Government Science College, Valod

Visual Basic Language

By

Mayurkumar Marolia

Program Structure

```
Module Module1
    Sub Main()
        System.Console.WriteLine("Hello from Visual Basic")
    End Sub
End Module
```

Visual Basic Statements

- **Visual Basic Statements**
- A Visual Basic statement is a complete instruction. It can contain:
- *keywords*—Words reserved for Visual Basic's use.
- *operators*—Symbols used to perform operations, like **+**, which performs addition operations; -, which performs subtraction operations; and so on.
- *variables*—Symbolic names given to values stored in memory and declared with the **Dim** keyword. For example, if you've declared a variable named **temperature** as an Integer type, you can store integer values like **72** or **83** in it.
- *literal values*—Simple values, like **5** or "**Hello**."
- *constants*—The same as variables, except that constants are assigned a value that cannot then be altered.
- *expressions*—Combinations of terms and/or keywords that yield a value. For example, if the variable **temperature** holds the value **72**, then the expression **temperature + 3** yields the value **75**.

Declaration Statement

- Each statement is one of the following:
- A *declaration statement*, which can name and create a
  - variable,
  - constant, or
  - procedure

    and can also specify a *data type*.
- The data type can be **Boolean**, **Byte**, **Char**, **Date**, **Decimal**, **Double**, **Integer**, **Long**, **Object**, **Short**, **Single**, or **String**; or the name of an enumeration (a series of constants defined at the same time), structure, class, or interface.

Executable Statements

- An *executable statement*, which can perform an action. These statements can execute a
  - Method or function
  - loop through code using one of the Visual Basic *loops*
  - an *assignment statement*, which assign a value or expression to a variable or constant such as this: **temperature = 72**, and so on.

Example of Statements

```
Module Module1
    Sub Main()
        Dim BankBalance As Single = 500.01 'Declaration Statement
        If (BankBalance < 0) Then 'Creating a block of statement
                'Declaration with Assignment Statements
                Dim strError As String strError = "Hey, your bank balance is negative!"
                    System.Console.WriteLine(strError) 'Executable Statement
        End if 'End of Statement Block
    End Sub
End Module
```

Dim Statement

- [and], for optional items, and curly braces, { and }, to indicate that you select one of the enclosed items, like this for the **Dim** statement:
- **[<*attrlist*>] [{ Public | Protected | Friend | Protected Friend | Private | Static }] [Shared] [Shadows] [ReadOnly] Dim [WithEvents] *name*[(boundlist)] [As [New] *type*] [= *initexpr*]**

Attributes of Identifiers

- ***attrlist***-A list of attributes that apply to the variables you're declaring in this statement.
  - You separate multiple attributes with commas.
- **Public**-Gives variables public access,
  - no restrictions
  - use **Public** only at module, namespace, or file level
  - if you specify **Public**, you can omit the **Dim** keyword if you want to.
- **Protected**-Gives variables protected access,
  - accessible only from base class or derived class
  - You can use **Protected** only at class level (which means you can't use it inside a procedure).
  - Note that if you specify **Protected**, you can omit the **Dim** keyword if you want to.

Attributes of Identifiers

- **Friend-**Gives variables friend access,
  - accessible from within the program that contains their declaration or from same assembly.
  - **Friend** can be used only at module, namespace, or file level (which means you can't use it inside a procedure).
  - Note that if you specify **Friend**, you can omit the **Dim** keyword if you want to.
- **Protected Friend-**Gives variables both protected and friend access,
  - which means they can be used by code in the same assembly, as well as by code in derived classes.
- **Private-**Gives variables private access,
  - accessible from a class where it declared, including any nested procedures.
  - You can use **Private** only at module, namespace, or file level (which means you can't use it inside a procedure).
  - Note that if you specify **Private**, you can omit the **Dim** keyword if you want to.
- **Static-**Makes variables static, which means they'll retain their values, even after the procedure in which they're declared ends.
  - You can declare static variables inside a procedure or a block within a procedure, but not at class or module level.
  - Note that if you specify **Static**, you can omit the **Dim** keyword if you want to, but you cannot use either **Shadows** or **Shared**.

