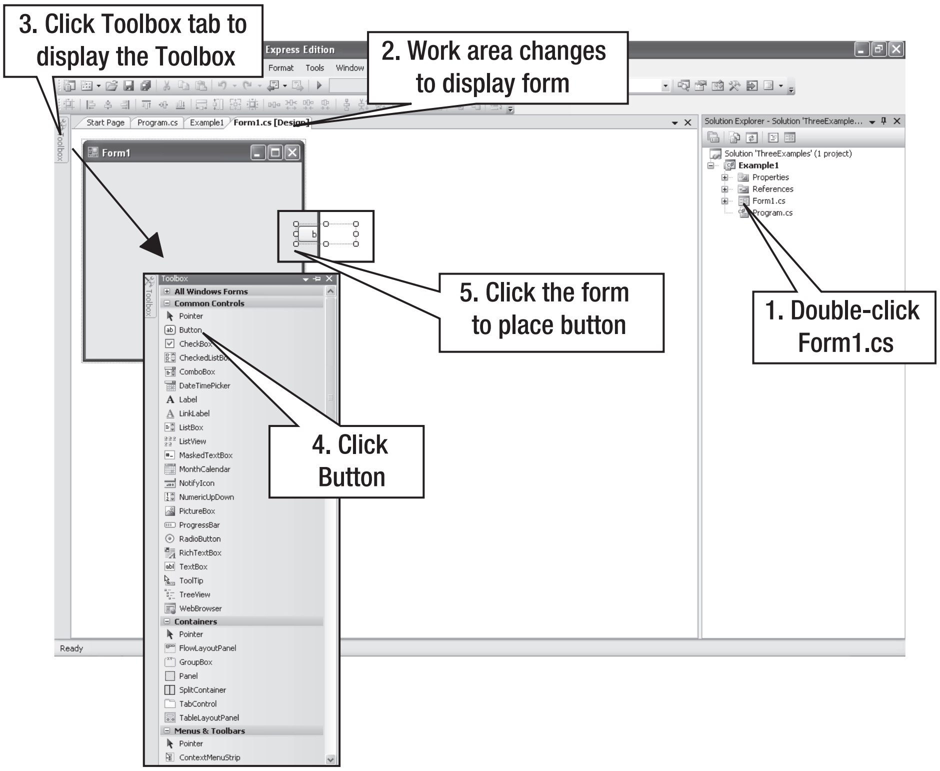


Figure 1-5. *Adding a button to the form*

Next, we'll add a TextBox control using the same basic procedure. Finally, we'll 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 C# Express will "snap" the edge of a control to the nearest geometrical edge as you drag the control. This kind of helpful support enables you to align controls more accurately.

