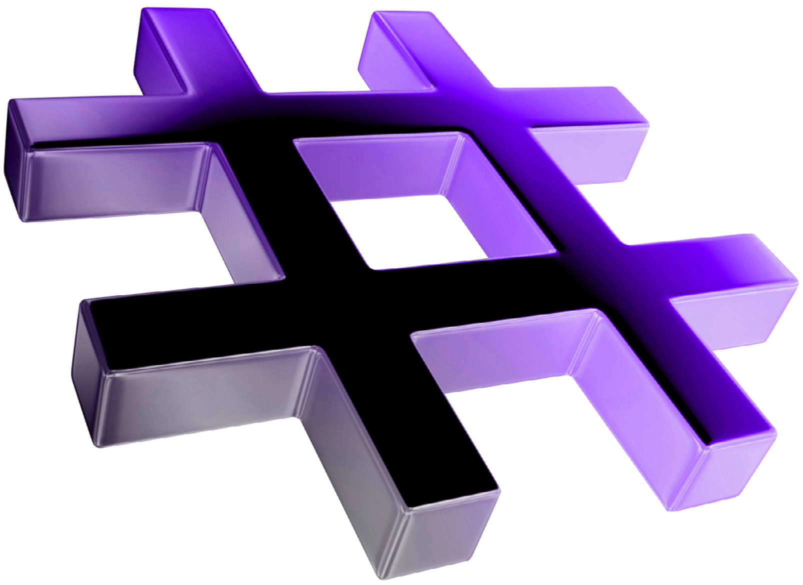
Using the Visual Studio Debugger

G

Objectives

In this appendix you will learn:

- To use breakpoints to pause program execution and allow you to examine the values of variables.
- To set, disable and remove breakpoints.
- To use the **Continue** command to continue execution from a breakpoint.
- To use the **Locals** window to view/modify variable values.
- To use the **Watch** window to evaluate expressions.
- To use the **Step Into**, **Step Out** and **Step Over** commands to execute a program line by line.
- To use **Just My Code**™ debugging.



Outline

G.1 Introduction
G.2 Breakpoints and the **Continue** Command
G.3 *DataTips* and Visualizers
G.4 The **Locals** and **Watch** Windows
G.5 Controlling Execution Using the **Step Into**, **Step Over**, **Step Out** and **Continue** Commands
G.6 Other Debugging Features
G.6.1 **Exception Assistant**
G.6.2 **Just My Code**™ Debugging
G.6.3 Other Debugger Features

G.1 Introduction

In this appendix, you'll learn about tools and techniques that can be used to address compilation errors and logic errors. Syntax errors are a type of **compilation error**—an error that prevents code from compiling. Logic errors, also called **bugs**, do not prevent a program from compiling successfully, but can cause a running program to produce erroneous results or terminate prematurely. Most compiler vendors, like Microsoft, package their IDEs with a tool called a **debugger**. Debuggers allow you to monitor the execution of your programs to locate and remove logic errors. A program must successfully compile before it can be used in the debugger. The debugger allows you to suspend program execution, examine and set variable values and much more. In this appendix, we introduce the Visual Studio debugger features for fixing errors in your programs. For this appendix, we used Visual Studio Community edition, but the features shown here apply to all versions of Visual Studio.

G.2 Breakpoints and the Continue Command

While compilation errors can be found automatically by the compiler, it can be much more difficult to determine the cause of logic errors. To help with this, we investigate the concept of **breakpoints**. Breakpoints are special markers that can be set at any executable line of code. They cannot be placed on comments or whitespace. When a running program reaches a breakpoint, execution pauses, allowing you to examine the values of variables to help determine whether logic errors exist. For example, you can examine the value of a variable that stores a calculation's result to determine whether the calculation was performed correctly. You also can examine the value of an expression.

To illustrate the debugger features, we use the program in Figs. G.1–G.2 that creates and manipulates an `Account` (Fig. G.1) object. This example is based on concepts from Chapter 4, so it does not use features that are presented after Chapter 4. The `Main` method (Fig. G.2) creates an `Account` object with an initial balance of $50.00. `Account`'s constructor (Fig. G.1) accepts one argument, which specifies the `Account`'s initial `balance`. `Main` outputs the initial account balance using `Account` property `Balance`. Then, `Main` prompts the user for and inputs the `withdrawalAmount`, subtracts the withdrawal amount from the `Account`'s `balance` using its `Debit` method and displays the new `balance`. Finally, `Main` performs similar steps to credit the account.

```
// Fig. G.1: Account.cs
// Account class with a Debit method that withdraws money from account.
using System;

public class Account
{
   private decimal balance; // instance variable that stores the balance

   // constructor
   public Account(decimal initialBalance)
   {
      Balance = initialBalance; // set balance using property
   }

   // credits (adds) an amount to the account
   public void Credit(decimal amount)
   {
      Balance = Balance + amount; // add amount to balance
   }

   // debit (subtracts) an amount from the account
   public void Debit(decimal amount)
   {
      if (amount > Balance)
      {
         Console.WriteLine("Debit amount exceeded account balance.");
      }

      if (amount <= Balance)
      {
         Balance = Balance - amount; // subtract amount from balance
      }
   }

   // property to get the balance
   public decimal Balance
   {
      get
      {
         return balance;
      }
      set
      {
         // validate that value is greater than or equal to 0;
         // if it is not, balance is left unchanged
         if (value >= 0)
         {
            balance = value;
         }
      }
   }
}
```

Fig. G.1 | `Account` class with a `Debit` method that withdraws money from account.