Attributes of Identifiers

- **Shared**-Declares a shared variable, which means it is not associated with a specific instance of a class or structure, but can be shared across many instances.
  - You access a shared variable by referring to it either with its class or structure name, or with the variable name of an instance of the class or structure.
  - You can use **Shared** only at module, namespace, or file level (but not at the procedure level).
  - Note that if you specify **Shared**, you can omit the **Dim** keyword if you want to.
- **Shadows**-Makes this variable a shadow of an identically named programming element in a base class.
  - A shadowed element is unavailable in the derived class that shadows it.
  - You can use **Shadows** only at module, namespace, or file level (but not inside a procedure).
  - This means you can declare shadowing variables in a source file or inside a module, class, or structure, but not inside a procedure.
  - Note that if you specify **Shadows**, you can omit the **Dim** keyword if you want to.
- **ReadOnly**-Means this variable only can be read and not written.
  - This can be useful for creating constant members of reference types, such as an object variable with preset data members.
  - You can use **ReadOnly** only at module, namespace, or file level (but not inside procedures).
  - Note that if you specify **ReadOnly**, you can omit the **Dim** keyword if you want to.

Attributes of Identifiers

- **WithEvents**-Specifies that this variable is used to respond to events caused by the instance that was assigned to the variable.
  - Note that you cannot specify both **WithEvents** and **New** in the same variable declaration.
- ***name***-The name of the variable.
  - You separate multiple variables by commas.
  - If you specify multiple variables, each variable is declared of the data type given in the first **As** clause encountered after its *name* part.
- ***boundlist***-Used to declare arrays;
  - gives the upper bounds of the dimensions of an array variable.
  - Multiple upper bounds are separated by commas. An array can have up to 60 dimensions.
- **New**-Means you want to create a new object immediately.
  - If you use **New** when declaring an object variable, a new instance of the object is created.
  - Note that you cannot use both **WithEvents** and **New** in the same declaration.

Attributes of Identifiers

- ***type***-The data type of the variable.
  - Can be **Boolean**, **Byte**, **Char**, **Date**, **Decimal**, **Double**, **Integer**, **Long**, **Object**, **Short**, **Single**, or **String**; or the name of an enumeration, structure, class, or interface.
  - To specify the type, you use a separate **As** clause for each variable, or you can declare a number of variables of the same type by using common **As** clauses.
  - If you do not specify *type*, the variable takes the data type of *initexpr*.
  - Note that if you don't specify either *type* or *initexpr*, the data type is set to **Object**.
- ***initexpr***-An initialization expression that is evaluated and the result is assigned to the variable when it is created.
- Note that if you declare more than one variable with the same **As** clause, you cannot supply *initexpr* for those variables.

Overview of Class, Module, Procedures, Functions, Methods

- ***procedure***-A callable series of statements that may or may not return a value.
- ***sub procedure***-A procedure that does not return a value.
- ***function***-A procedure that returns a value.
- ***method***-A procedure that is built into a class.
- ***constructor***-A special method that is automatically called when you create an object from a class. Used to initialize and customize the object. You can pass data to constructor methods just like other methods.

Overview of Class, Module, Procedures, Functions, Methods

- ***module***-Visual Basic modules are designed to hold code-that is, to separate the code in them from other, possibly conflicting, code. Their main purpose is to make your code more modular, and if your program is a long one, you should consider breaking it up into modules.
- ***class***-This is an OOP class, which can contain both code and data; you use classes to create objects. A class can have *members*, which are elements that can be accessible from outside the class if you so specify. Data members are called *fields*, procedure members are called *methods*.
- ***object***-This is an *instance* of a class, much like a variable is an instance of a data type.

Overview of Class, Module, Procedures, Functions, Methods

- ***shared or static members and instance members***-Fields and methods that apply to a class and are invoked with the class name are called *shared* or *static* fields and methods; the fields and methods that apply to objects created from the class are called *instance* fields and methods.
- ***structure***-Also an OOP element, just like a class but with some additional restrictions. In VB6 and earlier, you used structures to create user-defined types; now they are another form of OOP classes.

