Chapter 2 – Review of Visual Basic Essentials

# Objectives

* Review of Visual Basic 146 Concepts
* Setting up Visual Studio for VB
* Debugging Skills
* Understanding File Extensions
* Understanding Namespaces

# VB CS 146 Review

So the first thing that we want to do is to review the essentials that you should remember from CS 146. This will include:

* Comments, data types, variables and constants
* Simple expressions and the assignment operator
* Basic decision constructs
* Basic repetition constructs
* Subprograms
* Basic input and output
* Forms and basic controls

## Comments, Data Types, Variables and Constants

'Comments start with single quote marks and can go at the end of lines too...

'Each comment must start with a quote -- no spanning multiple lines!

Const MYTYPEDCONSTANT As Integer = 55

Const MYSIMPLECONSTANT = 6

Structure udtStudent

Dim strStudentName As String

Dim strSocSecNum As String

Dim intCredits As Integer

Dim sngTuitionAmount As Single

End Structure

Dim intVar As Integer

Dim sngVar As Single

Dim dblVar As Double

Dim chrVar As Char

Dim strVar As String

Dim blnVar As Boolean

Dim dtmVar As Date

Dim bytVar As Byte

Dim lngVar As Long

Dim arrAnArray(9) As Integer '10 elements from 0 to 9

Dim arrDeclareAndInitializeArray() As Integer = {5, 10, 15, 20}

Dim udtMyStudent As udtStudent

Dim arrMyCourse(29) As udtStudent '30 student structures in this array

Dim objMyObjectInstantiator As Object = New Object

Dim objAnotherObjectInstantiator = New Object

Dim objMyObjectDeclaror As Object

objMyObjectDeclaror = New Object

## Simple Expressions and the Assignment Operator

Remember the math operations:

|  |  |
| --- | --- |
| *Operation* | *Operator* |
| Addition | + |
| Subtraction | - |
| Multiplication | \* |
| Floating Point Division | / |
| Integer Division | \ |
| Modulus | Mod |
| Exponentiation | ^ |
| Unary Minus | - |
| Unary Plus | + |
| Parenthesis (for grouping/affecting order of operations) | ( ) |

The assignment operator is how we assign values to variables:

intCount = 0

Other assignment related operators:

|  |  |  |
| --- | --- | --- |
| *Operator* | *Shorthand form* | *How VB interprets the line* |
| += | x += 1 | x = x + 1 |
| -= | x -= 1 | x = x – 1 |
| \*= | x \*= y | x = x \* y |
| /= | x /= y | x = x / y |
| \= | x \= (y + 2) | x = x \ (y + 2) |
| ^= | x ^= (y + 2) | X = x ^ (y+2) |
| &= | strStart &= strNewPiece | strStart = strNewPiece & strNewPiece |

### Simple Functions You Should Have Seen in 146

|  |  |  |
| --- | --- | --- |
| *Function* | *Purpose* | *Example* |
| Math.Sqrt | Square Root of a number | Math.Sqrt(9) 🡪 3 |
| Math.Round | Rounds a number to nearest value | Math.Round(7.6) 🡪 8 |
| Int | Chops off fractional component | Int(7.6) 🡪 7 |
| CDate | Converts string to date | CDate(“11/20/21”) 🡪 #11/20/2021# |
| CDbl | Convert string to double | CDbl(“5.3”) 🡪 5.3 |
| CInt | Convert string to integer | CInt(“10”) 🡪 10 |
| CSng | Convert string to single | CSng(“-4.2”) 🡪 -4.2 |
| CStr | Converts Boolean or number to string | CStr(5<7) 🡪 “True” |
| Asc | Returns the ASCII value of a character | Asc(‘A’) 🡪 65 |
| Chr | Returns the character of an ASCII value | Chr(97) 🡪 ‘a’ |
| IsNumeric | Tells if string can be converted to a number | IsNumeric(“10.3”) 🡪 True |
| Val | Attempts to convert a string to number | Val(“53.7”) 🡪 53.7 |

## Basic Decision Constructs

The decision constructs allow us to ask the computer “questions” via if statements. They also allow us to compare two items to each other and make a decision based on the outcome. Finally, we can chain comparisons together using logic operators.

Decision Making Operators:

|  |  |
| --- | --- |
| *Operation* | *Operator* |
| Equal To | = |
| Not Equal To | <> |
| Greater Than | > |
| Greater Than or Equal To | >= |
| Less Than | < |
| Less Than or Equal To | <= |

Logic Operators:

|  |  |
| --- | --- |
| *Operation* | *Operator* |
| Logical And | And |
| Logical Or | Or |
| Logical Xor | Xor |
| Logical Not | Not |
| Short-circuited Logical And | AndAlso |
| Short-circuited Logical Or | OrElse |

Bitwise Note: we can also make use of the first 4 logic operators at the bitwise level too. For example:

Dim intAnswer As Integer

intAnswer = 2 Or 1

MessageBox.Show(intAnswer)

'Returns the answer 3 which is the result of 0010 Ored with 0001 = 0011

Additionally There are also bitwise shift operators << (left shift) and >> (right shift) as well as shorthand assignment operators of <<= and >>=.

Here’s the simple if:

If blnCondition Then

'Body

End If

Here’s an if-else statement:

If blnCondition Then

'Body - True Side

Else

'Body - False Side

End If

We can nest if statements inside one another:

If blnCondition Then

'Body - True Side

Else

If blnCondition2 Then

'Body 2 - True Side

Else

'Body 2 - False Side

End If

End If

Here is the abbreviated if-elseif construct (notice only one End If is required here):

If blnCondition Then

'Body - True Side

ElseIf blnCondition2 Then

'Body 2 - True Side

ElseIf blnCondition3 Then

'Body 3 - True Side

Else

'No condition was met – Essentially the “False side”

End If

The select case statement is usually a better and cleaner way to go than the if-elseif:

Select Case intValue

'Notice the variety of case tags that we can express in a Select Case

Case 0 : MessageBox.Show("This is case 0's body")

Case 1, 2, 3 : MessageBox.Show("This is case 1, 2 or 3's body")

Case 4 To 10 : MessageBox.Show("This is case 4, 5, 6, 7, 8, 9 or 10's body")

Case 11, 12 To 15, 16 : MessageBox.Show("This is case 11, 12, 13, 14, 15" &

"or 16's body")

Case Is < 0 : MessageBox.Show("This would be the body for all negative " &

"number values")

Case Else

MessageBox.Show("This would be the body for any positive number " &

" greater than 16")

End Select

There is one more if construct, the single line if but it can only have one line in the body and no End If:

If blnCondition Then intValue = 5

This is not a good programming practice to ram the condition and result on a single line!

## Basic Repetition Constructs

Repetition, or loop, constructs allow us to repeat program steps numerous times. There are both pre-test and post-test constructs and we see them in the form of for-next, while-end while and do-until structures.

Example of the basic For-Next variable controlled loop:

For intValue = 1 To 100

'Body of the loop

'For-Next loops are pre-test

Next

For-Next loops can jump by interval steps both forward and backwards:

For intValue = 1 To 100 Step 5

'Body of the loop

‘Runs 20 times: 1, 6, 11 ... 91, 96

Next

For intValue = 100 To 1 Step -5

'Body of the loop

'Runs 20 times: 100, 95, 90 ... 10, 5

Next

The remaining loops are all Boolean controlled. Here is the While-End While – the most common pre-test loop:

While blnCondition

'Body of the loop

'While-End While loops are pre-test

End While

The most common post-test loop is the Do-Loop Until:

Do

'Body of the loop

'Do-Loop loops are post-test

Loop Until blnCondition

There are several other variants of the Do and Loop structures:

Do While blnCondition

'Body of the loop -- pre-test since test is at top of loop

Loop

Do Until blnCondition

'Body of the loop -- still pre-test since test is at top of loop

Loop

Do

'Body of the loop – post-test since test is at bottom of loop

Loop While blnCondition

Some additional notes on the looping constructs:

1. Remember that you don’t have to declare the loop variable outside of the loop. We can declare the loop variable at the same time that we create the loop as in:

For intLoopCounter As Integer = 1 To 20

'Body of the loop

Next

This is okay since you are still declaring the data type of the intLoopCounter variable and it will cease to exist at the termination of the loop, e.g. its scope ends. If you try to create a variable by the same name but a different type at a higher scope level containing the for loop block, you will receive an error that you are hiding the variable – essentially this guarantees there aren’t scope issues!

VB will also allow you to type infer the loop index variable’s data type from the initializer and terminator types – however, this is not good programming practice:

For someLoopCounter = 10 To 20

'Hopefully the compiler guesses right and makes

'someLoopCounter integer in nature!

Next

1. You have the equivalents of Break and Continue in VB – they are Exit Do/Exit While/Exit For and Continue Do/Continue While/Continue For. These are technically goto statements in disguise and properly designed logic shouldn’t really need them!

For intLoopCounter As Integer = 1 To 5

If (intLoopCounter > 3) Then

'When the intLoopCounter hits 4, we bail...

Exit For

End If

'Loop body we'd normally execute each time

MessageBox.Show(intLoopCounter)

Next

For intLoopCounter As Integer = 1 To 5

If (intLoopCounter = 4) Then

'When the intLoopCounter hits 4, we skip it and go to 5...