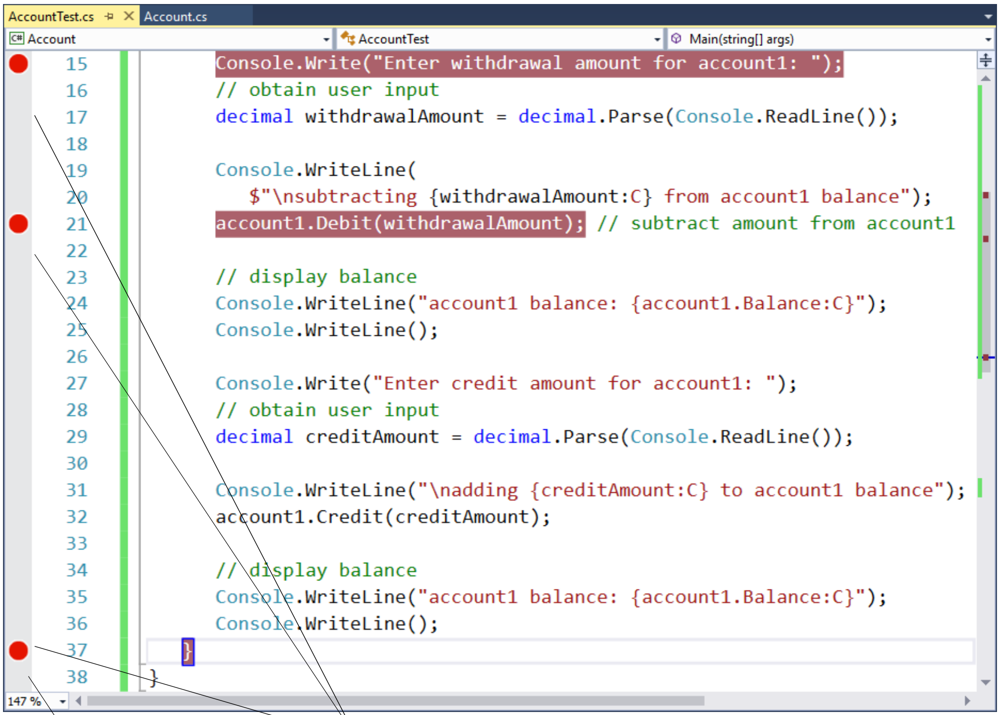
```csharp
// Fig. G.2: AccountTest.cs
// Creating and manipulating an Account object.
using System;

public class AccountTest
{
   // Main method begins execution of C# application
   public static void Main(string[] args)
   {
      Account account1 = new Account(50.00M); // create Account object

      // display initial balance of account object
      Console.WriteLine($"account1 balance: {account1.Balance:C}");

      Console.Write("Enter withdrawal amount for account1: ");
      // obtain user input
      decimal withdrawalAmount = decimal.Parse(Console.ReadLine());

      Console.WriteLine(
         $"\nsubtracting {withdrawalAmount:C} from account1 balance");
      account1.Debit(withdrawalAmount); // subtract amount from account1

      // display balance
      Console.WriteLine("account1 balance: {account1.Balance:C}");
      Console.WriteLine();

      Console.Write("Enter credit amount for account1: ");
      // obtain user input
      decimal creditAmount = decimal.Parse(Console.ReadLine());

      Console.WriteLine("\nadding {creditAmount:C} to account1 balance");
      account1.Credit(creditAmount);

      // display balance
      Console.WriteLine("account1 balance: {account1.Balance:C}");
      Console.WriteLine();
   }
}
```

```
account1 balance: $50.00
Enter withdrawal amount for account1: 25

subtracting $25.00 from account1 balance
account1 balance: $25.00

Enter credit amount for account1: 33

adding $33.00 to account1 balance
account1 balance: $58.00
```

Fig. G.2 | Creating and manipulating an `Account` object.

In the following steps, you'll use breakpoints and debugger commands to examine variable `withdrawalAmount`'s value (declared in Fig. G.2) while the program executes.