Option Statements

- You use **Option** statements to set the "ground rules" for your code, helping prevent syntax and logic errors.
- **Option Explicit—** Set to **On** or **Off**. **On** is the default. Requires declaration of all variables before they are used (this is the default).
- **Option Compare—** Set to **Binary** or **Text**. This specifies if strings are compared using binary or text comparison operations.
- **Option Strict—** Set to **On** or **Off**. **Off** is the default.
  - When you assign a value of one type to a variable of another type Visual Basic will consider that an error if this option is on and there is any possibility of data loss, as when you're trying to assign the value in a variable to a variable of less precise data storage capacity.
  - In that case, you must use explicit conversion functions of the kind we'll see in this chapter, like **CLng**.

Example

```
Option Strict Off

Module Module1

    Sub Main()

        System.Console.WriteLine("Hello from Visual Basic")

    End Sub

    End Module
```

Imports Statements

- You use **Imports** statements to import a namespace so you don't have to qualify items in that namespace by listing the entire namespace when you refer to them.
- Example

```
Option Strict Off

Module Module1

        Sub Main()

                System.Console.WriteLine("Hello from Visual Basic")

        End Sub

End Module
```

Imports Statements

```
Option Strict Off

Imports System.Console

Module Module1

        Sub Main()

                WriteLine("Hello from Visual Basic")

        End Sub

End Module
```

Creating Enumerations

- you can create an *enumeration*, which is a related set of constants. You create enumerations with the **Enum** statement at module, class, structure, procedure, or block level:
- Syntax:

[<*attrlist*>] [{ Public | Protected | Friend | Protected Friend | Private }]

[Shadows] Enum *name* [As *type*]

[<*attrlist1* >] *membname1* [= *initexpr1*]

[<*attrlist2* >] *membname2* [= *initexpr2*]

⋮

[<*attrlistn*>] *membnamen* [= *initexprn*]

End Enum

Creating Enumerations

- Example

```
Module Module1
        Enum Days
                Sunday = 1
                Monday = 2
                Tuesday = 3
                Wednesday = 4
                Thursday = 5
                Friday = 6
                Saturday = 7
        End Enum
        Sub Main()
                System.Console.WriteLine("Friday is day " & Days.Friday)
        End Sub
End Module
```

Data Types

Type	Storage size	Value range
Boolean	2 bytes	True or False
Byte	1 byte	0 to 255 (unsigned)
Char	2 bytes	0 to 65535 (unsigned)
Date	8 bytes	January 1, 0001 to December 31, 9999
Decimal	16 bytes	+/- 79,228,162,514,264,337,593,543,950,335 with no decimal point; 7.92281625142643 37593543950335 with 28 places to the right of the decimal; smallest non-zero number is 0.00000 00000000000000000001

Data Types

Type	Storage size	Value range
Double	8 bytes	-1.79769313486231E+308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486231E+308 for positive values
Integer	4 bytes	-2,147,483,648 to 2,147,483,647
Long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Object	4 bytes	Any type can be stored in a variable of type Object

Data Types

Type	Storage size	Value range
Short	2 bytes	-32,768 to 32,767
Single	4 bytes	-3.402823E to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E for positive values
String	Depends on implementing platform	0 to approximately 2 billion Unicode characters
User-Defined Type (structure)	Sum of the sizes of its members. Each member of the structure has a range determined by its data type and independent of the ranges of the other members	

Variable prefix

Data type	Prefix
Boolean	bln
Byte	byt
Collection object	col
Date (Time)	dtm
Double	dbl
Error	err
Integer	int
Long	lng
Object	obj
Single	sng
String	str
User-defined type	udt

Default values for data types

- **0** for all numeric types (including **Byte**).
- Binary **0** for **Char**.
- **Nothing** for all reference types (including **Object**, **String**, and all arrays). **Nothing** means there is no object associated with the reference.
- **False** for **Boolean**.
- **12:00 AM of January 1 of the year 1** for **Date**.

