# Part I

# Programming Basics

# Chapter 1

# Introduction to C# Coding and Debugging

## Key Skills & Concepts

- Brief History of C#
- Understanding Basic Code Terminology
- Writing and Running a Program
- Using the Integrated Development Environment
- Incorporating Debugging Techniques

At this point you must be anxious to get coding—me too! To prepare, after a brief history of C#, we will begin with a discussion on code terminology. Then we are going to write and run some simple programs while getting familiar with the development environment and debugging tools. When this chapter is done, you will be ready to immerse yourself in code for the rest of the book.

# Brief History of C#

C#, pronounced *C sharp,* is a multipurpose language that can be used to develop console, windows, web, and mobile applications. C# was first introduced as part of the Microsoft .NET Framework, which was announced and demonstrated in July 2000 at Microsoft's Professional Developers Conference. C# became an official language specification in 2002 with the official release of .NET 1.0. C# is popular due to its ease of use on different Windows platforms, its well-organized object-oriented methodology, and its C-style syntax that is familiar to most software developers today.

C# is a programming language that is enabled through the .NET Framework. The .NET Framework is a common environment and toolset for creating, building, and running Windows applications, Windows services, web applications, and web services with the Microsoft platform. The .NET Framework also includes common class libraries that provide functionality and structures for all languages that use the .NET Framework. Other languages within this framework include Visual Basic .NET (VB.NET) and Managed C++.

# Understanding Basic Code Terminology

To begin our discussion of C# as a programming language, let's define some basic code structures:

- **Variables** Variables are user-defined named references to stored data. Variable names are usually nouns that describe the items stored. Properly named variables help to narrate your code.
- **Methods** Methods are groups of instructions that perform a routine. Method names are usually verbs that describe the action they perform. Methods help reduce duplicate code by enabling code reuse in addition to making programs easier to read and debug.
- **Classes** Classes are templates that provide a related grouping of methods, data, and other constructs. Classes are used to create objects.
- **Namespaces** Namespaces are logical groupings of classes. Referencing namespaces (with "using" statements) at the top of your code file lets you use their classes in that file. In addition to using the namespaces provided by the .NET Framework, you will routinely want to create your own namespaces to organize your classes. In every example within this book, you will see a reference to at least one library of the .NET Framework. Most often, the *System* namespace is referenced in the examples to access the *Console* class for enabling writing to the console window. Namespaces also prevent conflicts that occur when classes have identical names. When more than one class share the same name, the namespace of the class is used in combination with the class name to identify it.

## Comments

In addition to making sure that you use proper naming of variables, methods, classes, and namespaces, you will at times want to add comments to narrate your code. Comments are visible to anyone who reads the code. Comments are not included in the executable application. There are three ways to write comments in C#:

- **Single-line comments** Single-line comments are preceded by two forward slashes, //. For example:

```
// Comments about the code structure go here.
```

- **Multiple-line comments** Multiple-line comments begin with /* and end with */, as shown next. This comment style may be used for narrative paragraphs.

```
/* Information about the code
   structure belongs here. */
```

- **XML tag comments** XML tag comments are preceded by triple forward slashes, ///, at the start of each line. This comment style enables auto-generation of HTML documentation for code blocks in your application through a transformation that is enabled in Microsoft Visual Studio. XML tag comments also enable tooltip hints for describing code blocks. The nine main comment tags are <remarks>, <summary>, <example>, <exception>, <param>, <permission>, <returns>, <seealso>, and <include>. XML tag comments may precede a method to provide a summary of its parameter and list return type. An example follows:

```
/// <summary>
/// Method description goes here.
/// </summary>
/// <param name=''varName''>Parameter description.</param>
/// <returns>Return value description.</returns>
```

## Syntax

Syntax defines the set of rules for structural correctness of a programming language. When writing code, aside from comments, every semicolon, brace, parenthesis, word, and letter case must be accurate. Improper syntax will stop your code from compiling into an executable program.

## Indentation

Opening and closing curly braces are required to enclose structures such as classes, methods, and more. For readability and by convention, it is essential that you indent and left-align your code within each pair of curly braces. Structures inside structures must also be indented and left-aligned:

```
namespace ConsoleApplication1 // Namespace declaration.
{
    class Program             // Class inside namespace.
    {
        static void Main()    // Method inside class.
        {
        }
    }
}
```

The material in this book often implements a common variation of code indentation to present code instructions in fewer lines by placing the opening curly brace at the end of the loop header, method header, or class header:

```
namespace ConsoleApplication1 { // Namespace declaration.
    class Program {             // Class inside namespace.
        static void Main() {    // Method inside class.
        }
    }
}
```

# Writing and Running a Program

When you write code, you are writing human-readable instructions. C# is considered a high-level language because the instruction set is user friendly.