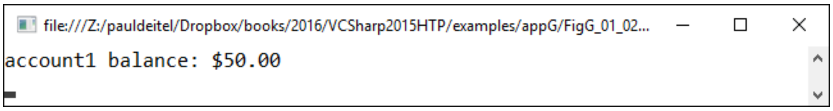
1. ***Inserting breakpoints.*** First, ensure that `AccountTest.cs` is open in the IDE's code editor. To insert a breakpoint, left click inside the **margin indicator bar** (the gray margin at the left of the code window in Fig. G.3) next to the line of code at which you wish to break, or right click that line of code and select **Breakpoint > Insert Breakpoint**. Additionally, you also can press *F9* when your cursor is on the line to toggle the breakpoint. You may set as many breakpoints as you like. Set breakpoints at lines 15, 21 and 37 of `Main`. [*Note:* If you have not already done so, have the code editor display line numbers by opening **Tools > Options...**, navigating to **Text Editor > C#** and selecting the **Line numbers** checkbox.] A solid circle appears in the margin indicator bar where you clicked, and the entire code statement is highlighted, indicating that breakpoints have been set (Fig. G.3). When the program runs, the debugger suspends execution at any line that contains a breakpoint. The program then enters **break mode**. Breakpoints can be set before running a program, both in break mode and during execution. To show a list of all breakpoints in a project, select **Debug > Windows > Breakpoints**.



```
15              Console.Write("Enter withdrawal amount for account1: ");
16              // obtain user input
17              decimal withdrawalAmount = decimal.Parse(Console.ReadLine());
18
19              Console.WriteLine(
20                 $"\nsubtracting {withdrawalAmount:C} from account1 balance");
21              account1.Debit(withdrawalAmount); // subtract amount from account1
22
23              // display balance
24              Console.WriteLine("account1 balance: {account1.Balance:C}");
25              Console.WriteLine();
26
27              Console.Write("Enter credit amount for account1: ");
28              // obtain user input
29              decimal creditAmount = decimal.Parse(Console.ReadLine());
30
31              Console.WriteLine("\nadding {creditAmount:C} to account1 balance");
32              account1.Credit(creditAmount);
33
34              // display balance
35              Console.WriteLine("account1 balance: {account1.Balance:C}");
36              Console.WriteLine();
37          }
38      }
```

Fig. G.3 | *Setting breakpoints.*

2. ***Beginning the debugging process.*** After setting breakpoints in the code editor, select **Build > Build Solution** to compile the program, then select **Debug > Start Debugging** (or press the *F5* key) to begin the debugging process. While debugging a console application, the **Command Prompt** window appears (Fig. G.4), allowing program interaction (input and output).

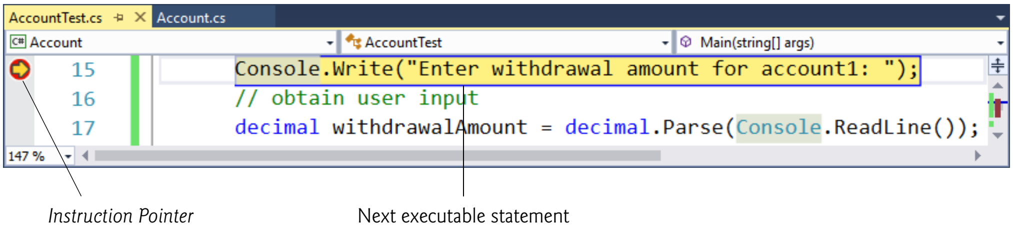


file:///Z:/pauldeitel/Dropbox/books/2016/VCSharp2015HTP/examples/appG/FigG_01_02...

```
account1 balance: $50.00
```

Fig. G.4 | **Account** program running.

3. ***Examining program execution.*** Program execution pauses at the first breakpoint (line 15), and the IDE becomes the active window (Fig. G.5). The yellow arrow to the left of line 18, also called the ***Instruction Pointer***, indicates that this line contains the next statement to execute. The IDE also highlights the line as well.

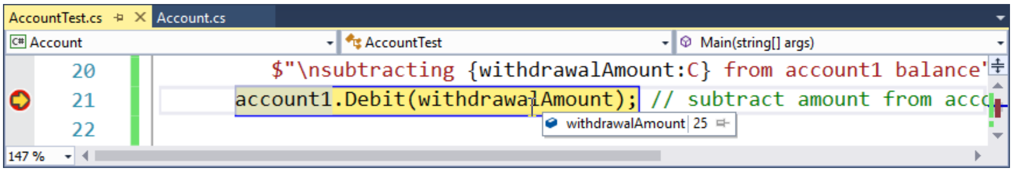


```
15        Console.Write("Enter withdrawal amount for account1: ");
16        // obtain user input
17        decimal withdrawalAmount = decimal.Parse(Console.ReadLine());
```

Fig. G.5 | Program execution suspended at the first breakpoint.