Conversion Functions

1. **CBool—** Convert to **Bool** data type.
2. **CByte—** Convert to **Byte** data type.
3. **CChar—** Convert to **Char** data type.
4. **CDate—** Convert to **Date** data type.
5. **CDbl—** Convert to **Double** data type.
6. **CDec—** Convert to **Decimal** data type.
7. **CInt—** Convert to **Int** data type.
8. **CLng—** Convert to **Long** data type.
9. **CObj—** Convert to **Object** type.
10. **CShort—** Convert to **Short** data type.
11. **CSng—** Convert to **Single** data type.
12. **CStr—** Convert to **String** type.

Ctype Function

- If you can't remember the name of a particular conversion function, you also can use the **CType** function, which lets you specify a type to convert to.

```
Option Strict On

Module Module1

        Sub Main()

                Dim dblData As Double

                Dim intData As Integer dblData = 3.14159

                intData = CType(dblData, Integer)

                System.Console.WriteLine("intData = " & Str(intData))

        End Sub

End Module
```

Data Conversion Functions

To convert	Use this
Character code to character	**Chr**
String to lowercase or uppercase	**Format**, **LCase**, **UCase**, **String.ToUpper**, **String.ToLower**, **String.Format**
Date to a number	**DateSerial**, **DateValue**
Decimal number to other bases	**Hex**, **Oct**
Number to string	**Format**, **Str**

Data Conversion Functions

To convert	Use this
One data type to another	**CBool**, **CByte**, **CDate**, **CDbl**, **CDec**, **CInt**, **CLng**, **CObj**, **CSng**, **CShort**, **CStr**, **Fix**, **Int**
Character to character code	**Asc**
String to number	**Val**
Time to serial number	**TimeSerial**, **TimeValue**

Checking Data Types

Function	Does this
IsArray()	Returns **True** if passed an array
IsDate()	Returns **True** if passed a date
IsDBNull()	Returns **True** if passed a database **NULL** value; that is, a **System.DBNull** value
IsError()	Returns **True** if passed an error value
IsNumeric()	Returns **True** if passed an numeric value
IsReference()	Returns **True** if passed an **Object** variable that has no object assigned to it; otherwise, returns **False**

Arrays

- Arrays are programming constructs that let you access your data by numeric index.
- To dimension arrays, you can use
  - **Dim** (standard arrays),
  - **ReDim** (dynamic arrays),
  - **Static** (arrays that don't change when between calls to the procedure they're in),
  - **Private** (arrays private to the form or module they're declared in),
  - **Protected** (arrays restricted to a class or classes derived from that class),
  - **Public** (arrays global to the whole program),

Standard Arrays

- You usually use the **Dim** statement to declare a standard array; here are a few examples of standard array declarations:

```
Dim Data(30)

Dim Strings(10) As String

Dim TwoDArray(20, 40) As Integer

Dim Bounds(10, 100)
```

- The **Data** array now has 30 elements, starting from **Data(0)**, which is how you refer to the first element, up to **Data(29)**.
- 0 is the *lower bound* of this array, and 29 is the *upper bound*.
- The **Bounds** array has two indices, one of which runs from 0 to 9, and the other of which runs from 0 to 99.
- You can also initialize the data in an array if you don't give an array an explicit size; here's the syntax to use, where I'm initializing an array with the values **10**, **3**, and **2**:

```
Dim Data() = {10, 3, 2}
```

Dynamic Arrays

- You can use the **Dim** statement to declare an array with empty parentheses to declare a *dynamic array*.
- Dynamic arrays can be dimensioned or redimensioned as you need them with the **ReDim** statement (which you must also do the first time you want to use a dynamic array). Here's how you use **ReDim**:

  ReDim [Preserve] *varname*(*subscripts*)

  - You use the **Preserve** keyword to preserve the data in an existing array.
  - The *varname* argument holds the name of the array to (re)dimension. The *subscripts* term specifies the new dimension of the array.

```
Dim DynaStrings() As String
ReDim DynaStrings(10)
DynaStrings(0) = "String 0" 'Need more data space!
ReDim DynaStrings(100)
DynaStrings(50) = "String 50"
```