## Writing and Editing Your Code

Normally you will want to write and edit your C# code inside some edition of Microsoft Visual Studio, since it integrates several different tools to help you to edit, manage, and compile your code solution. A *solution* is a collection of projects. It is possible to have more than one project in a solution, but the solutions in this book always include only one project. Figure 1-1 shows a solution within the Visual Studio integrated development environment (IDE) that contains a project named ConsoleApplication1. By default, the code editor is displayed in the large section on the left. The Solution Explorer is on the right. The Solution Explorer provides easy viewing and navigation for code files and other resources. Also, on the bottom right in Figure 1-1, you can see the Properties window. The Properties window provides a summary of the properties for any file or resource that is selected within the project. In this case, the Properties window displays the file name and path of the Program.cs file which is selected in the Solution Explorer. An Error List window also exists at the bottom of the IDE in Figure 1-1. The Error List window displays any errors that prevent your code from running. This window will be discussed in more detail later in this chapter during the section on debugging techniques.

***NOTE***

Until we begin Part II, all classes, methods, and variables that are declared at the class level must be declared with a static modifier. Variables that are declared within methods are nonstatic. The static modifier will be explained in Chapter 8.

## Program Compilation and Execution

After you finish writing your C# instructions, you then need to compile them before you can run your program. When you compile your C# code, you are actually converting the instructions into a bytecode-style format that runs on Windows.

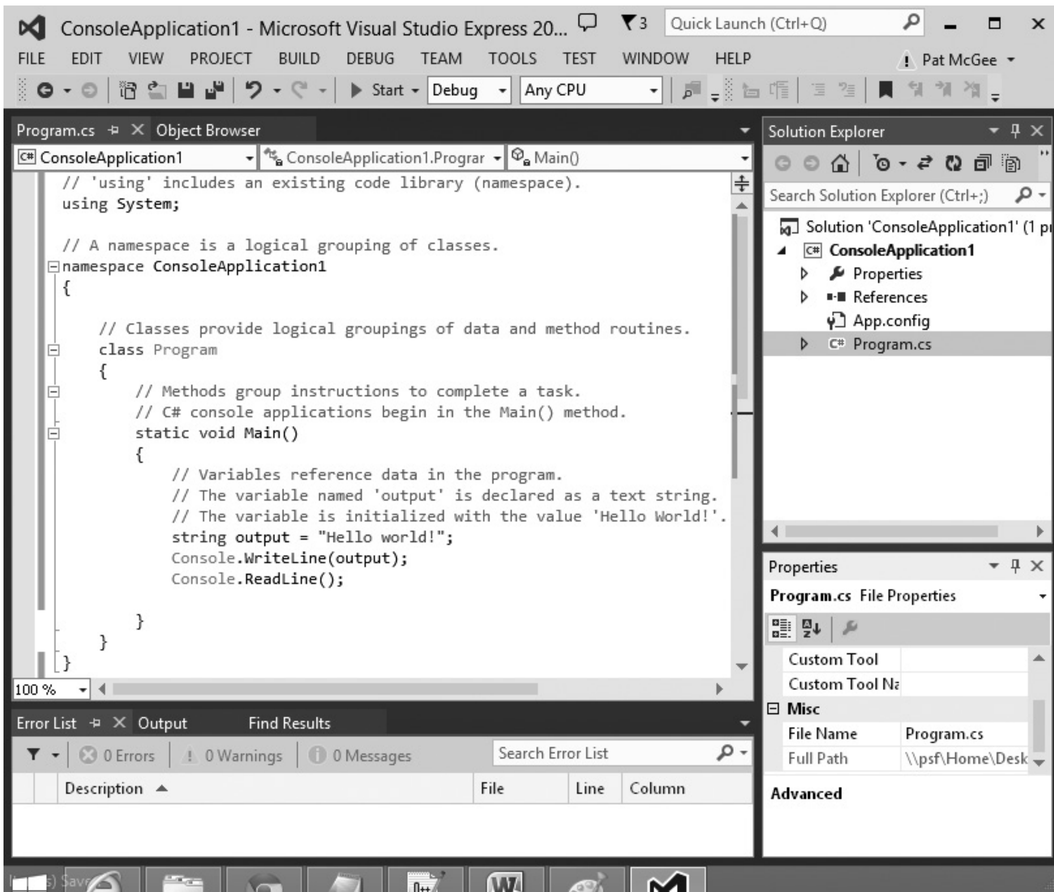


**Figure 1-1** The Visual Studio integrated development environment

## Try This 1-1 Creating Your First C# Program

This section provides your first opportunity to practice writing code and using Visual Studio to compile and run it. In this walkthrough, you will write a program that prints "Hello world!" to a console window.

1. Launch Microsoft Visual Studio from the Start menu.
2. Once Visual Studio is open, select File | New | Project. In the New Project dialog that opens, expand the Templates node and select the Visual C# node on the left. From there select Console Application from the list that appears in the center of the New Project dialog, as shown in Figure 1-2.

