Data Types, Arithmetic, Strings, Input

Visual Basic distinguishes between a number of fundamental data types. Of these, the ones we will use most commonly are:

Integer
Long
Single
Double
String
Boolean

The table below summarizes the different types:

Type	Bytes of Storage	Range of Values
Byte	1	0 to 255
Integer	4	−2,147,483,648 to 2,147,483,647
Long	8	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Short	2	−32,768 to 32,767
Single	4	−3.402823E38 to +3.402823E38
Double	8	−1.79769313486231E308 to +1.79769313486231E308
Char	2	Any Unicode character
String	2 per char	0 to approximately 2 billion Unicode characters
Boolean	1	True or False
Decimal	16	−79,228,162,514,264,337,593,543,950,335 to 79,228,162,514,264,337,593,543,950,335 with 29 Significant digits
Date	8	00:00:00 on January 1, 0001 to 11:59:59 on December 31, 9999
Object	4	(Reference to an object)

An Integer is a positive or negative number with no value past the decimal point. Note the limitation on the range of values it can hold. If we allocate more storage space (e.g., more bytes) then we can represent larger numbers.

The Long data type uses 8 bytes of storage instead of 4 like the Integer, so it can represent much larger values.

Similarly, VB has two commonly used floating point values: Single and Double. These data types are used to represent real numbers. The Single uses 4 bytes and the Double uses 8 bytes, so the Double can store larger values than the single.

If double has a larger data range than integer, and can store floating point numbers, you might wonder why we don't just always use 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.

Booleans are used to represent True or False. These are the only two values that are allowed. Booleans are useful in programming due to their ability to select a course of action based upon some outcome that is either true or false, so we will use Booleans extensively in decision-making.

Strings consist of textual data and are enclosed in double-quotes. Strings are represented as a sequence of bit patterns that match to alphanumeric values. For example, consider the following mapping for the letters A, B, and C:

A	01000001
B	01000010
C	01000011

To store the string "CAB" we would simply concatenate all of these codes:

01000011 01000001 01000010

Note that there is a difference between a string of numbers, and a number such as an Integer. Consider the string "0" and the number 0. The String "0" is represented by the bit pattern 00110000 while the integer 0 is represented by 00000000. Similarly, the string "10" would be represented as 00110001 00110000 while the integer 10 is represented as 00001010.

Strings are simply a sequence of encoded bit patterns, while integers use the binary number format to represent values. We will often convert back and forth between String and Number data types.

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:

Arithmetic Operator	VB.NET Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/ (floating point)
Division	\ (integer, truncation)
Exponent	^
Modulus	mod

Here are some examples of arithmetic operations and outputting the result to the console:

```
Console.WriteLine(3 + 2)
Console.WriteLine (3 - 2)
Console.WriteLine (5 * 2 * 10)
Console.WriteLine (14 mod 5)
Console.WriteLine (9 mod 4)
Console.WriteLine (10 / 2)
Console.WriteLine (11 / 2)
Console.WriteLine(11 \ 2)
Console.WriteLine (1 / 2)
Console.WriteLine (2 ^ 3)
Console.WriteLine ((2^3)*3.1)
```

The results are:

```
5
1
100
4
1
5
5.5
5
0.5
8
24.8
```

Extremely large numbers will be displayed in scientific notation, where the letter E refers to an exponent of 10^E :

```
Console.WriteLine(2^50)
```

outputs: 1.1259E+15

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.

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 Integer, Single, Double, String, etc.

It is a common notation to preface the first three letters of the variable with the data type. The first letter of subsequent words is capitalized. This is only a notation and is not required, but it's considered a good practice to follow. Here are common prefixes for several data types (we already talked about prefixes for controls like buttons and textboxes):

Data Type	Prefix	Data Type	Prefix
Boolean	bln	Integer	int
Byte	byt	Long	lng
Char	chr	Object	obj
Date	dat	Short	shr
Decimal	dec	Single	sng
Double	dbl	String	str

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.