Handling Strings

- Strings are supported by the .NET **String** class in Visual Basic.

```
Dim strText As String

Dim myString As String = "Welcome to Visual Basic"
```

```
Option Strict On Module Module1 Sub Main()
Dim strText1 As String = "welcome to visual
basic" Dim strText2 As String Dim strText3 As
String strText2 = UCase(strText1) strText3 =
strText1.ToUpper
System.Console.WriteLine(strText2)
System.Console.WriteLine(strText3) End Sub End
Module
```

```
Module Module1 Sub Main() Dim strText1 As String = "Hey, look here!" Dim strText2 As String Dim strText3 As String strText2 = Mid(strText1, 6, 4) strText3 = strText1.Substring(5, 4) System.Console.WriteLine(strText2) System.Console.WriteLine(strText3) End Sub End Module
```

String Handling Functions and Methods

To do this	Use this
Concatenate two strings	**&**, **+**, **String.Concat**, **String.Join**
Compare two strings	**StrComp**, **String.Compare**, **String.Equals**, **String.CompareTo**
Convert strings	**StrConv**, **CStr**, **String. ToString**
Copying strings	**=**, **String.Copy**
Convert to lowercase or uppercase	**Format**, **Lcase**, **Ucase**, **String.Format**, **String. ToUpper**, **String. ToLower**
Convert to and from numbers	**Str**, **Val.Format**, **String.Format**
Create string of a repeating character	**Space**, **String**, **String.String**

String Handling Functions and Methods

To do this	Use this
Create an array of strings from one string	**String.Split**
Find length of a string	**Len**, **String.Length**
Format a string	**Format**, **String.Format**
Get a substring	Mid, String.SubString
Insert a substring	**String.Insert**
Justify a string with padding	**LSet**, **Rset**, **String.PadLeft**, **String.PadRight**
Manipulate strings	**InStr**, **Left**, **LTrim**, **Mid**, **Right**, **RTrim**, **Trim**, **String.Trim**, **String.TrimEnd**, **String.TrimStart**

String Handling Functions and Methods

To do this	Use this
Remove text	**Mid**, **String.Remove**
Replace text	**Mid**, **String.Replace**
Set string comparison rules	Option Compare
Search strings	**InStr**, **String.Chars**, **String.IndexOf**, **String.IndexOfAny**, **String.LastIndexOf**, **String.LastIndexOf Any**
Trim leading or trailing spaces	**LTrim**, **RTrim**, **Trim, String.Trim**, **String.TrimEnd**, **String.TrimStart**
Work with character codes	**Asc**, **AscW**, **Chr**

- to concatenate (join) strings of multiline together, you can use the **&** or **+** operators

```
Dim Msg As String Msg = "Well, there is a problem " _
        & "with your program. I am not sure " _
        & "what the problem is, but there is " _
        & "definitely something wrong."
```

Converting Strings to Numbers and Back Again

```
Module Module1

    Sub Main()

        Dim strText1 As String = "1234"

        Dim intValue1 As Integer intValue1 = Val(strText1)

        strText1 = Str(intValue1)

        System.Console.WriteLine(strText1)

    End Sub

End Module
```

Converting between Characters and Character Codes

- The characters a program stores internally are stored using Unicode character codes; for example, the character code 65 stands for "A". How can you convert back and forth between characters and character codes? You can use the **Asc** and **Chr** functions:
- **Asc—** Takes a character and returns its character code. For example, **Asc("A")** returns 65.
- **Chr—** Takes a character code and returns the corresponding character. For example, **Chr(65)** returns "A".