Continue For

End If

'Loop body we'd normally execute each time

MessageBox.Show(intLoopCounter)

Next

As long as we’re on the subject of these items, there is also an Exit Select statement that can be used in one of the case bodies forcing the Select Case to immediately end. The designers of the language gave you the ability to quickly get out of any situation but I would caution you to design better logic so that these quick exit schemes aren’t necessary in the first place!

## Subprograms

Subprograms are the way that we break our programs up so that everything is not stuck in one huge monolithic block of code, e.g. everything is in Sub Main(). Remember that subroutines do not return values while functions do. Let’s look at a simple example of each:

This is a simple subroutine that prints out Hello There! when called:

Sub SayHello()

MessageBox.Show("Hello There!")

End Sub

This function returns the number 4 every time it’s called via the Return keyword. Also notice the return type is declared after the function’s name and parameter list:

Function ReturnsANumberFour() As Integer

Return 4

End Function

Here’s an invocation (call) of the subroutine:

SayHello()

This example calls the returned value from the function and then prints it:

intValue = ReturnsANumberFour()

MessageBox.Show(intValue)

We could call the function from within the printing object itself:

MessageBox.Show(ReturnsANumberFour())

We can also pass parameters through to the functions/subprograms and give programmatic permission to modify or not modify what goes through:

Sub SquaresTheArgument(ByRef NumToChange As Integer)

Dim intTempValue As Integer

'Notice that even though this is not a function

'that by passing by address/reference, I can get the

'subroutine to behave somewhat similar to a function

intTempValue = NumToChange \* NumToChange

NumToChange = intTempValue

End Sub

Function ReturnsTheArgumentSquared(ByVal NumCantBeChanged As Integer) As Integer

Dim intTempValue As Integer

'The parameter is passed by value, so it can’t be changed

'but we really don’t need to change because we are creating a function to

'return the squared number

intTempValue = NumCantBeChanged \* NumCantBeChanged

Return intTempValue

End Function

Of course we can mix and match what we pass through and have some parameters passed by value and some passed by address just by prefixing each parameter value with the passing mechanism type…

## Basic Input and Output

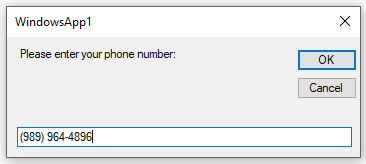
The quick and dirty way to get input and output between the user and the program is to use the MessageBox and InputBox objects. MessageBox.Show will pop up and display whatever information you want the user to see and InputBox will allow the user to type in information into a textbox and then place the contents of what was entered into a variable:

Dim strPhoneNumber As String

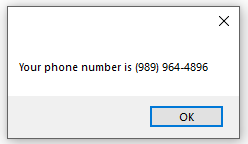
strPhoneNumber = InputBox("Please enter your phone number:")

MessageBox.Show("Your phone number is " & strPhoneNumber)

InputBox prompt:



MessageBox Response:

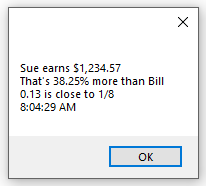


Remember that you can also pretty up numbers with these gems:  
MessageBox.Show("Sue earns " & FormatCurrency(1234.567) & vbCrLf &

"That's " & FormatPercent(0.3825) & " more than Bill " & vbCrLf &

FormatNumber(0.125, 2) & " is close to 1/8 " & vbCrLf &

FormatDateTime(Now(), DateFormat.LongTime))



What you should get in the habit of using is the String.Format command – remember this works based on the idea of placeholders…

Standard Numeric Format Placeholders:

|  |  |  |
| --- | --- | --- |
| Currency (C) | 123.456 | $123.46 |
| Decimal (D6) | -1234 | -001234 |
| Exponential (E) | 1052.0329112756 | 1.052033E+003 |
| Fixed-point (F4) | -1234.56 | -1234.5600 |
| General (G) – uses more compact of fixed-point or exponential | -123.456 | -123.456 |
| Number (N2) | 1234.567 | 1,234.57 |
| Percentage (P2) | 0.55555 | 55.56% |
| Hexadecimal (X) | 15 | F |

Standard Time Format Placeholders:

|  |  |  |
| --- | --- | --- |
| Short Date Pattern (d) | 6/15/2009 1:45:30 PM | 6/15/2009 |
| Long Date Pattern (D) | 6/15/2009 1:45:30 PM | Monday, June 15, 2009 |
| Full Date/Time Pattern Short (f) | 6/15/2009 1:45:30 PM | Monday, June 15, 2009 1:45 PM |
| Full Date/Time Pattern Long (F) | 6/15/2009 1:45:30 PM | Monday, June 15, 2009 1:45:30 PM |
| General Date/Time Short (g) | 6/15/2009 1:45:30 PM | 6/15/2009 1:45 PM |
| General Date/Time Long (G) | 6/15/2009 1:45:30 PM | 6/15/2009 1:45:30 PM |
| Month/Day Pattern (M or m) | 6/15/2009 1:45:30 PM | June 15 |
| Short Time Pattern (t) | 6/15/2009 1:45:30 PM | 1:45 PM |
| Long Time Pattern (T) | 6/15/2009 1:45:30 PM | 1:45:30 PM |
| Universal Sortable Date/Time (u) | 6/15/2009 1:45:30 PM | 2009-06-15 20:45:30Z |
| Universal Full Pattern (U) | 6/15/2009 1:45:30 PM | Monday, June 15, 2009 8:45:30 PM |
| Year/Month Pattern (Y or y) | 6/15/2009 1:45:30 PM | June, 2009 |

Dim intIntegerNumber As Integer = 2000

Dim sngRealNumber As Single = 1.0 / 3.0

Dim txtName As String = "Bill Jones"

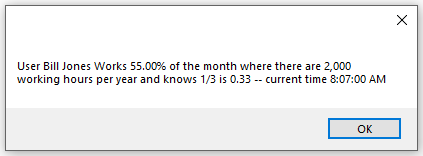
Dim sngPercentage As Single = 0.55

MessageBox.Show(String.Format("User {0} Works {1:P} of the month where " &

"there are {2:N0} working hours per year and knows 1/3 is " &

"{3:N2} -- current time {4:T}", txtName, sngPercentage,

intIntegerNumber, sngRealNumber, Now()))



It’s also pretty easy to get nice columnar output as long as we pick a non-proportional font:

Dim fmtStr As String = "{0, -15}{1, 10}{2, 8}"

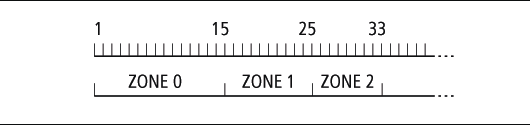
lstListBox1.Items.Add(String.Format(fmtStr, \_

data0, data1, data2))

'15 was preceded by a minus sign -- This

'produces left justification in the 0th zone. There will

'be right justification in the other two zones.



We can even combine the columnar zones with formatting types:

Dim fmtStr As String = "{0, -15:N1}{1, 10:C2}{2, 8:P0}"

'By assigning the layout to a string variable, our program could dynamically

'generate this at runtime adapting/changing the layout to whatever needs arise!

### File IO

Working with text files in Visual Basic is pretty simple thanks to the StreamReader and StreamWriter classes.

Dim objMyStreamReader As System.IO.StreamReader

Dim objMyStreamWriter As System.IO.StreamWriter =

System.IO.File.CreateText("C:\VBtest\test.txt")

'Make sure the folder in the previous line exists before you start the program.

'Remember that since Windows 7, you are not allowed to write to the root of the

'hard drive...

'If you don't include a path, the file will be created in the same folder as your

'executable (.exe) file

Dim strTheFileContents As String

'write some values out to the file

objMyStreamWriter.WriteLine("Bill" & vbTab & "15")

objMyStreamWriter.WriteLine("Kim" & vbTab & "18")

objMyStreamWriter.WriteLine("Sue" & vbTab & "25")

objMyStreamWriter.WriteLine("Mike" & vbTab & "37")

objMyStreamWriter.Close()

'Now read everything back in and print it out

objMyStreamReader = System.IO.File.OpenText("C:\VBtest\test.txt")

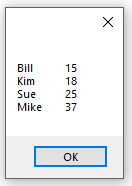
While Not (objMyStreamReader.EndOfStream)

strTheFileContents &= objMyStreamReader.ReadLine() & vbCrLf

End While

objMyStreamReader.Close()

MessageBox.Show(strTheFileContents)Here’s the results:



## Forms

I also expect that you will be comfortable creating Windows Forms App (.NET Framework) based applications. We will be creating numerous Forms apps as well as Console Apps (.NET Framework) over the course of the semester.

