Chapter 3 – Variables and Operations

Numbers

We have already seen a little bit of working with numbers – for example, setting the size or position of a window. When we put a numeric value directly into the program, these are called numeric literals.

VB.NET allows us to perform standard arithmetic operations:

<table>
<thead>
<tr>
<th>Arithmetic Operator</th>
<th>VB.NET Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
</tr>
</tbody>
</table>
| Floating Point Division  | /             | result is a floating point number
| Integer Division         | \             | truncates anything after decimal point
| Exponent                 | ^             |
| Modulus                  | mod           |

One way to show data on the screen is to display it in a list box. If there is a control named lstBox added to your project, and n is a number, then the instruction:

```
lstBox.Items.Add(n)
```

will display the number n as the last item in the list box. Add is called a method, sometimes referred to as a subprogram, procedure, or function. The code behind this method has been written by the developers of VB.NET for you to use.

To erase all items in the list box, use the method Clear:

```
lstBox.Items.Clear()
```

Here are some examples of arithmetic operations and storing the result in a list box:

```
lstBox.Items.Add(3 + 2)
lstBox.Items.Add(3 - 2)
lstBox.Items.Add(5 * 2 * 10)
lstBox.Items.Add(14 mod 5)
lstBox.Items.Add(10 / 2)
lstBox.Items.Add(11 / 2)
lstBox.Items.Add(1 / 2)
lstBox.Items.Add(2 ^ 3)
lstBox.Items.Add((2^3)*3.1)
```
The results are:

5
1
100
4
5
5.5
5
0.5
0
8
24.8

Note that VB.NET will convert numbers with a fractional result to a new format that stores the fractional part. Extremely large numbers will be displayed in scientific notation, where the letter E refers to an exponent of $10^E$:

```vbnet
ListBox.Items.Add(2^50)
```

outputs: $1.1259E+15$

Another often more convenient way to output variables is to print them to the Console screen. Use either of the following methods to output to the console window:

```vbnet
Console.WriteLine(n)  - outputs n with a carriage return and newline
Console.Write(n)  - outputs n with no carriage return and newline
```

The results will show up in the “Debug” window of VB.NET if you are running in Debug mode, or in a DOS window if the program was invoked from DOS directly.

**Variables**

In math problems quantities are referred to by names. For example, in physics, there is the well known equation:

\[
\text{Force} = \text{Mass} \times \text{Acceleration}
\]

By substituting two known quantities we can solve for the third. When we refer to quantities or values with a name, these are called **variables**. Variables must begin with a letter and may contain numbers or underscores but not other characters. This includes spaces. Names are not case sensitive. Although you are free to pick a wide range of whatever names you like for variables, try to pick names that are representative of what the variable means. For example, a name like `NumberOfStudents` is better than `X` (assuming the variable will store a number of students).

(In class – do examples of legal and illegal variables)
To use variables we must tell VB.NET what **data type** our variables should be. We do this using the **Dim** statement, which “Dimensions” a storage location for us using the format:

```
Dim varName as DataType
```

The Dim statement causes the computer to set aside a location in memory with the name varName. **DataType** can take on many different types, such as:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Format for integral data</td>
<td>-2,147,483,648 through 2,147,483,647</td>
</tr>
<tr>
<td>Short</td>
<td>Format for smaller values of integers</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>Double</td>
<td>Format for floating point data</td>
<td>-1.8 * 10^308 to 1.8 * 10^308</td>
</tr>
<tr>
<td>Single</td>
<td>Format for smaller floating point data</td>
<td>-3.4 * 10^38 through 3.4 * 10^38</td>
</tr>
<tr>
<td>Decimal</td>
<td>Format for high precision (28 digits past decimal)</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>Textual data, put in double quotes, like &quot; &quot;</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>Stores dates, e.g. #5/21/2002#</td>
<td></td>
</tr>
</tbody>
</table>

If varName is a numeric variable, the Dim statement also places the number zero in that memory location. (We say that zero is the initial value or default value of the variable.) Strings are set to blank text.