Arithmetic Operators

1. **^** Exponentiation
2. ***** Multiplication
3. **/** Division
4. **** Integer division
5. **Mod** Modulus
6. **+** Addition
7. - Subtraction

Assignment Operator

1. **=** Assignment
2. **^=** Exponentiation followed by assignment
3. ***=** Multiplication followed by assignment
4. **/=** Division followed by assignment
5. **\=** Integer division followed by assignment
6. **+=** Addition followed by assignment
7. **-=** Subtraction followed by assignment
8. **&=** Concatenation followed by assignment

Comparison Operator

1. **<** (Less than)—**True** if *operand1* is less than *operand2*
2. **<=** (Less than or equal to)—**True** if *operand1* is less than or equal to *operand2*
3. **>** (Greater than)—**True** if *operand1* is greater than *operand2*
4. **>=** (Greater than or equal to)—**True** if *operand1* is greater than or equal to *operand2*
5. **=** (Equal to)—**True** if *operand1* equals *operand2*
6. **<>** (Not equal to)—**True** if *operand1* is not equal to *operand2*
7. **Is**—**True** if two object references refer to the same object
8. **Like**—Performs string pattern matching

String Concatenation operators

- **&** String concatenation
- **+** String concatenation

Logical/Bitwise operators

- **And—** Performs an **And** operation (for logical operations: **True** if both operands are **True**, **False** otherwise; the same for bit-by-bit operations where you treat **0** as **False** and **1** as **True**).
- **Not—** Reverses the logical value of its operand, from **True** to **False** and **False** to **True**, for bitwise operations, turns **0** into **1** and **1** into **0**.
- **Or—** Operator performs an **Or** operation (for logical operations: **True** if either operand is **True**, **False** otherwise; the same for bit-by-bit operations where you treat **0** as **False** and **1** as **True**).
- **Xor—** Operator performs an **exclusive-Or** operation (for logical operations: **True** if either operand, but not both, is **True**, and **False** otherwise; the same for bit-by-bit operations where you treat **0** as **False** and **1** as **True**).
- **AndAlso—** Operator **A** "short circuited" **And** operator; if the first operand is **False**, the second operand is not tested.
- **OrElse—** Operator **A** "short circuited" **Or** operator, if the first operand is **True**, the second is not tested.

miscellaneous operators

- **AddressOf—** Gets the address of a procedure.
- **GetType—** Gets information about a type.

Understanding Visual Basic Operator Precedence

```
Module Module1
        Sub Main()
                Dim intGrade1, intGrade2, intGrade3, intNumberStudents As Integer
                intGrade1 = 60
                intGrade2 = 70
                intGrade3 = 80
                intNumberStudents = 3
                System.Console.WriteLine("Average grade = " & _ Str(intGrade1 +
intGrade2 + intGrade3 / intNumberStudents))
        End Sub
End Module
```

Understanding Visual Basic Operator Precedence

- Arithmetic operators
- Concatenation operators
- Comparison operators
- Logical/Bitwise operators

Arithmetic Operator Priority

1. Exponentiation (**^**)
2. Negation (-) (for example, **-intValue** reverses the sign of the value in **intValue**)
3. Multiplication and division (*, */*)
4. Integer division (\)
5. Modulus arithmetic (**Mod**)
6. Addition and subtraction (**+**,-)

Concatenation Operator Priority

1. String concatenation (**+**)
2. String concatenation (**&**)

Comparison Operator Priority

1. Equality (**=**)
2. Inequality (**<>**)
3. Less than, greater than (**<**,**>**)
4. Greater than or equal to (**>=**)
5. Less than or equal to (**<=**)
6. **Like**
7. **Is**

Logical/Bitwise Operator Priority

1. Negation-(**Not**)
2. Conjunction-(**And**,**AndAlso**)
3. Disjunction-(**Or**, **OrElse**, **Xor**)

Making Decision with If…Else Statements

- **If** statement, which is the bread-and-butter of Visual Basic conditionals

```
If condition Then
        [statements]
[ElseIf condition-n Then
        [elseifstatements] …]
[Else
        [elsestatements]]
End If
```

Example

```
Module Module1
        Sub Main()
                Dim intInput As Integer
                System.Console.WriteLine("Enter an integer…")
                intInput = Val(System.Console.ReadLine())
                If intInput = 1 Then
                        System.Console.WriteLine("Thank you.")
                ElseIf intInput = 2 Then
                        System.Console.WriteLine("That's fine.")
                ElseIf intInput = 3 Then
                        System.Console.WriteLine("Too big.")
                Else
                        System.Console.WriteLine("Not a number I know.")
                End If
        End Sub
End Module
```