You should be comfortable using the following controls (and events/properties):

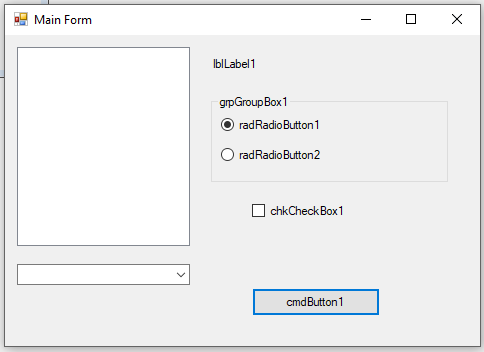
* Labels (lbl)
* Command Buttons (cmd)
  + Events:
    - Click
* Textboxes (txt)
  + Events:
    - TextChanged
    - TextLeave
* Masked Textbox (mtx)
  + Properties:
    - Mask
* Listboxes (lst)/Comboboxes (cbo)
  + Events:
    - Click
    - SelectedIndexChanged
    - DoubleClick
  + Properties:
    - Items (.Add, .Clear, .Remove, .RemoveAt, .Count)
    - ListIndex
    - SelectedIndex
    - Sorted
* Checkboxes (chk)/Radiobuttons (rad)
  + Events:
    - CheckedChanged
  + Properties:
    - Checked
* Groupboxes (grp)

General Properties you should know:

* Name
* Text
* Autosize
* Font.Name
* Font.Size
* ForeColor
* BackColor
* ReadOnly
* Visible

General Events you should know:

* Click
* Focus



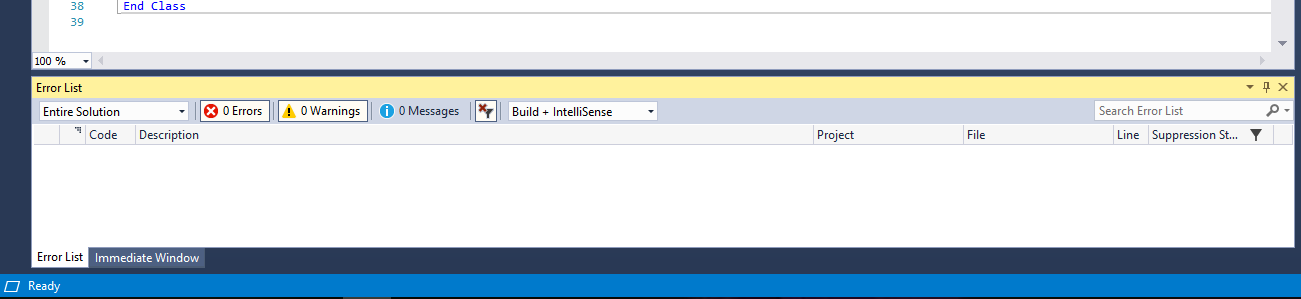
# Setting Up Visual Studio.NET for Optimal VB Development

When you install Visual Studio and run it for the first time, you are asked what development mode you prefer to start up in. You should select Visual Basic Development as it will make your life simpler since it configures itself to run optimally for this language.

I would then recommend that you add a few windows…

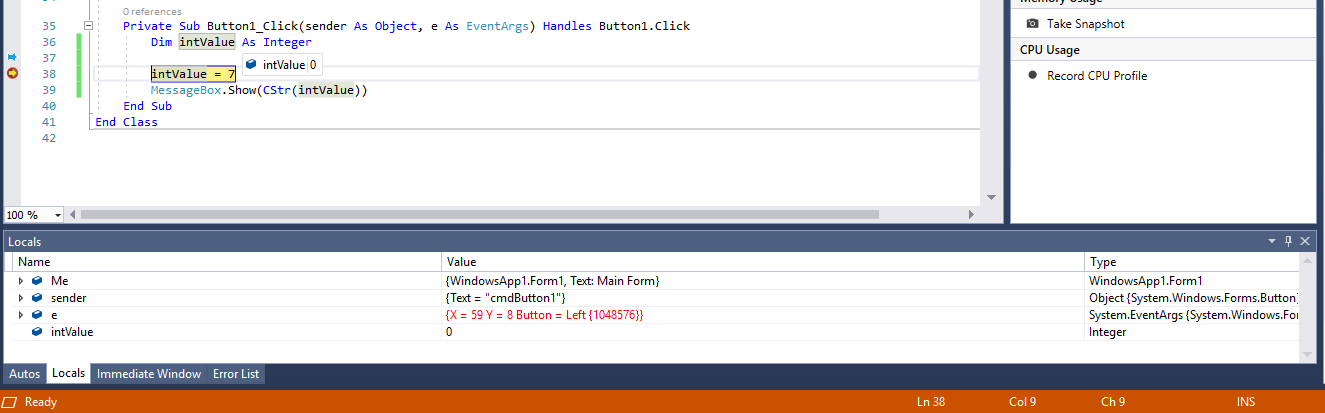
1. Select Error List under the View menu, if it is not already visible at the bottom of the IDE. This will add the ability to see compilation errors and warnings as type code in and as you actually compile.
2. Select Debug🡪Windows🡪Immediate Window. This adds the immediate window where you can actually change the contents of variables as your code is running and also instantiate objects and call methods on the fly.
3. The next time you have some VB code in your editor, add a breakpoint and when the code stops, click on Debug🡪Windows🡪Locals and Debug🡪Windows🡪Autos. The Autos will show you the data items related to the most recent line that you are executing and where the debugger is stopped. Locals will show you all of the data items related to the appropriate scope level of the object you are executing in, for example a form.

All of these windows show up tabbed at the bottom of the IDE. Here’s what I see when running on a small code snippet:

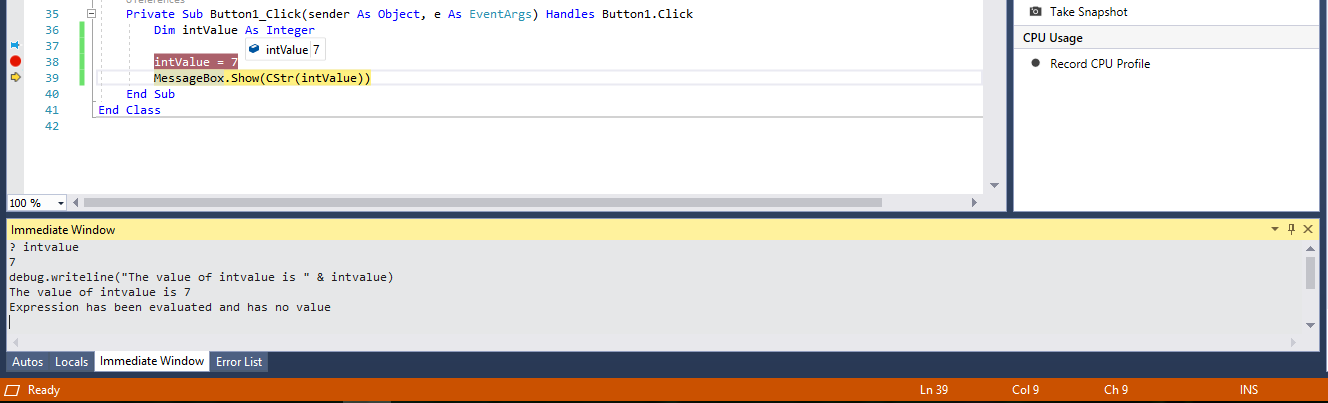


Notice that there are no errors, so the Error List window is empty. The only other window I can see when I am not running the application is the Immediate Window (also notice the status bar is blue indicating the application is not running).

The Autos and Locals will only appear when there is executing code (and the status bar turns amber)…makes sense and saves on clutter!

