Chapter 6 – Language Miscellany and Console Applications

# Objectives

* Visual Basic’s True and False Values
* Miscellaneous Relational Related Operators
* Other Decision Related Constructs
* More on Subprograms Arguments
* Variable Scope (Local vs. Global and Public vs. Private)
* Static Storage
* Subprogram Overloading
* Revisiting Option Explicit
* Building Console Applications

# Visual Basic’s True and False Values

Not that you necessarily care, but False's internal value is 0 in VB.NET (like Java). Originally True's internal value was 1, but due to issues with compatibility for VB 6, it was returned back to the VB 6 value of –1, which is somewhat weird compared to other .NET programming languages. In fact, all other languages in .NET use False (0) and True (1) – as long as you use the intrinsic True and False values, you don't have to worry about anything; in fact, the proper values will even be communicated to other programming languages behind the scenes.

Dim blnValue As Boolean

blnValue = True

Debug.WriteLine(blnValue)

blnValue = False

Debug.WriteLine(blnValue)

blnValue = True

Debug.WriteLine(CInt(blnValue))

blnValue = False

Debug.WriteLine(CInt(blnValue))

When run, we get:

True

False

-1

0

# Miscellaneous Relational Related Operators

## Like

Associated along with the relational operators is the Like operator. This is used to test whether a string matches a simple pattern or not. Here are the various characters that can be used to make up the patterns:

|  |  |
| --- | --- |
| *Character* | *Meaning* |
| ? | Match any single character |
| \* | Match zero or more characters |
| # | Match any single digit (0-9) |
| […] | Match any character included in the brackets |
| [!…] | Match any character that is not included in the brackets |
| X-X | Specifies a range of Unicode characters |

Here is a sample snippet of code showing how the Like operator works:

Dim x As String

x = "muddy"

If (x Like "m?[!aeiou]dy") Then

Debug.WriteLine("True")

'Returns True

End If

This program will print the value True out. If you examine the Like pattern, we are saying it must match a string must start with a m, can have any second character, the third character must not be a vowel and the last two characters must be dy. The string "muddy" conforms to that pattern.

What about the string "moody"? The problem with moody is that the third character is a vowel, consequently the WriteLine statement will not print because the Like operator did not find a pattern match.

## Is

Compares whether two object variables are pointing to the same object instance in memory. Remember that when you declare a variable which is of some class type, the class’ constructor will typically create an instance of that object in memory and your reference variable will point to it.

The following code snippet illustrates this issue:

Dim Obj1 As New Form1() 'Create a new form1 instance in Obj1

Dim Obj2 As New Form1() 'Create another form1 instance in Obj2

Dim blnResult As Boolean

blnResult = Obj1 Is Obj2 'Returns False

Debug.WriteLine(blnResult)

'The preceding line returns false since Obj1 and Obj2 do not

'point to the same object in memory. It is true that they

'both point to the same type of object, but they point to two

'distinct instances of that object type.

Obj1 = Obj2

blnResult = Obj1 Is Obj2 'Returns True

Debug.WriteLine(blnResult)

'The preceding line returns true, because we made Obj1 point at

'whatever Obj2 is pointing at. Now, both variables are pointing

'at the same instance in memory, thus Is now returns True

This discussion also brings into play the idea of deep copying versus shallow copying. Let’s say that we have an object ObjectA, that when instantiated also creates and points to an ObjectB. Now if we simply make a copy of ObjectA, say ObjectA2, ObjectA2 will be pointing to ObjectB for the reason shown above. This is known as a shallow copy and we have to be careful because if we change the ObjectB that ObjectA2 is pointing at, we will be changing the ObjectB that ObjectA is pointing at, because they are the same object – this may not be at all what we intend to do.

A deep copy, on the other hand, would not only clone ObjectA to ObjectA2, it would also clone ObjectB to ObjectB2 and make ObjectA2 point to ObjectB2. In other words, we have copied the entire object hierarchy (hence, a deep copy). Now we can change ObjectB2 to our liking without affecting the ObjectB that ObjectA points to…

## TypeOf…Is

Tests to see if an object is of a certain data type. The data type can be either a simple data type like Integer or Boolean, or an object instantiation of some class type.

Dim F As New Form1()

'Notice that the next line asks the question “Is

'variable F of type Form1?”

If TypeOf F Is Form1 Then

Debug.WriteLine("True")

## End If

# Other Decision Related Constructs

Visual Basic also provides some other non-traditional decision-making constructs that have been in the language for a while. This section takes a brief look at each of them.

## IIf

IIf is short for "If and only if" – a simplified version of the If/Then/Else statement in which one of two possible values is returned.

Syntax of the IIf statement:

IIf(*condition*, *true\_value\_to\_return*, *false\_value\_to\_return*)

Example:

lblStatus.Text = IIf(grade >= 60, "Passed", "Failed")

If grade >= 60 then lblStatus.Text will contain the value "Passed", otherwise it will contain the value "Failed." You may recognize this as being VB’s form of the ternary operator that other programming languages have.

## Choose

This function is used to return a specified value from a list of items. The first item in the list has an index value of 1.

