What is Programming?

What is Programming?

Writing software (computer programs) is a lot like writing down the steps it takes to complete a process. These written steps, often referred to as instructions, are passed to a computer which then processes each command in sequence. Often, in computing, these instructions manipulate things called objects; objects may be numbers, words and/or graphical interfaces.

Computer Rules

Always remember that a computer is dumb, but very obedient. In other words, a computer will do exactly what the user tells it to do… even if it is not necessarily what the user intended it to do. Below are three rules to consider when programming:

1. Computers do not make mistakes – programmers do
2. Programming will highlight the importance of 'clarity of expression'
3. Programming instructions are processed in **sequence** and one at a time

High-Level Programming Languages

Most computer programming today is achieved using high-level programming languages. There are lots of these languages available on the market and some are quite old, i.e. COBOL which was devised in the 1950s! More modern languages include Java, VB.NET, Python, C#, JavaScript, PHP, etc.

A high-level language basically makes it easier to write programs by allowing advanced programs to be written without any concerns in regard to computer architecture – i.e. specific CPU instructions. These languages also come with pre-written, reusable common code – known as libraries – which help to reduce development times. An example of a high-level language is shown below:

Code:	**Explanation:**

If x >= 5 Then
WriteLine("Hello World")
End IF
	If x is greater than or equal to 5, then write the line 'Hello World' to the console window.

All you need to remember about high-level programming languages (including VB.NET) is that:

- *The syntax is sort of like English*
- *They have pre-written code called libraries*
- *They 'sit on top' of an operating system*
- *They are **not** languages that a CPU understands*

Assembler Languages

Assembler language is 'one step above' a computer's native language, machine language (binary). In an assembler language, instructions are given human-friendly symbolic names. Unlike high-level languages, the programmer works with basic operations/instructions that the CPU can directly perform, such as bitwise operations (i.e. AND, OR, NOT). Remember that assembler language is very basic and therefore is impractical when writing large programs (which are normally achieved using high-level languages). Below is an example of assembler language (notice it is not as interpretable as the high-level language):

Code:	**Explanation:**
MOV EAX, [EBX]	Move the 4 bytes in memory at the address contained in EBX into EAX
MOV [ESI+EAX], CL	Move the contents of CL into the byte at address ESI+EAX

Machine Language

Computers only understand 'bits' – 0s and 1s, often referred to as binary; binary is machine language. A computer system uses these bits to represent information, whether that is numbers, characters, pixels, etc. The computer also uses bits to represent computer instructions; this is very difficult to do, although the pioneers of computer science once did this! However, today most programs are written in high-level languages which are then compiled into machine code using a compiler.

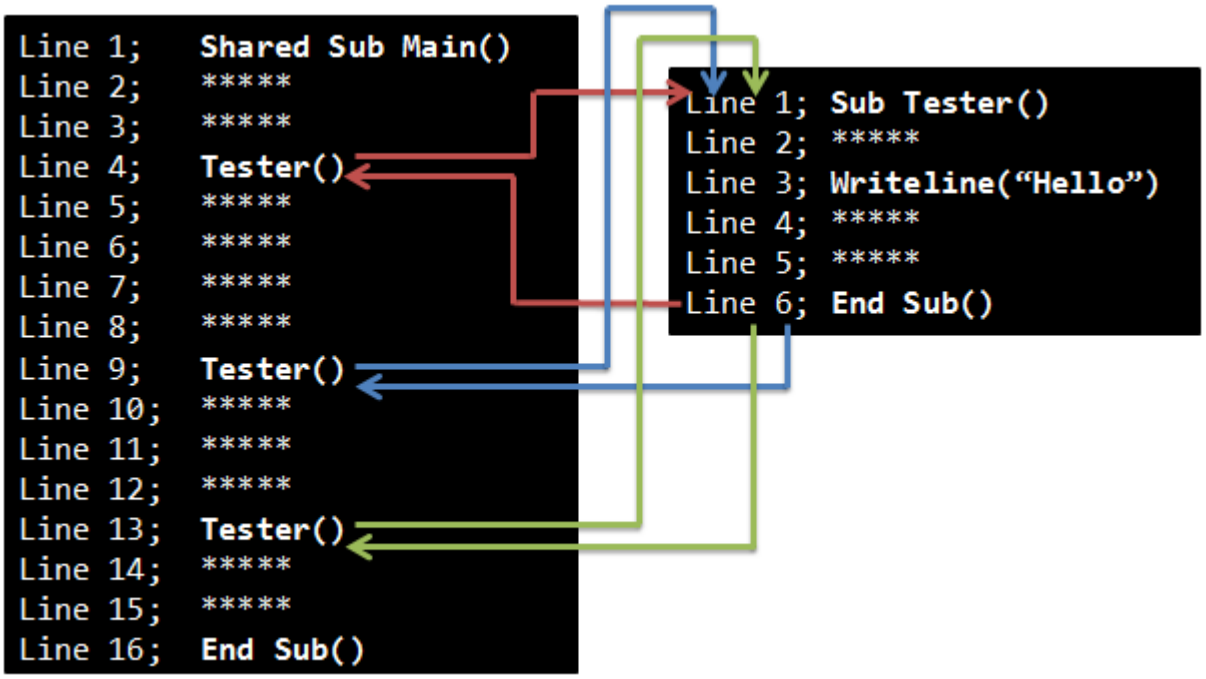
Compilers

As stated earlier, computers do not understand programs written in high-level languages, such as VB.NET and Java (they only understand binary, 0s and 1s). High-level languages must be compiled using a compiler to convert the source code into machine code. Every high-level programming language has a specific compiler; a compiler is a small computer utility program usually packaged with the SDK (Software Development Kit) of the programming language.

