The .NET Framework is divided into two main components — the .NET Framework Class Library and the Common Language Runtime.

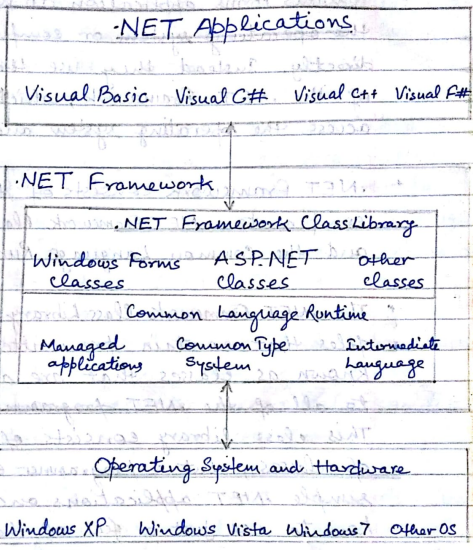
The .NET Framework Class Library consists of segments of pre-written code called classes that provide many of the functions that a programmer need for developing Windows Forms application. The ASP.NET classes are used for developing Web Forms applications. And other classes let programmer work with database, manage security, access files, and perform many other functions.

The classes in the .NET Framework Class Library are organized in a hierarchical structure. Within this structure, related classes are organized into groups called namespaces. Each namespace contains the classes used to support a particular function. Eg. the System.Windows.Forms namespace contain the classes used to create forms and System.Data namespace contains the classes programer use to access data.

The Common Language Runtime (CLR) provides the services that are needed for executing any application that's developed with one of the .NET languages. This is possible because all of the .NET languages compile to a common intermediate language. The CLR also provides the Common Type System that ~~specifies~~ defines the data types that are used by all .NET languages. Because all of the .NET applications are managed by the CLR, they are sometimes referred to as managed applications.

Prof. Madhavendra Dutt
9768741972

.NET Framework



.NET Framework provides a common set of services that application programs written in a .NET language such as Visual Basic can use to run on various operating systems and hardware platforms.

* Windows Forms application do not access the operating system or computer hardware directly. Instead they use the services of the .NET Framework, which in turn access the operating system and hardware.

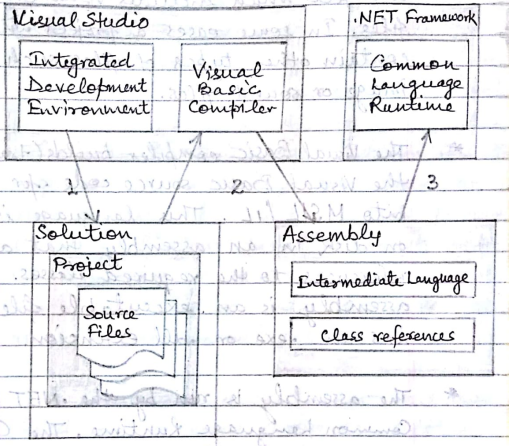
* .NET Framework consists of two main components - .NET Framework Class Library and the Common Language Runtime.

* The .NET Framework Class Library provides files that contain pre-written code known as classes that are available to all of the .NET programming languages. This class library consists of thousands of classes, but a programmer can create simple .NET applications once s/he learn how to use just a few of them.

* The CLR manages the execution of .NET programs by coordinating essential functions such as memory management, code execution, security, and other services. Because .NET applications are managed by the CLR, they are called managed applications.

* The CTS is a component of the CLR that ensures that all .NET applications use the same basic data types no matter what programming languages are used to develop the applications.

How a Visual Basic application is compiled and run –



To start, a programmer use Visual Studio to create a project, which is made up of source files that contain Visual Basic statements. A project may also contain other types of files, such as sound, image or text files.

After a programmer enters the Visual Basic code for a project, s/he uses the

Visual Basic compiler (it is built into Visual Studio) to build his/her Visual Basic source code into Microsoft Intermediate Language (MSIL).

At this point, the IL is stored on disk in a file that's called an assembly. In addition to the IL, the assembly includes references to the classes that the application requires. The assembly can then be run on any PC that has the Common Language Runtime installed on it. When the assembly is run, the CLR converts the IL to native code that can be run by the Windows system.

A solution is a container that can hold one or more projects. Although a solution can contain more than one project, the solution for a simple application usually contains just one project.

* The programmer uses Visual Studio to create a project which includes Visual Basic source files. In some cases a project will also contain other types of files, such as graphic image or sound files.

* The Visual Basic compiler builds (translates) the Visual Basic source code for a project into MSIL / IL. This language is stored on disk in an assembly that also contains references to the required classes. An assembly is an executable file that has an .exe or .dll extension.

* The assembly is run by the .NET Framework's Common Language Runtime. The CLR manages all aspects of how the assembly is run, including converting the IL to native code that can be run by the operating system, manging memory for the assembly, and enforcing security.