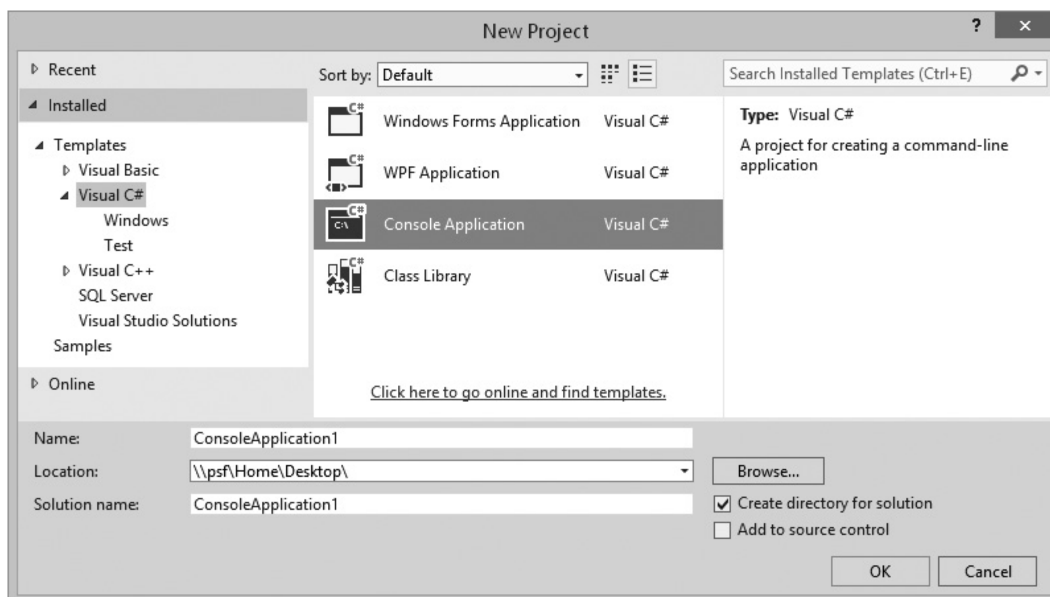


**Figure 1-2** Creating a console application

### NOTE

All examples in this book run in a console application project. When not stated, code that is presented for an example belongs in the Program.cs file.

3. Once the project has been created, double-click the Program.cs file in the Solution Explorer panel on the right to view the code. Replace the contents of Program.cs with the written instructions that are displayed in Figure 1-1. Comments are not compiled, so you are not required to include them.

### CAUTION

Remember that the spelling, letter case, and punctuation used in your code must all be correct or your program will not compile and run.

4. After writing your code, compile and run the program. To do this, either click the green arrow button, labeled Start, which is located in the Visual Studio toolbar (refer to Figure 1-1), or press F5. Running the program launches the console window to display the message “Hello world!” (see Figure 1-3).
5. You may terminate the program by either pressing ENTER in the console application or pressing SHIFT-F5 when in Visual Studio.

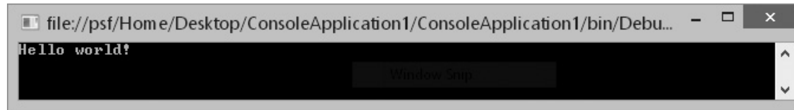


**Figure 1-3** Output from the console application

# Using the Integrated Development Environment

Now that you have written, compiled, and run your first program using Microsoft Visual Studio, this section covers some basics for managing your code projects with this tool.

## Creating a Console Application Project

The steps to create a console application project are the same as the first few steps you followed to create your first C# program. Launch Visual Studio, select File | New | Project, and, under the Templates node, select Visual C#. From there, select Console Application from the list in the center of the New Project dialog (refer to Figure 1-2).

## Compiling and Running a Program

Compiling a program refers to the process of converting C# human-readable instructions to a bytecode format that can be executed. To compile and run a program from Visual Studio, select Debug | Start Debugging or press the F5 key. Alternatively, just click the Start button in the Visual Studio toolbar, as shown earlier in Figure 1-1.

## Stopping the Application

When running the program from Visual Studio, you can stop its execution by placing your cursor in the Microsoft Visual C# code editor window and then pressing SHIFT-F5. You can also stop the program by clicking the toolbar button with the small red square icon. This button is only visible in Visual Studio while the program is running.

Unless specified, the console will close automatically when a program finishes running. When you're running a console application, to stop the command window from closing when the application finishes, you can add the instruction *Console.ReadLine()* at the end of the program to force it to wait for the user to press ENTER before terminating.

## Saving the Solution

By default, the act of compiling code from Visual Studio will automatically save any code changes. To save one file at a time while viewing each code file, select File | Save or press CTRL-S. Also, to save all code and resource changes in your solution, select File | Save All or press CTRL-SHIFT-S.

## Exiting the Solution

When the application is not running, you may exit the Visual Studio environment by navigating to File | Exit.

## Opening the Solution

To open a Visual Studio solution with one of your code projects in it, navigate to the folder where your project is stored and double-click the solution file. The solution file is generated whenever you create a console application. The solution file has a .sln extension. Double-clicking the solution file will launch Visual Studio with your code project inside it. When Visual Studio is running, you can open a solution from the File | Open menu. You can also use the CTRL-O shortcut from Visual Studio to launch the Open File dialog, which you can use to navigate to the .sln file.