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:

```
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
```

Constants

Sometimes we might like to make a variable whose value is set at declaration and cannot be changed later. These are called constants in VB. An example of where a constant might be used is a program that uses the value of pi (3.1415). This is a value that the program shouldn't change. To declare a constant use the keyword const instead of dim. It is also a common notation to make constants all uppercase letters:

```
Const sngSALES_TAX_RATE As Single = 0.06
Const sngPI As Single = 3.14159
```

If we had a program that was computing sales tax, it might contain something like:

```
sngTotal = sngAmount * 0.06 + sngAmount
```

We can make it more clear by using the constant:

```
sngTotal = sngAmount * sngSALES_TAX_RATE + sngAmount
```

The objective of our code is clearer using the constants instead of the direct value. This also has the benefit that if the tax rate is used in many places in the program, then there is only one place to modify it (where the constant is declared) in case the tax rate changes. Without using the constant we would have to find all the locations in the program that reference the old tax rate and change the value to the new tax rate.

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. multiplication, division, and modulus have the next highest precedence
3. 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:

Dim x As Integer	Value stored in X
x = 1 + 2 + 6 / 6	4
x = (1 + 2 + 3) / 6	1
x = 2 * 3 + 4 * 5	26
x = 2 / 4 * 4	2
x = 2 / (4 * 4)	0
x = 10 Mod 2 + 1	1

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

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, 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.

More on 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 output three strings:

```
Dim strDay1 As String
Dim strDay2 As String
strDay1 = "Monday"
strDay2 = "Tuesday"
Console.WriteLine(strDay1)
Console.WriteLine (strDay2)
Console.WriteLine ("Wednesday")
```

This outputs "Monday", "Tuesday", and "Wednesday".

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 str1 as String = "hello"
Dim str2 as String = "world"
Console.WriteLine(str1 & " " & str2)
```

This outputs: hello world

Note that if we output: Console.WriteLine(str1 & str2)

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 text 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?)
```

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:

```
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:

```
Dim s as String
s = txtBox.Text.ToUpper()
txtBox.Text = s
```

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.

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** (there are others too, using CDataType):

```
CSng(string)        ; Returns the string converted to a single
CInt(string)        ; Returns the string converted to an integer
CStr(number)        ; Returns the number converted a string
```

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

```
Dim i as Integer

i = CInt(txtBox.Text)
i = i + 1
txtBox.Text = CStr(i)
```

Option Strict

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 this reason, VB.NET includes a way to require type-casting. At the top of the code, add the line:

```
Option strict on
```

This forces type-casting or a program will not compile.

For example, consider the statement:

```
Dim intCount as Integer = “abc123”
```

With option strict on, this program will not compile. With option strict off, this program will compile but when it is run and the statement is executed, “abc123” is not a valid integer. A **runtime error** will result, in this case **a type mismatch** error. By using option strict on, you can catch these types of errors, although it may result in slightly more verbose programming code.

Here is another example that might give the undesired results:

```
Dim strVal1 as String = “1”
Dim strVal2 as String = “2”
Dim intValSum as Integer = strVal1 + strVal2
```

If option strict on is enabled, what would happen?
If option strict off is set, what value is stored in intValSum?

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:

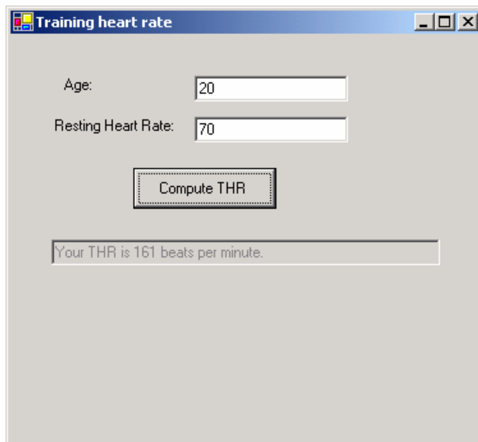
Highlight the text to comment and click the icon shown above on the left.
Highlight the text to uncomment and click the icon shown above on the right.

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) mod 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, labels, 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)
```

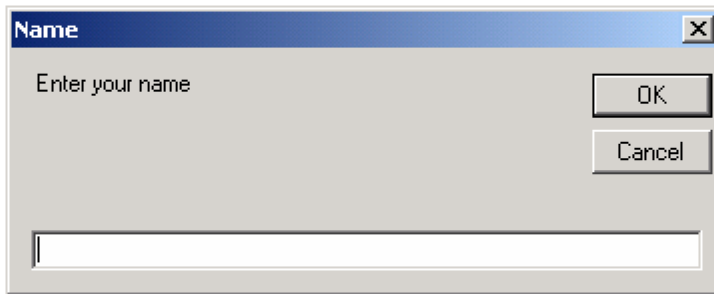
To output data via a popup use a statement of the form:

```
MessageBox.Show(string)
```

for example:

```
Dim s as String
s = InputBox("Enter your name", "Name")
MessageBox.Show("You entered " & s & ".")
```

Generates a window like the following:



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:

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

Exceptions

Consider the following code:

```
sngNum = CSng(txtNum.Text)
```

This looks harmless, but txtNum might contain a non-numerical value, like “abc123” which could cause problems with the logic of the program.