Data Types

VB keyword	Bytes	.NET type	Description
Boolean	1	Boolean	True / False value.
Byte	1	Byte	+ve integer value 0 - 255
SByte	1	SByte	signed integer value -128 to 127
Short	2	Int16	Integer From -32,768 to +32,767
UShort	2	UInt16	Unsigned integer 0 to 65,535
Integer	4	Int32	Integer from -2147483648 to +2147483647
UInteger	4	UInt32	Unsigned integer from 0 to 4294967295
Long	8	Int64	Integer from -9223372036854775808 to +9223372036854775807
ULong	8	UInt64	Unsigned integer from 0 to +18446744073709551615
Single	4	Single	A non-integer number with 7 significant digits
Double	8	Double	A non-integer number with 14 significant digits
Decimal	16	Decimal	Upto 28 significant digits (Integer & fraction) 79228×10^{24}
char	2	Char	A single Unicode character

All the built-in data types are actually aliases for the data types defined by the Common Type System of the .NET Framework.

All of the data types shown here are value-types, which means that they store their own data. In contrast, reference types store a reference to the area of memory where the data is stored.

The Unicode character set provides for over 65,000 characters, with two bytes used ~~to store~~ for each character. Each character maps to an integer value.

The older ASCII character set that's used by most operating systems provides for 256 characters with one byte used for each character. In the Unicode character set, the first 256 characters corresponds to the 256 ASCII characters.

How to declare and Initialize a variable

Syntax

Dim variablename [As type] [= expression]

Example

```
Dim counter As Integer
Dim noOfBytes As Long = 20000
Dim price As Double = 14.95
Dim total As Decimal = 24218.1928D
Dim letter As Char = "A"C
Dim valid As Boolean = True
Dim x, y As Integer
Dim i As Integer, d As Decimal
```

How to declare and initialize a constant

Syntax

Const constantName As type = expression

Example

```
Const DayInNovember As Integer = 30
Const SalesTax As Decimal = .075D
```

Arithmetic operators

Operator	Name	Description
+	Addition	Adds two operands
−	Subtraction	Subtracts the right operand from the left operand.
*	Multiplication	Multiplies the right operand and left operand
/	Division	Divides the right operand into the left operand. If both operands are integers, then the result is an integer.
\	Integer Division	Divides the right operand into the left operand and returns an integer quotient.
Mod	Modulus	
^	Exponentiation	Raises the left operand to the power of the right operand.
+	Positive sign	Returns the value of the operand
−	Negative sign	Changes a positive value to negative, and vice versa.

Assignment operators

Operator	Name	Description
=	Assignment	Assigns a new value to the variable.
+=	Addition	Adds the right operand to the value stored in the variable and assigns the result to the variable.
-=	Subtraction	
*=	Multiplication	
/=	Division	
\=	Integer Division	

Statements that use the shortcut assignment operators

```
total += 100
price *= .8D
```

Statements that use the same variable on both sides of the equals sign

```
total = total + 100
price = price * .8D
```

Simple assignment statement

```
total = subtotal - discountAmount
```

```
Dim A, B, C, D As Integer
A = 45
B = 10
C = 4
D = A \ B / C          ' 22
```

Calculation using the default order of precedence —

```
Dim discountPercent As Decimal = .2D
Dim price As Decimal = 100
price = price * 1 - discountPercent   '79.8
```

Calculation using the parentheses to specify the order of precedence

```
Dim discountPercent As Decimal = .2D
Dim price As Decimal = 100
price = price * (1 - discountPercent)   '80
```

* Unless parantheses are used, the operations in an expression take place from left to right in the order of precedence.

↳ 45 \ 10 / 4
45 \ 2.5 (not 2.5 as integer)
22

The order of precedence for arithmetic operations

1. Exponentiation
2. Positive and Negative
3. Multiplication, division, integer division and modulus.
4. Addition and subtraction

* Unless parentheses are used, the operations in an expression take place from left to right in the order of precedence

* To specify the sequence of operations, you can use parentheses. Then, the operations in the innermost sets of parentheses are done first, followed by the operations in the next sets, and so on.

Casting

In many cases Visual Basic does casting automatically as follows —

Widening conversions:

Byte → Short → Integer → Long → Decimal → Single → Double

Implicit casts for widening conversions:

```
Dim grade As Double = 93        ' convert Integer to double.

Dim a As Double = 6.5
Dim b As Integer = 6
Dim c As Integer = 10
Dim avg As Double = (a+b+c)/3   ' avg=7.5
```

Implicit casts for narrowing conversions:

```
Dim grade As Integer = 93.75    ' grade = 94 due to rounding
Dim stotal As Decimal = txtstotal.Text   ' may throw exception
Dim avg As Integer = (a+b+c)/3  ' avg=8
```

To override implicit narrowing casts, you can code explicit casts that use Visual Basic functions like CInt and CDec

Code explicit casts with the CInt, and CDec .... functions —

```
Dim grade As Integer = CInt(93.75)   'grade=94

Dim avg As Integer = CInt((CInt(a)+b+c)/3)
  'convert a and the calculated result to
  'Integer values; avg = 7

Dim stotal As Decimal = CDec(txtStotal.Text)
  'CDec converts to Decimal
  'May throw an exception
```

* Casting refers to the process of converting an expression from one data type to another.