Getting Started

Procedural Programming

Procedural programming is a method of developing computer programs – programs that are sequential and linear in fashion, with specific focus on the order in which tasks are completed. Consider procedural programming as A before B and B before C. Procedural programs are normally command-line based and thus are not normally graphical in appearance. The example below shows the flow of a procedural program; each instruction is executed in order and one at a time.



My First Program

Open Notepad or Notepad++ and type the following code (pay attention to the indentation used):

Code:	**Explanation:**

Imports System.Console
Public Class MyFirstProgram
  Shared Sub Main()
  End Sub
End Class
	1. `Imports System.Console` – this is an imported library
	2. `Public Class MyFirstProgram` – a class is a 'blueprint' for an object. However, for now consider this as indicating the start of the program, as well as the name.
	3. `Shared Sub Main()` – every program has to have a Sub Main() (an initial starting method)
	4. `End Sub` – end of Sub Main()
	5. `End Class` – end of the program

Next, add the following programming instruction to Sub Main():

```
Imports System.Console
Public Class MyFirstProgram
    Shared Sub Main()
        WriteLine("Hello World")
    End Sub
End Class
```

Once the above line of code has been added to Sub Main(), save the file as 'first.vb'. Be sure to change the 'Save As Type' to 'All Files'. Now open the Visual Studio Command Prompt (this is a utility program that is packaged with visual studios – Microsoft's software development environment) and type the following instructions (note that the 'c:\student temp' directory should be replaced with the location and folder name of where your VB file has been saved):

```
cd c:\student temp
vbc first.vb /t:exe
first.exe
```

Program Output:

Test Your Skills

- ✓ *Now modify the code so that the console prints your full name, address and phone number on a separate line each. Remember you will need to compile your program each time you change it.*

  *Use the '**vbc first.vb /t:exe**' command to achieve this. You can also use the up arrow key to reprint previous commands typed.*

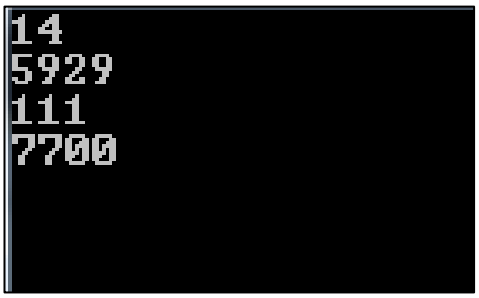
My Lucky Seven

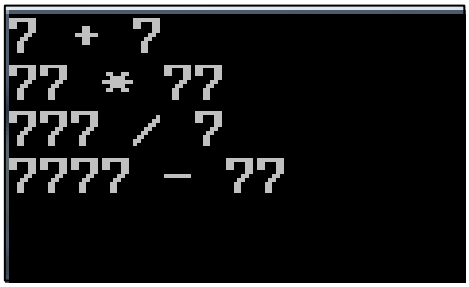
Open Notepad or Notepad++ and type the following code (pay attention to the indentation used):

```
Imports System.Console
Public Class MyLuckySeven
    Shared Sub Main()
        WriteLine(7 + 7)
        WriteLine(77 * 77)
        WriteLine(777 / 7)
        WriteLine(7777 - 77)
    End Sub
End Class
```

Once the code has been typed up, compile and run the program. See previous steps if you cannot remember how to achieve this. Try this code both with and without double quotes, i.e. 7+7 and "7+7".

Program Output:





Concatenation can be used to join 'parts' together. This can consist of both 'string' (text) and program instructions. Modify the original code so that it appears the same as the code shown below. Then compile and run the program again.

```
Imports System.Console
Public Class MyLuckySeven
    Shared Sub Main()
        WriteLine("7 + 7 = " & 7+7)
        WriteLine("77 * 77 = " & 77 * 77)
        WriteLine("777 / 7 = " & 777 / 7)
        WriteLine("7777 + 77 = " & 7777 - 77)
    End Sub
End Class
```

Test Your Skills

- ✓ *Create a short program that displays the first five sums of the five times table.*

Using Variables

A variable is an allocation in memory. Variables are very useful as a programmer can use them to store user input, answers from calculations and/or values for later use. There are many types of variable; one such variable type is Integer, which is used to store whole numbers. To declare a variable, the following syntax is used (note X is the name of the variable and As Integer is the type):