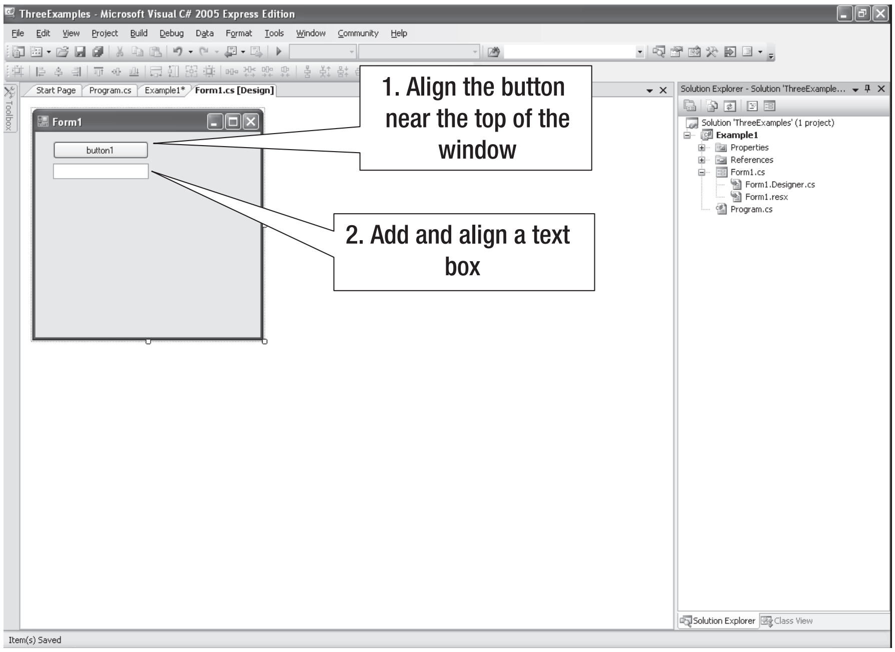

ThreeExamples - Microsoft Visual C# 2005 Express Edition
File Edit View Project Build Debug Data Format Tools Window Community Help
Start Page Program.cs Example1* Form1.cs [Design]
Form1
button1
1. Align the button near the top of the window
2. Add and align a text box
Solution Explorer - Solution 'ThreeExamples'
Solution 'ThreeExamples' (1 project)
Example1
Properties
References
Form1.cs
Form1.Designer.cs
Form1.resx
Program.cs
Solution Explorer
Class View
Item(s) Saved

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

If you've executed `Example1` by pressing Ctrl+F5, you should then see a window that contains the button and text box shown in Figure 1-6. You can click the button, and you can add or delete text from the text box. But whatever you do has no effect right now, because neither control has been associated with any code. At this point, the button and text box are just static user interface elements.

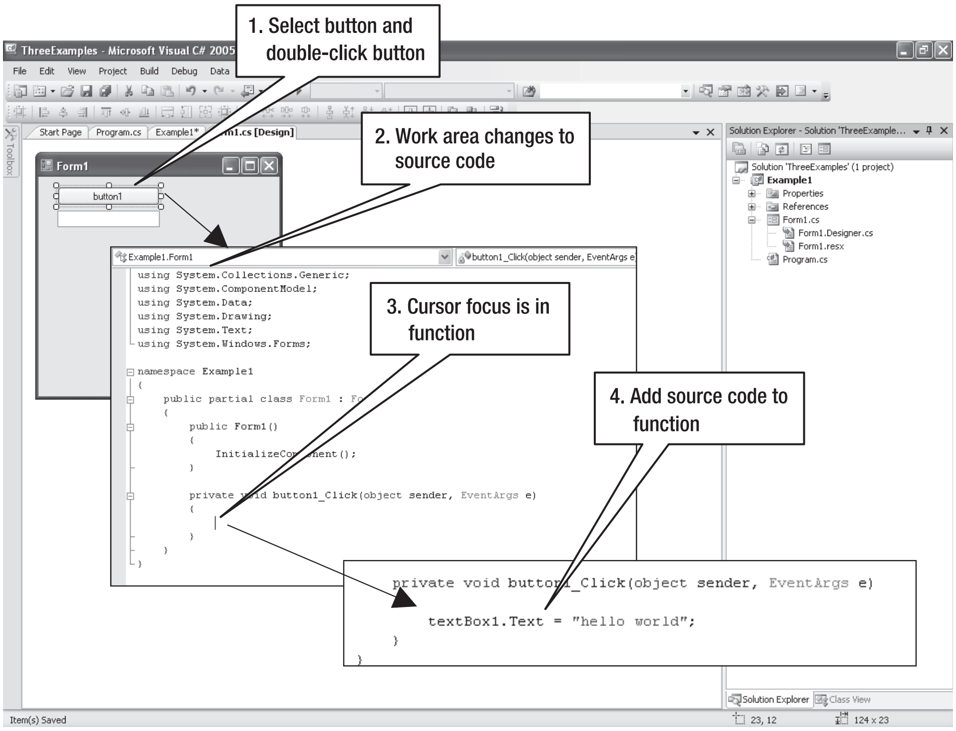
To make the application do something, you need to think in terms of *events*. For example, if you have a garage with an automatic door opener, you would expect that pressing the remote control button would open the garage door when it's closed and that the button would close the door when it's open. In `Example1`, we'll associate the clicking of the button with an action that displays text in the text box.

Select the button on the form and double-click it. The work area changes to source code, with the cursor in the `button_Click` function. Add this source code:

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

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

Note that `textBox1` is the name of the text box you added to the form. This name is generated by Visual C# 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 in place for this example.