* A widening conversion is one that casts data from a data type with a narrower range of possible values to a data type with a wider range of possible values. Visual Basic uses implicit casts to do widening conversions automatically.

* When we code an arithmetic expression, Visual Basic does widening conversions implicitly so all operands have the widest data type used in the expression.

* A narrowing conversion is one that casts data from a wider data type to a narrower data type.

* By default Visual Basic also does narrowing conversions through implicit casts. But if the receiving data type can not hold the original data then a casting exception is thrown.

* To override implicit narrowing cast, code explicit casts with Visual Basic functions like CInt and CDec.

* In arithmetic expressions, explicit casts are done before any of the other operations.

Permissive type semantics –

By default Visual Basic uses permissive type semantics, ie narrowing conversions are done automatically and you don't get error messages when you compile an application that requires implicit narrowing conversion.

Strict type semantics –

In this semantics you have to code explicit casts for any narrowing conversions. If you fail to do this, your code won't compile and you'll get error message.
When you use strict type semantic, you still get casting exceptions when a narrowing conversion can't be done.

Common .NET structures that define value types

Structure	VB Keyword	What the value type holds
Byte	Byte	An 8-bit unsigned integer
Int16	Short	A 16-bit signed integer
Int32	Integer	A 32-bit signed integer
Int64	Long	A 64-bit signed integer
Single	Single	A single-precision floating point number
Double	Double	A double-precision "
Decimal	Decimal	A 96-bit decimal value
Boolean	Boolean	A True or False value
Char	Char	A single character

Common .NET classes that define reference types

Class name	VB Keyword	What the reference type holds
String	String	A reference to a String object
Object	Object	A reference to any type of object

The .NET structures and classes that define data types

The previous table summarizes the structures and classes that define the data types. To work with the Decimal structure, we use Decimal keyword. To work with Int32 structure, we use Integer keyword. And to work with String class, we use String keyword.

When we declare a variable as one of the data types that's supported by a structure, that variable is a value type. That means that the variable stores its own data.

When we declare a variable as one of the data types that's supported by a class, an object is created from the class. Then the variable stores a reference to the object, not the object itself. Because of that object data types are called reference types.

* Each built-in data type is supported by a structure or a class within the .NET Framework. When we use a Visual Basic keyword to refer to a data type, we are actually using an alias for the associated structure or class.

* A structure defines a value type which stores its own data.

* A class defines a reference type. A reference type doesn't store the data itself. Instead, it stores a reference to the area of memory where data is stored.

How to code Boolean expressions

When we code an expression that evaluates to a True or False value, that expression can be called a Boolean expression.

Relational operators

Operator	Name	Description
=	Equal to	Returns a True value if the left and right operands are equal
<>	Not equal to	Returns a True value if the left and right operands are not equal.
>	Greater than	Returns a True value if the left operand is greater than the right operand.
<	Less than	Returns a True value if the left operand is less than the right operand.
>=	Greater than or equal to	Returns a True value if the left operand is greater than or equal to the right operand.
<=	Less than or equal to	Returns a True value if the left operand is less than or equal to the right operand

Operator Name	Description
Is Is	Returns a True value if the left operand refers to the same object as the right operand.
IsNot IsNot	Returns a True value if the left operand does not refer to the same object as the right operand.

* We can use the relational operators to create a Boolean expression that compares two operands and returns a Boolean value.

* If we compare two numeric operands with different data types, Visual Basic will cast the less precise operand to the type of more precise operand.

* We can use the relational operators with nullable types. If the value of a nullable type is null, the result of the Boolean expression is always null rather than True or False.

FirstName = "Madhavendra"
txtYear.Text = " "
isValid = False
lastName <> "Jones"
i < months
subTotal >= 500
msg Is Nothing 'equal to a null value
address IsNot Nothing 'not equal to a null value

Logical Operators

Name	Description
And	Returns a True value if both expressions are True. This operator always evaluates both expressions.
Or	Returns a True if either expression is True. This operator always evaluates both expression.
AndAlso	Returns a True value if both expressions are True. This operator only evaluates the second expression if necessary.
OrElse	Returns a True value if either expression is True. This operator only evaluates the second expression if necessary.
Not	Reverses the value of the expression.