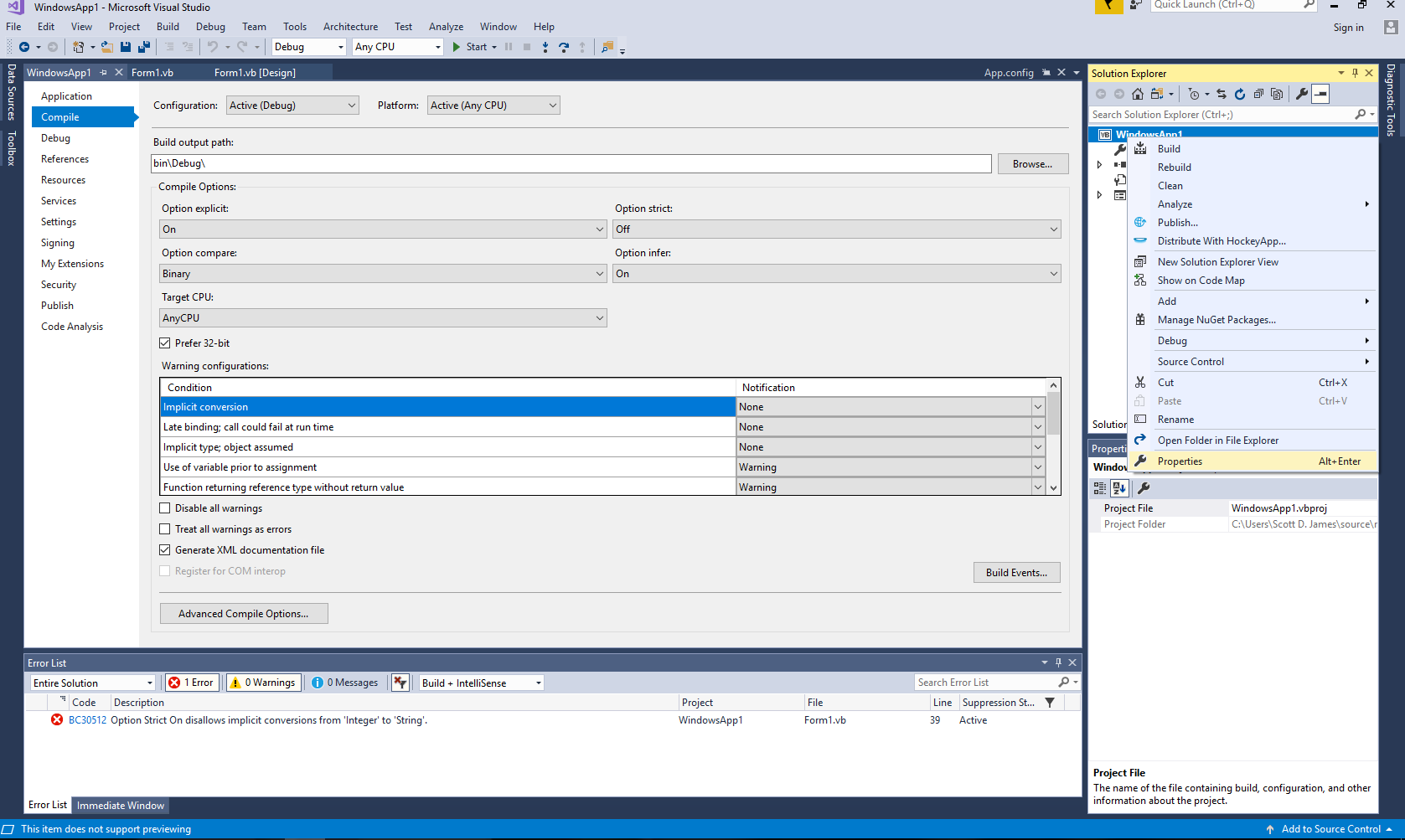


Look at what we can do with the immediate window:



Pretty cool, eh? And, a lot of really useful information as you try to debug your programs!

1. Oops…but what if I didn’t pick Visual Basic as my preferred start up environment when I initially installed. Not to fret! Select Tools🡪Import and Export Settings🡪Reset All Settings. When you exit and restart Visual Studio, you will again be prompted for the preferred environment.
2. Check your Option settings. Take a look at the project’s properties by right clicking on the project name under the Solution Explorer tab (far right pane), selecting Properties from the context menu and finally the Compile tab on the properties page in the left pane:



What do these settings mean?

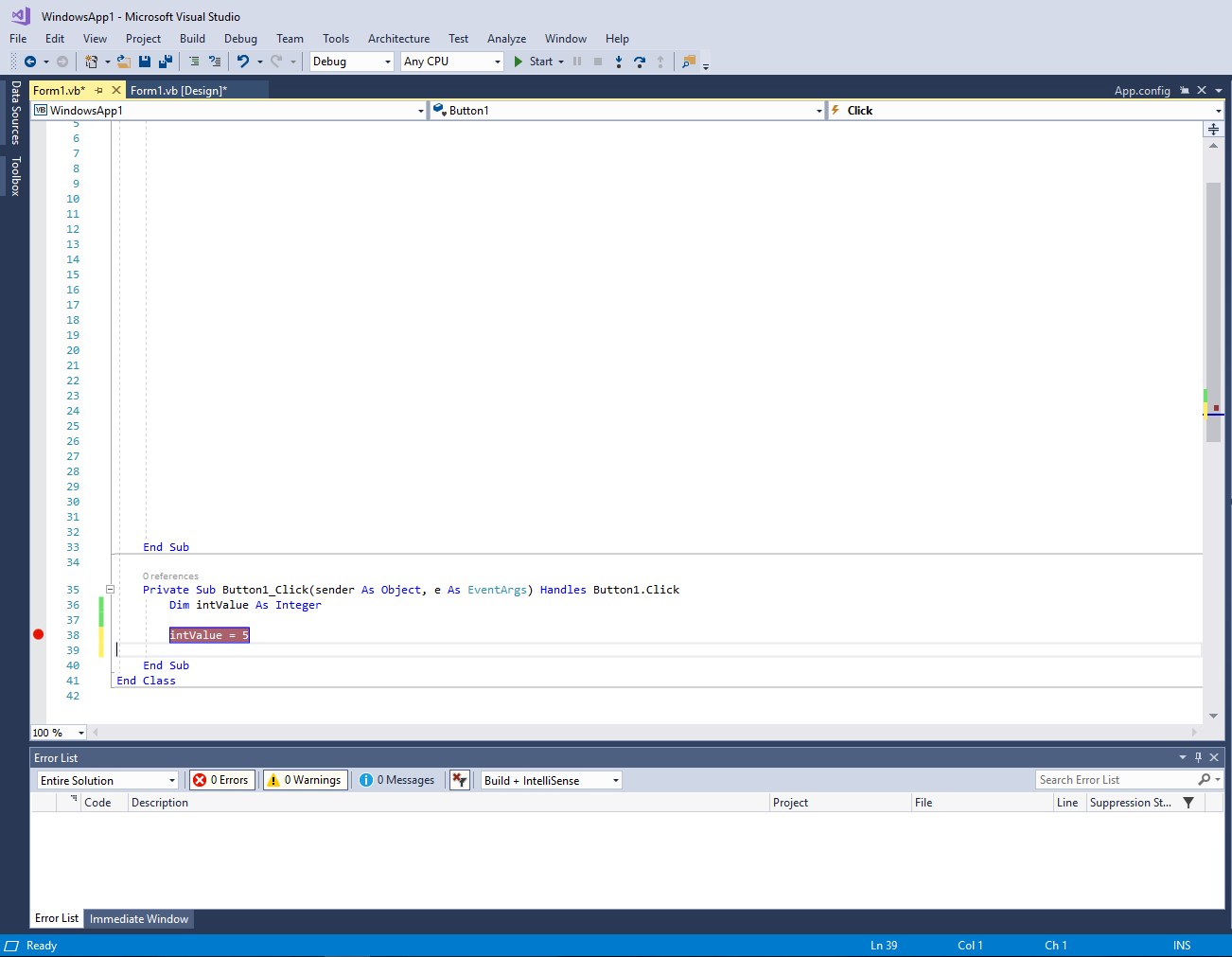
* Option Explicit On – means that all variables have to be declared before you can use them. This is a good thing otherwise a misspelled variable name is a new variable!
* Option Strict Off – will allow you to perform conversions between compatible data types without having to explicitly code the conversions. Having it off allows numeric values to be assigned to strings and what not – On stops this, but requires that you do all of the appropriate conversions yourself.
* Option Compare Binary – says to use a case sensitive comparison. The other mode, Text, is case insensitive.
* Option Infer On – allows the compiler to infer a variable’s type based on the initializer variable. This is what allows the statement For x = 1 To 10 to create a for loop scoped integer variable x. In my opinion, this should be changed to Off. I don’t like the compiler guessing things for me!

# Basics of Debugging

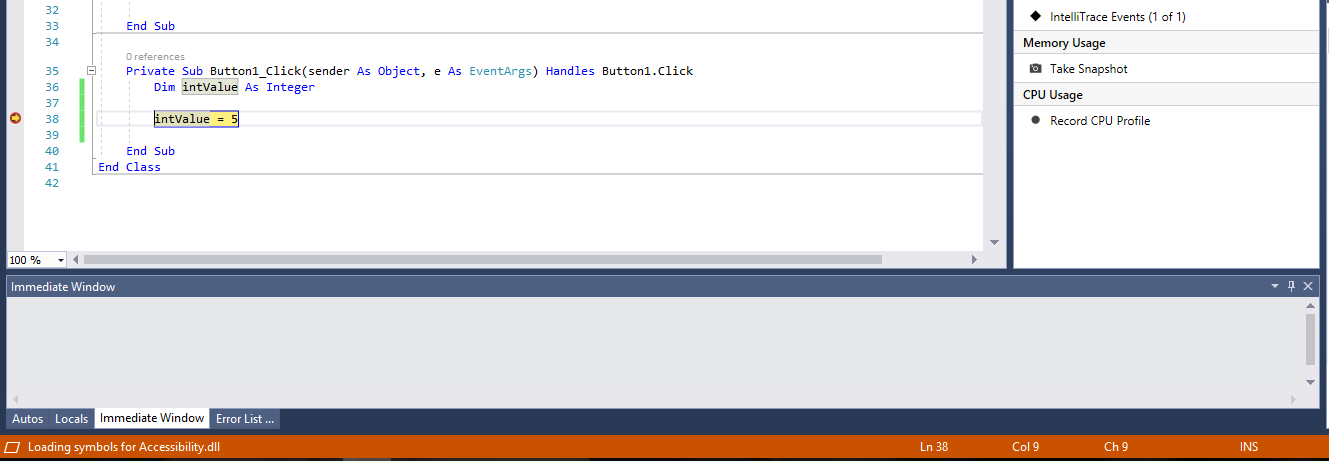
## Breakpoints and Stepped Execution

Breakpoints are going to be your best friend in the Visual Studio IDE. You can add as many as you want and whenever you run your application, it will run uninterrupted until it encounters a breakpoint. When it hits one, it stops and puts you in the debugging seat to see what’s going on under the hood with your code.

