

# ITD121 Programming

## Debugging

### Learning Objectives

Become familiar with the Microsoft Visual Studio .NET debugging tools

#### Visual Studio Debugger

Now that you know more about methods in C#, the following material shows you how to use the Debugger to step into (and back out of) such methods. It also shows you a few errors that programmers often make, so that you can avoid these in your own work.

This material assumes that you have completed the Week 4 Workshop on using the Debugger. If not, do that first!

Each of the following programs includes a **Pause** method, so that the program always ends with the prompt "Press any key to continue ..." when run with/without the Debugger. It is not necessary for you to understand the lines of code inside that **Pause** method. (You will not be expected to reproduce such lines in any item of assessment.)

### A. Stepping Over Methods

1. Download the **Debugging Workshop Files\_Part 2.zip** file from Blackboard, and unzip it.
2. From the **Ch05\_LargestValue** subfolder, open the **LargestValue** solution in Visual Studio.
3. Double-click on **LargestValue.cs** to open it.

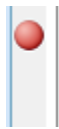
This is essentially the same program as Example 5-11 in the Doyle textbook with some modifications. You can refer to the textbook if you'd like more explanation about the details of the program. But you can still continue here, without reading the textbook.

4. When the program runs, it asks you to enter two values. The program determines which is the larger, then displays it and its square root. Try it out by running the program. Enter any numbers you like.
5. As you can see, the program's **Main** method calls three methods: **InputValues**, **DetermineLargest** and **PrintResults**.

```
static void Main() {  
    int value1,  
        value2,  
        largestOne;  
  
    InputValues(out value1, out value2);  
    largestOne = DetermineLargest(value1, value2);  
    PrintResults(largestOne);  
}
```

Each of these three methods is defined further down in the program. You may have to scroll down to see them all.

6. Use **F9** (or **Debug → Toggle Breakpoint**) to set a breakpoint on the line containing the call to `InputValues`.



```
InputValues(out value1, out value2);  
largestOne = DetermineLargest(value1, value2);  
PrintResults(largestOne);
```

7. Run the program, but make sure that you are Debugging by pressing **F5** (or **Debug → Start Debugging**).
8. When the program pauses at the breakpoint, press **F10** (or **Debug → Step Over**) to execute this method.

When prompted, enter any two numbers. Once you do that, the program will pause because it has finished executing your **Step Over** command, i.e. finished executing the `InputValues` method.

Though paused, the console window still has the keyboard focus, so you must click back in Visual Studio so that the following keystrokes go to it. Click anywhere within the `LargestValue.cs` window.

The yellow line should now be as shown here:



```
InputValues(out value1, out value2);  
largestOne = DetermineLargest(value1, value2);  
PrintResults(largestOne);
```


9. Press **F10** to step over – i.e. execute – the call to the `DetermineLargest` method.
10. Press **F10** to step over – i.e. execute – the call to the `PrintResults` method.
11. Press **F10** to exit from the `Main` method, terminating the program. Don't close anything.

## B. Stepping Into a Method

As you have seen, **F10** (Step Over) executes each method as a single step. But what if you want to see what is going on inside a method. That's where **F11** (Step Into) is very useful.

For this example, assume that you don't want to see what is going on inside the `InputValues` method, but you do want to see the details of how the `DetermineLargest` method is executed.

12. With the same breakpoint as before, run the program by pressing **F5** (or **Debug → Start Debugging**).
13. Step over the `InputValues` method by repeating Step 8 above, but make sure that the second number is bigger than the first, this time. For example, enter 12 and 34. (You'll see why soon.)
14. The yellow line will be in the same place as shown before. Now instead of pressing **F10**, press **F11** (or **Debug → Step Into**) to step into the `DetermineLargest` method, as shown here:




```
public static int DetermineLargest(int value1, int value2) {
    int largestOne; //local variable declared to
                    //facilitate single exit from
                    //method

    if (value1 > value2) {
        largestOne = value1;
    } else {
        largestOne = value2;
    }
    return largestOne;
}
```


If you like, you can hover the mouse over the `value1` and `value2` parameters, to see their values as this method is executed.