Eg. isValid = True And Counter + 1 < years
isValid = False Or Counter + 1 >= years

subTotal >= 250 AndAlso subTotal < 500
timeInService <= 4 OrElse timeInService >= 12

Not (counter + 1 >= years)

* We can use the logical operators to create Boolean expressions that combine two or more Boolean expressions.
* Since the AndAlso and OrElse operators only evaluate the second expression if necessary, they're sometimes reffered to as short-circuit operators. These operators are slightly more efficient than the And and Or operators.
* If a nullable Boolean variable with a value of null is used with the And operator, the result is Null unless the value of ~~the value of~~ the other expression is False. In that case, the result is False.
* If a nullable Boolean variable with a value of Null is used with the Or operator, the result is Null unless the value of the other expression is True. In that case, the result is True.
* By default, Not operations are performed first, followed by And operations, and then Or operations. These operations are performed after arithmetic operations and relational operations.

Conditional Statements

If statement without an ElseIf or Else

```
If subTotal >= 100 Then
    discountPercent = .2D
End If
```

If statement with an Else clause

```
If subtotal >= 100 Then
    discountPercent = .2D
Else
    discountPercent = .1D
End If
```

If statement with ElseIf clause

```
If subtotal >= 100 And subtotal < 200 Then
    discountPercent = .2D
ElseIf subtotal >= 200 And subtotal < 300 Then
    discountPercent = .3D
ElseIf subtotal >= 300 Then
    discountPercent = .4D
Else
    discountPercent = .1D
End If
```

Nested If statement

```
If customerType = "R" Then
    If subtotal >= 100 Then
        discountPercent = .2D
    Else
        discountPercent = .1D
    End If
Else
    discountPercent = .4D
End If
```

* An If statement always contains an If clause. It can contain one or more ElseIf clauses and a final Else clause.

* If we declare a variable within an If statement, the variable has block scope, which means that it can only be referred to within If statement.

Select Case Statements

Syntax:
```
Select Case testexpression
    [Case expressionlist
        statements ] ...
    [Case Else
        statements ]
End Select
```

Example:
```
Select Case orderQuantity
    Case 1,2
        discountPercent = 0D
    Case 3 To 9
        discountPercent = .1D
    Case 10 To 24
        discountPercent = .2D
    Case Is >= 25
        discountPercent = .3D
    Case Else
        discountPercent = 0
End Select
```

To - Specifies a range of values
Is - Precedes a conditional expression

```
Select Case customerType
    Case "R"
        discountPercent = .1D
    Case "C"
        discountPercent = .2D
    Case Else
        discountPercent = 0
End Select
```

* After evaluating the test expression, the Select Case statement transfers control to the case that matches the value in the expression. If none of the cases matches, the Case Else clause is executed.

* If you declare a variable within a Select Case statement, the variable has block scope so we can only refer to it within that statement.

* We can code If statements within the cases of a Select Case statement. We can also code Select Case statements within the cases of a Select Case statement.

For Loop

Syntax: For counter [As type] = start To End [Step step]
    statements
Next [counter]

Example:

```
Dim sum As Integer
Dim i As Integer
For i = 2 To 12 step 2
    sum += i
Next
Console.WriteLine(sum)
```

42

* If we declare a variable within a For loop, the variable has block scope so we can only refer to it within the loop.

For Each Loop

The For Each statement iterates through all the items in a list. The list may be an array or a collection of objects.

```
For Each element [As datatype] In group
    [statements]
    [Continue For]
    [Exit For]
    [statements]
Next [element]
```

```
Dim Names As String() = New
    String() {"Ram", "Shyam", "Mohan"}
For Each i As String in Names
    Console.WriteLine(i)
Next i
```

Ram
Shyam
Mohan

While Loop

The While statement executes a statement or a block of statements until a Boolean expression evaluates to True.

Syntax

```
While condition
    [ statements ]
    [ Exit While ]
    [ statements ]
End While
```

Example:

```
Dim i As Integer = 0
Do while i <=5      ' True
    Console.WriteLine("I = {0}", i)
    i += 1
Loop
Do While i <=5      ' False
    Console.WriteLine("I = {0}", i)
    i += 1
Loop
```

I = 0
I = 1
I = 2
I = 3
I = 4
I = 5

Do While Loop

Syntax:
```
Do
    [ statements ]
    [ Exit Do ]
    [ statements ]
Loop { While | Until } condition
```

Example:
```
Dim i As Integer = 0
Do
    Console.WriteLine("I = {0}", i)
    i += 1
Loop While i >= 5   'False
```

I = 0

②
```
Dim a As Integer = 0
Do
    Console.WriteLine("I = {0}", i)
    i += 1
Loop Until i >= 5   'False
```

I = 0
I = 1
I = 2
I = 3
I = 4

With Statement

The With statement is not a loop, but it can be useful as a loop. We can use the With statement to execute statements using a particular object of the class within which the With statement is defined.