## Renaming a Code File

To rename a code file, right-click the code file name in the Solution Explorer and then choose Rename in the drop-down list. You will be prompted to enter a new name for your code file. Alternatively, while the file is selected, press F2 and rename it when prompted.

## Renaming Code Structures

Renaming variables, methods, classes, and namespaces is a very important part of development. Good coders rely heavily on proper names for variables, methods, and classes to accurately describe how the code works. Due to the iterative nature of code development, you will often think of better names for your structures much later, after declaring them. For these reasons, you will want to use both the Find and Replace dialog and the Rename dialog to rename your variables, methods, and classes as you think of better names.

### The Find and Replace Dialog

To launch the Find and Replace dialog, select Edit | Find and Replace | Replace In Files or use the keyboard shortcut CTRL-SHIFT-H. Figure 1-4 shows dialog settings used to rename a

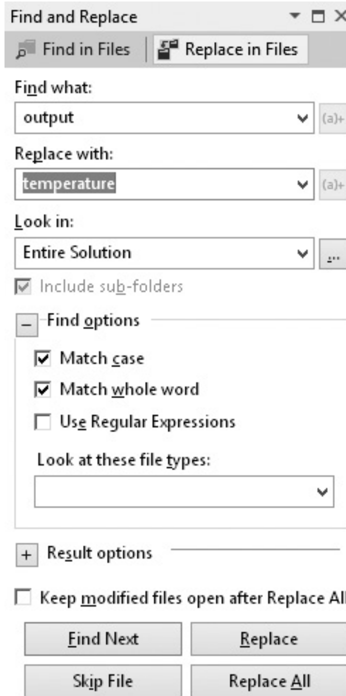


**Figure 1-4** The Find and Replace dialog

variable called *output* to *temperature*. You can apply changes to one reference at a time, to a code block such as a method, to an entire code page, or to the entire solution, depending on which option you select from the Look In drop-down menu. If you are applying changes one at a time, then you must click Replace. Clicking Replace applies the change and advances the cursor to the next reference that matches the search criteria. If you do not wish to change the currently selected text, click Find Next to advance the cursor to the next reference. When searching an entire solution while replacing text, you can also click Skip File to avoid making further changes to the current file. To apply changes to all references in a defined range, click Replace All.

Sometimes you may not know the case or full spelling of a structure that you must locate but you know part of the spelling. You can adjust your search in these situations by toggling the Match Case and Match Whole Word check boxes. For additional search flexibility, you can also select the Regular Expressions check box to search based on a regular expression in the Find What area. Regular expressions define matching patterns, and they are discussed in Chapter 6.

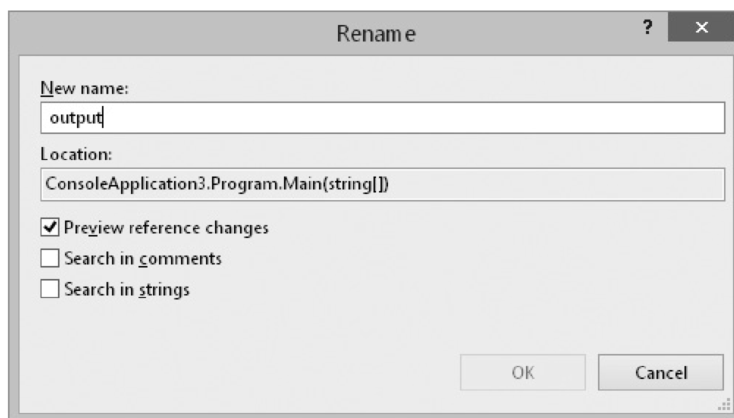


**Figure 1-5** Refactoring with the Rename dialog

### Refactoring with the Rename Dialog

The Rename dialog offers a focused way to rename code structures within their program context. For example, renaming a namespace with the Find and Replace dialog may bluntly change the namespace name, the assembly reference, comments, and other similarly named structures. The Rename dialog, on the other hand, provides more relevant options to change only the code references to it. To launch the Rename dialog, right-click the text to be renamed and then select Refactor | Rename. The shortcut CTRL-R-R also launches the Rename dialog with the selected text in it. You can also open this dialog by choosing Edit | Refactor | Rename. Figure 1-5 shows the Rename dialog with text that is to be renamed in the New Name field.

## Incorporating Debugging Techniques

Debugging is the process of eliminating errors. To be effective at debugging, you must be able to combine several different techniques to track bugs in your code. There are times when you will need to be creative in your bug-tracking efforts, so familiarity with many debugging techniques will prove to be valuable.

Understanding how the IDE can help you debug your code will make you far more productive as a developer. Some errors may be caught prior to run time, but others will be apparent only when the program runs. For C# developers, Visual Studio provides a rich set of tools to help identify and understand errors in either case.

# Errors and Warnings

Errors that are caught prior to compilation prevent your code from compiling. In Visual Studio, errors that are caught prior to compilation are highlighted with red wavy lines.