If double has a larger data range than integer, and can store floating point numbers, you might wonder why we don’t just declare everything as a double. We could do this, but it would be wasteful – the double format takes up more space than an integer. Also it is slower to perform arithmetic operations on a number stored as double than it is on a number stored as integer. The integer data type is better to use if that is all your application needs. If you try to store a value that is too large for a data type, you will get an **overflow** error. We’ll see ways to prevent this later.

To assign or copy a value into a variable, use the = or assignment operator:

```
myVar = newValue
```

We can also assign an initial value when we declare a variable:

```
Dim myVar as Integer = 10
```
Here are some examples using numeric variables:

```vbnet
Dim dblVal as Double
Dim intVal as Integer

dblVal = 5 * 2 * 10
intVal = 5 * 2 * 10
Console.WriteLine(dblVal)
Console.WriteLine(intVal)

dblVal = 11 / 2
intVal = 11 / 2
Console.WriteLine(dblVal)
Console.WriteLine(intVal)

dblVal = 1 / 2
intVal = 1 / 2
Console.WriteLine(dblVal)
Console.WriteLine(intVal)
```

Output:

```
100
100
5.5
6
0.5
0
```

VB.NET will round floating point values up or down when converted to an integer (although 0.5 seems to be an exception).

A common operation is to increment the value of a variable. One way to do this is via:

```
intVal = intVal + 1
```

This is common enough that there are shortcuts:

```
x = x + y  <->  x += y
x = x * y   <->  x *= y
x = x - y   <->  x -= y
x = x / y   <->  x /= y
```

**Naming Conventions**

In the previous example I prefaced the variables by “int” and “dbl”. This is not necessary, but is a convention that indicates the data type of the variable just by looking at the name. The following chart illustrates some typical data type prefixes:
### Data Type

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>bln or b</td>
</tr>
<tr>
<td>Date</td>
<td>dte</td>
</tr>
<tr>
<td>Decimal</td>
<td>dec</td>
</tr>
<tr>
<td>Double</td>
<td>dbl</td>
</tr>
<tr>
<td>Integer</td>
<td>int or i</td>
</tr>
<tr>
<td>Long</td>
<td>lng</td>
</tr>
<tr>
<td>Short</td>
<td>sht</td>
</tr>
<tr>
<td>Single</td>
<td>sng</td>
</tr>
<tr>
<td>String</td>
<td>str or s</td>
</tr>
</tbody>
</table>

### Precedence Rules

The precedence rules of arithmetic apply to arithmetic expressions in a program. That is, the order of execution of an expression that contains more than one operation is determined by the precedence rules of arithmetic. These rules state that:

1. parentheses have the highest precedence
2. exponent has the next highest precedence
3. multiplication, division, and modulus have the next highest precedence
4. addition and subtraction have the lowest precedence.

Because parentheses have the highest precedence, they can be used to change the order in which operations are executed. When operators have the same precedence, order is left to right.

**Examples:**

<table>
<thead>
<tr>
<th>Dim x As Integer</th>
<th>Value stored in X</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1 + 2 + 6 / 6</td>
<td>4</td>
</tr>
<tr>
<td>x = (1 + 2 + 3) / 6</td>
<td>1</td>
</tr>
<tr>
<td>x = 2 * 3 + 4 * 5</td>
<td>26</td>
</tr>
<tr>
<td>x = 2 / 4 * 4</td>
<td>2</td>
</tr>
<tr>
<td>x = 2 / (4 * 4)</td>
<td>0</td>
</tr>
<tr>
<td>x = 10 Mod 2 + 1</td>
<td>1</td>
</tr>
</tbody>
</table>

In general it is a good idea to use parenthesis if there is any possibility of confusion. There are a number of built-in math functions that are useful with numbers. Here are just a few:
Math.Sqrt(number) returns the square root of number

Ex:
    Console.WriteLine(Math.Sqrt(9))  ' Displays 3
    Dim d as Double
    d = Math.Sqrt(25)
    Console.WriteLine(d)  ' Displays 5
    Console.WriteLine(Math.Sqrt(-1))  ' Displays NaN

Math.Round(number) returns the number rounded up/down

Ex: Math.Round(2.7) returns 3

Math.Abs(number) returns the absolute value of number

Ex: Math.Abs(-4) returns 4

Math.Floor(number) returns the floor of the number, as an integer
This is sometimes useful to convert a floating point variable to an integer data type

Ex: Math.Floor(3.14) returns 3

There are many more, for sin, cos, tan, atan, exp, log, etc.