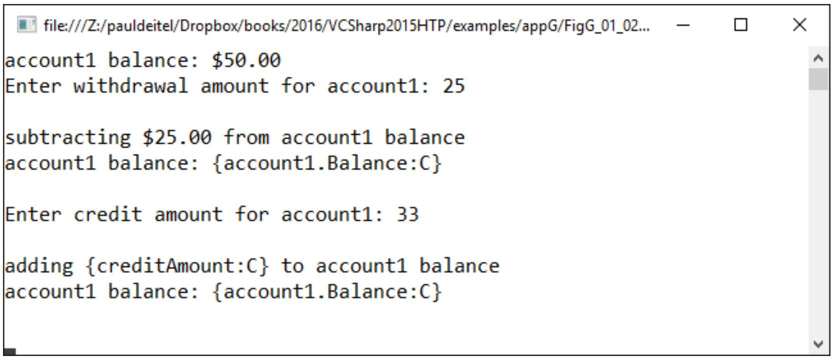
4. ***Using the* Continue *command to resume execution.*** To resume execution, select **Debug > Continue** (or press the *F5* key). The **Continue command** executes the statements from the current point in the program to the next breakpoint or the end of `Main`, whichever comes first. It is also possible to drag the *Instruction Pointer* to another line in the same method to resume execution starting at that position. Here, we use the **Continue** command, and the program continues executing and pauses for input at line 17. Enter `25` in the **Command Prompt** window as the withdrawal amount. When you press *Enter*, the program executes until it stops at the next breakpoint (line 21). Notice that when you place the mouse pointer over the variable name `withdrawalAmount`, its value is displayed in a ***Quick Info* box** (Fig. G.6). As you'll see, this can help you spot logic errors in your programs.

5. ***Continuing program execution.*** Use the **Debug > Continue** command to execute line 21. The program then asks you to input a credit (deposit) amount. Enter `33`, then press *Enter*. The program displays the result of its calculation (Fig. G.7).



```
$"\nsubtracting {withdrawalAmount:C} from account1 balance"
account1.Debit(withdrawalAmount); // subtract amount from acco
```

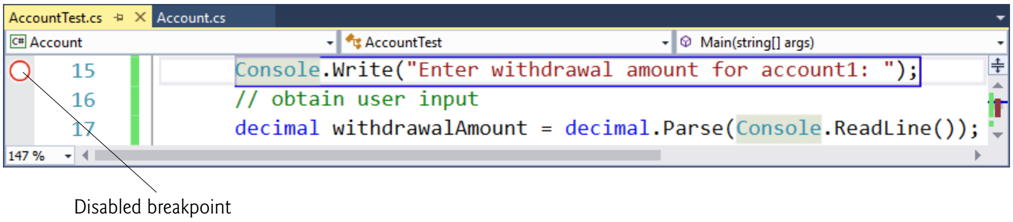
Fig. G.6 | *Quick Info* box displays value of variable `withdrawalAmount`.


file:///Z:/pauldeitel/Dropbox/books/2016/VCSharp2015HTP/examples/appG/FigG_01_02...

```
account1 balance: $50.00
Enter withdrawal amount for account1: 25

subtracting $25.00 from account1 balance
account1 balance: {account1.Balance:C}

Enter credit amount for account1: 33

adding {creditAmount:C} to account1 balance
account1 balance: {account1.Balance:C}
```

Fig. G.7 | Sample execution of `Account.exe` in debug mode.

6. ***Disabling a breakpoint.*** To **disable a breakpoint**, right click the breakpoint and select **Disable Breakpoint**. The disabled breakpoint is indicated by a hollow circle (Fig. G.8)—the breakpoint can be reenabled by right clicking the hollow circle and selecting **Enable Breakpoint**.



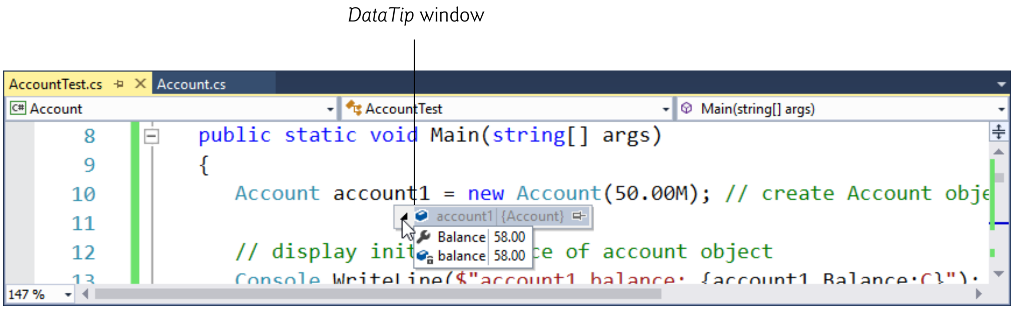
```
Console.Write("Enter withdrawal amount for account1: ");
// obtain user input
decimal withdrawalAmount = decimal.Parse(Console.ReadLine());
```

Fig. G.8 | Disabled breakpoint.

7. ***Removing a breakpoint.*** To remove a breakpoint that you no longer need, right click the line of code on which the breakpoint has been set and select **Breakpoint > Delete Breakpoint**. You also can remove a breakpoint by clicking the circle in the margin indicator bar or pressing *F9* when the cursor is on the line.
8. ***Finishing program execution.*** Select **Debug > Continue** to execute the program to completion. Then delete all the breakpoints.