To set a breakpoint, all you need to do is left click to the left of a legitimate line of code in the gray margin. Don’t click on a comment line or a line of syntactically broken code, since that won’t work. If where you left clicked is valid, you’ll see a red circle show up – this is a set breakpoint:

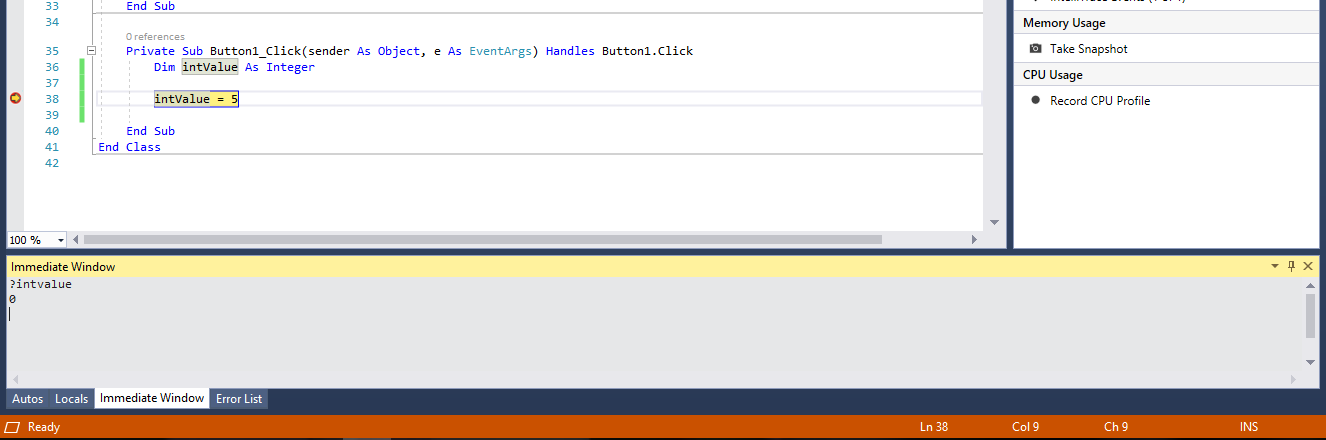


You can toggle breakpoints on and off just by clicking on the red circles. Now the really cool part is when you run your program – it’ll stop at any breakpoint it hits:

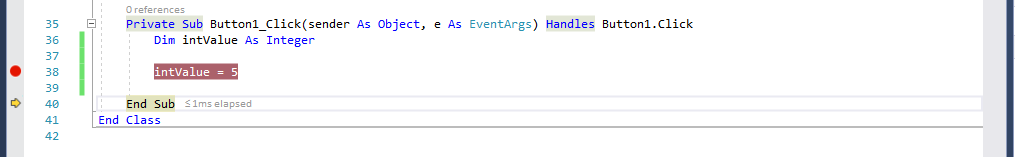


Notice that the breakpoint line is highlighted in yellow. This indicates that the yellow line will be the next to execute. You also will see a yellow arrow out in the gray gutter area – again, this shows where the next executing line lives.

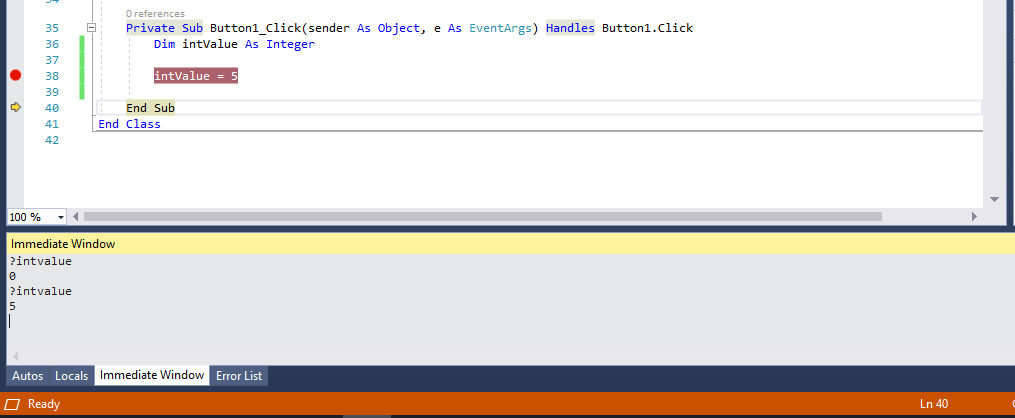
Go down to the Immediate Window and type ?intValue:



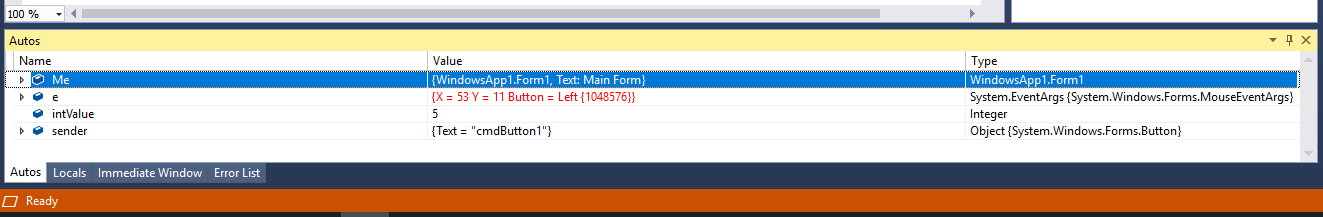
The ? stands for print, so we effectively asked the debugger to tell us what value is in intValue, and since we haven’t done anything with it since we declared it, it should be 0 – which it is. Now press the F10 key once.



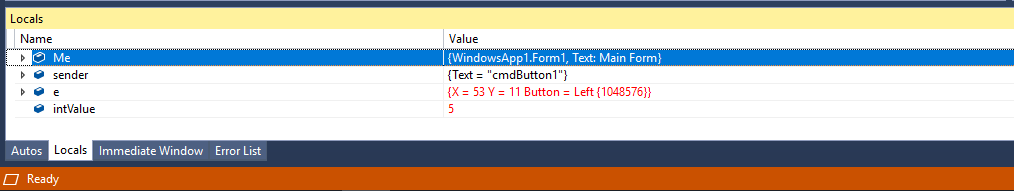
Did you see how the yellow arrow moved down one line to the End Sub? The End Sub would be the next line to execute, meaning that our intValue = 5 line just ran. Let’s confirm that by checking the value in the Immediate Window.



Sure enough the value is 5, just like the code said it should be. If we were to press F10 again, the End Sub would execute and then we would be taken to the next line of code we need to execute. If we take a look at Autos we see the variables that are related to the line we are currently executing/where the debugger has stopped:

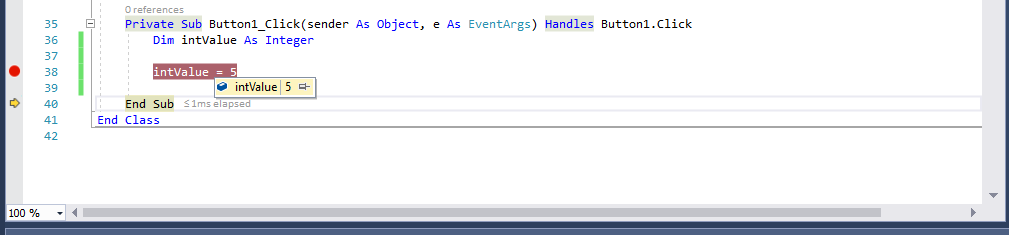


In this case I see e and sender which are the parameters sent to this command button’s Click event handler and I also see intValue. e and sender make sense to show up now because I am about ready to end the event handler that contained them. If I click on Locals, I will see the object scope level which contains the code block I am currently executing and that will be the Form. The items that are shown in the Locals and Autos windows are identical in this case, which isn’t surprising considering there are very few variables and/or data items in this program. Here’s the Locals window:



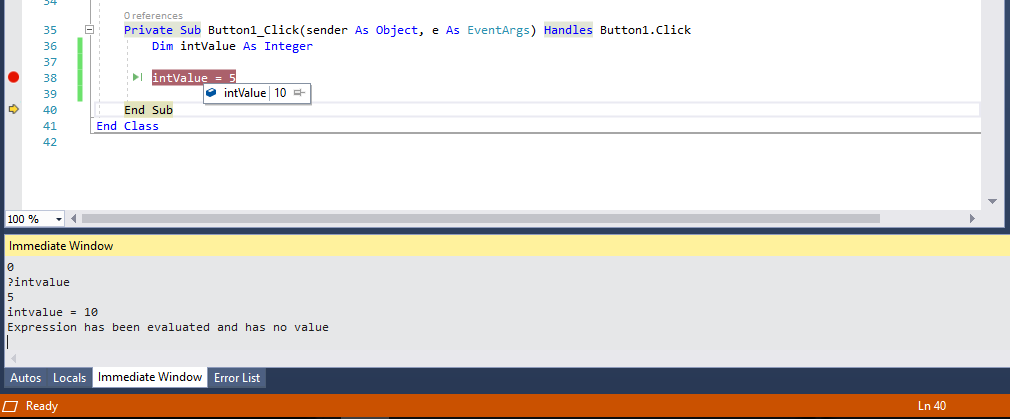
At the top of the list is Me, referring to the form that is currently being executed. Notice the arrow symbols in front of the three data items – those mean there are additional properties/attributes within. Click on the arrow and it will expand. Click again and it will contract. This behavior is true for arrowed items in both the Locals and Autos windows.