Syntax:
```
With object
    [ statements ]
End With
```

Example.
```
Module Module1
    Public Class Rectangle
        Public length, width As Integer
        Public Function Area() As Integer
            Dim ar As Integer = length * width
            Return ar
        End Function
    End Class
    Sub Main()
        Dim rct As New Rectangle
        With rct
            .length = 100
            .width = 50
        End With
        Console.WriteLine("Area is:" & rct.Area())
    End Sub
End Module
```

Area is: 5000

Exit and Continue Statement

* We can code an Exit For or Exit Do statement to exist from a For loop or a Do loop.
* We can use other forms of Exit statements to exist from other types of structures.
  Eg. - We can use the Exit Sub statement to exit from a Sub procedure.
* We can code a Continue For or Continue Do statement to jump to the start of a For or Do loop.

①
```
Dim i, sum As Byte Integer
For i = 1 To 100
    sum += i
    If sum >= 255 Then
        Exit for
    End If
Next i                                  276
Console.Writeline(sum)
```

②
```
Dim sumNum, sumBigNum As Integer
for i As Byte = 1 To 6
    sumNum += i                         B1 B2 B3 B4 B5 A4 A5 A6
    If i < 4 Then                       21 15
        Continue for
    End If
    sumBigNum += i
Next
Console.WriteLine(sumNum & " " & sumBigNum)
```

GoTo Statement

The GoTo statement is used to transfer the program control to a statement that has the same label as that of the GoTo statement. More than one GoTo statements can transfer the program control to the same labeled statement. One of the most important thing is that the GoTo statement can transfer the program control outside a loop control statement, but it can never transfer the program control inside a loop control statement.

```
GoTo label
label:
[Statements]
```

```
Module Module1
  Sub Main()
    Dim x As Integer = 100, y As Integer = 4
    Dim count As Integer = 0
    Dim myArray As String(,) = New _
                String(x-1, y-1) {}
    For i As Integer = 0 To x-1
      For j As Integer = 0 To y-1
        count += 1
        myArray(i, j) = count.ToString()
      Next j
    Next i
```

```
Console.Write("Enter the Number to search:")
Dim myNumber As String = Console.ReadLine()

For i As Integer = 0 To x-1
  For j As Integer = 0 To y-1
    If myArray(i, j).Equals(myNumber) _
        Then
      GoTo Found
    End If
  Next j
Next i
Console.WriteLine("The Number {0} was not _
        Found.", myNumber)
'GoTo Finish Exit Sub
Found:
  Console.WriteLine("The Number {0} is _
        Found.", myNumber)
  Console.WriteLine("End of search.")
  Console.ReadLine()
End Sub
End Module
```

Return Statement

The Return statement returns the program control to the code that called a Function, Sub, Get, Set or Operator procedure. Eg - If a function contains a Return statement to return an expression, the Return statement terminates execution of the function and returns the value of the expression to the code, which calls that particular function.

The Return statement without an expression is equivalent to an Exit Sub or Exit Property statement within a Sub or Set procedure. This means that we do not need to provide the expression that is to be returned.

Function, Get or Operator procedure must include an expression with the Return statement, and the value of the expression must be the same as the data type of the procedure.

Syntax -

Return

or

Return expression

Scope

The scope of an element in our code enables us to access an element without qualifying its name, and makes the element available through an Imports statement.

An element's scope defines its accessibility in our code. The scope in Visual Basic 2010 is categorized depending on the following access modifiers:

Public - Declares elements to be accessible from anywhere within the same project, from other projects that refer the project, and from an assembly built from the project. We can use Public only at the module, namespace, or file level. We can declare a Public element in a source file or inside a module, class, or structure but not within a procedure.

Protected - Declares elements to be accessible only from within the same class where the element is declared, or from a class derived from the class containing the element. We can use Protected only at class level, and when declaring a member of a class.

Friend – Declares elements to be accessible from within the same project, but not from outside the project. We can use Friend only at module, namespace, or file level. We can declare a Friend element in a source file or inside a module, class, or structure but not within a procedure.

Protected Friend – Declares elements to be accessible either from derived classes or within the same project, or both. We can use Protected Friend only at class level and only when declaring a member of a class.

Private – Declares elements to be accessible only from within the same module, class, or structure. We can use Private only at module, namespace, or file level. We can declare a Private element in a source file or inside a module, class, or structure but not within a Procedure.

In Visual Basic an element can have scope at one of the following levels:

1. Block scope – Available only from within the code block in which it is declared.
2. Procedure scope – Available only from within the procedure in which it is declared.
3. Module Scope – Available to all code within the module, class, or structure in which it is declared.
4. Namespace scope – Available to all code in the namespace.

Block Scope – A block is a set of statements enclosed within initializing and terminating declaration statements, such as If and End If, For and Next, While and End While, With and End With.