When we have many variables of the same type it can sometimes be tedious to declare each one individually. VB.NET allows us to declare multiple variables of the same type at once, for example:

    Dim a, b as Double
    Dim a as Double, b as Integer
    Dim c as Double = 2, b as integer = 10

Variable Scope

When we DIM a variable inside an event or subroutine, the variable only “exists” within the scope of the event. This means we are free to define other variables of the same name in different events (which is often quite useful to keep variables from stomping on each other’s values!) For example

    Private Sub MyClick1(...) Handles MyButton.Click
        Dim i As Integer
        i = 10 / 3
    End Sub
Private Sub MyClick2(...) Handles MyButton2.Click
    Dim i As Integer
    i = 30
End Sub

The variable i in the two subroutines is a different i; the first exists only within the scope of MyClick1 and the second only exists within the scope of MyClick2.

These types of variables are called **local** variables, because they only exist locally within the scope of the subroutine.

Another type of variable is a **global** variables. When you declare a variable within the Declarations section of a form, which is usually at the top of the form outside of all subroutines, then these variables are visible to the entire form. This is useful if you want to share the same variable among many different subroutines. Sometimes these variables are declared using the word **public** instead of dim. The word public is used when we have multiple forms and objects; for now we can just use the word Dim.

Examples:

If the following three Click events are coded, what would the output be if the buttons were clicked in the following order: btnInitialize, btnAdd, and btnOutput?

```vb
Public Class frmDrills
    Inherits System.Windows.Forms.Form
    Dim intDrillValue As Integer
    Private Sub btnInitialize_Click(...)
        intDrillValue = 10
    End Sub
    Private Sub btnAdd_Click(...)
        intDrillValue = intDrillValue + 10
    End Sub
    Private Sub btnOutput_Click(...)
        MsgBox(intDrillValue)
    End Sub
End Class
```
Strings

A string variable is a variable that refers to a sequence of textual characters. A string variable is declared by using the data type of String:

    Dim s as String

To assign a literal value to a string, the value must be in double quotes. The following shows how to add three strings to a listbox:

    Dim day1 As String
    Dim day2 As String
    day1 = "Monday"
    day2 = "Tuesday"
    lstBox.Items.Add(day1)
    lstBox.Items.Add(day2)
    lstBox.Items.Add("Wednesday")

This fills the listbox with “Monday”, “Tuesday”, and “Wednesday”.

If we are adding a lot of items to the listbox then it can be tedious to write “lstBox.Items” every single time. A shorter way is to use the With command:

    With lstBox.Items
       .Add(day1)
       .Add(day2)
       .Add("Wednesday")
    End With

The with command takes a collection (lstBox.Items) which denotes the beginning of a block of statements that all pertain to lstBox.Items. Since VB.NET knows the statements are in reference to lstBox.Items, we can leave it off and only use the dot followed by the method name.

Two strings can be combined to form a new string consisting of the strings joined together. The joining operation is called concatenation and is represented by an ampersand (&).

For example, the following outputs “hello world”:

    Dim s1 as String = "hello"
    Dim s2 as String = "world"
    Console.WriteLine(s1 & " " & s2)

This outputs: hello world
Note that if we output:  
`Console.WriteLine(s1 & s2)`

Then we would get:  
```
helloworld
```

Sometimes with strings we can end up with very long lines of code. The line will scroll off toward the right. You can keep on typing to make a long line, but an alternate method is to continue the line on the next line. To do that, use the line continuation character. A long line of code can be continued on another line by using underscore ( _ ) preceded by a space:

```
msg = "640K ought to be enough " & _
    "for anybody. (Bill Gates, 1981)"
```

is the same as:

```
msg = “640K ought to be enough “ & “for anybody. (Bill Gates, 1981)”
```

### String Methods and Properties

There are a number of useful string methods and properties. Just like control objects, like list boxes, that have methods and properties, strings are also objects and thus have their own properties and methods. They are accessed just like the properties and methods: use the name of the string variable followed by a dot, then the method name.