1. Select button and double-click button
2. Work area changes to source code
3. Cursor focus is in function
4. Add source code to function
ThreeExamples - Microsoft Visual C# 2005
File Edit View Project Build Debug Data
Start Page Program.cs Example1* Form1.cs [Design]
Form1
button1
Example1.Form1
button1_Click(object sender, EventArgs e)
Solution Explorer - Solution 'ThreeExamples'
Solution 'ThreeExamples' (1 project)
Example1
Properties
References
Form1.cs
Form1.Designer.cs
Form1.resx
Program.cs
Solution Explorer
Class View
Item(s) Saved
23, 12
124 x 23

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Example1
{
    public partial class Form1 : Fo
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            |
        }
    }
}
```

```
    private void button1_Click(object sender, EventArgs e)

        textBox1.Text = "hello world";
    }
}
```

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

Adding an action to an event is very simple if you follow the instructions shown in Figure 1-7. The simplicity is due to Visual C# Express's automation capabilities; not necessarily because the event or action itself is simple. Visual C# 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-change 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 C# application! You have associated an event with an action: that is, the button-click event is associated 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's a good idea to document what the program does—within the source code itself. If you've ever programmed in another language, you probably already know the value of internal documentation. If you come back to the application in the future (sometimes many months or even years later), you won't be puzzled by your previous work. In fact, you might not even be the person who maintains your code, so leaving comments in the code to help explain it is definitely good practice.

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

```
// A single-line comment
```

Anything after the // 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 void button1_Click(object sender, EventArgs e)
{
    textBox1.Text = "hello, world";
}
```

It's always worth leaving simple comments like this as you go, because it helps greatly when working out an application's logic. But what if you want to leave lengthier comments—several lines long—such as a more detailed comment describing a whole class? In such a case, you would use the syntax for a multiline comment:

```
    /* Here's the first line of a multiline comment.
     * Here's the second line.
     * Here's the third line.
     */
```

As you can see, in a multiline comment, the /* starts the comment and the */ ends it; anything in between is ignored by the compiler as before. Note that the asterisks before the second and third lines are added by Visual C# Express as extra dressing, but are not a requirement of a multiline comment. (In other words, you don't *need* to include the asterisks yourself, although you can if you want. However, you do need to make sure you include the closing */ symbols.)

Let's write a multiline comment for our Windows application:

```
namespace Example1
{
    /* This is the example simple form for Chapter 1,
     * which displays text when the user clicks the button.
     * It is a first look at event-driven programming.
     */

    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        // When the user clicks the button, we
        // display text in the text box
        private void button1_Click(object sender, EventArgs e)
        {
            textBox1.Text = "hello, world";
        }
    }
}
```

Visual C# Express supports other types of comments that can be of benefit to end users as well as other programmers who review your code. You'll learn about these other approaches in Chapter 10.

Navigating the User Controls of the Solution

When you're writing code, your most important form of navigation is the Solution Explorer, which is the tree control that contains 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 might inadvertently make are not saved.

The left pane of the Solution Explorer is your work area—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 `Program.cs` in the Solution Explorer, the work area displays the code related to the `Program.cs` file.

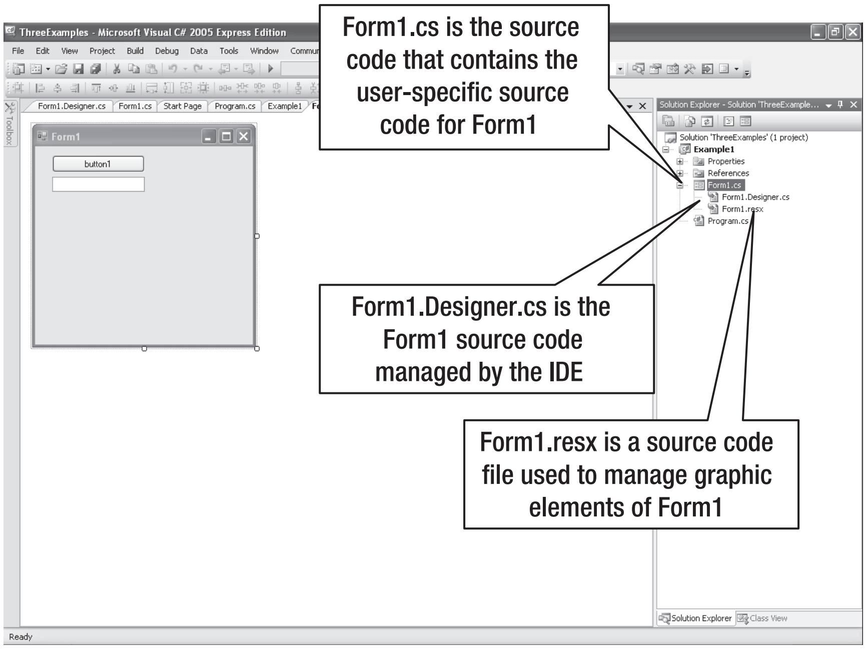
`Program.cs` is a plain-vanilla source code file from `Example1`. Plain-vanilla source code files have no special representation in Visual C# Express and simply contain source code. `Program.cs` contains source code to initialize the application and looks like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace Example1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Plain-vanilla source code files contain the logic that makes your application do something useful. The advantage of plain-vanilla source code files is that they provide a complete view of your application's logic. A typical application will contain many plain-vanilla source code files.