G.3 *DataTips* and Visualizers

You already know how to use the *Quick Info* window to view a variable's value. However, often you may want to check the status of an `object`. For example, you may want to check the `Text` value of a `TextBox` control. When you hover the mouse over a reference-type variable while debugging, the ***DataTip*** window appears (Fig. G.9). When you click the arrow to expand the object in the *DataTip*, the *DataTip* window gives information about the object's data. There are some limitations—references must be instance variables or local variables, and expressions involving method calls cannot be evaluated.



```
public static void Main(string[] args)
{
    Account account1 = new Account(50.00M); // create Account obje

    // display init                ce of account object
    Console.WriteLine($"account1 balance: {account1.Balance:C}");
```

Fig. G.9 | A *DataTip* displayed for the `account1` variable.

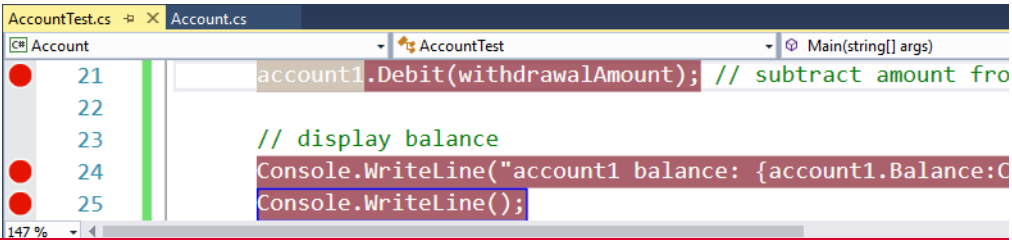
For the `Account` object, this means that you can see the `balance` inside it (as well as the `Balance` property used to access it). Just like the *Quick Info* window, you also can change the value of a property or variable inside it by clicking on one of the values listed, then typing the new value.

DataTips do not intuitively display information for all variables. For example, a variable representing an XML document cannot be viewed in its natural form using most debugging tools. For such types, **visualizers** can be useful. Visualizers are specialized windows to view certain types of data. They are shown through *DataTip* windows by clicking the small magnifying glass next to a variable name. There are three predefined visualizers—advanced programmers may create additional ones. The **Text Visualizer** lets you see `string` values with all their formatting included. The **XML Visualizer** formats XML objects into a color-coded format. Finally, the **HTML Visualizer** parses HTML code (in `string` or XML form) into a web page, which is displayed in the small window.

G.4 The Locals and Watch Windows

In the preceding section, you learned how to use the *Quick Info* and *DataTip* features to examine the variable's value. In this section, you'll learn how to use the **Locals window** to view all variables that are in use while your program is running. You'll also use the **Watch window** to examine the values of expressions.

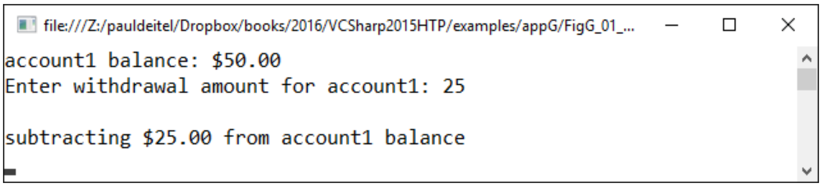
1. *Inserting breakpoints.* Set a breakpoint at line 21 (Fig. G.10) in the source code by left clicking in the margin indicator bar to the left of line 21. Use the same technique to set breakpoints at lines 24 and 25 as well.



```
account1.Debit(withdrawalAmount); // subtract amount fro

// display balance
Console.WriteLine("account1 balance: {account1.Balance:C
Console.WriteLine();
```

Fig. G.10 | Setting breakpoints at lines 24, 27 and 28.

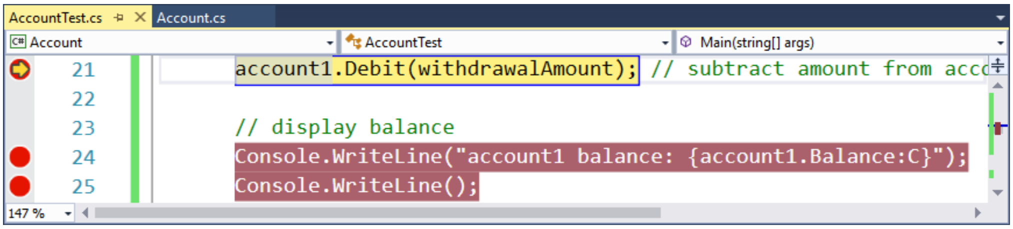
2. ***Starting debugging.*** Select **Debug > Start Debugging**. Type `25` at the **Enter withdrawal amount for account1:** prompt (Fig. G.11) and press *Enter*. The program executes until the breakpoint at line 24.


file:///Z:/pauldeitel/Dropbox/books/2016/VCSharp2015HTP/examples/appG/FigG_01_...

```
account1 balance: $50.00
Enter withdrawal amount for account1: 25

subtracting $25.00 from account1 balance
```

Fig. G.11 | Entering the withdrawal amount before the breakpoint is reached.