Example 1.
```
Sub Main()
    Dim i As Integer = 1
    If i = 1 Then
        Dim str As String = "No worries"
        Console.WriteLine(str)
    End If
    Console.WriteLine(str) 'will not work
End Sub
```

2.
```
Module Module1
    Sub Main()
        Console.WriteLine(Module2.Function1())
        Console.WriteLine(Module2.Function2())
    End Sub
End Module
Module Module2
    Public Function Function1() As String
        Return "Hello"          'Accessible
    End Function
    Private Function Function2() As String
        Return "Hi"             'Not Accessible
    End Function
End Module
```

Procedure Scope — Declaring a variable in a procedure gives the variable procedure scope. These variables are not accessible outside that procedure. Variables at this level are known as local variables. Procedure scope and block scope are closely related.

```
Sub Main()
    Dim flag = False
    If flag = False Then
        Dim str As String = "Visual Basic"
    End If
    str = "Visual Basic"    'will not work
End Sub
```

Module Scope — The module scope controls the elements within a module. We can declare elements at this level by placing the declaration statement outside any procedure or block but within the module, class or structure. When we declare an element at the module level, the access level that we choose specifies the scope. The namespace that contains the module, class, or structure also affects the scope.

- If we declare an element to be at private access level, the element is available to every procedure in that module, but not to any procedures or code in different modules. In Visual Basic the access level for a module is private by default.

```
Module Module1
    Private msg As String    ' can be used anywhere in module
    Sub Main()
    End Sub
    Sub setMessage()
        msg = "This variable is within the scope of
        Module1 and cannot be accessible outside of it"
    End Sub
End Module
Module Module2
    msg = "This variable is not declared and therefore
End Module  declaration expected"
```

Namespace Scope — If we declare an element at the module level by using the Friend or Public keyword, it becomes available to all procedures throughout the namespace in which the element is declared. Namespace scope includes nested namespaces.

```
Module Module1
  Private msg As String
  Sub Main()
  End Sub
  Sub sMessage()
    msg = "This variable is within the scope of
      Module1 and cannot be accessible
      outside of it"
  End Sub
End Module
Namespace mynamel
  Module Module2
    Dim str1 As String
    Sub MyProc1()
      str1 = "New Procedure"
    End Sub
  End Module
End Namespace
```

```
Namespace myname2
  Module Module3
    Sub MyProc2()
      str1 = "This variable is declared
      within myname1 namespace and
      cannot be accessible within
      the current namespace"
    End Sub
  End Module
End Namespace
```

Here in the above example the variable named str1 is declared within the myname1 namespace and when we want to access this variable from within the myname2 namespace, we get an error message that the variable is not declared. That means that the variable is bound to the namespace scope.

Exception Handling

Exception is a runtime error that arises because of some abnormal conditions, such as division of a number by zero, passing a string to a variable that holds an integer value.

Whenever compiler encounters such type of exception, it creates an exception object and stores the information about the error in the object.

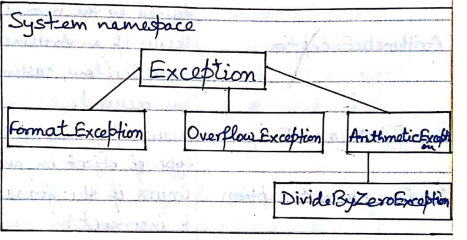
There is another type of error known as compile-time errors which occurs during compilation of a program due to wrong coding or incorrect syntax. We can correct these compile-time errors after looking at the error message that the compiler generates.

On the other hand, runtime errors are those errors that occur during the execution of a program, thus they cannot be corrected at the same time. Therefore necessary actions need to be taken beforehand to prevent such type of errors.

For this we should identify the following two aspects:

* Find out those parts of a program which can cause runtime errors.
* How to handle those errors, when they occur.

Exception hierarchy for five common exceptions —



Handling the errors and exceptions requires sorting out the type of exception that occurs by using different classes.

Some common exception classes :

Exception Class	Description
SystemException	Defines a base class for other exception classes.
AccessViolationException	Occurs when there is an attempt to read or write protected memory.

ArgumentException	Occurs when an invalid argument is passed to a method.
ArgumentNullException	Occurs if a null argument is passed to a method that does not accept it.
ArgumentOutOfRangeException	Occurs if an argument value is not within the allowable range of values defined by the invoked method.
ArithmeticException	Occurs if an Arithmetic overflow or underflow, casting, or conversion has occured.
ArrayTypeMismatchException	Occurs if you store the incorrect type of object in an array.
BadImageFormatException	Occurs if the format of an image is incorrect.
DivideByZeroException	Occurs if you divide a value by zero.
FormatException	Occurs if ~~the index of an array~~ is an incorrect format of argument
IndexOutOfRangeException	Occurs if the index of an array is out of bounds of the array.
InvalidCastException	Occurs if invalid casting or explicit conversion is performed.
InvalidOperationException	Occurs if when a method call is invalid for the object's current state.

MissingMemberException	Occurs if you access an invalid version of a DLL.
NotFiniteNumberException	Occurs if a number is invalid or infinite.
NotSupportedException	Occurs if a class cannot support a method.
NullReferenceException	Occurs if you attempt to dereference a null object reference.
OutOfMemoryException	Occurs if an exception stops due to lack of memory.
StackOverflowException	Occurs if a stack has overflown.