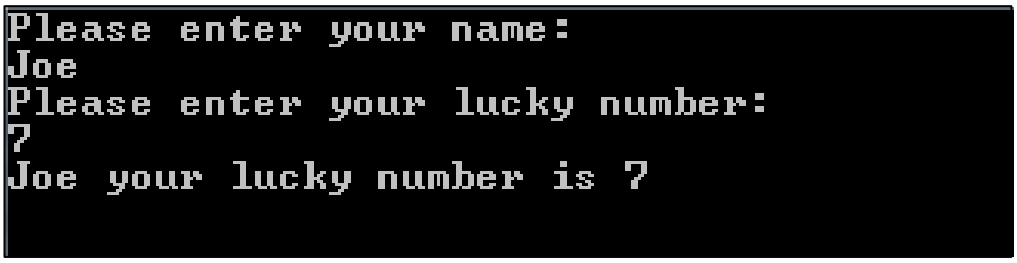
```
Dim X As Integer
```

Open Notepad or Notepad++ and type the following code (pay attention to the indentation used):

```
Imports System.Console
Public Class UsingVariables
    Shared Sub Main()
        Dim X As Integer
        Dim Y As String
        WriteLine("Please enter your name:")
        Y = ReadLine()
        WriteLine("Please enter your lucky number:")
        X = ReadLine()
        WriteLine(Y & " your lucky number is " & X)
    End Sub
End Class
```

Once the code has been typed up, compile and run the program. See previous steps if you cannot remember how to achieve this. Note that the ReadLine command is the opposite of WriteLine; ReadLine allows a programmer to gain input from the user and store the value entered into a variable ready for processing.

Program Output:



Test Your Skills

- ✓ *Create a short program that allows the user to enter two numbers. The program must then calculate the answer of those two numbers added together, multiplied together, divided together and subtracted by each other. The program must then display all the calculations and all of their respective answers.*

Applying Comments

Comments can be added to programming code; comments are not executed by the computer, but are instead visual aids for the programmer. For example, comments can be used for:

- Specifying the author and version of the software
- 'Commenting out' unwanted or old code
- Testing purposes (commenting out segments of code)
- Recording changes made to the software
- Leaving notes for other programmers

Modify the program that you wrote earlier (the 'Using Variables' program) to include your name, date and software version in the comments of the code. Comments are added by using the apostrophe symbol (') at the start of a line.

For example:

```
'*************************
'Joe Smith
'Version 1.1
'*************************
```

Using Procedures

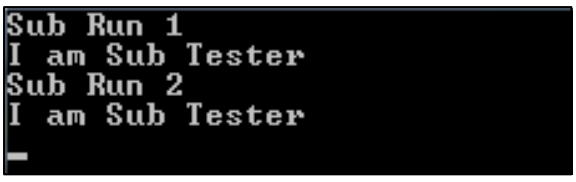
In procedural programming, code is broken down into 'procedures'. In VB.NET this is achieved using Subs. For clarification purposes, a method, sub, procedure and function refer to the same thing. They refer to an executable block of code (with a specific purpose) that can be called by a program. Procedures can be used to 'break up' programs, making them easier to build and maintain.

Open Notepad or Notepad++ and type the following code (pay attention to the indentation used). Once the code has been typed up, compile and run the program.

```
Imports System.Console

Public Class UsingProcedures
        Shared Sub Main()
            WriteLine("Sub Run 1")
            Tester()
            WriteLine("Sub Run 2")
            Tester()
        End Sub
        Shared Sub Tester()
            WriteLine ("I am Sub Tester")
        End Sub
End Class
```

Program Output:



Using Selection

'If' Statements (Selection)

'If' statements are fundamental in all programming languages; they allow selection. In other words, they allow alternative paths in the program to be followed or avoided. The principles of an 'if' statement are the same in all languages; only the syntax differs. 'If' statements use operators to determine if a condition is true or false. Operators include **<** (less than), **<=** (less than or equal to), **=** (equal to), **>** (greater than) and **>=** (greater than or equal to). The logic of an 'if' statement is shown below:

```
If [this condition is true] Then
      Do this code
End If
```

Open **Notepad** or **Notepad++** and type the following code (pay attention to the indentation used). Once the code has been typed up, compile and run the program.

```
Imports System.Console
Public Class UsingIFS
        Shared Sub Main()
         WriteLine("Enter a number")
                Dim Y As Integer
                Y = ReadLine()
                If Y > 3 Then
                        WriteLine("Hello World")
                End If
        End Sub
End Class
```

A second (and more) condition(s) can be added using the keyword **AND**. When using the **AND** keyword, both conditions must be true before the code contained in the 'if' statement will run. The keyword **OR** can be used instead to make the code execute if either of two (or more) conditions is true. Amend the previous code so that it includes a second condition; compile and run the code to ensure it works as expected. The syntax for using multiple conditions is shown below:

```
If Y > 3 AND Y < 7 Then
```

Program Output:

```
Enter a number
4
Hello World
```

Test Your Skills

- ✓ *Amend the previous code so that the program writes 'Hello World' if the number is between 5 and 10, and 'Hello Universe' if the number is between 11 and 20.*