3. ***Suspending program execution.*** When the program reaches line 21, the IDE suspends program execution and switches the program into break mode (Fig. G.12). At this point, the statement in line 17 (Fig. G.2) has input the `withdrawalAmount` that you entered (`25`), the statement in lines 19–20 has output that the program is subtracting that amount from the `account1` balance and the statement in line 21 is the next statement that executes.

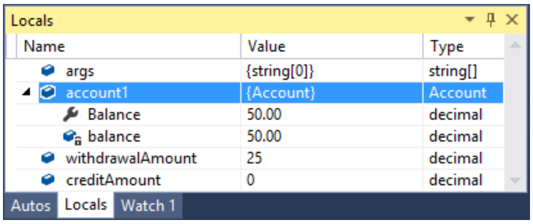


```
21  account1.Debit(withdrawalAmount); // subtract amount from acco
22
23  // display balance
24  Console.WriteLine("account1 balance: {account1.Balance:C}");
25  Console.WriteLine();
```

Fig. G.12 | Program execution pauses when debugger reaches the breakpoint at line 24.

4. ***Examining data.*** Once the program enters break mode, you can explore the local variable values using the **Locals** window at the bottom of the IDE. If this window is not displayed, select **Debug > Windows > Locals**. Click the plus to the left of `account1` in the **Name** column (Fig. G.13) to view each of `account1`'s instance variable values individually, including the value for `balance` (`50`). The **Locals**

window displays a class' properties as data, which is why you see both the `Balance` property and the `balance` instance variable in the window. In addition, the current value of local variable `withdrawalAmount` (`25`) is displayed.

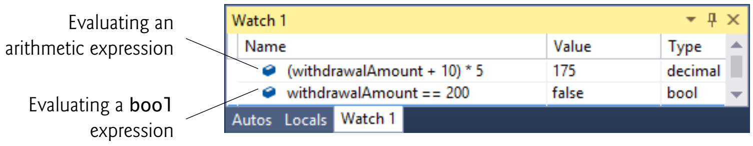


Locals

Name	Value	Type
args	{string[0]}	string[]
account1	{Account}	Account
Balance	50.00	decimal
balance	50.00	decimal
withdrawalAmount	25	decimal
creditAmount	0	decimal

Fig. G.13 | Examining local variables.

5. ***Evaluating arithmetic and boolean expressions.*** You can evaluate arithmetic and `bool` expressions using the **Watch** window. If it is not displayed, select **Debug > Windows > Watch > Watch 1** to display the window (Fig. G.14). In the **Name** column's first row, type `(withdrawalAmount + 10) * 5`, then press *Enter*. The value `175` is displayed. In the **Name** column's next row in the **Watch** window, type `withdrawalAmount == 200`, then press *Enter*. This expression determines whether the value contained in `withdrawalAmount` is `200`. Expressions containing the `==` symbol are boolean expressions. The value returned is `false`, because `withdrawalAmount` does not currently contain the value `200`.



Watch 1

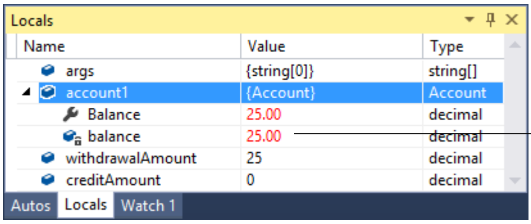
Name	Value	Type
(withdrawalAmount + 10) * 5	175	decimal
withdrawalAmount == 200	false	bool

Evaluating an arithmetic expression
Evaluating a bool expression
Autos
Locals
Watch 1

Fig. G.14 | Examining the values of expressions.

6. ***Resuming execution.*** Select **Debug > Continue** to resume execution. Line 21 executes, subtracting the account with the withdrawal amount, and the program enters break mode again at line 24. Select **Debug > Windows > Locals**. The updated `balance` instance variable and `Balance` property value are now displayed (Fig. G.15). The values in red in the window are those that have just been modified.

7. ***Modifying values.*** Based on the value input by the user (`25`), the account balance output by the program should be `25`. However, you can use the **Locals** window to change variable values during program execution. This can be valuable for experimenting with different values and for locating logic errors in programs. In the **Locals** window, click the **Value** field in the `balance` row to select the value `25`. Type `37`, then press *Enter*. The debugger changes the value of `balance` (and the `Balance` property as well), then displays its new value in red. Now select **Debug > Continue** to execute lines 27–28. Notice that the new value of `balance` is displayed in the **Command Prompt** window.