This type of error is called a **runtime** error and is not caught until the program runs, because the result depends on the data input to the program (as opposed to compiler errors, which are caught when the program is compiled).

In the case of the error above, and others that are similar, Visual Basic will throw an **exception** and crash. An exception is some condition that was not expected and caused an error. If we want the program to fail more gracefully we can use the **try/catch** block:

```
Try
    try-block
Catch [exception-type]
    catch-block
End Try
```

The try-block contains program statements that might throw an exception.
The catch-block contains statements to execute if an exception is thrown.

Here is a short example:

```
Try
    sngNum = CSng(txtNum.Text)
    Console.WriteLine("You entered " & sngNum)
Catch ex As Exception
    MessageBox.Show(“There was an error “ + ex.Message)
End Try
```

If the user enters a value such as “123” then the program will “catch” the error in the conversion and skip directly to the catch block:

There are more detailed ways to handle exceptions and specific types of exceptions. We’ll look at that later when we cover file I/O.

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 MessageBox 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 the 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:

```
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:

```
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:

```
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.

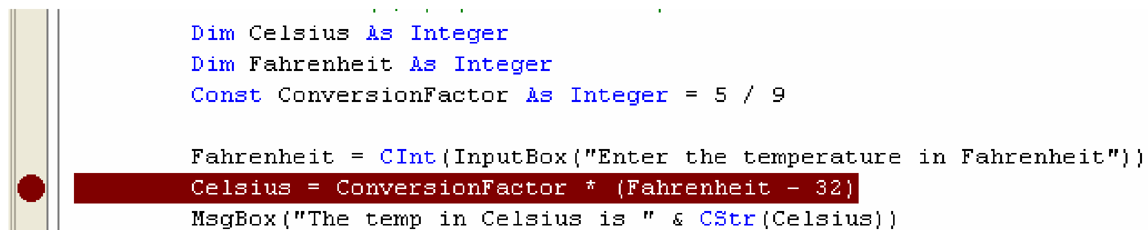
Using the Integrated Debugger

While the process described above works, it is somewhat tedious to all of the WriteLine statements and them 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.



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:

```
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))
```

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
            Public Const ConversionFactor As Integer = 1
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:

Autos

Name	Value
Celsius	0
Fahrenheit	100

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

Locals

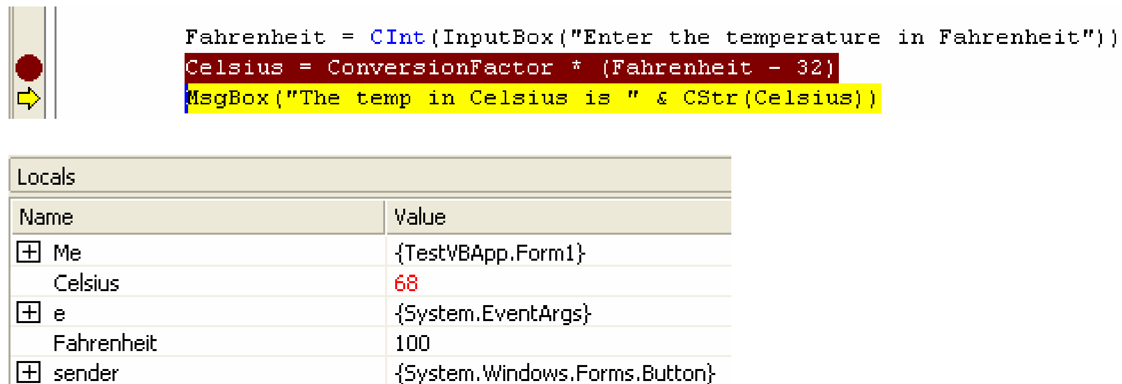
Name	Value
⊞ Me	{TestVBApp.Form1}
Celsius	0
⊞ e	{System.EventArgs}
Fahrenheit	100
⊞ sender	{System.Windows.Forms.Button}

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:



Name	Value
Me	{TestVBApp.Form1}
Celsius	68
e	{System.EventArgs}
Fahrenheit	100
sender	{System.Windows.Forms.Button}

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!