If, ElseIf and Else

ElseIf can be used to extend an 'if' statement (by adding additional conditions to check if the previous condition returns false). This method is more efficient than writing multiple separate 'if' statements. The **Else** keyword can be used to catch any other possibilities if all conditions return false. The logic of an extended 'if' statement is shown below:

```
If [this condition is true] Then
      Do this code
ElseIf [this condition is true] Then
      Do this code
Else
      If all conditions are false do this code
End If
```

Open **Notepad** or **Notepad++** and type the following code (pay attention to the indentation used). Once the code has been typed up, compile and run the program. Note that the instruction **Main()** will call Sub Main() again, in other words looping the program (this is not ideal, but it works). **MsgBox** is a simple message box that is displayed to the user.

```
Imports System.Console
Public Class UsingIFS
    Shared Sub Main()
        WriteLine("Enter a number")
            Dim y As Integer
            y = ReadLine()
                If y > 3 Then
                    MsgBox("This is bigger than 3")
                ElseIf y > 6 Then
                    MsgBox("This is bigger than 6")
                ElseIf y > 10 Then
                    MsgBox("This is bigger than 10")
                Else
                    MsgBox("Any other response")
                End If
            Main()
    End Sub
End Class
```

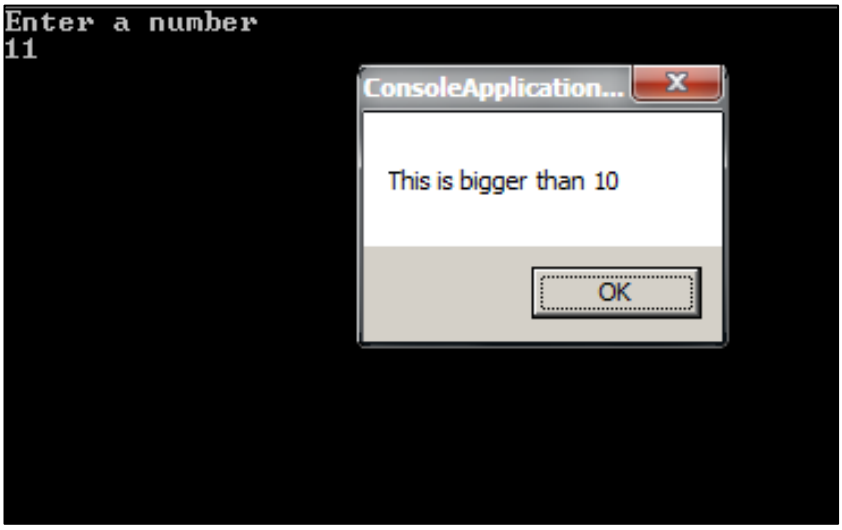
Test Your Skills

- ✓ Once the code has been compiled and run, notice that it does not work as expected, i.e. when entering the value 11, the message 'This is bigger than 3' is displayed, as opposed to 'This is bigger than 10'. See if you can amend the error before continuing.

The previous code did not function as expected because the order of an 'if' statement is important. In the previous example, when the user entered the value 11, the message box 'This is bigger than 3' was displayed to the user, as opposed to 'This is bigger than 10'. This is because an 'if' statement checks all conditions in order (it does not 'pick' the most applicable) and as the first condition is true (10 is bigger than 3), the first message box is displayed. Therefore, to resolve this problem, the 'if' statement must be reversed (as shown below):

```
Imports System.Console
Public Class UsingIFS
    Shared Sub Main()
        WriteLine("Enter a number")
            Dim y As Integer
            y = ReadLine()
                If y > 10 Then
                    MsgBox("This is bigger than 10")
                ElseIf y > 6 Then
                    MsgBox("This is bigger than 6")
                ElseIf y > 3 Then
                    MsgBox("This is bigger than 3")
                Else
                    MsgBox("Any other response")
                End If
            Main()
    End Sub
End Class
```

Program Output:



Using CASE

CASE is an alternative method of selection to an 'if' statement. The principles of CASE are similar to that of an 'if' statement, and both techniques can often be used interchangeably. However, CASE is particularly useful when testing the content of a variable. The logic of a CASE statement is shown below:

```
Select Case [Variable Name]
      Case [if this is the 'case' then]
            Do this code
       Case Else
            If all conditions are false do this code
End Select
```

Open **Notepad** or **Notepad++** and type the following code (pay attention to the indentation used). Once the code has been typed up, compile and run the program.

```
Imports System.Console
Public Class UsingIFS
      Shared Sub Main()
            WriteLine("Please enter a name")
            Dim Name As String
            Name = ReadLine()
               Select Case Name
                        Case "Joe"
                                 MsgBox("Your name is Joe")
                        Case "Neal"
                                 MsgBox("Your name is Neal")
                        Case Else
                                 MsgBox("Don't know name..")
                  End Select
       End Sub
End Class
```

Program Output:

Using Iteration

If you were asked to write a simple program that displays 1 to 5 in the console window, with each number on a separate line, what would your code look like? Would it look like this?

```
Imports System.Console
Public Class UsingIFS
    Shared Sub Main()
        WriteLine("1")
        WriteLine("2")
        WriteLine("3")
        WriteLine("4")
        WriteLine("5")
    End Sub
End Class
```

If your code would be similar to that above, how would you make your program write 1 to 100 or even 1 to 10,000? Let us agree that the previous example is not very efficient! So instead, amend your code so that it is the same as the code below. Once the code has been typed up, compile and run the program.

```
Imports System.Console
Public Class UsingIteration
    Shared Sub Main()
        For X = 1 To 5
            WriteLine(X)
        Next X
    End Sub
End Class
```

Program Output:

Congratulations – you just used a 'for loop' to reduce the amount of code written! Most software languages have some form of 'for loop'; remember, the principles are the same – only the syntax changes. 'For loops' are helpful if the programmer is aware of the number of times they need to execute the same code. A 'for loop' has a starting number and an ending number/condition (that marks the end of the loop); this example will loop from 1 to 5 (in total five times). A 'for loop' also has an increment value; by default this is set to 1. In this example, the integer variable X will initially have the numeric value 1; once the loop executes its containing code once, X will then increment to 2 and will repeat this process until the end condition is met (in this case when X reaches the value 5).

While Loop

Another useful loop is the 'Do While' loop; again, this type of loop is found in most software languages. This loop works by repeating the contained code while a condition is true; the moment the condition becomes false, the loop is ended. The logic of a 'While Loop' is shown below:

```
Do While [this condition is true]
   Repeat this code
Loop
```

'Do While' loops are helpful if the programmer is unsure how many times the code will need to be repeated, as a condition can be used instead of a fixed numeric value. An example of a 'Do While' loop is shown below; try it yourself!

```
Imports System.Console
Public Class UsingIteration
    Shared Sub Main()
        Dim x As Integer = 1
        Do While x < 5
            WriteLine(x)
            x = x + 1
        Loop
    End Sub
End Class
```

This example has a condition that states the variable x must be less than 5. As x has the value 1 to begin with, the given condition is true and thus the code will execute. The second line of code inside the loop (after the **WriteLine** instruction) will increment the value of x by 1, before the loop returns to the start and checks the condition again. This loop will continue until x has the value of 5, because at this point the condition becomes false (if x equals 5 then this is no longer less than 5) and the code ceases to execute. This example above will only loop four times; if the programmer wanted it to run five times then the condition needs to be amended to '**Do While x<=5**'.

Program Output:

Test Your Skills

- ✓ *Replace the keyword **While** with **Until** and replace the less than symbol (<) with the more than symbol (>). What is the difference between **While** and **Until**?*

Nesting

Global and Local Variables

From the previous examples, you are familiar with how to declare a variable in VB.NET:

```
Dim X As Integer = 1
```

However, did you realise there are different ways of declaring a variable? Variables can either be declared as **global** or **local**. Global variables can be accessed by other routines and functions, and thus are declared outside of any routine/function. Local variables however, can only be accessed by the routine/function that they are declared in and thus their life expires when the routine/function ends.

Local Variable Example

```
Shared Sub Main
   Dim Name As String
   Name = "Joe"
   WriteLine(Name)
End Sub

Shared Sub UberProgram
   Name = "James"
   WriteLine(Name)
End Sub
```

Global Variable Example

```
Shared Name As String

Shared Sub Main
   Name = "Joe"
   WriteLine(Name)
End Sub

Shared Sub UberProgram
   Name = "James"
   WriteLine(Name)
End Sub
```

The example above illustrates how the local variable **Name** (which belongs to Sub Main()) cannot be accessed by other functions/procedures, in this scenario Sub UberProgram(). Global variables, however (as illustrated on the right), are visible to all methods. Therefore the second example demonstrates how both Sub Main() and Sub UberProgram() can now use the **Name** variable.

Data Types

Learning the data types available in a programming language is important. You have already used two data types – **Integer** and **String** – but there are many others available, i.e. Boolean, Double, Float, Char, etc. Each data type has a specific purpose; using them correctly will increase your program's performance and reduce programming errors. Conduct some research into VB.NET data types and see what other data types are available to use. Some examples are shown below:

Dim A As String = "Hello There!"
Dim B As Integer = 2
Dim C As Decimal = 200.234
Dim D As Char = "C"
Dim E As Boolean = True
Dim F As Date = "23/01/2011"

Nesting

There are times when a programmer requires extended functionality of their iteration and selection techniques. This functionality can be extended by 'nesting' these techniques inside other techniques. For example, an 'if' statement can be nested inside a 'do while loop' and, further still, nest another 'do while loop' inside the 'if' statement. Below are some examples of nesting – try them yourself!