Name	Value	Type
args	{string[0]}	string[]
account1	{Account}	Account
Balance	25.00	decimal
balance	25.00	decimal
withdrawalAmount	25	decimal
creditAmount	0	decimal

Locals
Autos
Locals
Watch 1
Updated value of the balance variable appears in red on the screen

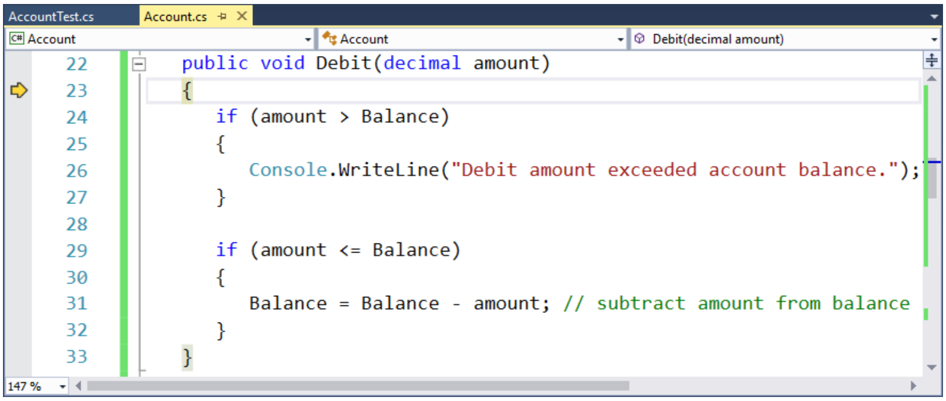
Fig. G.15 | Displaying the value of local variables.

8. ***Stopping the debugging session.*** Select **Debug > Stop Debugging**. Delete all breakpoints, which can be done by pressing *Ctrl* + *Shift* + *F9*.

G.5 Controlling Execution Using the Step Into, Step Over, Step Out and Continue Commands

Sometimes you need to execute a program line by line to find and fix logic errors. Stepping through a portion of your program this way can help you verify that a method's code executes correctly. The commands you learn in this section allow you to execute a method line by line, execute all of a method's statements or execute only its remaining statements (if you have already executed some statements in the method).

1. ***Setting a breakpoint.*** Set a breakpoint at line 21 by left clicking in the margin indicator bar.
2. ***Starting the debugger.*** Select **Debug > Start Debugging**. Enter the value `25` at the **Enter withdrawal amount for account1:** prompt. Program execution halts when the program reaches the breakpoint at line 21.
3. ***Using the* Step Into *command.*** The **Step Into command** executes the next statement in the program and immediately halts. If the statement to execute is a method call, control transfers to the called method. The **Step Into** command allows you to follow execution into a method and confirm its execution by individually executing each statement inside the method. Select **Debug > Step Into** (or press *F11*) to enter class `Account`'s `Debit` method (Fig. G.16).
4. ***Using the* Step Over *command.*** Select **Debug > Step Over** (or press *F10*) to enter the `Debit` method's body and transfer control to line 24. The **Step Over command** behaves like the **Step Into** command when the next statement to execute does not contain a method call or access a property. You'll see how the **Step Over** command differs from the **Step Into** command in *Step 10*.
5. ***Using the* Step Out *command.*** Select **Debug > Step Out** or press *Shift-F11* to execute the remaining statements in the method and return control to the calling method. Often, in lengthy methods, you may want to look at a few key lines of code, then continue debugging the caller's code. The **Step Out command** executes the remainder of a method and returns to the caller.
6. ***Setting a breakpoint.*** Set a breakpoint at line 24 of Fig. G.2. This breakpoint is used in the next step.



```
public void Debit(decimal amount)
{
    if (amount > Balance)
    {
        Console.WriteLine("Debit amount exceeded account balance.");
    }

    if (amount <= Balance)
    {
        Balance = Balance - amount; // subtract amount from balance
    }
}
```

Fig. G.16 | Stepping into the `Debit` method.

7. *Using the* **Continue** *command.* Select **Debug > Continue** to execute until the next breakpoint is reached at line 24. This feature saves time when you do not want to step line by line through many lines of code to reach the next breakpoint.
8. *Stopping the debugger.* Select **Debug > Stop Debugging** to stop debugging.
9. *Starting the debugger.* Before we can demonstrate the next debugger feature, you must restart the debugger. Start it, as you did in *Step 2*, and enter the same value (`25`). The debugger pauses execution at line 21.
10. *Using the* **Step Over** *command.* Select **Debug > Step Over**. Recall that this command behaves like the **Step Into** command when the next statement to execute does not contain a method call. If the next statement to execute contains a method call, the called method executes in its entirety (without pausing execution at any statement inside the method—unless there is a breakpoint in the method), and the arrow advances to the next executable line (after the method call) in the current method. In this case, the debugger executes line 21 in `Main` (Fig. G.2), which calls the `Debit` method. Then the debugger pauses execution at line 24, the next executable statement.
11. *Stopping the debugger.* Select **Debug > Stop Debugging.** Remove all remaining breakpoints.

G.6 Other Debugging Features

Visual Studio provides many other debugging features that simplify the testing and debugging process. We discuss some of these features in this section.