Using Select Case

Select Case *testexpression*

[Case *expressionlist-n* [*statements-n*]]

…

[Case Else [*elsestatements*]]

End Select

Example

```
Module Module1

    Sub Main()

        Dim intInput As Integer

        System.Console.WriteLine("Enter an integer…")

        intInput = Val(System.Console.ReadLine())

        Select Case intInput

            Case 1

                System.Console.WriteLine("Thank you.")

            Case 2

                System.Console.WriteLine("That's fine.")

             Case 3

                System.Console.WriteLine("OK.")

            Case 4 To 7

                System.Console.WriteLine("In the range 4 to 7.")

            Case Is > 7

                System.Console.WriteLine("Definitely too big.")

            Case Else

                System.Console.WriteLine("Not a number I know.")

        End Select

    End Sub

End Module
```

The Switch Function

- The **Switch** function evaluates a list of expressions and returns an **Object** value or an expression associated with the first expression in the list that is true. Here's the syntax:

  Switch(*expr-1, value-1*[, *expr-2, value-2* … [, *expr-n, value-n*]])

- In this case, *expr-1* is the first expression to evaluate; if true, **Switch** returns ***value-1***. If *expr-1* is not true but *expr-2* is, **Switch** returns ***value-2*** and so on.

```
intAbsValue = Switch(intValue < 0, -1 * intValue, intValue >= 0, intValue)
```

The Choose Function

- You use the **Choose** function to return one of a number of choices based on an index. Here's the syntax:

  Choose(*index*, *choice-1*[, *choice-2*, … [, *choice-n*]])

- If the index value is **1**, the first choice is returned, if index equals **2**, the second choice is returned, and so on. Here's an example using **Choose**. In this case, we have three employees, Bob, Denise, and Ted, with employee IDs 1, 2, and 3. This code uses an ID value to assign the corresponding employee name to **strEmployName**:

  strEmployeeName = Choose(intID, "Bob", "Denise", "Ted")

Using the *Do* Loop

- The **Do** loop keeps executing its enclosed statements while or until (depending on which keyword you use, **While** or **Until**) *condition* is true. You can also terminate a **Do** loop at any time with an **Exit Do** statement. The **Do** loop has two versions; you can either evaluate a condition at the beginning:

  Do [{While | Until} *condition*]

  [*statements*]

  [Exit Do]

  [*statements*]

  Loop

  or at the end:

  Do

  [*statements*]

  [Exit Do]

  [*statements*]

  Loop [{While | Until} *condition*]

Do Loop Example

- Here's an example where the code keeps displaying the message "What should I do?" until the user types "Stop" (note that I'm using **UCase** to uppercase what the user types and comparing it to "STOP" to let them use any combination of case when they type "Stop"):

```
Module Module1
        Sub Main()
                Dim strInput As String
                Do Until UCase(strInput) = "STOP"
                        System.Console.WriteLine("What should I do?")
                        strInput = System.Console.ReadLine()
                Loop
        End Sub
End Module
```

Using the *For* Loop

- The Do loop doesn't need a*loop index*, but the **For** loop does; a loop index counts the number of loop iterations as the loop executes. Here's the syntax for the **For** loop—note that you can terminate a **For** loop at any time with **Exit For**:

  For *index* = *start* To *end* [Step *step*]

  [*statements*]

  [Exit For]

  [*statements*]

  Next [*index*]

- The ***index*** variable is originally set to *start* automatically when the loop begins. Each time through the loop, ***index*** is incremented by ***step*** (***step*** is set to a default of **1** if you don't specify a value) and when ***index*** equals end, the loop ends.