The example below shows an 'if' statement nested inside another:

```
Dim X As Integer = 1
Dim Y As Integer = 2

    If X = 1 Then
        If Y = 2 Then
            WriteLine("I like cake")
        Else
            WriteLine("I don't like cake")
        End If
    ElseIf X = 2 Then
        WriteLine("I like cheese")
    Else
        WriteLine("Invalid Input")
    End If
```

Here is a second example; however, this time it illustrates an 'if' statement inside a 'do while loop':

```
Dim X As Integer = 0
    Do While X < 1
        Dim Y As Integer = 0
        Y = ReadLine()
            If Y = 1 then
                X = X + 1
            End If
    Loop
```

Here is a third example. This example demonstrates a loop inside a loop:

```
Dim X As Integer = 0
    Do While X < 5
        WriteLine("")
        Dim Y As Integer = 0
            Do While Y < 5
                Write("*")
                Y = Y + 1
            Loop
        X = X + 1
    Loop
```

Methods and Functions

Methods

In programming, a method can be defined as a block of code that contributes to a whole program – a block that performs a specific task or function. Programs can be written in a single (and inflexible) method; for example, most of your applications so far have been written in a single method: Shared Sub Main(). However, using a single method will put limitations on any application, as well as making it difficult to debug, maintain and upgrade/adapt in the future. Breaking an application's functionality down into separate methods and applicable modules/objects will help eliminate some of these problems. In theory, it should make applications more robust, efficient and maintainable. An example of separating code in separate methods is shown below:

```
Shared NumberX As Integer
    Shared Sub Main()
            NumberX = 10
            JoeBloggs()
    End Sub
    Shared Sub JoeBloggs()
        NumberX = NumberX + 10
        WriteLine(NumberX)
    End Sub
```

In the previous example, a global variable (NumberX) was used to pass information between two methods. This is not an efficient technique of passing information between methods and can cause problems later on in the development of applications (especially as they become more complex). Instead, 'arguments' and 'parameters' can be used to pass information efficiently between methods. A **parameter** is what the method expects. An **argument** is what is passed to the accepting method. An example of this is shown below; this time the method JoeBloggs() expects an Integer value to be passed (the method will not run unless it is given the required parameters). When JoeBloggs() is called in Sub Main(), the value 10 is passed as an argument into the method JoeBloggs() where it is now stored in the variable NumberX ready for use by the method.

```
Shared Sub Main()
        JoeBloggs(10)
    End Sub
    Shared Sub JoeBloggs(NumberX As Integer)
        NumberX = NumberX + 10
        WriteLine(NumberX)
    End Sub
```

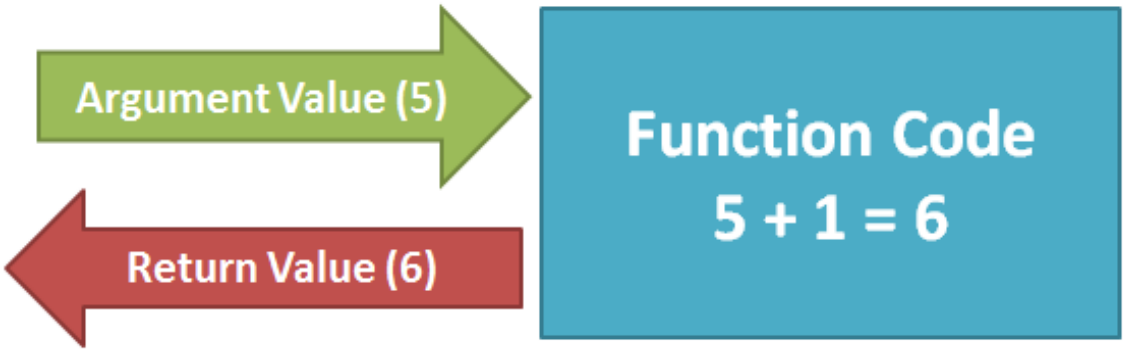
Please note that arguments are passed in order from left to right and do not rely on any naming conventions. Therefore, a second parameter would take the second given argument. A comma is used to separate multiple arguments and parameters.

Test Your Skills

- ✓ *Write a short application that consists of two methods. The application will need to:*
  1. *Read two values from the user in Sub Main()*
  2. *The application must then pass the two values (from the user) to a second method*
  3. *The second method must then add those two values together and display the answer to the user*

Functions

Functions are like ATMs: input a card, a pin number and an amount (arguments) and receive a cash value on return. An example of a function is shown below;



In VB.NET, functions have a different purpose to methods. When using a method, arguments can be passed to it and the method may use the arguments in its own execution. This works fine, but on occasion the programmer may require the method to return a value and this cannot be achieved when using Subs (methods). However, in VB.NET functions can be used to get a return value – functions are the same as methods in every principle, except that they have to return a value before finishing. Please be aware that a function can only return one value; however, a single object or array can be used to return multiple values.