- `str.Length()` ; returns number of characters in the string
- `str.ToUpper()` ; returns the string with all letters in uppercase
  does not change the original string, returns a copy
- `str.ToLower()` ; returns the string with all letters in lowercase
  does not change the original string, returns a copy
- `str.Trim()` ; returns the string with leading and trailing whitespace removed. Whitespace is blanks, tabs, cr’s, etc.
- `str.Substring(m,n)` ; returns the substring of str starting at character m and fetching the next n characters. M starts at 0 for the first character! If n is left off, then the remainder of the string is returned starting at position m.

Here are some examples:

```
Dim s As String = "eat big macs   
Console.WriteLine(s.Length())
Console.WriteLine(s.ToUpper())
Console.WriteLine(s & "!")
s = s.Trim()
Console.WriteLine(s & "!")
```
Console.WriteLine(s.Substring(0, 3))
Console.WriteLine(s.Substring(4))
Console.WriteLine(s.Substring(20))

Output:

15
EAT BIG MACS
eat big macs  !
eat big macs!
eat
big macs
CRASH! Error message  (do you know why?)

Sometimes it is useful to convert individual characters back and forth from their ANSI representation to the number that represents that character. Recall that a string is really a sequence of codes, where each code represents a different character. A small subset of the ASCII code is shown below:

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>(space)</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>68</td>
<td>c</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>125</td>
<td>}</td>
</tr>
<tr>
<td>126</td>
<td>-</td>
</tr>
</tbody>
</table>

The string “Ab” is really represented by storing the values 65 and 98 consecutively. To access the ASCII value of a character use ASC:

    Console.WriteLine(Asc("A"))  ' Outputs 65

To go in the reverse direction, from the code to the character, use Chr:

    Console.WriteLine(Chr(65))   ' Outputs A

On occasion you may be interested in generating the **empty string**, or a string with nothing in it. This is a string of length 0. It is referenced by simply "" or two double quotes with nothing in between.

Finally, if you would like to create a string that contains the " character itself, use two ""'s:

Wrong:

    s = "Dan Quayle said, "I love California; I practically grew up in Phoenix.""
What is the problem?

Right:

\[
\text{s = "Dan Quayle said, ""I love California; I practically grew up in Phoenix.""
}\]

Using Text Boxes for Input and Output

It turns out that any text property of a control is also a string, so what we just learned about strings also applies to the controls! A particularly useful example is to manipulate the content of text boxes.

For example, say that we create a text box control named txtBox. Whatever the user enters into the textbox is accessible as a string via txtBox.Text. For example:

\[
\begin{align*}
\text{Dim } s & \text{ as String} \\
\text{s} & = \text{txtBox.Text.ToUpper()} \\
\text{txtBox.Text} & = s
\end{align*}
\]

This changes the txtBox.Text value to be all upper case letters.

Text Boxes provide a nice way to provide textual input and output to your program. However, recall that other items also have a text property, such as Me.Text, which will change the caption on the title bar of your application. Similarly you can change items in a list box.

Because the contents of a text box is always a string, sometimes you must convert the input or output if you are working with numeric data. You have the following functions available for type-casting:

\[
\begin{align*}
\text{CDbl(string)} & ; \text{Returns the string converted to a double} \\
\text{CInt(string)} & ; \text{Returns the string converted to an integer} \\
\text{CStr(number)} & ; \text{Returns the number converted a string}
\end{align*}
\]

For example, the following increments the value in a text box by 1:

\[
\begin{align*}
\text{Dim } i & \text{ as Integer} \\
\text{i} & = \text{CInt(txtBox.Text)} \\
\text{i} & = i + 1 \\
\text{txtBox.Text} & = \text{CStr(i)}
\end{align*}
\]

For strings, you can also use the toString method:

\[
\text{txtBox.Text} = i.\text{ToString}
\]
**Options**

It turns out that VB.NET actually allows you to perform these operations without the conversion functions:

```
i = txtBox.Text  ' implicitly converts the string to a number
```

However, this practice is not recommended because it can often lead to errors when the programmer really didn’t intend to convert the variables. For example, consider the following:

```
Dim val as Integer

...  
val = 3.4  ; perhaps we forgot val only stores integers
          ; val is converted to the integer value 3 for us
          ; but we might have been expecting 3.4 to be stored
```

For this reason, VB.NET includes a way to require type-casting. At the very top of the code, as the first line in the form, add the line:

```
Option strict on
```

This forces type-casting or a program will not compile. It is recommended that you add this to the top of your program to enforce type checking. It will help eliminate errors in the future.