For Loop Example

```
Module Module1
        Sub Main()
                Dim intLoopIndex As Integer
                For intLoopIndex = 0 To 3
                        System.Console.WriteLine("Hello from Visual Basic")
                Next intLoopIndex
        End Sub
End Module
        For intLoopIndex = 0 To 3 Step 2
                 System.Console.WriteLine("Hello from Visual Basic")
        Next intLoopIndex
```

Using the *For Each…Next* Loop

- **For Each…Next** loop to loop over elements in an array or a Visual Basic collection.
- This loop is great, because it automatically loops over all the elements in the array or collection

```
For Each element In group
        [statements]
        [Exit For]
        [statements]
Next [element]
```

Using the *For Each…Next* Loop Example

```
Module Module1
        Sub Main()
                Dim intIDArray(3), intArrayItem As Integer
                 intIDArray(0) = 0
                intIDArray(1) = 1
                intIDArray(2) = 2
                intIDArray(3) = 3
                For Each intArrayItem In intIDArray
                        System.Console.WriteLine(intArrayItem)
                Next intArrayItem
        End Sub
End Module
```

Using the *While* Loop

- **While** loops keep looping while the condition they test remains true, so you use a **While** loop if you have a condition that will become false when you want to stop looping.

    While *condition*

        [*statements*]

    End While

Using the *While* Loop Example

```
Sub CheckWhile()
        Dim intCounter As Integer = 0
        Dim intNumber As Integer = 10
        While intNumber > 6
                intNumber -= 1
                intCounter += 1
        End While
        MsgBox("The loop ran " & intCounter & " times.")
End Sub
```

Handling Maths

- The built-in Visual Basic math functions appear in Table are belongs to **System.Math** namespace.

New Visual Basic .NET method	Description
System.Math.Abs	Yields the absolute value of a given number.
System.Math.Atan	Yields a Double value containing the angle whose tangent is the given number.
System.Math.Cos	Yields a Double value containing the cosine of the given angle.
System.Math.Exp	Yields a Double value containing e (the base of natural logarithms) raised to the given power.
System.Math.Log	Yields a Double value containing the logarithm of a given number.
System.Math.Round	Yields a Double value containing the number nearest the given value.
System.Math.Sign	Yields an Integer value indicating the sign of a number.
System.Math.Sin	Yields a Double value specifying the sine of an angle.
System.Math.Sqrt	Yields a Double value specifying the square root of a number.
System.Math.Tan	Yields a Double value containing the tangent of an angle.

Using Math Function Example

```
Imports System.Math

Module Module1

        Sub Main()

                System.Console.WriteLine("Pi =" & 4 * Atan(1))

        End Sub

End Module
```

Handling Dates and Times

To do this	Use this
Get the current date or time	**Today**, **Now**, **TimeofDay**, **DateString**, **TimeString**
Perform date calculations	**DateAdd**, **DateDiff**, **DatePart**
Return a date	**DateSerial**, **DateValue**
Return a time	**TimeSerial**, **TimeValue**
Set the date or time	**Today**, **TimeofDay**
Time a process	**Timer**

Using Date Functions Example

```
Imports System.Math

Module Module1

        Sub Main()

                Dim FirstDate As Date

                FirstDate = #12/31/2001#

                 System.Console.WriteLine("New date: " & DateAdd_
(DateInterval.Month, 22, FirstDate))

        End Sub

End Module
```

New date: 10/31/2003

Date Format Representations

Format Expression	Yields this
Format(Now, "M-d-yy")	"1-1-03"
Format(Now, "M/d/yy")	"1/1/03"
Format(Now, "MM - dd - yy")	"01 /01 / 03"
Format(Now, "ddd, MMMM d, yyy")	"Friday, January 1, 2003"
Format(Now, "d MMM, yyy")	"1 Jan, 2003"
Format(Now, "hh:mm:ss MM/dd/yy")	"01:00:00 01/01/03"
Format(Now, "hh:mm:ss tt MM-dd-yy")	"01:00:00 AM 01-01-03"

Handling Financial Data

To do this	Use this
Calculate depreciation	**DDB**, **SLN**, **SYD**
Calculate future value	**FV**
Calculate interest rate	**Rate**
Calculate internal rate of return	**IRR**, **MIRR**
Calculate number of periods	**NPer**
Calculate payments	**IPmt**, **Pmt**, **PPmt**
Calculate present value	**NPV**, **PV**

Thank You….
End of Unit-2

Unit – 3

Windows Forms and Controls