Warnings, on the other hand, identify questionable or redundant sections of code. Warnings often identify variables that are no longer in use so that you can get rid of them to clean up your code. It is usually best to address all warnings as they appear, but you have the option to skip over them if needed. Warnings are denoted with green wavy lines on your code pages in Visual Studio.

The red wavy line on line 6 in Figure 1-6 highlights an error that indicates that a semicolon is missing for this instruction. Also in Figure 1-6, the green wavy line on line 5 warns that a variable is declared but is never used. The errors and warnings are summarized in the Error List window at the bottom of Figure 1-6. The Error List window appears by default, but if it is not displayed, you can open it by selecting View | Error List or by pressing CTRL-W-E.

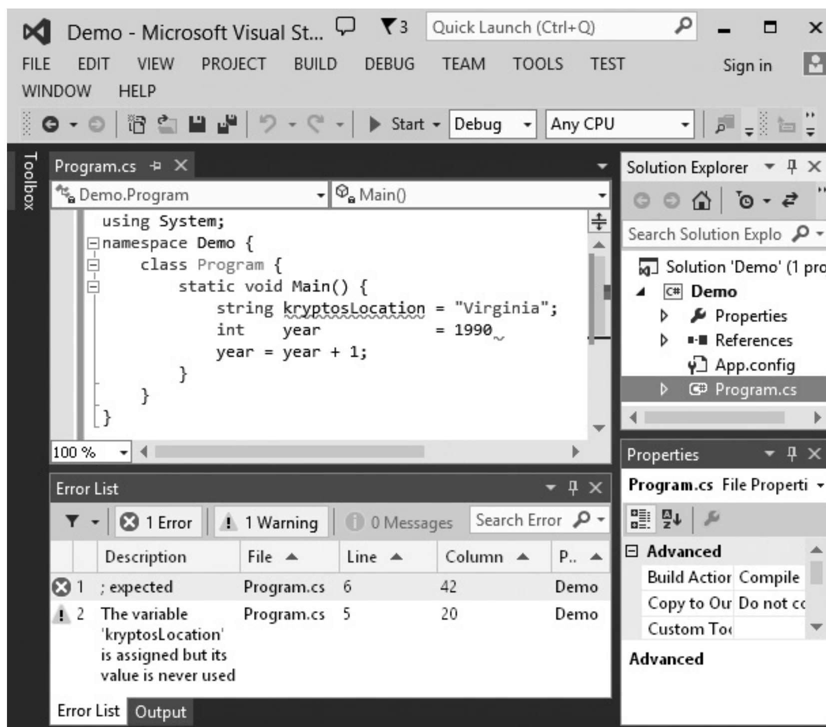


**Figure 1-6** Viewing errors and warnings

***TIP***

The most critical errors are listed at the top of the Error List window, so fix those first when debugging. You can navigate to the line where the error occurs by double-clicking the error description in the Error List.

## Breakpoints

Breakpoints allow you to halt your program at any instruction you choose. When stopping your program at a breakpoint in the code, you can inspect the variable values at that point in the application, or you can step through your code from that point one line at a time. In Figure 1-7, breakpoints are set at lines 9 and 13.