**Comments**

As your code gets more complex, it is a good idea to add comments. You can add a comment to your code by using the ‘ character. Anything from the ‘ character to the end of the line will be ignored. If you neglect to add comments, it is very common to forget how your code works when you go back and look at it later!

Another common use of comments is to “comment out” blocks of code. For example, if you insert code for testing purposes or if code is not working properly, you can comment it out and have the compiler ignore it. However, the code will still be there if you want to use it again later without having to type it in again – just uncomment the code.

VB.NET has a button to comment and uncomment blocks of code:

Highlight the text to comment and click the icon shown above on the left.
Highlight the text to uncomment and click the icon shown above on the right.
**Constants**

Often during programming you will want to represent values that will not change during the execution of the program. However, they might change at some future point. These values are called **constants**. For example, perhaps you have created a tax program. The tax rate is 0.05.

You could embed the value 0.05 directly in your program where it is used, e.g.:

\[
\text{dblValue1} = 0.05 \times \text{dblAmount} \\
\text{...} \\
\text{dblValue2} = 0.05 \times \text{dblValue3} \\
\text{...}
\]

Two disadvantages to this approach are:

1. The value 0.05 is somewhat mysterious and it is not immediately clear why it is there and what it represents. If there was a name, like TaxRate, it would be clearer what the value means.
2. If the tax rate ever changes, you would have to find all the 0.05’s and change them to the new value.

You might wonder why not just use a variable. A variable would work, but a variable may also change within the program. To make it clear that a value should not change, and in fact may not change, you should use a constant.

To define a constant use this format:

```
Const CONSTANTNAME = Value
```

For example:

```
Const TAXRATE = 0.05
```

It is a common convention to name constants in all uppercase letters.
In-class Exercise:

It is recommended that you maintain your training heart rate during an aerobic workout. Your training heart rate is computed as:

\[ 0.7(220-a)+(0.3*r) \]

where \( a \) is your age in years and \( r \) is your resting heart rate. Write a program to compute the training heart rate as shown below:

Example:

You are running a marathon (26.2 miles) and would like to know what your finishing time will be if you run a particular pace. Most runners calculate pace in terms of minutes per mile. So for example, let’s say you can run at 7 minutes and 30 seconds per mile. Write a program that calculates the finishing time and outputs the answer in hours, minutes, and seconds.

Input:

Distance : 26.2  
PaceMinutes: 7  
PaceSeconds: 30

Output:

3 hours, 16 minutes, 30 seconds

Here is one algorithm to solve this problem:
1. Express pace in terms of seconds per mile, call this SecsPerMile
2. Multiply SecsPerMile * 26.2 to get the total number of seconds to finish. Call this result TotalSeconds.
3. There are 60 seconds per minute and 60 minutes per hour, for a total of 60*60 = 3600 seconds per hour. If we divide TotalSeconds by 3600 and throw away the remainder, this is how many hours it takes to finish.
4. TotalSeconds mod 3600 gives us the number of seconds leftover after the hours have been accounted for. If we divide this value by 60, it gives us the number of minutes, i.e. (TotalSeconds mod 3600) / 60
5. TotalSeconds mod 3600 gives us the number of seconds leftover after the hours have been accounted for. If we mod this value by 60, it gives us the number of seconds leftover. (We could also divide by 60, but that doesn’t change the result), i.e. (TotalSeconds mod 3600) % 60
6. Output the values we calculated!