15. Press **F10 (Step Over)** to step to the `if` statement.



```
if (value1 > value2) {
```


16. Press **F10** twice, to see that the `else` part is executed (assuming that you entered a second number that was bigger than the first).



```
    } else {
        largestOne = value2;
    }
```

You should be able to explain why this has happened. If you're not sure, ask for help.


17. Press **F10** twice, to reach the `return` statement.



```
    return largestOne;
```

Hover the mouse over the `largestOne` variable in this line of code. You will see the value that this method is going to give back to the `Main` method (since the `Main` method called the `DetermineLargest` method that we're currently in).

18. Press **F10** twice, to return from the `DetermineLargest` method, so that the yellow line looks like below:



```
InputValues(out value1, out value2);
largestOne = DetermineLargest(value1, value2);
PrintResults(largestOne);
```

At this point, the `DetermineLargest` method has calculated the larger value, but that value has not yet been stored in the `largestOne` variable that is part of the `Main` method.

Hover the mouse over the `largestOne` variable in this line of code. It will appear to have the value of 0, although it doesn't really have any value assigned to it yet.

19. Press **F10** to step down one line. Hover the mouse over the `largestOne` variable again. It will now have the value calculated by the `DetermineLargest` method.
20. Use **F5 (or Debug → Continue)** to let the program run to its end.

### C. Stepping Out of a Method

When you're inside a method, there are times when you'd really like to get out of that method as fast as possible, and then pause.

As you've seen above, you can use F10 to step through and return from a method. That's fine when you're first learning to program and want to see exactly what is going on. But doing this can be tedious in certain situations, for example when you stepped into a method that you really meant to step over, or you accidentally pressed F11 when you meant to press F10, or when the method has a loop, and you don't want to go around and around it.

Now you could tackle this by setting a breakpoint at the end of the method, use **F5** (Continue) to reach that breakpoint, and then remove the breakpoint. But the Debugger gives you a faster way, **Shift+F11 (Step Out)**.

21. With the same breakpoint as before, run the program by pressing **F5** (or **Debug → Start Debugging**).
22. When the program pauses at the breakpoint, press **F11 (Step Into)** to step into the `InputValues` method, as shown here.



```
public static void InputValues(out int v1, out int v2) {
```

Press **F10** once or twice to check that you really are inside this method.

23. Now to get out of this method in the fastest way, press **Shift+F11** (or **Debug → Step Out**). The remaining lines of code in the method will still be executed, so you'll still be prompted to enter two numbers. But you won't have to press **F10** for every line in this method.

As before, the program will pause when it has finished executing the `InputValues` method.



```
InputValues(out value1, out value2);
largestOne = DetermineLargest(value1, value2);
PrintResults(largestOne);
```

Now click back in Visual Studio so that the following keystrokes go to it.

24. Press **F10** to step to the next line. Then press **F5** (or **Debug → Continue**) to let the program run to its end.
25. Use **F9** (or **Debug → Toggle Breakpoint**) to remove the current breakpoint, i.e. the one on the line that calls `InputValues`.

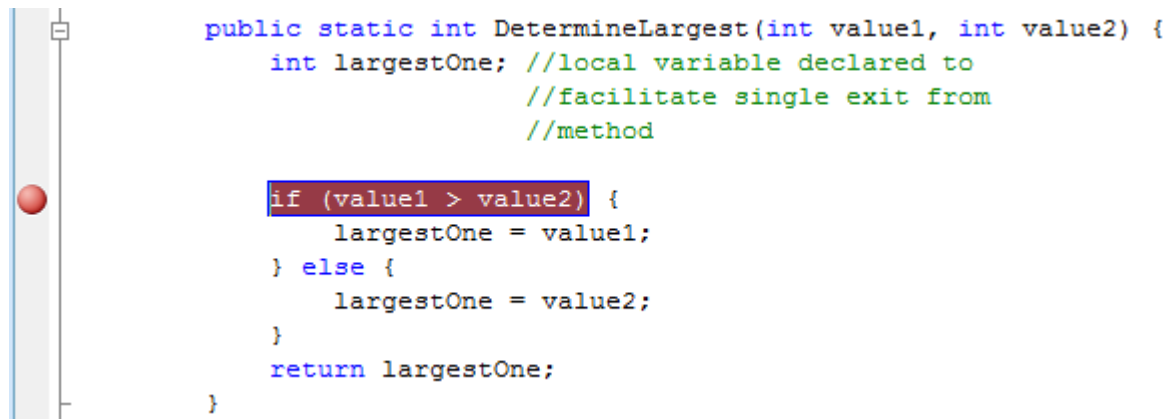
#### An extra note:

With C# statements that are not method-calls, it usually makes no difference whether you use **F10** or **F11** to execute each statement. But because it can be annoying when you accidentally step into a method – the screen changes dramatically which can cause you to lose track of where you are – most people normally prefer to use **F10**, and only use **F11** when they deliberately want to go inside a method.

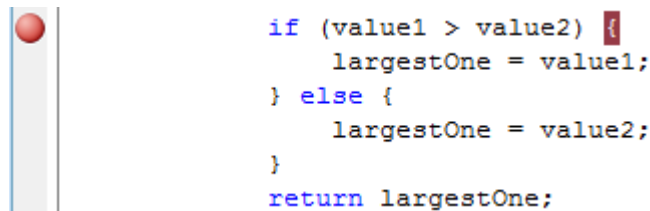
## **D. Setting a Breakpoint *Inside* a Method**

There is another way to see what is going on inside a method - set a breakpoint on a line of code that is inside the method. When you already understand most of what a program is doing, and just want to focus on a particular method, this can be easier than stepping from the start of the program.

26. Use **F9** (or **Debug → Toggle Breakpoint**) to set a breakpoint on the line containing the `if` condition, in the `DetermineLargest` method.



**Warning:** click somewhere inside the main part of the **if** condition, before pressing **F9**. Don't click at or near the right-hand end, or your breakpoint will look like the one below. The breakpoint above will always pause before the **if** condition is tested, whereas the breakpoint below will only pause when the **if** condition happens to be **true**. This means that the one above will always pause, but the one below will only pause sometimes.



27. Run the program by pressing **F5** (or **Debug** → **Start Debugging**). Enter a pair of numbers again.
28. When the program pauses at the breakpoint, you can then use **F10** to show that the computer executes the **if** part or the **else** part, depending on which value is bigger.
29. You can then use **F5** (or **Debug** → **Continue**) to let the program run to its end.  
If you'd like to see how it works a few more times, just go back to step 27.
30. You can now close this solution.

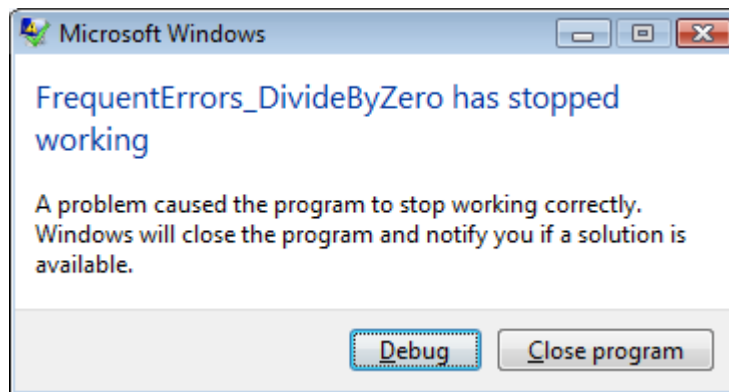
## E. Trapping Runtime Errors and Exceptions

When a program runs, it may encounter a situation that prevents it from continuing to execute. The Debugger can help you to see exactly where the problem occurs. The program used here is a deliberately simplified version of what can happen in other programs.