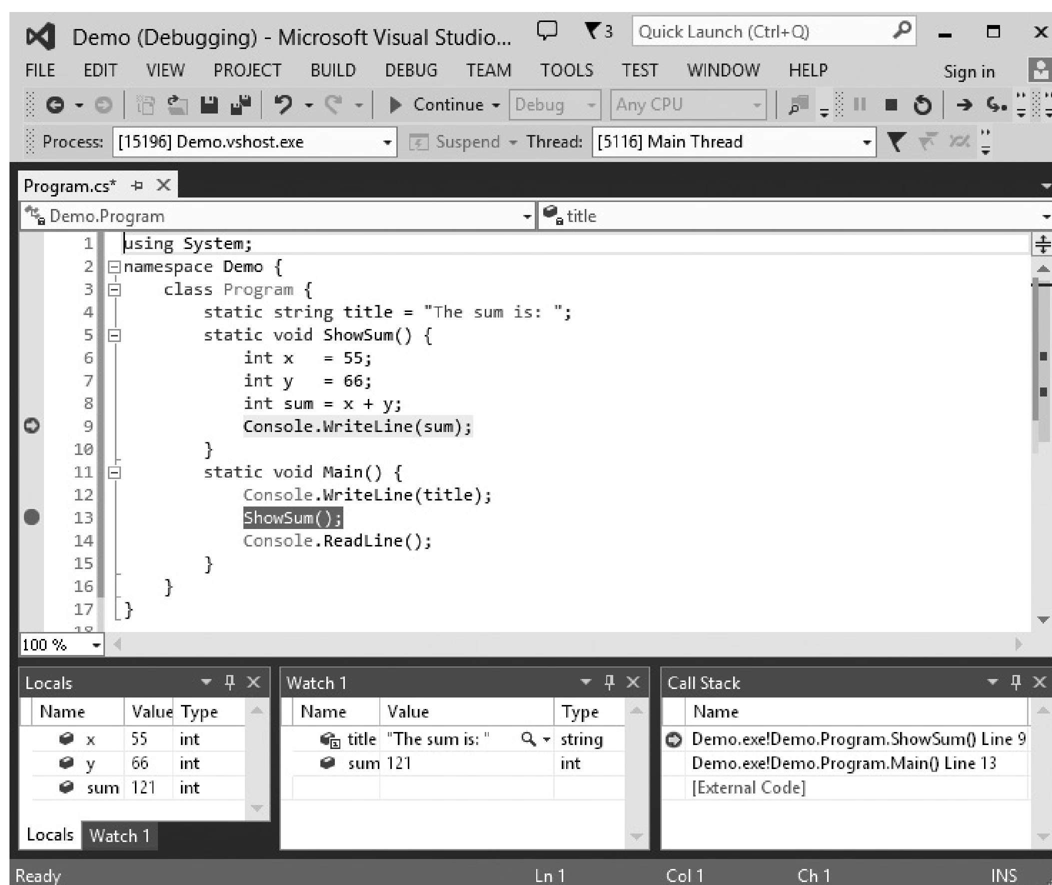


**Figure 1-7** Viewing local variables and the call stack while halted

To set breakpoints, click in the gray margin, also called the *gutter,* beside the instruction where the program is halted. You can set or clear breakpoints by placing the cursor beside the instruction where the program is to halt and then pressing F9. Toggling F9 at the breakpoint location clears and sets the breakpoint. When you run the program, it will stop at the breakpoint until you take further action.

## Resuming Program Execution

When your program is halted, or when you are stepping through your code, you can return to normal program execution by pressing F5. You may also resume program execution by selecting Debug | Continue, or you could instead click the Continue button in the toolbar.

## Tooltips

While a program is halted at a breakpoint, you can hover the cursor over a variable's reference to see a tooltip that displays the variable's value. When your cursor is hovering over a complex variable that stores more than one value, usually you can expand the tooltip to show each value.

## Stepping into Methods

When stepping through your code, you can step into a method from the calling instruction by pressing F11 or by selecting Debug | Step Into. To resume the program, press F5. At the breakpoint on line 13 in Figure 1-7, for example, you would need to press F11 to step into the *ShowSum()* method.

## Stepping over Methods

During debug sessions, you do not want to waste time stepping through methods that you know already work properly. You can skip past proven or irrelevant methods, or methods defined in the framework, by pressing F10 at the calling instruction or by selecting Debug | Step Over. To resume the program, press F5. In Figure 1-7, the program first halts at line 13. To avoid stepping through the *ShowSum()* method, pressing F10 allows you to step over the calling instruction so you can advance to line 14.

***TIP***

To speed up inspection of your variables and routines while debugging, you will want to memorize the keyboard shortcuts for stepping into and stepping over code. Here is a memory trick: First, remember that both routines are triggered with the F10 and F11 keys. To remember that F10 is for Step Over, think of the zero as the letter "O" in Over. To remember that F11 is for Step Into, think of the one as the letter "I" in Into.

## The Call Stack Window

When debugging, you may sometimes want to know where a method was called if it can be called from several locations. Or, you might want to know the sequence of the methods that were accessed prior to the failing instruction. The Call Stack window shows where the calling instruction is located and shows the most recently accessed methods in sequence of access. The Call Stack window appears only while the program is running. To view the Call Stack window while debugging, select Debug | Windows | Call Stack or press ALT-F7. Visual Studio's display of the call stack appears at the bottom right of Figure 1-7.

## The Locals Window

When stepping through your code, the Locals window automatically displays the current values of all variables that are declared locally in the method. By default, the Locals window is positioned at the bottom left of the development environment, as shown in Figure 1-7. If you do not see the Locals window, you can launch it while the program is halted at a breakpoint by choosing Debug | Windows | Locals or pressing ALT-F4.

## Try This 1-2 Inspecting Your Code

This exercise offers practice for

- Creating and writing a console application program
- Setting breakpoints
- Examining code line by line while the program is halted
- Stepping into methods
- Stepping over methods
- Examining variable values with tooltips and the Locals window

Try stepping through your code and examining the variable values at run time:

1. Open Visual Studio and create a console application project, as outlined earlier in the chapter.
2. Replace all code in the Program.cs file with the code that appears in Figure 1-8.
3. Set a breakpoint beside the call to the *MultiplyTwoNumbers()* method and run the program until it halts.