**In-Class Exercise:** Write the code to implement the algorithm given above.

**In-Class Exercise:** Write a program that takes as input an amount between 1 and 99 which is the number of cents we would like to give change. The program should output the minimum number of quarters, dimes, nickels, and pennies to give as change assuming you have an adequate number of each coin.

For example, for 48 cents the program should output;
   1 quarter
   2 dimes
   0 nickels
   3 pennies

First write pseudocode for the algorithm to solve the problem. Here is high-level pseudocode:

- Dispense max number of quarters and re-calculate new amount of change
- Dispense max number of dimes and re-calculate new amount of change
- Dispense max number of nickels and re-calculate new amount of change
- Dispense remaining number of pennies

**Input and Output**

We have already seen how to get input via textboxes and we can also output data via textboxes, listboxes, or the console window.

We will not cover this in class but to format numbers, currency, or percents, there is a format function (for example, FormatNumber(1.23456,1) turns the number into only a single decimal point, 1.2). We can also format to pad numbers with spaces.
Another way to input and output data is through “pop-up” windows. To input data through an input dialog box, use a statement of the form:

    stringVar = InputBox(prompt, title)

To output data via a popup use a statement of the form:

    MsgBox(string)

for example:

    Dim s as String
    s = InputBox(“Enter your name”, “Name”)  
    MsgBox(“You entered “ & s & “.”)

Generates a window like the following:

![InputBox Window](image)

Whatever the user types into the text area is stored into variable s when the user presses OK. The output is then shown in a message box:

![MessageBox Window](image)

If the user presses cancel, then the string returned is empty. There are additional options on the InputBox and MsgBox to set the title, icon, and buttons that appear on the pop-up window. See the VB.NET reference for more information.
**Introduction to Debugging**

If a program is not running the way you intend, then you will have to debug the program. Debugging is the process of finding and correcting the errors. There are two general ways to go about debugging:

1. Add `Console.WriteLine` or `MsgBox` statements at strategic points in the program to display the values of selected variables or expressions until the error is detected.

2. Use an integrated debugger that lets you pause, view, and alter variables while the program is running. Such a tool is called a debugger.

**Debugging with WriteLines**

Let’s first examine the `WriteLine` method. Although somewhat “primitive” it is useful since it works in virtually any programming environment. Consider the following program which converts a temperature from Fahrenheit to Celsius using the formula:

```vbnet
Private Sub btnConvert_Click(...) Handles Button1.Click
    Dim Celsius As Integer
    Dim Fahrenheit As Integer
    Const ConversionFactor As Integer = 5 / 9

    Fahrenheit = CInt(InputBox("Enter temp in Fahrenheit"))
    Celsius = ConversionFactor * (Fahrenheit - 32)
    MsgBox("The temp in Celsius is " & CStr(Celsius))
End Sub
```

When run, it compiles and executes but gives incorrect outputs. For example, on an input of 100 F, we get 68 C, which is incorrect. What is wrong?

One technique is to add `WriteLine` statements to output intermediate values of interest:

```vbnet
Private Sub btnConvert_Click(...) Handles Button1.Click
    Dim Celsius As Integer
    Dim Fahrenheit As Integer
    Const ConversionFactor As Integer = 5 / 9

    Fahrenheit = CInt(InputBox("Enter temp in Fahrenheit"))
    Console.WriteLine("Fahrenheit = " & Fahrenheit)
    Console.WriteLine("Conversion = " & ConversionFactor)
    Celsius = ConversionFactor * (Fahrenheit - 32)
    MsgBox("The temp in Celsius is " & CStr(Celsius))
End Sub
```

The program outputs:

```
Fahrenheit = 100
Conversion = 1
```

The Conversion factor is obviously incorrect! This should give you enough information to see that the variable was defined incorrectly as an Integer and rounded up to 1, since an Integer cannot store the number 5 / 9.

The easy correction is to change this to a Double:

```vbnet
Const ConversionFactor As Double = 5 / 9
```

Note that if we had used “option strict on” at the top of our program, then this error would have been detected for us!

Once the error is found and detected, then using the WriteLine method we would then remove or comment out the WriteLine statements that helped us track down the source of the error.