Syntax of the Choose statement:

##### Choose(index\_of\_item, item1, item2 … itemN).

Example:

Dim intTime As Integer

intTime = 2

MessageBox.Show(Choose(intTime, "Good Morning",

"Good Afternoon",

"Good Evening"))

The MessageBox would print "Good Afternoon" since the index value, intTime, contains a 2 – again note that Choose is not 0 based.

# More on Subprogram Arguments

VB.NET technically requires a ByVal or ByRef to be explicitly stated in front of each subprogram parameter. If you type variable names into a subprogram heading without a passing type specified, VB uses ByVal as the default. Note that the IDE doesn’t type this in – it just assumes the ByVal. This is a bad idea on Microsoft’s part – I tend to always preface things with ByVal or ByRef so there is no confusion!

In other programming languages, you have the ability to create optional and named arguments in your subprograms. Visual Basic provides these options as well, so we will take a very quick look at how to implement these items.

Here’s an example of an Optional argument:

Sub MySub(ByVal intA As Integer, Optional ByVal blnB As Boolean = True)

The first parameter is a required parameter that must be provided by the user. The second parameter on the other hand, is optional. If the user supplies a value, that's great; if not, then the subroutine will automatically use the value specified default value of True.

This subroutine could be called in the following manner:

MySub(10) 'Default value is used on second parameter

MySub(10, False) 'Here we supplied both arguments

There are just a couple of simple rules that we have to obey when working with Optional arguments: (1) We must always provide a default value for each Optional argument. (2) In addition, all non-optional parameters must be declared to the left of the first Optional argument.

The second special type of parameter notation is that of named parameters. Notice that the next subroutine declaration has every parameter listed as an optional parameter.

Sub MySub(Optional A As Integer = 10, Optional B As Integer = 20, \_

Optional C As Integer = 30)

There are several different ways that this routine could be called:

MySub(0, 0, 10) 'No defaults used

MySub(, , 10) 'Use first two as defaults

MySub(B:=10)

'We provide value for parameter 2, use

'defaults for the other two. Hence

'we named the parameter we were

'going to provide a value for

# Variable Scope

Another issue to throw in this whole subprogram stew deals with where and when a variable can be seen. This is known as the scope of a variable. Let's start out with an easy example of scope: any variable that is created within a subprogram gets destroyed when that subprogram ends.

For example:

Sub MySub()

Dim X As Integer

X = 5

End Sub

X will be created when MySub starts up. X will then continue to exist until MySub ends, at which point X will be destroyed. This X is known as a local variable. We can extend the visibility of a variable through passing parameters. Look at the following example:

Sub MySub2(ByVal Y As Integer)

MessageBox.Show(Y) 'This is really X from MySub

End Sub

Sub MySub()

Dim X As Integer 'Create X here

X = 5

MySub2(X) 'X will temporarily live "outside" in MySub2

End Sub 'X still gets destroyed here

I still have X as a local variable created in MySub, but I can extend its visibility by passing the variable into other subroutines, in this case MySub2.

What do you do if you need variables that are available over a longer duration of time though? The answer: you can create variables of a more “global” nature. These are variables that are created when the item you house them in begins running and have a lifetime until that item is destroyed. The item could be a form, a class or a module – therefore the lifetime of “global” variables created within an entity are dependent on how your application executes. The rule of thumb is to minimize your use of global variables.

To create a variable that has a more “global” scope, define the variable outside of any subprogram code block and preface it with the keyword Public, indicating that you want this to be a public (global) variable. For example, if you want to create a form level global variable, I recommend placing it at the top of your form as the first line after the Class name line. The following example shows how to do this:

Public Class Form1

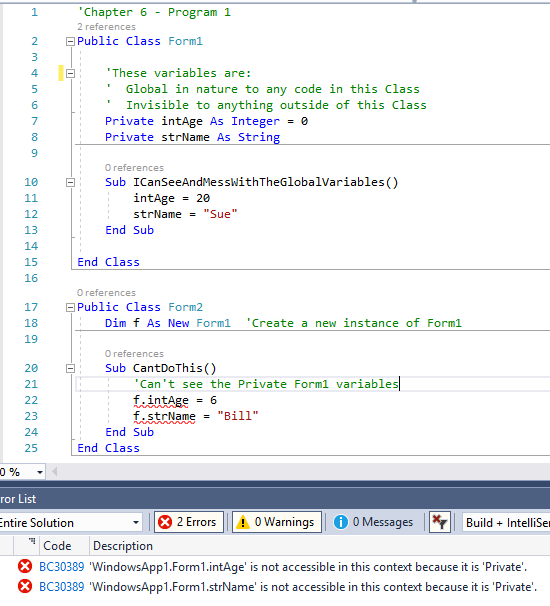
Dim intAge As Integer = 0

Dim strName As String

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

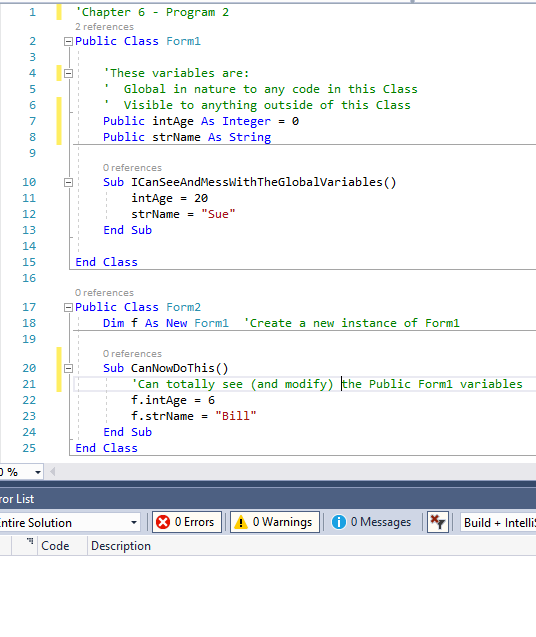
I have just created two global variables that are visible everywhere within this form. You really want to keep the number of global variables to a minimum though. These variables will get created when the form is created and will live as long as the form exists. It shouldn’t surprise you that you could also create global variables in a code module and they would last the duration that that module runs – after all a form, which is a class, is usually implemented in its own standalone code module.

We now want to consider the difference between Private and Public scope. Technically by using the Dim keyword above, the global variable that I created were of Private scope. This means that anything inside the code body of the form in which the variables were created can see and modify them. Anything outside of the form has no idea they exist though. Try this and note that you get syntax errors:



By declaring a variable with the keyword Private (which is better than Dim since it is makes the scope level that must more explicit), it confines the variable’s scope level to the module in which it is declared. Therefore, when we want a variable accessible to the majority of a module or a form, but not outside of it, we should use the Private scope identifier.

On the other hand, if we use the keyword Public in declaring our variables, then we are extending the scope outside of the entity in which it is declared. Take a look at this code:



Notice there are no syntax errors when the Form2 instance attempts to access the Form1 global variables now. Since global variables are visible everywhere, they can be modified (accidentally) at any time, in any piece of code. By using local variables and passing techniques, you have control as a programmer to determine which subprograms can modify what data.

Another problem with using global variables in subprograms is the fact that your subprograms become dependent upon certain global variables existing in order for that subprogram to execute properly. If I want to try to reuse one of my global variable code modules in another program, I’d have to ensure the same global variables exist in the second program as I had in the first program. How effective would the Sqrt() function be if I had to always create a global variable named HMPzValue1E and put the value that I wanted to send to the square root function in that variable before I called the function? The Sqrt() function certainly wouldn’t be very portable or reusable…

So we have looked at extending scope by using passing mechanisms and declaring variables with module scope level. Now what if you really want to create a “true” global variable? In other words, everybody can see it and manipulate it across the entire project (every form, module and class)? The answer lies in the Application Events section again. Since Application Events allows us to control the startup and shutdown of our programs, it should make sense that declaring any public variables here will be seen everywhere. I started this example out by clicking View Application Events in the Project’s properties and I added a global variable inside of Class MyApplication:

Namespace My

'Chapter 6 - Program 3

Partial Friend Class MyApplication

'This will be a global variable that can be seen across the application!

Public gintCounter As Integer = 25

End Class

End Namespace

This variable can be seen anywhere which is evidenced by the following code in my Form1 form. And, since the variable is public I can print it as well as change it!

Public Class Form1

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

'Accessing Global Variable:

Debug.WriteLine("gintCounter in Form1 value:" &

My.Application.gintCounter)

'Modifying Global Variable:

My.Application.gintCounter = 30

'Calling a subroutine in a code module:

DoSub()

End Sub

End Class

Notice that Form1 is going to call a subroutine called DoSub. I put that into a code module called Module1 and again I can change and print the global variable. Note that no passing was involved here!

Module Module1

Sub DoSub()

Debug.WriteLine("gintCounter in Sub DoSub value on entry:" &

My.Application.gintCounter)

My.Application.gintCounter = 35

Debug.WriteLine("gintCounter in Sub DoSub value on exit:" &

My.Application.gintCounter)

End Sub

End Module

Three completely separate code entities and yet all could see, share and modify the variable – it truly is global in nature. Here’s the output when run:

gintCounter in Form1 value:25

gintCounter in Sub DoSub value on entry:30

gintCounter in Sub DoSub value on exit:35

# Static Storage

You may have heard of static storage in Java – the idea where a variable is created and initialized only once during the program’s execution, even if the routine is called over and over again. Variables in functions and subroutines in Visual Basic may be prefaced with the Static keyword which makes those variables static in nature. This means that storage is allocated and initialized only once when the subprogram is initially executed.

In other words, static local variables are created once and kept intact for the duration of the program. Even if you call the same subprogram multiple times, the variable will not be recreated, nor will the value of the variable be reinitialized. This is in stark contrast to how local variables are usually created. Here’s some code that shows how this works:

'Chapter 6 - Program 4

Public Class Form1

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

For intI As Integer = 1 To 5

DoIt()

Next

End Sub

Sub DoIt()

'Notice it looks like we are creating a normal scoped variable

'when we start DoIt. intX gets created and initialed upon entry, right?

Static intX As Integer = 0

'Add 5 to the value that's in intX

intX += 5

'Shouldn't this never exceed 5, since we create & destroy intX here?

Debug.WriteLine("Value in intX is: " & intX)

End Sub

End Class

What does the output tell us though?