There are two ways to handle errors that occur at run time in Visual Basic 2010 - structured and unstructured exception handling.

Structured Exception Handling —
Structured exception handling encapsulates blocks of code, with each block associated with one or more handlers. Eg.- We can associate multiple exception handlers such as NullReferenceException and IndexOutOfRangeException to a single block of code within the Try block.

Syntax for a simple Try...Catch statement -

```
Try
    trystatement
Catch
    catchstatement
End Try
```

Example

```
Try
    subtotal = CDec(txtSubtotal.Text)
    discountPercent = .2D
    discountAmount = subtotal * discountPercent
    invoiceTotal = subtotal - discountAmount
Catch
    MessageBox.Show("Please enter a valid number
        for the Subtotal field.", "Entry Error")
End Try
```

* The Try block contains the code that is to be tested for exceptions, and the Catch block contains the code that handles the exceptions.
* If the Try block contains calls to other procedures, the Catch block catches any exceptions that are not handled by the called procedures.
* All variables that are declared within a Try ... Catch statement have block scope.

Syntax for a Try...Catch statement that accesses the exception -

```
Try
    trystatements
Catch exceptionName As Exceptionclass
    catchstatements
End Try
```

Two common Properties for all exceptions -

- Message - Gets the message that briefly describes the current exception.
- StackTrace - Gets a string that lists the procedures that were called before the exception occured.

A common method for all exceptions -

- GetType() - Gets the type of the current exception.

~

```
Dim subtotal As Decimal
Try
    subtotal = CDec(txtSubtotal.Text)
Catch ex As Exception
    MessageBox.Show(ex.Message & vbCrLf &
        ex.StackTrace, ex.GetType.ToString)
End Try
```

The complete syntax for the Try...Catch statement —

```
Try
    trystatements
Catch exceptionname As mostspecificexception
    catchstatements
[Catch exceptionname As nextmostspecificexception
    catchstatements]
...
[Catch exceptionname As lastspecificexception
    catchstatements]
[Finally
    statements]
End Try
```

* We can code one Catch block for each type of exception that may occur in the Try block. If we code more than one Catch block, we must code the Catch blocks for the most specific types of exceptions first.
* Since all exceptions are subclasses of the Exception class, a Catch block for the Exception class will catch all types of exceptions.
* We can code a Finally block after all the Catch blocks. The code in this block is executed whether or not an exception occurs. It's often used to free any system resources.

Example

```
Try
    monthlyInvestment = Convert.ToDecimal(txtMI.Text)
    yearlyInterestRate = Convert.ToDecimal(txtIR.Text)
    years = Convert.ToInt32(txtY.Text)
Catch ex As FormatException
    MessageBox.Show("A format exception has
      occured. Please check all entries.", "Entry Error")
Catch ex As OverflowException
    MessageBox.Show("An overflow exception has occured.
      Please enter smaller values.", "Entry Error")
Catch ex As Exception
    MessageBox.Show(ex.Message, ex.GetType.ToString)
Finally
    PerformCleanup()
End Try
```

The Throw statement —

We can use the Throw statement to throw a new or existing exception in case an error occurs in a program. An exception usually occurs when some abnormal tasks are performed but in Visual Basic it is also possible to throw an exception programmatically. When we create a new exception, we can specify a string that's assigned to the Message property of the exception.

Syntax for throwing a new exception

```
Throw New ExceptionClass([message])
```

Syntax for throwing an existing exception

```
Throw exceptionName
```

When to throw an exception

* When a procedure encounters a situation where it is not able to complete its task.
* When you want to generate an exception to test an exception handler.
* When you want to catch the exception, perform some processing, and then throw the exception again.

```
'A Function procedure that throws a FormatException
Private Function FutureValue(mI As Decimal,
    mIR As Decimal, m As Integer) As Decimal
  If monthly
  If mI <= 0 Then
    Throw New FormatException("Monthly Invest-
        ment must be greater than 0.")
  End If
  If mIR <= 0 Then
    Throw New FormatException("Monthly
      Interest Rate must be greater than 0.")
  End If
  :
  :

'Code that throws an Exception for testing purposes
  Try
    subtotal = Convert.ToDecimal(txtSubtotal.Text)
    Throw New OverflowException
  Catch ex As Exception
    MessageBox.Show(ex.Message & vbCrLf &
        ex.GetType.ToString & vbCrLf &
        ex.StackTrace, "Exception")
  End Try

'Code that rethrows an exception
Try
  Convert.ToDecimal(txtSubtotal.Text)
Catch ex As FormatException
  txtSubtotal.Select()
  Throw ex
End Try
```