Finally, I have one other way to see what’s in intValue – the mouse. If I just park it over the code window – the value will pop up in what’s called a Quick Info box:

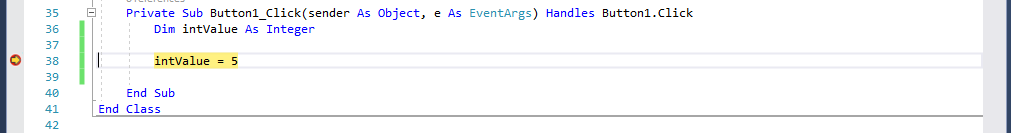


You can even pin the Quick Info box onto your source code canvas so that it stays there all the time you are debugging, and, as the value in the variable changes, you’ll see those changes immediately. Pretty slick – 3 different windows to look at what’s happening inside of my code.

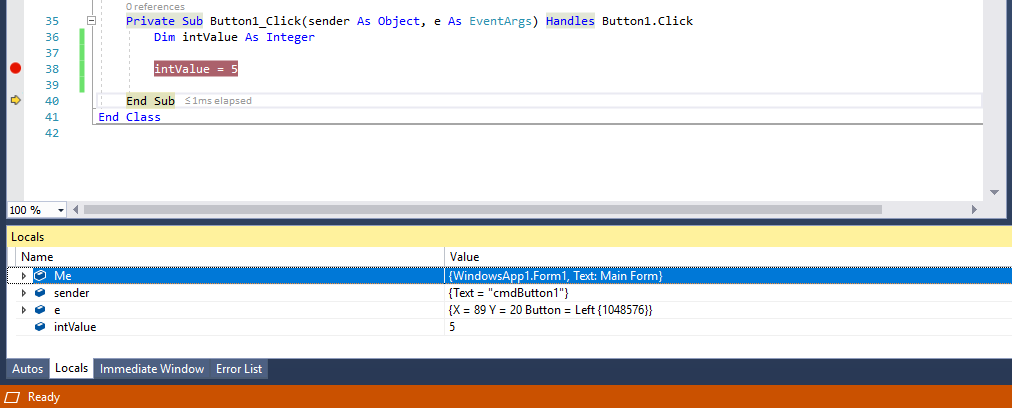
Let’s say though, that when I get to this point in my code, the value of intValue shouldn’t be 5 but instead I know that it should be 10. I’m curious to see if I could intValue was set to 10 if the rest of my code would work properly. No sense, stopping to fix this – we can do it through the Immediate Window by typing intValue = 10. Do this and then hover the mouse over the variable to see the updated Quick Info box:



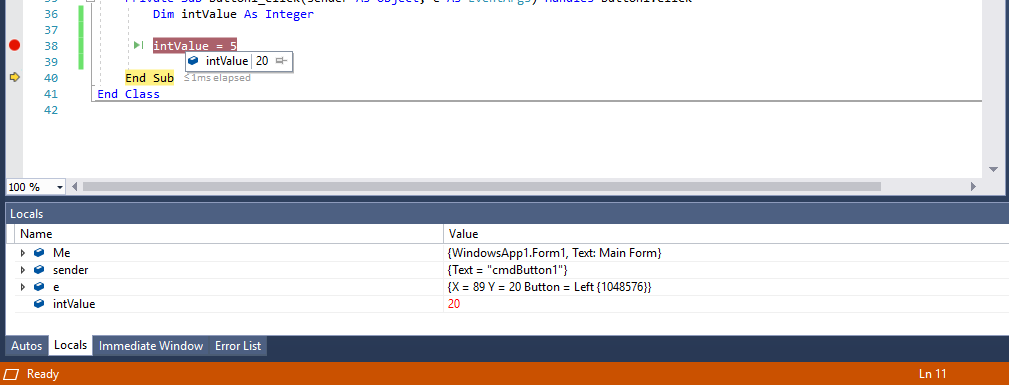
I could ask the IDE to continue running from this point and see what happens. We really don’t want this change to take place though, so you know that little yellow arrow: click and drag it back up to the intValue = 5 line…



We just reset where we want execution to take place…press F10 once and observe the value of intValue in the Locals window:



Right back to our good old 5! Now let’s say we want the 5 to actually be a 20. Well, I could jump back to the immediate window and make the change there. But you can actually change the values shown in your Autos and Locals windows directly – just click on the value you want to change, type in the new value and then click off of that value entry line:



The Quick Info Box shows 20 is the value and the Locals window highlighted where I typed the value in as red, meaning it had been changed! That’s pretty cool.

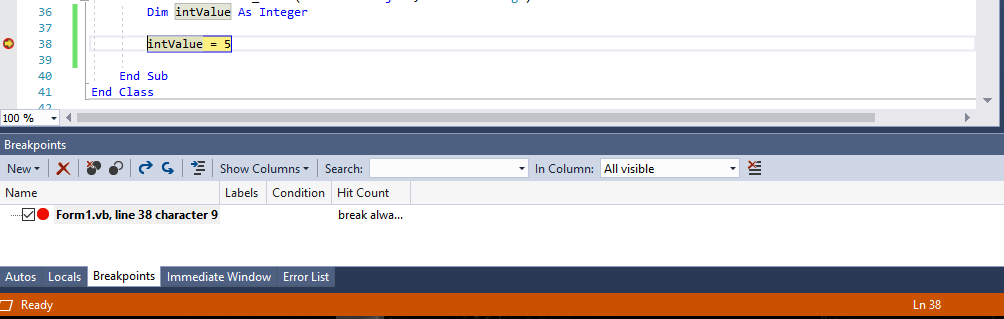
So we’ve seen how to put in breakpoints and I told you about using the F10 key to execute the next line. There are a few more keystrokes that we want to look at:

|  |  |
| --- | --- |
| *Keystroke* | *Purpose* |
| F10 | Single step execute the next line |
| F9 | Toggles breakpoints on and off |
| Shift-F8 | Steps over a subprogram (executes the subprogram and advances to next line of code after the subprogram call) |
| F11 | Steps Into a subprogram (debugging goes into the subprogram) |
| Shift-F11 | Steps out of a subprogram (finishes executing the current subprogram then pauses) |
| F5 | Start debugging |
| Ctrl-F5 | Executes the program and temporarily ignores all debugging information |

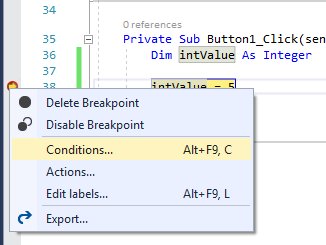
You can also get most of these same functions through the Debug button bar of your Visual Studio IDE, which should be on by default, but if it’s not you can activate it through View🡪Toolbars🡪Debug.



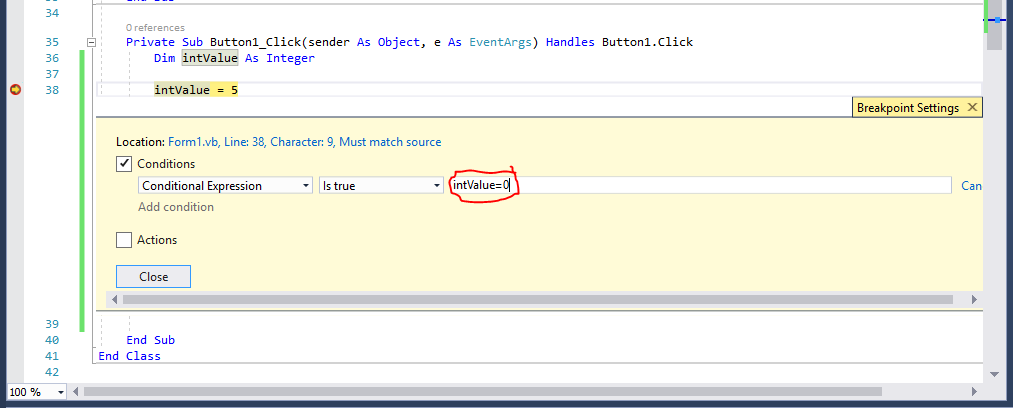
I want to talk about one more breakpoint related item and that is stopping when a given condition is met during program execution. If we had a loop that was supposed to run 100 times and it works right for the first 80, single stepping through until we hit loop execution 81 would be tedious. Here’s where another debugging feature can really help out. If we select Debug🡪Windows🡪Breakpoints, we get a new window that’s tabbed in with the others:



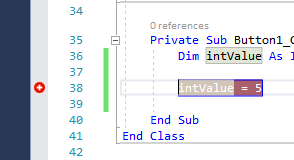
It may not look like much, but there’s a ton of power there! We can add conditions when we want the execution to stop. I’m going to change it so that my program pauses at this point only when intValue = 0. If I right click on the debug point line in the gray gutter, I am given a context menu. I will select Conditions:



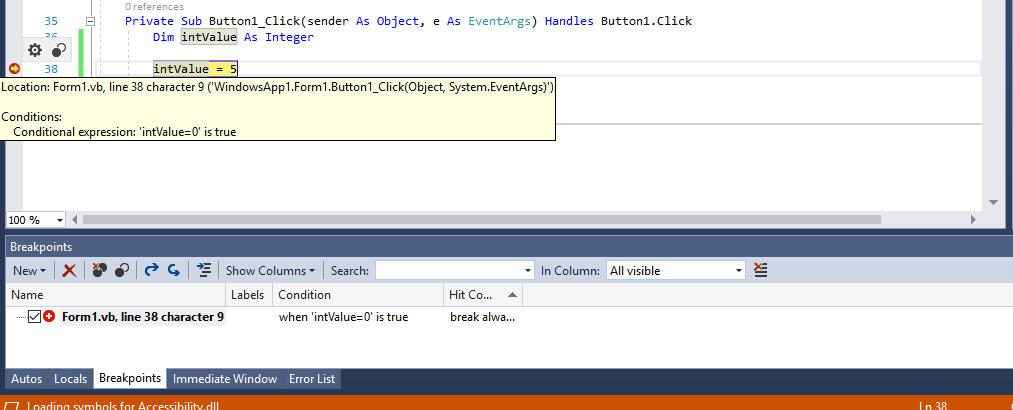
Once I’ve selected conditions, there’s an edit screen that appears right in my source code that allows me to describe the stopping condition. Here’s that screen filled out, in which I entered intValue=0 in the condition line.