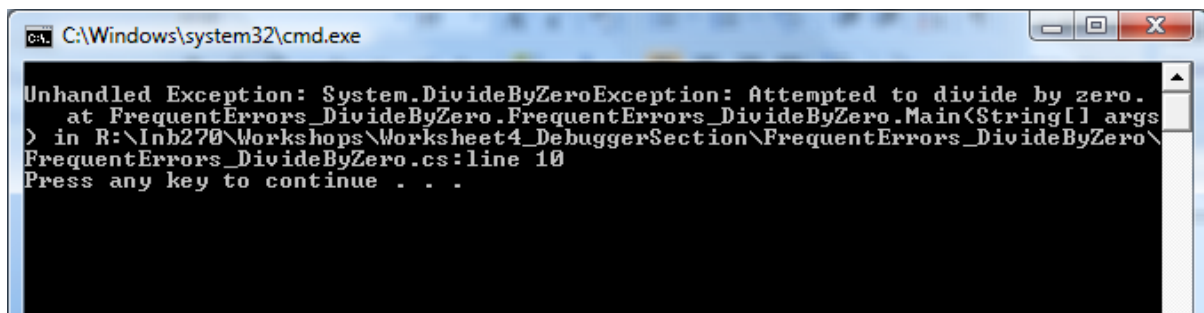
1. From the **FrequentErrors\_DivideByZero** subfolder, open the **FrequentErrors\_DivideByZero** solution in Visual Studio.
2. In the Solution Explorer window, double-click on **FrequentErrors\_DivideByZero.cs** to open it.
3. To see what happens when not using the Debugger, press **Ctrl+F5** (or **Debug** → **Start Without Debugging**).

Don't press **F5** by itself. If you do so by mistake, close the dialog box that appears and then press **Shift+F5** (**Stop Debugging**), and then try **Ctrl+F5**.

You will get a message box something like this. (The exact message box format depends on which version of Windows you're using.)

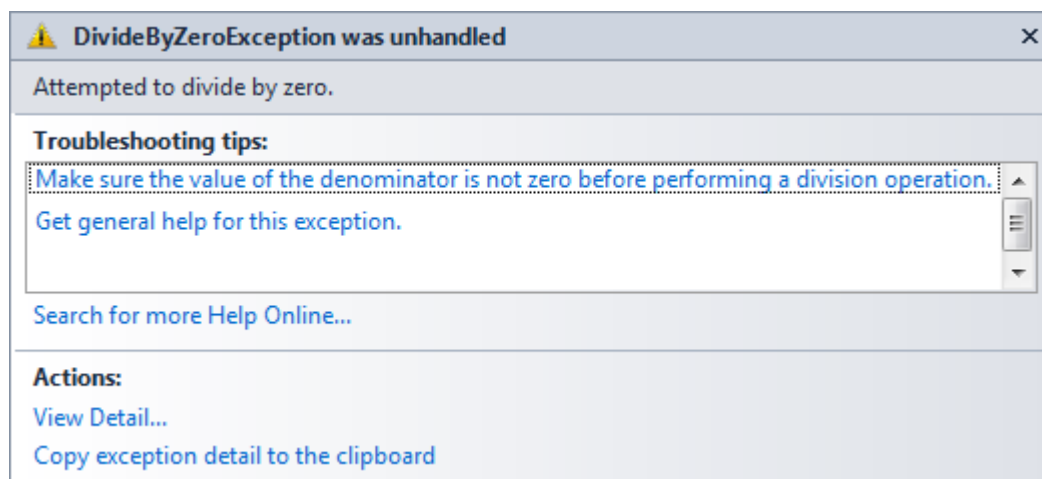



Click the *Close program* button to close that message box, not the *Debug* button. Your console window will then display some information about the error.



While this console window gives us some information about the error, there is a better way to see what is wrong. So press any key to close that window.

4. Now press **F5 (Start Debugging)**. This time the Debugger catches the error, showing the line at which the error occurred, as well as one of the exception helper dialog boxes shown below. (Again, the exact dialog box format depends on which version of Windows you're using.)



In general, it is possible to click on one or more of the *Troubleshooting tips* shown to try to get more information about the problem. But in this simple example, it is clear that the program is making the error of trying to divide by zero. So click on the close button  to close this dialog box.