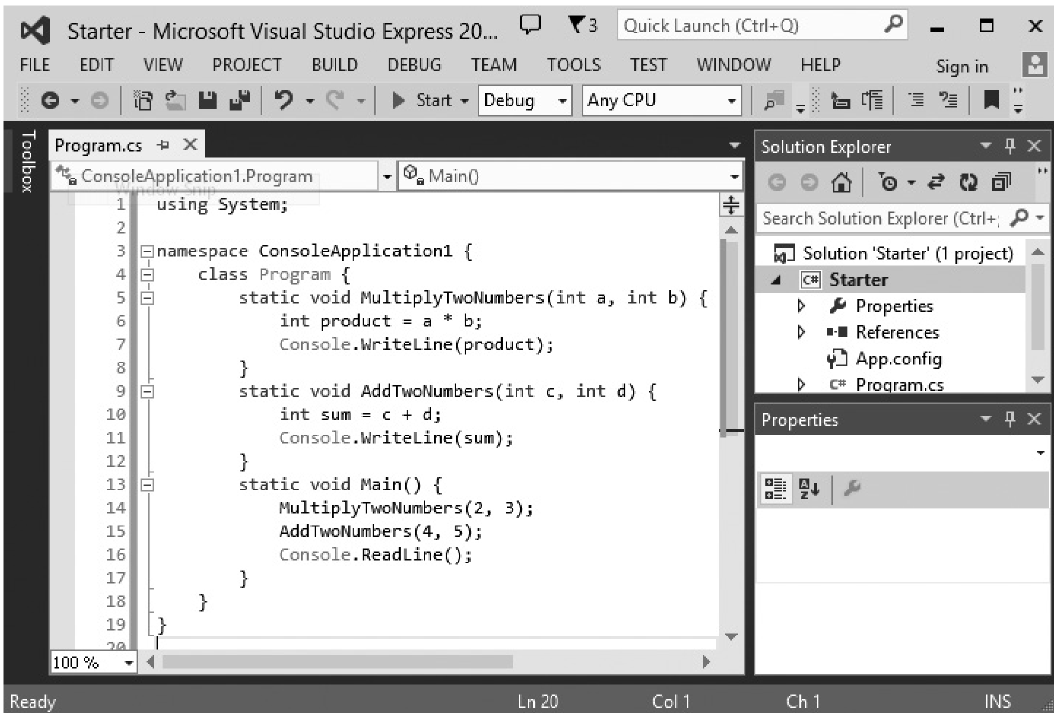


**Figure 1-8** Code inspection project

4. Step into the *MultiplyTwoNumbers()* method with the F11 shortcut.
5. When inside *MultiplyTwoNumbers()*, press F10 repeatedly to advance line by line to the end of this method.
6. While halted inside the *MultiplyTwoNumbers()* method, notice how the variables *a, b,* and *product* appear in the Locals window. Also, hover your cursor over each variable reference and observe how values stored in each variable appear inside a tooltip.
7. Press F10 to advance line by line out of the *MultiplyTwoNumbers()* method, and press F10 again to advance over the call to *AddTwoNumbers()*.
8. To shut down the program, allow it to run until completion by pressing F5 to resume execution. Press ENTER when prompted.

## The Watch Window

The Watch window allows you to customize the list of variables and their corresponding values when you are halted at a breakpoint. The Watch window is different from the Locals window because it shows both local and class-level variables simultaneously. The Watch window can only be viewed at run time. The Watch window does not appear by default. To launch it and to create a list, first halt the program at a breakpoint, then right-click a variable name and select Add Watch from the pop-up menu. The window appears as shown at the bottom middle of Figure 1-7. You can delete an item from the watch list by right-clicking the item and selecting Delete Watch.

***NOTE***

By default, Visual Studio runs with debugging. At times, you might choose to run the project without debugging to avoid having to disable breakpoints. While not relevant for the material in this book, extremely large solutions can take much longer to start when debugging is enabled. To run the project without debugging, select Debug | Start Without Debugging or press CTRL-F5.

## Try-Catch Blocks

To avoid program crashes and to generate helpful responses for critical errors, you may place your suspect or vulnerable code inside a try-catch block. When an error is encountered in the try block, the application skips the remaining instructions in the block and resumes execution in the catch block. Code implemented in the catch block can display or log information about the error and also provide a suitable response for the error. The program then resumes after leaving the catch block.

### The Exception Class

The *Exception* class, from the *System* namespace, creates an object that stores and provides information about errors when they are caught. The catch block can be set up to receive an *Exception* object that contains information about the error. Here is a simplified view of the try-catch syntax:

```
try {                              // 'try' block header.
    // Code that may lead to run-time errors is placed here.
}
catch (Exception e) {              // 'catch' block header receives
                                   // the Exception object named ''e''.
    // Respond to the error.
    Console.WriteLine(e.Message); // Display error information with the
                                   // 'Message' property.
}
```

| Property | Description |
|---|---|
| Data | Gets user-defined information about the error. |
| HelpLink | Gets or sets a link to information associated with a specific exception. |
| InnerException | Gets the System.Exception instance that caused the error. |
| Message | Gets a relatively user-friendly message that describes the error. |
| Source | Gets or sets the name of the application that caused the error. |
| StackTrace | Displays a listing of items on the call stack that led to the error. |
| TargetSite | Gets the name of the method that throws the error. |

**Table 1-1** Properties of the Exception Class

The *Exception* class provides properties that offer additional information about errors that are found in a catch block. These properties are listed and described in Table 1-1. Note that not all of the properties in Table 1-1 apply to every run-time error.