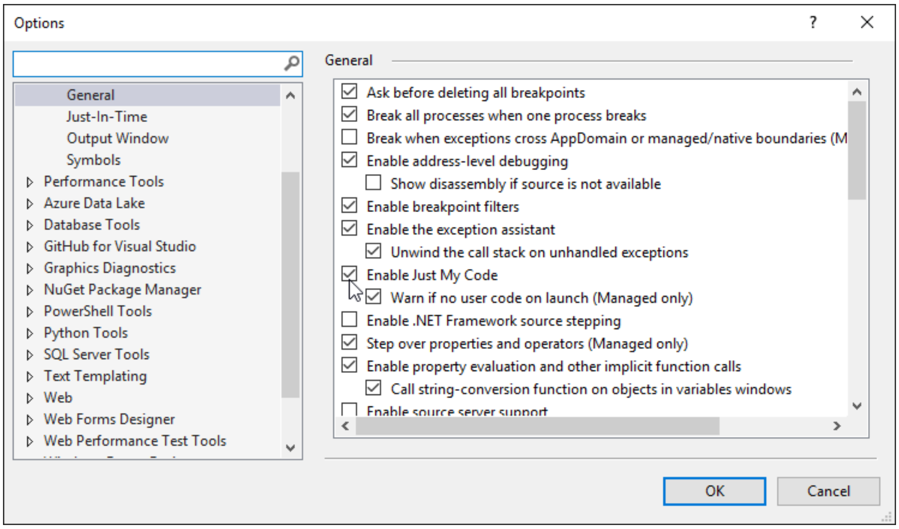
G.6.1 Exception Assistant

You can run a program by selecting either **Debug > Start Debugging** or **Debug > Start Without Debugging.** If you select the option **Debug > Start Debugging** (in full versions of Visual Studio) and the runtime environment detects uncaught exceptions, the application pauses, and a window called the **Exception Assistant** appears, indicating where the exception occurred, the exception type and links to helpful information on handling the exception.

G.6.2 Just My Code™ Debugging

Throughout this book, we produce increasingly substantial programs that often include a combination of code written by the programmer and code generated by Visual Studio. The IDE-generated code can be difficult to understand—fortunately, you rarely need to look at this code. Visual Studio provides a debugging feature called **Just My Code™** that allows programmers to test and debug only the portion of the code they have written. When this option is enabled, the debugger always steps over method calls to methods of classes that you did not write.

This option is enabled by default. If you need to change this setting, select **Tools > Options**. In the **Options** dialog, select the **Debugging** category to view the available debugging tools and options. Click the checkbox that appears next to the **Enable Just My Code**option (Fig. G.17) to enable or disable this feature.


Options

- General
- Just-In-Time
- Output Window
- Symbols
- Performance Tools
- Azure Data Lake
- Database Tools
- GitHub for Visual Studio
- Graphics Diagnostics
- NuGet Package Manager
- PowerShell Tools
- Python Tools
- SQL Server Tools
- Text Templating
- Web
- Web Forms Designer
- Web Performance Test Tools

General

- ☑ Ask before deleting all breakpoints
- ☑ Break all processes when one process breaks
- ☐ Break when exceptions cross AppDomain or managed/native boundaries (M
- ☑ Enable address-level debugging
  - ☐ Show disassembly if source is not available
- ☑ Enable breakpoint filters
- ☑ Enable the exception assistant
  - ☑ Unwind the call stack on unhandled exceptions
- ☑ Enable Just My Code
  - ☑ Warn if no user code on launch (Managed only)
- ☐ Enable .NET Framework source stepping
- ☑ Step over properties and operators (Managed only)
- ☑ Enable property evaluation and other implicit function calls
  - ☑ Call string-conversion function on objects in variables windows
- ☐ Enable source server support

Fig. G.17 | Enabling the **Just My Code** debugging feature.

G.6.3 Other Debugger Features

You can learn more about the Visual Studio debugger at

```
https://msdn.microsoft.com/library/sc65sadd
```

G.7 Wrap-Up

In this appendix, you learned how to use the debugger and set breakpoints so that you can examine your code and results while a program executes. This capability enables you to locate and fix logic errors in your programs. You also learned how to continue execution after a program suspends execution at a breakpoint and how to disable and remove breakpoints.

We used *DataTips* to find additional information on nonprimitive variables, and **Visualizers** to view certain data types in a detailed view. We showed how to use the debugger's **Watch** and **Locals** windows to evaluate arithmetic and boolean expressions. We also demonstrated how to modify a variable's value during program execution so that you can see how changes in values affect your results.

You learned how to use the debugger's **Step Into** command to debug methods called during your program's execution. You saw how the **Step Over** command can be used to execute a method call without stopping the called method. You used the **Step Out** command to continue execution until the current method's end. You also learned that the **Continue** command continues execution until another breakpoint is found or the program terminates.

Finally, we discussed the Visual Studio debugger's additional features, including **Edit and Continue**, the **Exception Assistant** and **Just My Code**™ debugging.