There are two types of functions available: Utility Functions and Object Functions. An example of an Object Function and a Utility Function are shown below:

- A utility function (sometimes known as static) can be addressed directly from the code. This function converts a String variable's content to upper case and returns it:

```
Variable = Variable.ToUpper
```

- An object utility is object-orientated and thus has to be initiated first before it can be used. This function generates a random number between 1 and 50 and returns it:

```
Dim RandomClass As New Random()
    Dim answer as Integer
    answer = RandomClass.Next(1, 51)
```

A user does not need to use pre-built functions; they can also define their own. It is considered good practice to define code into its own function if it is an operation that is going to be performed multiple times by the same (or another) program. Below is an example of a self-defined function:

```
Shared Sub Main()
            Dim NumberX As Decimal
              NumberX = Test(10)
              WriteLine(NumberX)
    End Sub
Shared Function Test(X As Decimal) As Decimal
            Dim Answer As Decimal
              Answer = X + 5
            Return Answer
    End Function
```

There are some key differences to notice when declaring a method and declaring and a function. First is that the keyword '**Function**' is used instead of the keyword '**Sub**'. Another difference is that it is important to state the data type of the return value; this is achieved by adding the data type at the end of the function declaration, i.e. 'Shared Function Test(X As Decimal) **As Decimal**'. The last difference is that a function must have a return value; this is achieved by using the keyword '**Return**'. In this case the variable **Answer** is returned. If the code above was to run, what would be the value of **Answer**?

Test Your Skills

- ✓ *Write a short application that consists of one method and one function. The first method (Sub Main()) should ask the user for two values. The application should then pass those two values to a separate function which adds them together and returns the answer to Sub Main(). The method Sub Main() should then write the answer to the screen.*

Error Handling

Try Catch

Making programs as robust as possible is very important in software development; nobody wants a reputation for writing buggy software! Applications should be bug-free and also flexible enough to handle any unexpected exceptions that may occur. There is nothing more frustrating to a software user than an application that continuously crashes.

In VB.NET (and many other languages), 'Try Catch' is a programming technique that is used to handle any exceptions that may arise during the execution of an application. This technique can be used to catch exceptions, such as String conversion to Integer-type error, which would normally cause an application to crash.
The example below demonstrates that if a String value is entered instead of an Integer value, the 'Try Catch' will handle the exception without crashing the program. Try it yourself both with and without the 'Try Catch' syntax.
Please be aware that this is only a basic example; this technique is especially useful when working with the unknown, i.e. file handling, as a file may or may not be available.

```
Try
        Dim X As Integer
        X = ReadLine()
        WriteLine(X)
Catch ex As Exception
        WriteLine("Problem Caught here")
Finally
        WriteLine("This will always happen!")
End Try
```

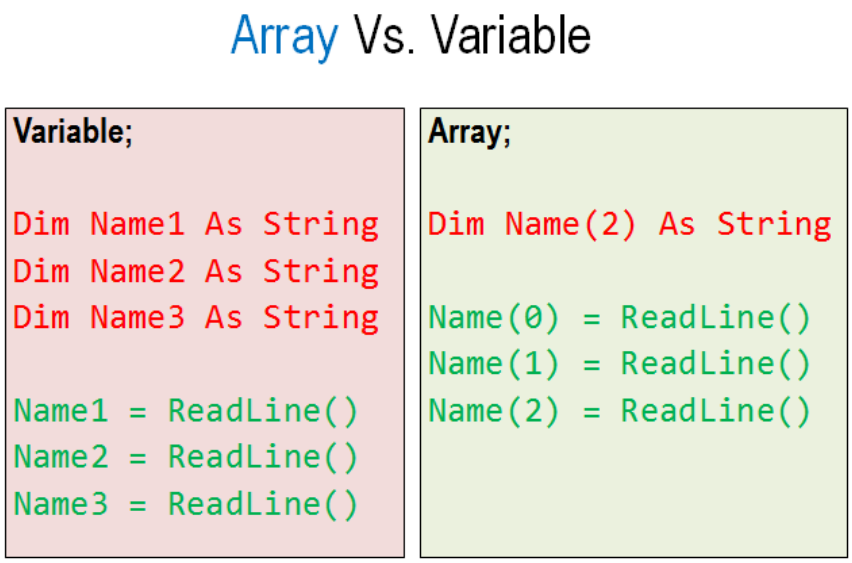
Using Arrays

What is an Array?

Currently you have been using variables to store one piece of information at a time; for example:

```
Dim X As Integer = 5
```

Although variables are useful, storing only one piece of information at each instance will make some tasks very tedious. Fortunately, an array is a type of variable that allows a programmer to store multiple values as opposed to a single value. Arrays are available in most programming languages and there are also many different types; we shall focus on native arrays.