**Using the Integrated Debugger**

While the process described above works, it is somewhat tedious to all of the WriteLine statements and then remove them. A much nicer technique is to use the built-in debugger.

VB.NET programs run in one of three modes – design mode, run mode, or break mode. The current mode is displayed in parentheses in the VB.NET title bar. Design mode is where you design the program. Run mode is when you run the program. Break mode is when you pause the program to debug it.

If we return to the original program with the bugs, one way to enter break mode is to add a breakpoint. A breakpoint stops execution at a particular line of code and enters Break mode. This is useful when you know that a particular routine is faulty and want to inspect the code more closely when execution reaches that point.

To set a breakpoint, click in the border to the left of the code. A red dot will appear. Click the same dot to turn the breakpoint off.

```vbnet
Dim Celsius As Integer
Dim Fahrenheit As Integer
Const ConversionFactor As Integer = 5 / 9

Fahrenheit = CInt(InputBox("Enter the temperature in Fahrenheit"))
Celsius = ConversionFactor * (Fahrenheit - 32)
MsgBox("The temp in Celsius is " & CStr(Celsius))
```

When we run the program and reach this code, the program automatically enters Break mode and stops. Execution stops **before** the line with the breakpoint is executed. The current line is indicated in yellow:
The first thing we can do is inspect the value of variables. One way to do this is to hover the mouse over the variable or constant, and a popup window will display its contents:

```
Dim Celsius As Integer
Dim Fahrenheit As Integer
Const ConversionFactor As Integer = 5 / 9

Fahrenheit = CInt(InputBox("Enter the temperature in Fahrenheit"))
Celsius = ConversionFactor * (Fahrenheit - 32)
MsgBox("The temp in Celsius is " & CStr(Celsius))
```

In this case, I have hovered over “ConversionFactor” and its value is displayed as 1. This by itself would give us enough information to debug the program. Note that we did not have to add any WriteLine statements!

We can also immediately see the contents of all the active variables by looking in the “ Autos” window:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Celsius</td>
<td>0</td>
</tr>
<tr>
<td>Fahrenheit</td>
<td>100</td>
</tr>
</tbody>
</table>

We can also click on the “Locals” tab to see all local variables in the current procedure:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Me</td>
<td>{TestVBApp.Form1}</td>
</tr>
<tr>
<td>Celsius</td>
<td>0</td>
</tr>
<tr>
<td>Fahrenheit</td>
<td>100</td>
</tr>
<tr>
<td>sender</td>
<td>{System.Windows.Forms.Button}</td>
</tr>
</tbody>
</table>

If a value is displayed in red this indicates that the variables has just been changed.

To illustrate this, we can now step through the program one line at a time using the buttons:
These buttons are used respectively to step into a procedure, step over a procedure, or step out of a procedure. We can use these buttons and view our variables change as we run the program. When we define our own procedures this will make more sense, but for now the first two buttons do the same thing when we’re executing code within a subroutine.

Click on the “Step Into” or “Step over” buttons in our example and we get:

```
Fahrenheit = CInt(InputBox(“Enter the temperature in Fahrenheit”))
Celsius = ConversionFactor * (Fahrenheit - 32)
MsgBox(“The temp in Celsius is ” & CStr(Celsius))
```

<table>
<thead>
<tr>
<th>Locals</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Me</td>
<td>{TestVBApp.Form1}</td>
</tr>
<tr>
<td>Celsius</td>
<td>68</td>
</tr>
<tr>
<td>e</td>
<td>{System.EventArgs}</td>
</tr>
<tr>
<td>Fahrenheit</td>
<td>100</td>
</tr>
<tr>
<td>sender</td>
<td>{System.Windows.Forms.Button}</td>
</tr>
</tbody>
</table>

Here we can see that the Celsius variable was just changed to 68.

As a shortcut, F11 steps into a procedure, and F10 steps over a procedure. These commands are the same for non-procedures (i.e. the move to the next statement).

Whenever you are done debugging your program, you must make sure that the debugging session is ended before you go back to edit your code. Click the “Stop Debugging” button to exit the debugger.

A common error is to attempt to change and fix code while still in debugging mode. If you attempt to do this, the program will beep at you until you stop the program from executing!