Value in intX is: 5

Value in intX is: 10

Value in intX is: 15

Value in intX is: 20

Value in intX is: 25

The static keyword created intX once and set it initially to 0. After that, the value of the variable was preserved between executions and not reinitialized upon each entry into the DoIt subprogram…

In Java, static has an additional meaning: it can be applied to a method. A static method means that there can only be one instance of that method created and all classes that use that method would point to the same instance. Probably the most obvious example is the main() function in Java programs. It should be pretty clear that we would only ever want one main(), so making it a static method solves that problem. We will visit this concept later when we get to proper object-orientation in Visual Basic; for now the keyword that we would use to create the equivalent of a Java static method in VB is *Shared* – just in case you care.

# Subprogram Overloading

VB.NET has the ability for functions and subroutines to be overloaded, just as in languages like Java. Each overloaded subprogram has the same name and purpose, just different parameter types that are sent to it.

When you place a call to an overloaded subprogram, VB examines the parameters types in your actual arguments and then will call the appropriate version of your overloaded subprogram to handle it.

Here's an example program that calculates the Squares of Integers or Doubles:

'Chapter 6 - Program 5

Public Class Form1

'Notice that the same subroutine is written twice – the only

'difference between the two versions is the parameter type of

'X passed in

Sub SquareIt(ByVal X As Integer)

MessageBox.Show("Integer version called: " & X \* X)

End Sub

Sub SquareIt(ByVal X As Double)

MessageBox.Show("Double version called: " & X \* X)

End Sub

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

SquareIt(5) 'Calls Integer Version

SquareIt(2.5) 'Calls Double Version

SquareIt(3.0) 'Calls Double Version -- there's a decimal in it!

End Sub

End Class

Overloaded functions may also have different return types as long as the parameter signatures between the functions are different. Think about, if there wasn't a difference in the signatures, how would you determine which version to call?

'Chapter 6 - Program 6

Public Class Form1

Function SquareIt(ByVal X As Integer) As Integer

MessageBox.Show("Integer Version")

Return (X \* X)

End Function

Function SquareIt(ByVal X As Double) As Double

MessageBox.Show("Double Version")

Return (X \* X)

End Function

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

MessageBox.Show(SquareIt(5)) 'Calls Integer Version

MessageBox.Show(SquareIt(2.5)) 'Calls Double Version

MessageBox.Show(SquareIt(3.0)) 'Calls Double Version -- the ToString

'method in .Show knows it can print the

'result as a simple integer

End Sub

End Class

# Revisiting Option Strict

We have already discussed the need for Option Strict, which worries about data type conversions. Option Strict is set to Off by default. If this option is turned On, it forces the programmer to perform all data type conversions manually, including when going from a narrower data type to a larger data type (this is called widening). An example of this would be converting a byte to an integer. Obviously, any value that is valid for a byte should fit inside of an integer. VB will guess right most of the time when doing this type of conversion, therefore you may want to leave the default Option Strict value alone unless you want to be responsible for ensuring that every conversion is done correctly and by you.

Summary of data conversion issues:

* Widening example -- assigning an Integer value to a Double variable
* Widening always works. (Every Integer is a Double.)
* No conversion function needed
* Narrowing example -- assigning a Double value to an Integer variable
* Narrowing might not work. (Not every Double is an Integer.)
* Narrowing requires CInt or setting Option Strict to Off…

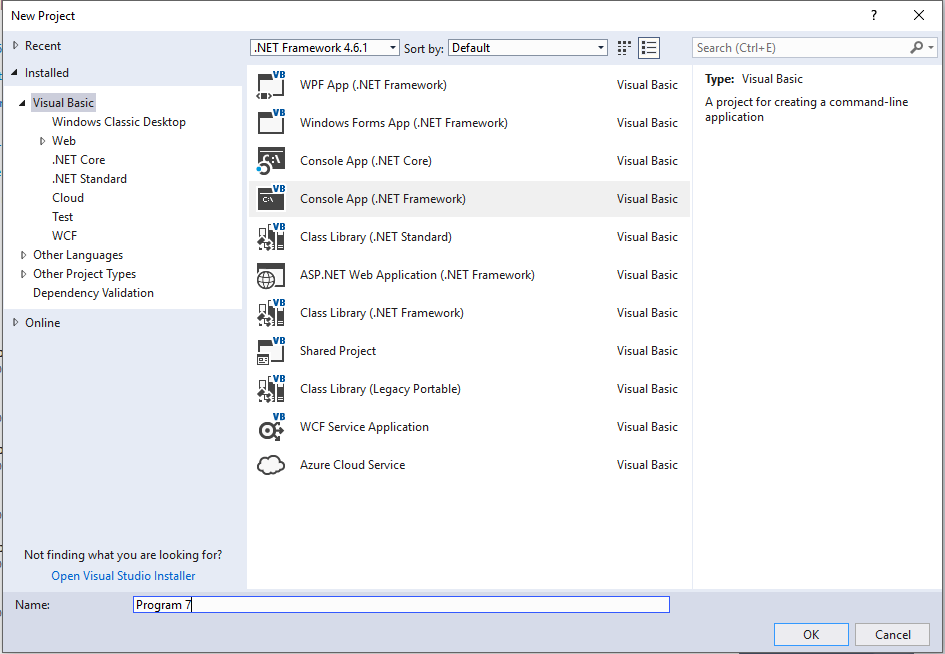
Know that Option Strict affects more than just data type conversions. It also affects the implicit typing that results in an Object type and, in addition, late binding. So basically, Option Strict On really means that you have to explicitly define types for all of your data! There are some advantages to doing this:

* IntelliSense can show up all of the properties and members of data types as you type.
* The compiler can perform type checking. Type checking helps you find statements that can fail at run time because of type conversion errors. It also identifies calls to methods on objects that do not support those methods.
* It speeds up the execution of code. One reason for this is that if you do not specify a data type for a programming element, the Visual Basic compiler assigns it the Object type. Any derived object type like Form or MessageBox can be held in the Object type since Object is the granddaddy of all objects! Compiled code might have to convert back and forth between Object and other data types, which would slow things down.

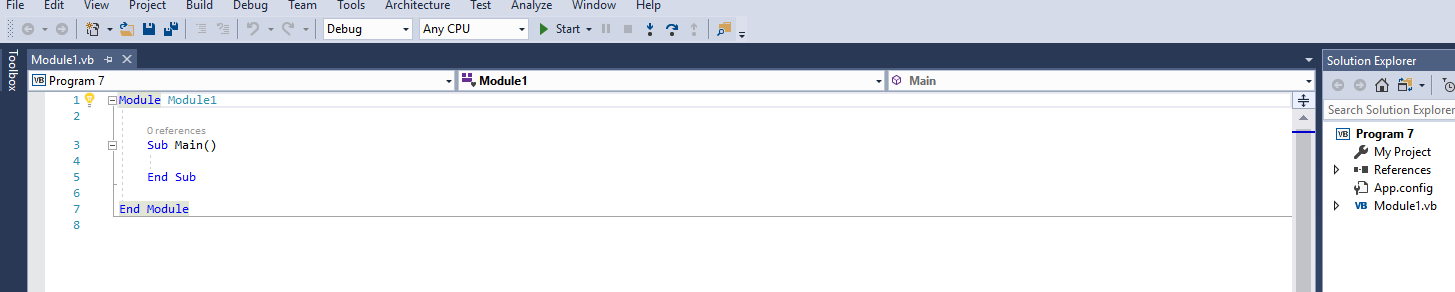
# Building Console Applications

While the majority of our focus in this class is on Windows Forms Applications, many times we need a simple, quick and dirty program to do something once. In other instances, we want to build a Windows service or background program that needs to run but does not need a GUI. These are the times when it is necessary to build a console application.

We start this out by selecting a Console Application from the New Project menu:



And we’re dropped off in a pretty simple place – Sub Main…just like in good old CS 116 command line Java!



We can run the program, just like a Forms application. The output will go to a console output window and nothing else will happen since we haven’t told it to do anything yet! Try it – you’ll see the screen flash as a console (command) window appears and disappears. Let’s write a little code:

'Chapter 6 - Program 7

Module Module1

Sub Main()

Dim intHours As Integer

Dim sngWage As Single

Dim sngPayDue As Single

'Get the input

Console.Write("Please enter the number of hours worked: ")

intHours = Console.ReadLine()

Console.Write("Please enter the hourly pay rate: ")

sngWage = Console.ReadLine()

'Do the calculations

If intHours <= 40 Then

sngPayDue = intHours \* sngWage

Else

sngPayDue = ((intHours - 40) \* 1.5 \* sngWage) + (sngWage \* 40)

End If

'Generate output

Console.Beep()

Console.WriteLine(StrDup(30, "\*"))

Console.WriteLine(String.Format("Hours: {0:N2}", intHours))

Console.WriteLine(String.Format("Wage: {0:C}", sngWage))

Console.WriteLine(String.Format("Total Earnings: {0:C}", sngPayDue))

Console.WriteLine(StrDup(30, "\*"))

Console.Beep()

'Pause so the user can read the screen

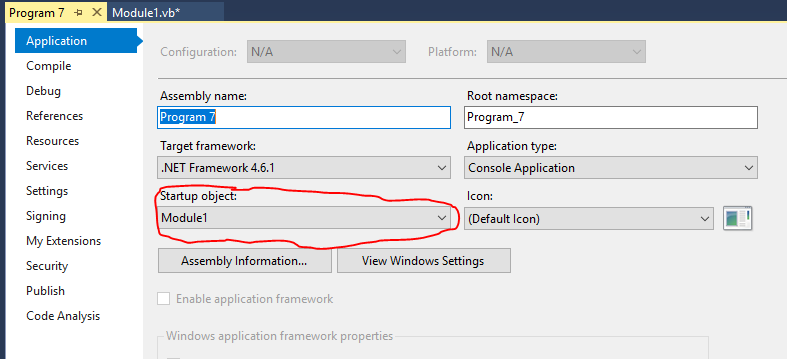
Console.WriteLine("Press Enter to exit the program...")