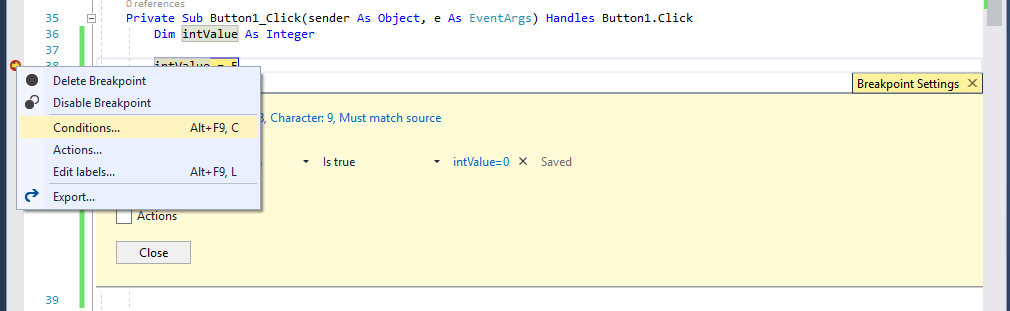
This tells Visual Studio to stop execution on the highlighted line whenever intValue’s value changes to 0. Click off of the textbox and the condition is saved. We can then press the Close button to close that conditional breakpoint editing window up. If we were to stop the program’s execution and take a look at the source code window, we would see that the normal breakpoint symbol has been changed to one with a plus sign on it, indicating this is a conditional breakpoint:



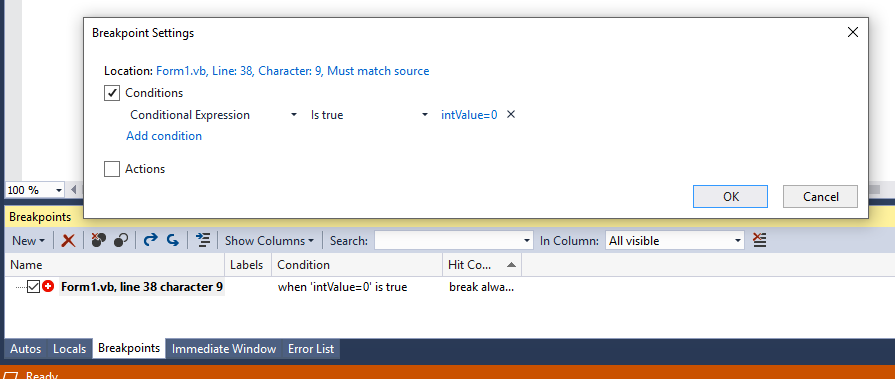
If we run the program until the breakpoint is encountered, we see a couple of things. (1) If we hover over the breakpoint in the source code window, the conditional breakpoint is displayed. (2) In the Breakpoints tab, we also see the condition there.



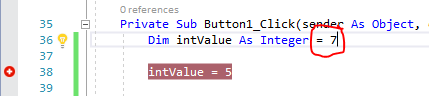
In the case that we decide we want to edit the condition on the breakpoint, we can simply right click the breakpoint in the gutter and select Conditions from the context menu. The Conditions sheet reappears in the source code window and changes (edits/adds/deletes) can be made:



We can also get to that same kind of breakpoint informational screen, by right clicking anywhere on the breakpoint information line in the Breakpoints tab and selecting the Settings option. A floating settings window appears rather than the sheet that appears in the source code window, however all the same functionality is there:

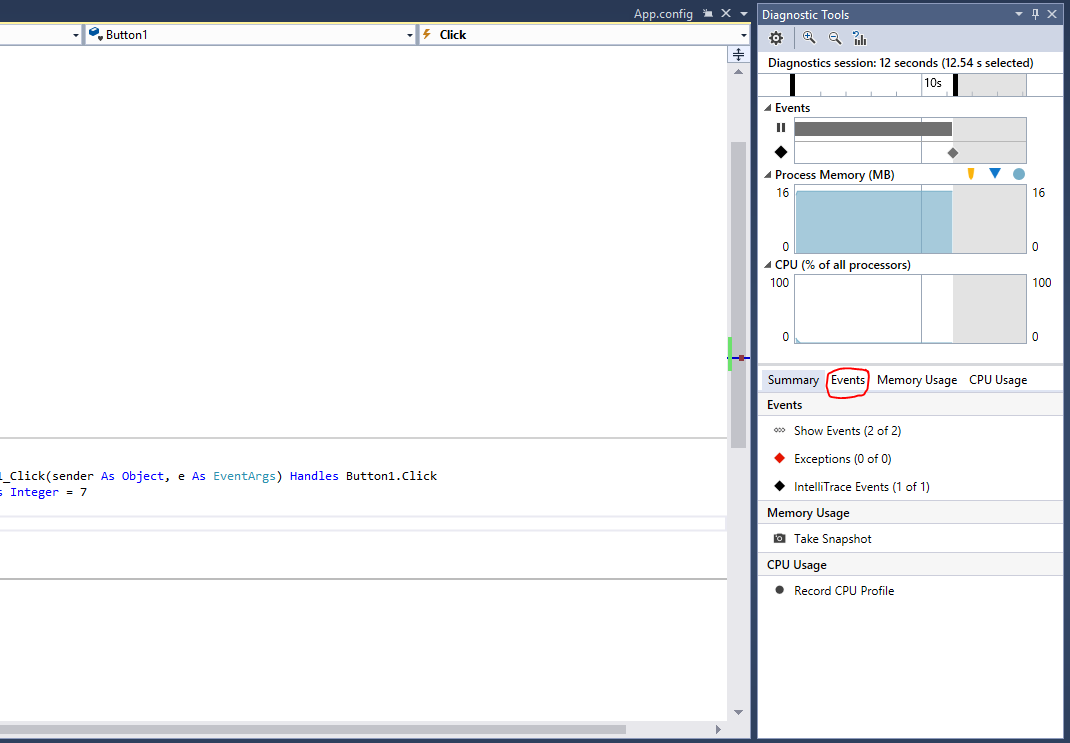


Now when I run the application it will pause only if intValue = 0… If I make a change to my code, setting intValue to 7 when it gets created, it will never be zero: therefore, the line will never break. Very, very cool!

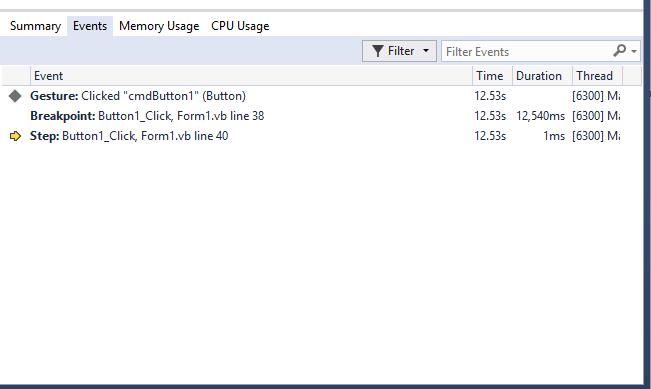


If you really want to see what’s going on, add additional normal (non-conditional) breakpoints on the Dim line and on the End Sub line. Single step through and watch you Autos window – you’ll see the variable is not zero when it reaches the conditional breakpoint line, hence, there is no reason to halt the program on that line because the condition is not met!