The Solution Explorer also shows specialized groupings, which are specific items that Visual C# Express recognizes and organizes. A specialized grouping contains a number of files that rely on each other and implement a specific piece of functionality. `Form1` is an example of a specialized grouping that manages the layout of the user interface, elements of the user interface, and your custom code. The individual file pieces of `Form1` are illustrated in Figure 1-8.


ThreeExamples - Microsoft Visual C# 2005 Express Edition
File Edit View Project Build Debug Data Tools Window
Form1.Designer.cs Form1.cs Start Page Program.cs Example1
Form1
button1
Solution Explorer - Solution 'ThreeExamples'
Solution 'ThreeExamples' (1 project)
Example1
Properties
References
Form1.cs
Form1.Designer.cs
Form1.resx
Program.cs
Solution Explorer Class View
Ready

Form1.cs is the source code that contains the user-specific source code for Form1

Form1.Designer.cs is the Form1 source code managed by the IDE

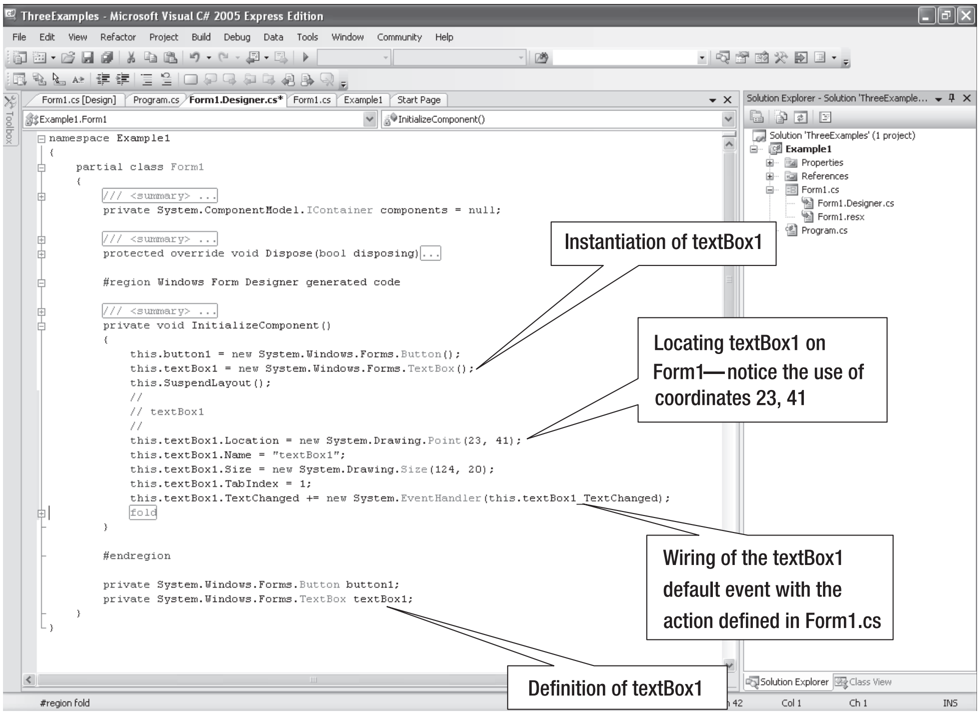
Form1.resx is a source code file used to manage graphic elements of Form1

Figure 1-8. *Specialized grouping with three files*

In Figure 1-8, the Solution Explorer contains a top-level item called `Form1.cs`, which is a source code file that contains the user-defined pieces of `Form1`, which can be represented in one of two ways in the work area: graphically and textually (source code). For the most part, you will be editing `Form1.cs` using source code and graphical means, and will let Visual C# Express handle the `Form1.Designer.cs` and `Form1.resx` files.