Console.ReadLine()

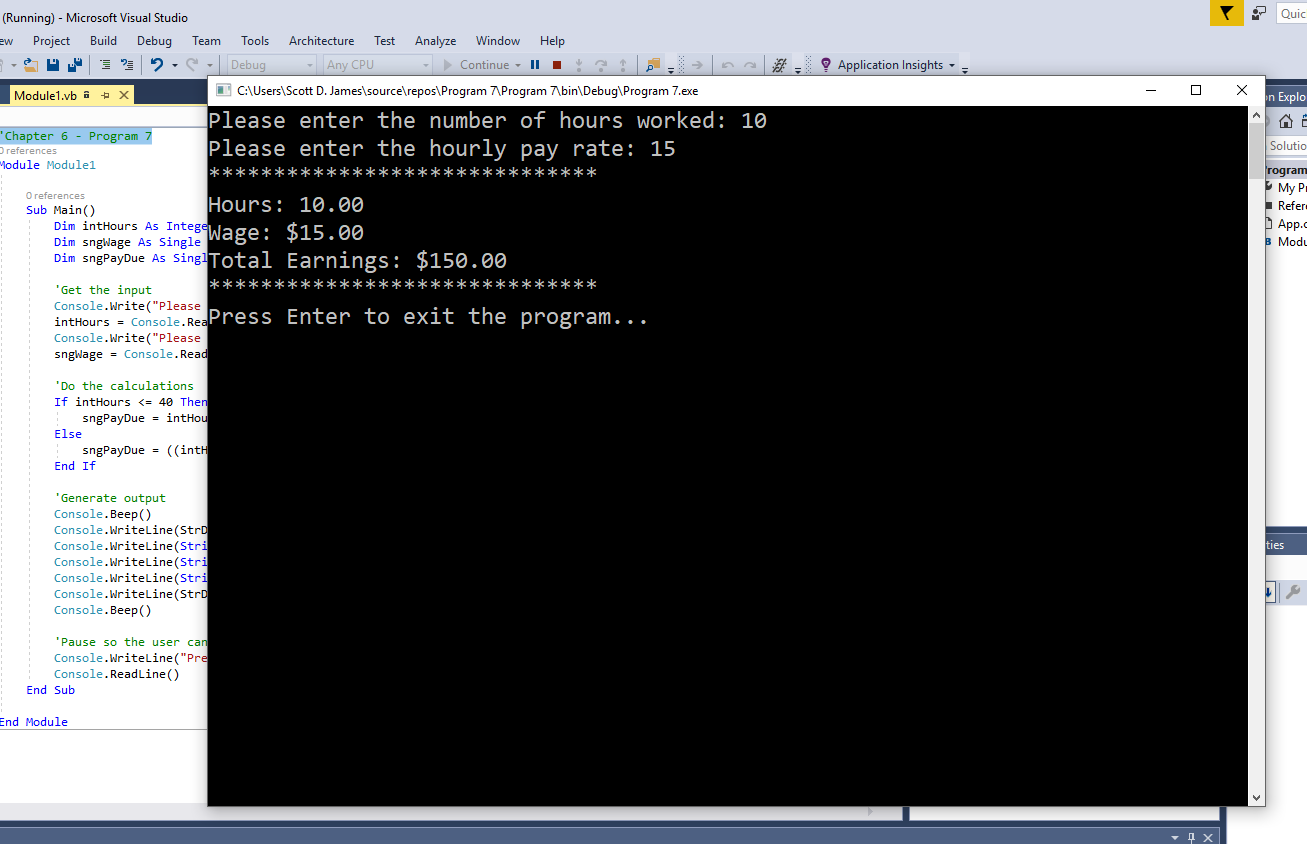
End Sub

End Module

Notice that the code goes into a Module and everything starts with Sub Main. If we were to check the Project Properties, we’d see the Module is the startup object – we can pick Sub Main if we want, but things automatically default there anyway:



Running the application gives us a console window:



Now for some reason, people seem to forget about using subroutines and functions when they switch back over to console applications…don’t do that! All you have technically changed is how the input and output work with the program. You went from a GUI to command line, but that has nothing to do with the programmatic logic of modules, classes, subroutines and functions.

The majority of the notes that we’ve looked as so far (and will look at in the future) can be used in console applications as long as it doesn’t involve Windows Forms or Forms’ Controls. About the only exception is that the graphical InputBox and MsgBox keywords can be used. Note: MessageBox.Show cannot be used, but the old MsgBox keyword can – nice inconsistency Microsoft! Here’s the code for the “windowed” version of the same application:

'Chapter 6 - Program 8

Module Module1

Sub Main()

Dim intHours As Integer

Dim sngWage As Single

Dim sngPayDue As Single

Dim strOutput As String

'Get the input

intHours = InputBox("Please enter the number of hours worked:")

sngWage = InputBox("Please enter the hourly pay rate:")

'Do the calculations

If intHours <= 40 Then

sngPayDue = intHours \* sngWage

Else

sngPayDue = ((intHours - 40) \* 1.5 \* sngWage) + (sngWage \* 40)

End If

'Generate output

Console.Beep()

strOutput &= StrDup(30, "\*") & vbCrLf

strOutput &= String.Format("Hours: {0:N2}", intHours) & vbCrLf

strOutput &= String.Format("Wage: {0:C}", sngWage) & vbCrLf

strOutput &= String.Format("Total Earnings: {0:C}", sngPayDue) & vbCrLf

strOutput &= StrDup(30, "\*") & vbCrLf

MsgBox(strOutput)