Open Notepad or Notepad++ and type the following code (pay attention to the indentation used). Once the code has been typed up, compile and run the program. In this example, a one-dimensional array is used to store three name items. A one-dimensional array could be visualised as a single column in a spreadsheet. Notice that arrays always start at position 0, not position 1.

```
Imports System.Console
Public Class UsingArrays
    Shared Sub Main()
        Dim FirstNames(2) As String
        FirstNames(0) = "Joe"
        FirstNames(1) = "James"
        FirstNames(2) = "Alex"
        WriteLine(FirstNames(0))
        WriteLine(FirstNames(1))
        WriteLine(FirstNames(2))
    End Sub
End Class
```

Test Your Skills

- ✓ *Write a short application that stores your favourite films in a one-dimensional array and then writes your list of favourite films back to the user.*

Arrays and Loops

Outputting the results of an entire array individually (i.e. displaying each array item using separate **WriteLine** commands) can be a tedious process, and is inefficient and costly (in terms of time and LOC (Lines Of Code)). Fortunately, iteration (loops) can be used to output multiple items of an array in just a few lines of code; most programming languages support this functionality. An example of this is shown below; now try it yourself!

```
Shared Sub Main()
        Dim FirstNames(3) As String
        FirstNames(0) = "Joe"
        FirstNames(1) = "James"
        FirstNames(2) = "Alex"
        FirstNames(3) = "Bobby"
        For X = 0 To 3
                WriteLine(FirstNames(X))
        Next X
End Sub
```

Determining the length of an array is important because there are occasions when a programmer will not know the specific number of items in an array. There is a function that is part of the **array** class in VB.NET (and the same principle in other languages) that allows a programmer to determine the number of items in an array. The function in VB.NET is called **UBound()**. The first argument in **UBound()** specifies the name of the array; the second specifies the dimension. An example of the function is shown below:

```
Dim X As Integer = UBound(ArrayName, 1)
```

This example shows the **UBound()** function implemented in code:

```
Shared Sub Main()
        Dim FirstNames(2) As String
        FirstNames(0) = "Joe"
        FirstNames(1) = "James"
        FirstNames(2) = "Alex"
        For X = 0 To UBound(FirstNames, 1)
            WriteLine(FirstNames(X))
        Next X
End Sub
```

The following example uses two loops: one to read the values from the user to the array, and another to write the responses back to the screen; try this one yourself!

```
Shared Sub Main()
        Dim FirstNames(4) As String
        Dim X As Integer = 0
        WriteLine("Please enter five first names;")
            For X = 0 To UBound(FirstNames, 1)
                FirstNames(X) = ReadLine()
                X = X + 1
            Next X
        WriteLine("These are your first names;")
            For X = 0 To UBound(FirstNames, 1)
                WriteLine(FirstNames(X))
            Next X
End Sub
```

Two-Dimensional Arrays

Arrays do not have to be one-dimensional; in fact, they can have numerous dimensions. A two-dimensional array is similar to that of a spreadsheet; it is like a grid in memory that consists of rows and columns, and each cell has a unique reference. As arrays start at 0, a 3-by-3 two-dimensional array could be visualised similar to that of the image below. A coded example has also been provided.

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

```
    Shared Sub Main()
        Dim Details(2, 2) As String
        WriteLine("Please enter two first names;")
                Details(0, 0) = ReadLine()
                Details(1, 0) = ReadLine()
        WriteLine("Please enter two last names;")
                Details(0, 1) = ReadLine()
                Details(1, 1) = ReadLine()
        WriteLine("Please enter two ages;")
                Details(0, 2) = ReadLine()
                Details(1, 2) = ReadLine()
        WriteLine("Here are your details;")
        WriteLine(Details(0, 0) & " " & Details(0, 1) & " " & Details(0, 2))
        WriteLine(Details(1, 0) & " " & Details(1, 1) & " " & Details(1, 2))
End Sub
```

File Handling

Reading Files

File handling is an important aspect of programming, as many programs need to be able to interact with different file types, whether this is to save content or to load content. In VB.NET, this can be achieved by using pre-built objects called **StreamReader** to read, and **StreamWriter** to write.

The example below will read the content from a text file (in this case 'test.txt') and display the content in the console window. Note that the variable **File** is used to contain the directory of the file to be read; this is then passed to the **StreamReader** when it is created. The keyword **New** is used to create a new instance of the StreamReader before it can be used; the new instance is called RDR. The **StreamReader's** method **ReadToEnd()** is then called to read the content of the text file and display it to the console window. The method **Close()** is then called to destroy the object as it is no longer needed.

```
Shared Sub Main()
    Dim File As String = "U:\test.txt"
    Dim RDR As New System.IO.StreamReader(File)
  WriteLine(RDR.ReadToEnd)
RDR.Close()
End Sub
```

Writing Files

Writing content to a text file is similar to reading it, except that we use the **StreamWriter** object as opposed to the **StreamReader**. The example below will write the line 'I Like Pie' twice to the text file 'test.txt'. Note that, similarly to **StreamReader**, to use **StreamWriter** a new instance is required as well as the directory of the text file. The **Write()** method is used to write the content to the text file. The instruction '**System.IO.File.Exists(FILE)**' is used to determine if the file exists first, because if it did not it would cause a runtime error.

```
Shared Sub Main()
 Dim FILE As String = "U:\test.txt"
        If System.IO.File.Exists(FILE) = True Then
                Dim objWriter As New System.IO.StreamWriter(FILE)
                objWriter.Write("I Like Pie" & vbNewLine)
                                objWriter.Write("I Like Pie" & vbNewLine)
                objWriter.Close()
                MsgBox("Text written to file")
        Else
                MsgBox("File Does Not Exist")
        End If
End Sub
```