The `Form1` specialized grouping exists to make the organization of the code that represents the user interface of `Form1` easier to manage for you and the IDE. You can still edit the `Form1.Designer.cs` and `Form1.resx` files. If you double-click `Form1.Designer.cs`, you will see source code, which you can modify. However, be forewarned that if you mess up the source code in that file, Visual C# Express might stop functioning properly when editing `Form1`.

Knowing that the specialized grouping called `Form1` should be taken as a whole, you might wonder where the definition of `textBox1` came from. The answer is that `textBox1` is defined and assigned in one of the IDE-generated source code files. Figure 1-9 illustrates what the generated source code file does with `textBox1`.



```
namespace Example1
{
    partial class Form1
    {
        /// <summary> ...
        private System.ComponentModel.IContainer components = null;

        /// <summary> ...
        protected override void Dispose(bool disposing)...

        #region Windows Form Designer generated code

        /// <summary> ...
        private void InitializeComponent()
        {
            this.button1 = new System.Windows.Forms.Button();
            this.textBox1 = new System.Windows.Forms.TextBox();
            this.SuspendLayout();
            //
            // textBox1
            //
            this.textBox1.Location = new System.Drawing.Point(23, 41);
            this.textBox1.Name = "textBox1";
            this.textBox1.Size = new System.Drawing.Size(124, 20);
            this.textBox1.TabIndex = 1;
            this.textBox1.TextChanged += new System.EventHandler(this.textBox1_TextChanged);
            fold
        }

        #endregion

        private System.Windows.Forms.Button button1;
        private System.Windows.Forms.TextBox textBox1;
    }
}
```

ThreeExamples - Microsoft Visual C# 2005 Express Edition
File Edit View Refactor Project Build Debug Data Tools Window Community Help
Form1.cs [Design] Program.cs Form1.Designer.cs* Form1.cs Example1 Start Page
Example1.Form1
InitializeComponent()
Solution Explorer - Solution 'ThreeExample...
Solution 'ThreeExamples' (1 project)
Example1
Properties
References
Form1.cs
Form1.Designer.cs
Form1.resx
Program.cs
Instantiation of textBox1
Locating textBox1 on Form1—notice the use of coordinates 23, 41
Wiring of the textBox1 default event with the action defined in Form1.cs
Definition of textBox1
Solution Explorer
Class View
#region fold
Col 1
Ch 1
INS

Figure 1-9. *Visual C# Express–generated code*

Notice that everything—definitions, wiring of events to actions, and placement of the controls—is managed by Visual C# Express. For example, if you were to change the placement of `textBox1` by altering the location coordinates, Visual C# Express would read and process the change. However, if you were to make larger changes that Visual C# Express could not process, you would corrupt the user interface.

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 ➤ Accessories ➤ Command Prompt. Alternatively, select Start ➤ Run and type `cmd` in the dialog box.

Visual C# 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. Right-click the solution name, `ThreeExamples`, in the Solution Explorer.
2. Select Add ➤ New Project.
3. Select Console Application and change the name to `Example2`.

The Solution Explorer changes to show the additional project, and the work area displays the source code in the new `Program.cs`.

Notice the simplicity of the console application. It contains a single plain-vanilla source code file, called `Program.cs`. 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:

```
namespace Example2
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("hello, world");
        }
    }
}
```

The bold line in the preceding code 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 `Example1` 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 `Example1` project is in bold type in the Solution Explorer? That means `Example1` is the startup project. Whenever you run or debug an application, the startup project is executed or debugged.

To switch the startup project to `Example2`, right-click the `Example2` project and select Set As Startup Project. `Example2` will now be in bold, meaning it is the startup project of the `ThreeExamples` solution.

Running the Console Project

With `Example2` set as the startup project, you can now press Ctrl+F5 to run the console application. The output is as follows:

```
hello, world
Press any key to continue
```

Executing the console application does not generate a window, as did the Windows application. Instead, a command prompt is started with `Example2` as the application to execute. Executing that application generates the text “hello, world.” You can also see that you can press any key to close the command prompt window. Visual C# Express automatically generated the code to show this output and executed 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 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 that the Windows application and console application can 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 `Example3`.

The resulting solution project should look like Figure 1-10.

The added `Example3` project has a single file called `Class1.cs`, which is a plain-vanilla source code file.