5. The Debugger is active, so you can hover the mouse over the `x` variable to confirm that it is zero.

6. Although the Debugger is active, you can't really make the program run any further. Try using **F10 (Step Over)** or **F5 (Continue)**, to see that the program will again try to execute the same statement, and the same divide-by-zero error will happen again.  
So, the usual thing is to stop the Debugger at this point. Press **Shift+F5 (Stop Debugging)** to stop.
7. You can now close this solution.

## F. Frequent Kind of Errors: Missing Braces

When you are learning to write your own programs in C#, we try to help you from making the kinds of errors that beginners often make. (Visual Studio also tries to assist you. You will have noticed that it automatically positions your code as you type, so that is indented properly.)

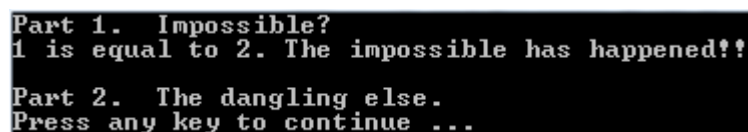
Nonetheless, there are certain errors that people often make when first learning to program. The program described below contains two of these errors. As before, this program is a deliberately simplified version of what people have done in their own programs.

1. From the **FrequentErrors\_MissingBraces** subfolder, open the **FrequentErrors\_MissingBraces** solution in Visual Studio.
2. In the Solution Explorer window, open **FrequentErrors\_MissingBraces.cs** by double-clicking on it.
3. Have a look at the program code. The first part of the **Main** method is shown below.

```
Console.WriteLine("Part 1. Impossible?");  
  
if (1 == 2)  
    Console.WriteLine("??");  
    Console.WriteLine("1 is equal to 2. The impossible has happened!!");
```

Here the aim is that the first **WriteLine** statement will be executed – producing the output: "Part 1. Impossible?". The other two **WriteLine** statements aren't expected to be executed, because the number 1 cannot be equal to the number 2.

4. Run the program by pressing **F5 (Start Debugging)**. The following output should appear.



```
Part 1. Impossible?  
1 is equal to 2. The impossible has happened!!  
  
Part 2. The dangling else.  
Press any key to continue ...
```

As you can see, the third **WriteLine** statement was executed. How did that happen?

The problem is that the author forgot to put braces **{ }** around the two statements that follow the **if** test. Because of that, the system treats the statements as though they had been written like this.

```
if (1 == 2)  
    Console.WriteLine("??");  
Console.WriteLine("1 is equal to 2. The impossible has happened!!");
```

So, the third **WriteLine** statement will always be executed.



If you like, use **F10** to step through the code and see which statements are being executed.

- Fix the problem by adding the missing braces, as shown below. This tells the compiler precisely where the **if** statement begins and ends.

```
if (1 == 2) {
    Console.WriteLine("??");
    Console.WriteLine("1 is equal to 2. The impossible has happened!!");
}
```

Then run the program again to confirm that this part now works correctly.

```
Part 1. Impossible?
Part 2. The dangling else.
Press any key to continue ...
```

- Now take a look at the second part of the **Main** method, as shown below.

```
Console.WriteLine("Part 2. The dangling else.");
bool happy = false;
bool friend = true;

if (happy)
    if (friend)
        Console.WriteLine("both");
else
    Console.WriteLine("not happy");
```

As before, the first **WriteLine** statement will be executed – producing the output: “Part 2. The dangling else.”.

And, because **happy** is **false**, the expectation is that the last **WriteLine** statement “not happy” will also be executed. But the Console window shows that this doesn’t happen. There is no “not happy” output before the “press any key” line.

- Fix the problem by adding the missing braces, as shown below. They tell the compiler that the **else** part belongs to the **if (happy)** rather than to the **if (friend)** condition.

```
if (happy) {
    if (friend) {
        Console.WriteLine("both");
    }
} else {
    Console.WriteLine("not happy");
}
```

Opening and closing braces for  
**if (happy)**

Then run the program again to confirm that this part now works correctly.

As these examples show, leaving out braces can cause a program to do different things to what you might think. The safe approach is to always include braces, even when the compiler might not insist on them. In your assessments, you are expected to always use braces. See the C# Coding Style Guide on Blackboard for more detail.