Lastly, I want to introduce the idea of IntelliTrace here. This will keep track of everything you’ve done up to a breakpoint being hit. It should be automatically showing up over in the Diagnostic Tools section in the middle of the IDE when you execute. If it’s not, select Debug🡪Windows🡪IntelliTrace Calls. Here’s what I see when I make my program break on the intValue = 5 line and then F11 single step:



If I click on Events and then slide the pane divider over to the left to get a bit more real estate on the screen, I see something that looks like this:

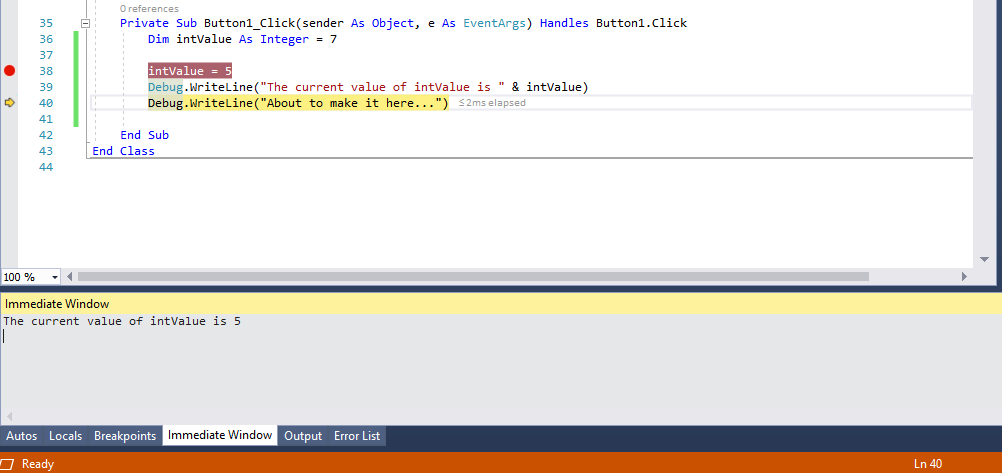


You can see the application started, I clicked (Gesture) on the Button1 command button, the debugger hit my breakpoint in the Click handler at the intValue = 5 line and then I stepped forward. This might not seem like much now, but you actually have the ability to see what’s happening over on servers like IIS and SQL Server with this little gem. No more guessing and trying to put both client and server side code in just to determine where things are going haywire! There’s way more that you can do with both Intellitrace and the Diagnostics Tools set but that’s all we are going to look at for now… As you can see, the time and duration to carry out tasks is recorded. There are even other tabs that show you your memory and CPU usage.

Hands down, Visual Studio is the most comprehensive development suite on the planet bar none. Other toolkits are constantly trying to catch up and match the capabilities that Microsoft builds into their toolchain. If you’re really, really interested in learning about all the things that Visual Studio can do, I would recommend “Professional Visual Studio 20XX” by Bruce Johnson. He keeps updating his fantastic book as new releases of Visual Studio become available.

## The Debug Object

As if there weren’t already a great number of debugging tools available, you can work with the Debug object. The output of the Debug object will be written to the Immediate window. For example, I can print now print diagnostic messages as well as the output of variables to the Immediate window by inserting an additional line of code using the Debug.WriteLine() statement. Here is the new code and the Immediate window displayed:



You can clear the Immediate window at any time by right clicking on its canvas and then selecting Clear All from the context menu that appears.

# Understanding File Extensions

Another thing that radically changed in VB.NET is the number of file extensions that are produced in a typical project. VB 6 created a handful of files, but as you can see from the next list, there are a multitude of files that you may run across when examining what VB placed in your project directory.

|  |  |
| --- | --- |
| *Name* | *Description* |
| .asax | ASP.NET configuration file that handles global session scripts |
| .asmx | Web service source file |
| .asp | Active Server Page source file |
| .aspx | Web application form |
| BIN directory | Where builds are loaded from |
| .css | Cascading Style Sheets |
| .disco | Dynamic Discovery document source file that enumerates Web services and schema in a Web services project |
| .html | HTML source files |
| .js | Jscript .NET source file |
| OBJ directory | Used for the output of specific configurations and builds |
| .resx | The assembly resource files used for the definition of application resources |
| .rpt | Crystal Reports Designer files |
| .sln | The solution file – details about the projects and their locations in the solution |
| .suo | Solutions User-Options – stores all of your custom settings |
| .tdl | Template Description Language file |
| .vb | A source code file: could be a class, form or component |
| .vbproj | The Visual Basic project file – contains all information about the files in a project |
| .vbs | Visual Basic Script source file |
| .web | ASP.NET configuration file |
| .wsf | Windows Scripting source file |
| .xml | An XML document |
| .xsd | An XML schema file |
| .xslt | XML files containing transformation instructions for xml and xsd documents |

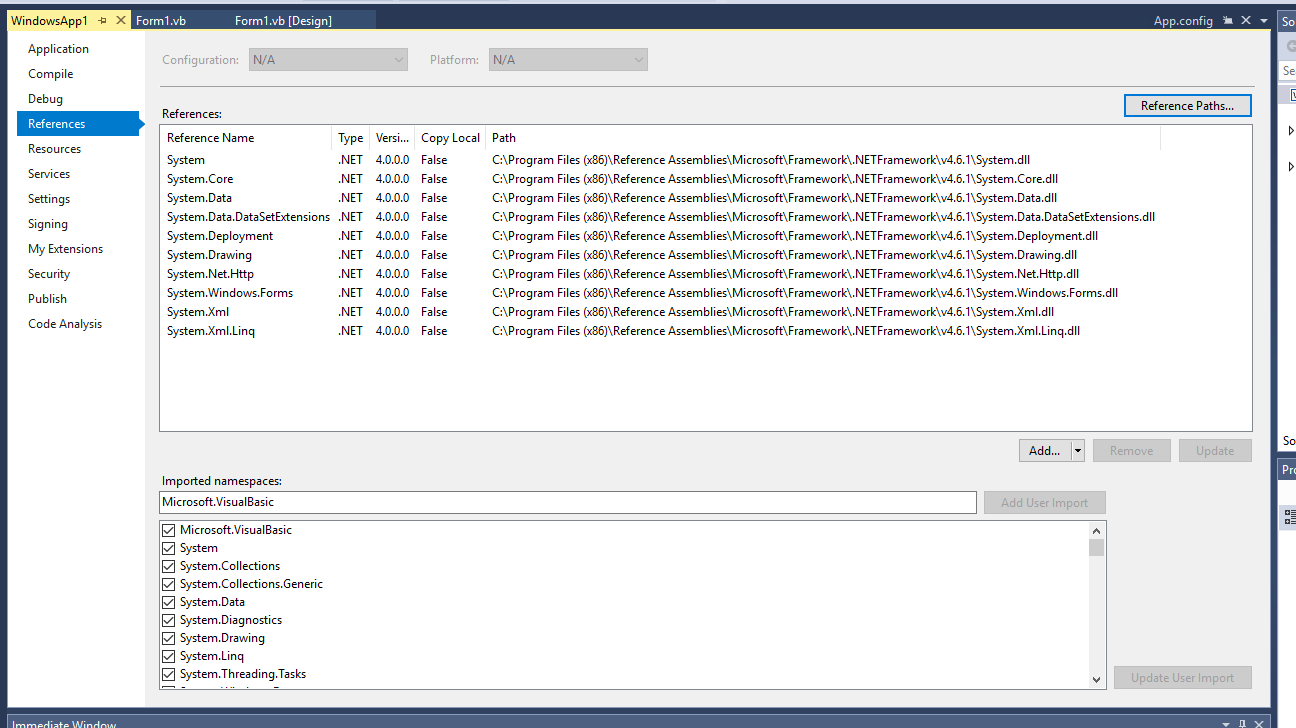
# Understanding Namespaces

VB.NET has organized its architecture around the idea of namespaces. These can essentially be thought of as C++ header files or Java import statements. The idea is that each set of related classes of functionality gets its own namespace.

To help introduce you to the richness of the namespaces, the following list identifies some of the key players that you will commonly use as you develop applications.

|  |  |
| --- | --- |
| *Namespace* | *Description* |
| System.Collections | Classes that describe a variety of collection-based data structures, such as arrays, lists, queues and stacks |
| System.Configuration | Classes that programs can use to access .NET framework configuration information |
| System.Data | Classes that support data input from sources such as ADO.NET |
| System.DirectoryServices | Classes that provide programs with an interface to the Active Directory structure which is used in the newer versions of Windows |
| System.Drawing | Classes that programs can use to access the new GDI+ graphics functionality available in .NET |
| System.Globalization | Classes the support international features such as date, time and currency formats |
| System.IO | Classes that support stream-based input and output operations |
| System.Messaging | Classes that programs can use to send and receive messages across various message queues |
| System.Net | Classes that programs can use to perform network related operations |
| System.Reflection | Classes that programs can use to query an object about that particular object's capabilities |
| System.Runtime | Classes that programs can use to interact with the low level .NET architectural components |
| System.Security | Classes that programs can use for encryption, access control and so forth |
| System.Text | Classes that programs can use to manipulate ASCII, Unicode and other text formats |
| System.Timers | Classes that provide timer support |
| System.Web | Classes that programs can use to interact with a browser |
| System.Windows.Forms | Classes that programs can use to create Windows-based forms |
| System.XML | Classes that support XML-based operations, such as reading and writing XML content |

Note that some of the necessary namespaces are automatically brought in when you start a project. You can see these by looking at the project’s properties (again right click the project name in the Solution Explorer window, select Properties and then the References tab). At the bottom is a list of the imported namespaces.



If you want to use a namespace, you can either add the entire namespace to a code file at the very top by using the Imports statement, as in:

'Here is a namespace that I want to import --

'it must be at the first line of a code file

Imports System.IO

Public Class frmMain

…

Now if we don’t want to import the whole namespace, we can explicitly define the class path to where an object lives in a namespace such as:

Dim myTextFile As System.IO.TextReader

But, if we had a ton of objects that we wanted to use out of the System.IO namespace, specifying that prefix every time would get pretty old – so it would be better to just use the Imports statement.