Console.Beep()

'Pause so the user can read the screen

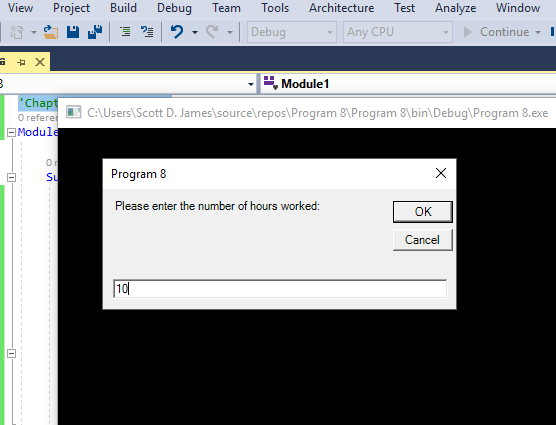
Console.WriteLine("Press Enter to exit the program...")

Console.ReadLine()

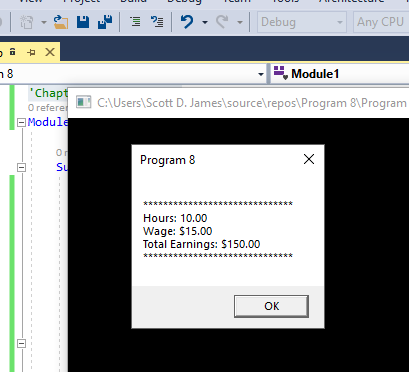
End Sub

End Module

This is the first graphical input prompt:



Here’s the graphical output:



Let’s take a look at the more interesting Console Properties:

|  |  |
| --- | --- |
| *Name* | *Description* |
| [BackgroundColor](http://msdn.microsoft.com/en-us/library/system.console.backgroundcolor.aspx) | Gets or sets the background color of the console. |
| [BufferHeight](http://msdn.microsoft.com/en-us/library/system.console.bufferheight.aspx) | Gets or sets the height of the buffer area. |
| [BufferWidth](http://msdn.microsoft.com/en-us/library/system.console.bufferwidth.aspx) | Gets or sets the width of the buffer area. |
| [CapsLock](http://msdn.microsoft.com/en-us/library/system.console.capslock.aspx) | Gets a value indicating whether the CAPS LOCK keyboard toggle is turned on or turned off. |
| [CursorLeft](http://msdn.microsoft.com/en-us/library/system.console.cursorleft.aspx) | Gets or sets the column position of the cursor within the buffer area. |
| [CursorSize](http://msdn.microsoft.com/en-us/library/system.console.cursorsize.aspx) | Gets or sets the height of the cursor within a character cell. |
| [CursorTop](http://msdn.microsoft.com/en-us/library/system.console.cursortop.aspx) | Gets or sets the row position of the cursor within the buffer area. |
| [CursorVisible](http://msdn.microsoft.com/en-us/library/system.console.cursorvisible.aspx) | Gets or sets a value indicating whether the cursor is visible. |
| [Error](http://msdn.microsoft.com/en-us/library/system.console.error.aspx) | Gets the standard error output stream. |
| [ForegroundColor](http://msdn.microsoft.com/en-us/library/system.console.foregroundcolor.aspx) | Gets or sets the foreground color of the console. |
| [In](http://msdn.microsoft.com/en-us/library/system.console.in.aspx) | Gets the standard input stream. |
| [InputEncoding](http://msdn.microsoft.com/en-us/library/system.console.inputencoding.aspx) | Gets or sets the encoding the console uses to read input. |
| [IsErrorRedirected](http://msdn.microsoft.com/en-us/library/system.console.iserrorredirected.aspx) | Gets a value that indicates whether the error output stream has been redirected from the standard error stream. |
| [IsInputRedirected](http://msdn.microsoft.com/en-us/library/system.console.isinputredirected.aspx) | Gets a value that indicates whether input has been redirected from the standard input stream. |
| [IsOutputRedirected](http://msdn.microsoft.com/en-us/library/system.console.isoutputredirected.aspx) | Gets a value that indicates whether output has been redirected from the standard output stream. |
| [KeyAvailable](http://msdn.microsoft.com/en-us/library/system.console.keyavailable.aspx) | Gets a value indicating whether a key press is available in the input stream. |
| [LargestWindowHeight](http://msdn.microsoft.com/en-us/library/system.console.largestwindowheight.aspx) | Gets the largest possible number of console window rows, based on the current font and screen resolution. |
| [LargestWindowWidth](http://msdn.microsoft.com/en-us/library/system.console.largestwindowwidth.aspx) | Gets the largest possible number of console window columns, based on the current font and screen resolution. |
| [NumberLock](http://msdn.microsoft.com/en-us/library/system.console.numberlock.aspx) | Gets a value indicating whether the NUM LOCK keyboard toggle is turned on or turned off. |
| [Out](http://msdn.microsoft.com/en-us/library/system.console.out.aspx) | Gets the standard output stream. |
| [OutputEncoding](http://msdn.microsoft.com/en-us/library/system.console.outputencoding.aspx) | Gets or sets the encoding the console uses to write output. |
| [Title](http://msdn.microsoft.com/en-us/library/system.console.title.aspx) | Gets or sets the title to display in the console title bar. |
| [TreatControlCAsInput](http://msdn.microsoft.com/en-us/library/system.console.treatcontrolcasinput.aspx) | Gets or sets a value indicating whether the combination of the [Control](http://msdn.microsoft.com/en-us/library/system.consolemodifiers.aspx) modifier key and [C](http://msdn.microsoft.com/en-us/library/system.consolekey.aspx) console key (Ctrl+C) is treated as ordinary input or as an interruption that is handled by the operating system. |
| [WindowHeight](http://msdn.microsoft.com/en-us/library/system.console.windowheight.aspx) | Gets or sets the height of the console window area. |
| [WindowLeft](http://msdn.microsoft.com/en-us/library/system.console.windowleft.aspx) | Gets or sets the leftmost position of the console window area relative to the screen buffer. |
| [WindowTop](http://msdn.microsoft.com/en-us/library/system.console.windowtop.aspx) | Gets or sets the top position of the console window area relative to the screen buffer. |
| [WindowWidth](http://msdn.microsoft.com/en-us/library/system.console.windowwidth.aspx) | Gets or sets the width of the console window. |