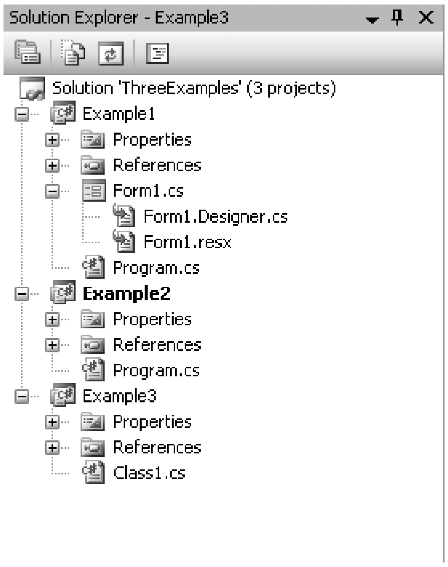


Figure 1-10. *Updated solution structure that contains three projects*

Moving Functionality

Now we will move the code used to say "hello, world" from `Example2` to `Example3`. Add the code to `Class1.cs` as follows (the bold code):

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Example3
{
    public class Class1
    {
        public static void HelloWorld()
        {
            Console.WriteLine("hello, world");
        }
    }
}
```

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

For different applications to actually share the code that's in a class library, you must make the projects aware of each other's existence.

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 A project only knows about the functionality that has been declared as being public. Public functionality, or what C# 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 `Example2` aware of the functionality in `Class1`, you need to set a physical reference, as follows:

1. Click and expand the References node under `Example2`. Notice that three references already exist. When you typed `Console.WriteLine()` in `Class1.cs`, you were using functionality from the `System` reference.
2. Right-click References and select Add Reference.
3. Click the Projects tab.
4. Select `Example3`, and then click OK. `Example3` will be added to the `Example2` references.

Once the reference has been assigned, `Example2` can call the functionality in `Example3`.

Note In `Class1.cs`, the first three lines begin with the keyword `using`. The keyword `using` tells Visual C# Express you want to use the functionality defined in the reference after the `using` keyword. This is a shortcut, which we purposely didn't use in this example so that you could see another way of using a reference.

Calling Class Library Functionality

Now we need to change `Example2` so that it calls the function in `Example3`. Let's modify the `Program.cs` file in `Example2` as follows:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Example2
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("hello, world");
            Example3.Class1.HelloWorld();
        }
    }
}
```

Run `Example2` 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 `Example3.Class1.HelloWorld()` generates the second “hello, world.”

USING REFERENCE SHORTHAND

`Example3.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, we have used the `using System` line, so we don’t need to do it this way.

To use shorthand for the `Example3` call, we would include a new `using` line at the beginning of `Program.cs` in `Example2` and change the call to `Class1`’s `HelloWorld()` method:

```
using System;
using System.Collections.Generic;
using System.Text;
using Example3;

namespace Example2
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("hello, world");
            Class1.HelloWorld();
        }
    }
}
```

But shorthand like this has a downside. What if we had many references, each containing a class called `Class1`? In this case, Visual C# 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.

Using Variables and Constants

One of the core concepts in a C# program is the use of 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 Example3 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, because we can define some data (the message to print), and then use it later in our program.

```
namespace Example3
{
    public class Class1
    {
        public static void HelloWorld()
        {
            // The message to display, held in a variable
            string message = "hello, world";
            Console.WriteLine(message);
        }

    }
}
```

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 shown before. This time, however, we've set the message to display in a separate statement.

This is very useful for us, but there is more to variables than this. 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 static void HelloWorld()
        {
            // The message to display
            string message = "hello, world";
            Console.WriteLine(message);
        }

        public static void DisplayMessageText()
        {
            Console.WriteLine("The message text is: ");
            Console.WriteLine(message);
        }
```

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 `static`, it must also be `static`):

```
public class Class1
{
    // The message to display
    static string message = "hello, world";

    public static void HelloWorld()
    {
        Console.WriteLine(message);
    }


    public static void DisplayMessageText()
    {
        Console.WriteLine("The message text is: ");
        Console.WriteLine(message);
    }
}
```

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 `static` 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:

```
    // The message to display
    const string MESSAGE = "hello, world";

    public static void HelloWorld()
    {
        Console.WriteLine(MESSAGE);
    }

    public static void DisplayMessageText()
    {
        Console.WriteLine("The message text is: ");
        Console.WriteLine(MESSAGE);
    }
```

Constant names should always be all uppercase (capital letters). The contents of a constant cannot be changed at any point. The following would not compile, for instance:

```
// The message to display
const string MESSAGE = "hello, world";

public static void HelloWorld()
{
    MESSAGE = "goodbye, world";
    Console.WriteLine(MESSAGE);
}
```

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

Understanding How the .NET Framework Works

When you write C# source code, you are creating instructions for the program to follow. The instructions are defined using the C# 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. You probably already know that this conversion utility is the 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 that converting the source code into an intermediate language is inefficient, but in truth, it really is a good approach. Let's use an analogy. 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 the German shepherd 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 1-11.

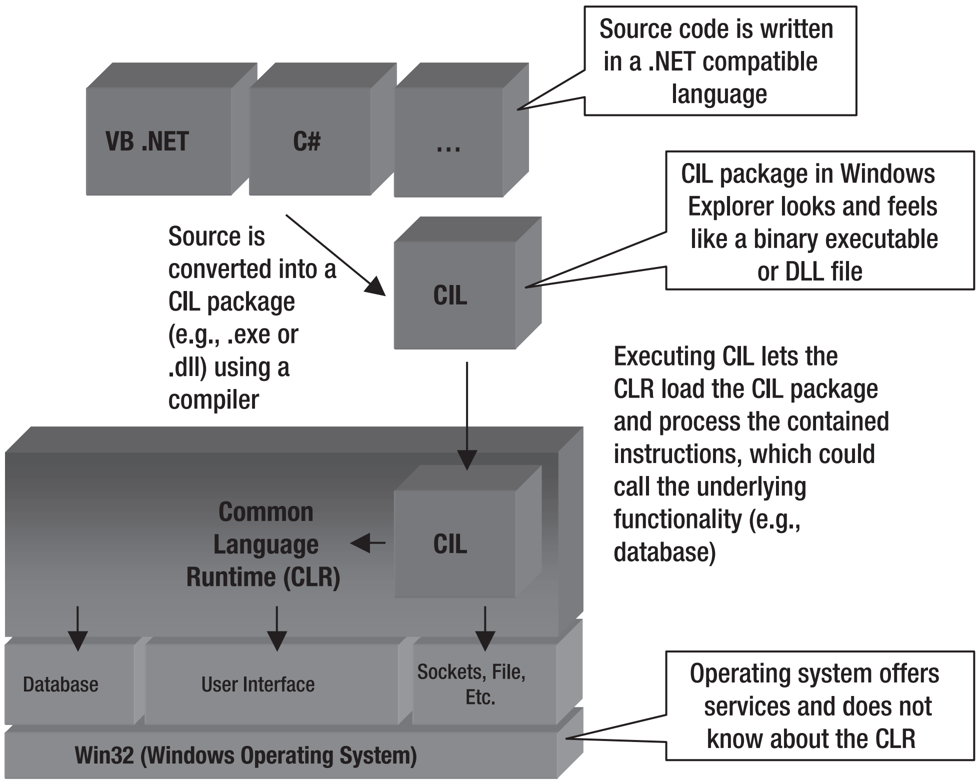

Source code is written in a .NET compatible language
VB .NET
C#
...
CIL package in Windows Explorer looks and feels like a binary executable or DLL file
Source is converted into a CIL package (e.g., .exe or .dll) using a compiler
CIL
Executing CIL lets the CLR load the CIL package and process the contained instructions, which could call the underlying functionality (e.g., database)
Common Language Runtime (CLR)
CIL
Database
User Interface
Sockets, File, Etc.
Operating system offers services and does not know about the CLR
Win32 (Windows Operating System)

Figure 1-11. *.NET architecture*

In Figure 1-11, Visual C# Express is responsible for converting the C# 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 C# Express, you installed the CLR in the background as a separate package. Visual C# Express is an application that allows you to develop for the CLR, but of course it also allows you to use 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#, or 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 C#. 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, because, with it, 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 still a string in C#. This ensures that when a CIL package generated by C# talks to a CIL package generated by Visual Basic, there will be no data type misrepresentations.

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

The benefit of the CLR is its ability to allow developers to focus on application-related problems, because developers don't 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 C# in an IDE. Here are the key points to remember:

- There are three major types of C# 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 C# Express, use Ctrl+S to save your work and Ctrl+F5 to run your application without debugging.
- In Visual C# Express projects, there are 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 C# 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 `.cs` file, you would be manipulating a C# file that would contain C# code. Should a single C# file reference a single C# class or namespace? And if not, how would you organize your C# code with respect to C# files?
4. You have learned about how a .NET application generates 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. `Example3` 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?