### Example 1-1 Setting Up a Try-Catch Block

This example shows how to set up a try-catch block to prevent a crash from a divide-by-zero error. While the code in the try block is trivial, the solution offers a simple demonstration of how to quickly set up run-time error handling.

When the error in this example is encountered in the try block, the catch block is entered and information about the error is obtained with an *Exception* object named *e*. Error details are printed with the help of the *Exception* object's *StackTrace* and *Message* properties.

```
using System;

namespace Starter {
    class Program {
        public static void Main() {
            // 'try' block contains vulnerable code.
            try {
                int x = 0;
                int y = 1;
                int z = y / x; // Invalid operation.
            }
            // 'catch' block handles error and
            // provides information about the error.
            catch (Exception e) {
```

```
                Console.WriteLine("Error:" + e.StackTrace);
                Console.WriteLine(e.Message);
            }
            Console.ReadLine();
        }
    }
}
```

When you run this program, as soon as the application encounters the fatal error, it enters the catch block. Information about the location of the error in the program is displayed with the *StackTrace* property. Detail about the error is printed with the *Message* property.

```
Error:   at Starter.Program.Main() in C:\ConsoleApplication\Program.cs:line 10
Attempted to divide by zero.
```

**NOTE**

This example can be rewritten without a try-catch block to avoid a program crash. However, try-catch blocks are especially helpful for catching errors and preventing crashes when referencing external resources such as files or database content.

## Logging Data

For really large programs, where hundreds or thousands of lines of code must execute first to create conditions where errors may occur, it can sometimes be very difficult to determine where the error occurred. For example, a variable value may fall outside a valid range. Stepping through your code to find this error might not be feasible since potentially thousands of instructions must execute to find the stray value. In a situation like this during development, you might consider writing your output to a log file to track changes in your data values so that you can run your application at normal speed.

C# allows you to quickly output text to a file with the *StreamWriter* class, which is defined in the *System.IO* namespace. There are several ways to create a *StreamWriter* object. The syntax we will use, shown next, defines the object with two parameters. The first parameter is a string that represents the file location. The second parameter is a Boolean value that indicates whether the object overwrites the existing file or appends output to any data in the existing file:

```
StreamWriter streamWriter = new StreamWriter(string fileLocation, bool append);
```

When the *StreamWriter* object is created, if a file does not exist at the location specified, then one is created.

## Example 1-2 Logging Data

This example program generates a simple log file in the same directory where the project code files are stored. It then writes error details in the file when an error occurs.

```
using System;    // Namespace of Console and Exception classes.
using System.IO; // Namespace of StreamWriter class.

namespace Starter {
    class Program {
        public static void Main() {
            // 'try' block contains vulnerable code.
            try {
                int x = 0;
                int y = 1;
                int z = y / x; // Invalid operation.
            }
            // 'catch' block handles error and
            // provides information about the error.
            catch (Exception e) {
                LogError(e);
                Console.WriteLine("Error logged to file.");
            }
            Console.ReadLine();
        }

        public static void LogError(Exception e) {
            const string FILE_NAME = "log.txt";
            const bool APPEND = true;
            string filePath = "../../" + FILE_NAME;

            // Create writer and append to file.
            StreamWriter sw = new StreamWriter(filePath, APPEND);

            // Show output.
            sw.WriteLine(e.StackTrace);
            sw.WriteLine(e.Message);
            sw.Close();
        }
    }
}
```

When you run the project, you will notice a text file named log.txt is generated inside the directory where the code exists. Every time you run the program, you will see an additional line of text that reads "Here is some log data." when viewing the log file with a text editor such as Notepad.

---

## Chapter 1 Self Test

The following questions are intended to help reinforce your comprehension of the concepts covered in this chapter. The answers can be found in the accompanying online Appendix B, "Answers to the Self Tests."

1. Fill in the blanks:
   - **A.** A ___________ is a logical group of instructions.
   - **B.** ___________ are templates that provide logical groupings of methods and data.
   - **C.** ___________ are logical groupings of classes.
   - **D.** ___________ reference stored data.
2. Indicate whether this statement is true or false:

   ____ It is usually more effective to fix errors at the top of the Error List window before fixing other errors in the list.
3. List shortcuts for the following routines:
   - **A.** ____ Stopping the program.
   - **B.** ____ Setting or clearing a break point.
   - **C.** ____ Stepping into a method.
   - **D.** ____ Stepping over a calling instruction.
   - **E.** ____ Resuming program execution.
4. Write a program that prints your name in the console window.
5. List at least two techniques you can follow to write more readable code.
6. Describe ten debugging techniques discussed in this chapter and briefly describe an advantage of each one.

7. Select the most definitive statement:
   A. Commenting is an essential part of any functional program.
   B. A divide-by-zero error is an example of a syntax error.
   C. Indentation of code within C# structures will not help you write a better program.
   D. Try-catch blocks allow programs to gracefully handle errors when referencing external resources such as files or database content.
8. What variables and corresponding values appear in the Locals window while halted at line 11 of Figure 1-8?