Here are the various Console Methods that are commonly used:

|  |  |
| --- | --- |
| *Name* | *Description* |
| [Beep](http://msdn.microsoft.com/en-us/library/8hftfeyw.aspx) | Plays the sound of a beep through the console speaker. |
| [Beep(Int32, Int32)](http://msdn.microsoft.com/en-us/library/4fe3hdb1.aspx) | Plays the sound of a beep of a specified frequency and duration through the console speaker. |
| [Clear](http://msdn.microsoft.com/en-us/library/system.console.clear.aspx) | Clears the console buffer and corresponding console window of display information. |
| [OpenStandardError](http://msdn.microsoft.com/en-us/library/5y49kb86.aspx) | Acquires the standard error stream. |
| [OpenStandardInput](http://msdn.microsoft.com/en-us/library/tx55zca2.aspx) | Acquires the standard input stream. |
| [OpenStandardOutput](http://msdn.microsoft.com/en-us/library/16f09842.aspx) | Acquires the standard output stream. |
| [Read](http://msdn.microsoft.com/en-us/library/system.console.read.aspx) | Reads the next character from the standard input stream. |
| [ReadKey](http://msdn.microsoft.com/en-us/library/471w8d85.aspx) | Obtains the next character or function key pressed by the user. The pressed key is displayed in the console window. |
| [ReadLine](http://msdn.microsoft.com/en-us/library/system.console.readline.aspx) | Reads the next line of characters from the standard input stream. |
| [ResetColor](http://msdn.microsoft.com/en-us/library/system.console.resetcolor.aspx) | Sets the foreground and background console colors to their defaults. |
| [SetBufferSize](http://msdn.microsoft.com/en-us/library/system.console.setbuffersize.aspx) | Sets the height and width of the screen buffer area to the specified values. |
| [SetCursorPosition](http://msdn.microsoft.com/en-us/library/system.console.setcursorposition.aspx) | Sets the position of the cursor. |
| [SetError](http://msdn.microsoft.com/en-us/library/system.console.seterror.aspx) | Sets the [Error](http://msdn.microsoft.com/en-us/library/system.console.error.aspx) property to the specified [TextWriter](http://msdn.microsoft.com/en-us/library/system.io.textwriter.aspx) object. |
| [SetIn](http://msdn.microsoft.com/en-us/library/system.console.setin.aspx) | Sets the [In](http://msdn.microsoft.com/en-us/library/system.console.in.aspx) property to the specified [TextReader](http://msdn.microsoft.com/en-us/library/system.io.textreader.aspx) object. |
| [SetOut](http://msdn.microsoft.com/en-us/library/system.console.setout.aspx) | Sets the [Out](http://msdn.microsoft.com/en-us/library/system.console.out.aspx) property to the specified [TextWriter](http://msdn.microsoft.com/en-us/library/system.io.textwriter.aspx) object. |
| [SetWindowPosition](http://msdn.microsoft.com/en-us/library/system.console.setwindowposition.aspx) | Sets the position of the console window relative to the screen buffer. |
| [SetWindowSize](http://msdn.microsoft.com/en-us/library/system.console.setwindowsize.aspx) | Sets the height and width of the console window to the specified values. |
| Write(value) | Writes the specified value to the standard output stream. |
| [WriteLine](http://msdn.microsoft.com/en-us/library/zdf6yhx5.aspx)(value) | Writes the current value and line terminator to the standard output stream. |

Finally, these are the Console Events we care about:

|  |  |
| --- | --- |
| *Name* | *Description* |
| [CancelKeyPress](http://msdn.microsoft.com/en-us/library/system.console.cancelkeypress.aspx) | Occurs when the [Control](http://msdn.microsoft.com/en-us/library/system.consolemodifiers.aspx) modifier key (Ctrl) and either the [ConsoleKey.C](http://msdn.microsoft.com/en-us/library/system.consolekey.aspx) console key (C) or the Break key are pressed simultaneously (Ctrl+C or Ctrl+Break). |