Unstructured Exception Handling

The unstructured exception handling allows you to write the On Error statement at the starting of a block of code, and any errors occur during the execution of that block handled by On Error statement. You can use this statement to instruct Visual Basic what to do if an exception is raised.
The unstructured exception handling supports two types of On Error statements:

- ✓ On Error Resume Next
- ✓ On Error

On Error Resume Next Statement

The On Error Resume Next statement specifies that when a run-time error occurs, the program control passes to the statement immediately following the statement in which the error occured. At that point the execution continues.

```
Sub Main()
  On Error GoTo Handler
     :
  Handler:
  On Error Resume Next
     :
End Sub
```

In the above program, the Main sub procedure contains the On Error statement that transfers the control to the label named Handler when an error occurs. The label Handler contains the On Error Resume Next statement that enables you to move to the next statement followed by the statement that caused an exception.

The On Error Statement

The On Error statement assumes that error-handling code starts at the line specified in the required line argument. The line argument can be a line number or line label indicating the exception handler location. If a run-time error occurs, control branches to the line label or line number specified in the argument, activating the error handler. The specified line must be in the same procedure as the On Error statement otherwise Visual Basic generates a compile-time error.

The On Error statement is used in three different ways that are :

GoTo line - Enables the error-handling routine that starts at the line specified in the required line argument.

GoTo 0 - Disables enabled error handler in the current procedure and resets it to Nothing.

GoTo -1 - Disables enabled exception in the current procedure and resets it to Nothing.

```
Sub Main()
    On Error GoTo Handler
    :
    Exit Sub
    Handler:
    :
End Sub
```

In the above program, the program control will be transferred to the label Handler if there has been an exception. When an error is raised the control jumps to the labeled statement and

the execution continues from the statement where the program control is transfered.

On Error Resume Next Example:

```
Module Module1
  Sub Main()
    Dim int1 = 0, int2 = 1, int3 As Integer
    On Error GoTo Handler
    int3 = int2 / int1
    Console.WriteLine("Program completed...")
    Exit Sub
  Handler:
    If(TypeOf Err.GetException() Is OverflowException)
      Then
    Console.WriteLine("Overflow error!")
    On Error Resume Next
    Console.WriteLine("Error skipped")
    Console.WriteLine("Now continue with the next
          statements...")
    End If
  End Sub
End Module
```

Output

```
Overflow error!
Error Skipped
Now continue with the next statements...
```

On Error Example :

```
Module Module1
  Sub Main()
    Dim int1 = 0, int2 = 1, int3 As Integer
    On Error GoTo Handler
    int3 = int2 / int1
    Console.WriteLine("Program completed...")
  Handler:
    If(TypeOf Err.GetException() Is OverflowException) Then
      Console.WriteLine("Overflow error!")
      Resume Next
    End If
  End Sub
End Module
```

Output

```
Overflow error!
Program completed...
```

OOP

Classes and Objects

Syntax for declaring a class is:

```
Public Class <Classname>
    'Class Members
End Class
```

Syntax for declaring and instantiating an object:

```
Dim <objectname> As <classname>
<objectname> = New <classname()>
```

Constructors and Destructor –

A constructor is the first method that is executed when an instance of a class is created. It never returns any value and can be overriden. You can use a constructor to initialize objects and set parameters.

A destructor is a method that is called when the object is finally destroyed. It is the last procedure run by the class.

```
Class Human
    Private Age As Integer
    Public Sub New()          'default constructor
      Age = 0
      Console.WriteLine("Human is created with the
                    age zero")
    End Sub
    Public Sub New(ByVal val As Integer)  'parameterized
      Age = val                           'constructor
      Console.WriteLine("Human is created with the
            age" + Convert.ToString(val))
    End Sub
    Protected Overrides Sub Finalize()  'destructor
      Console.WriteLine("Human is destroyed")
    End Sub
Sub Main()
    Dim Joseph, Robert As Human
    Joseph = New Human          ' default constructor
    Robert = New Human(20) ' parameterized constructor
End Sub
End Class
```

Shared Members —

Shared member of a class or structor helps in making a class or structor available to every instance, in a non-shared class or structor each instance keeps its own copy.

You can declare members shared by using the Shared keyword. This keyword makes a member into a class member that can be used just with the class name - no object required.

The Shared keyword also enables the shared method or data member to be shared across all instances of its class. Eg. If a class has a shared data member named Count, then every object of that class uses the same memory location for Count.

```
Public Class Mathematics
  Shared Function Add(ByVal x As Integer,
        ByVal y As Integer) As Integer
    Return x + y
  End Function
End Class
```

You can now use the Add function by using the name of the class

```
Sub Main()
    Result = Mathematics.Add(10, 15)
End Sub
```

Structure and Modules —

A structure is a composite data type that contains the different types of element in one place. It is a user defined data type which is similar to a class. It support many features supported by classes, including the ability to support fields, methods, and properties. The following features of classes are not supported by structures —

* Structures cannot explicitly inherit from any value type.
* Structures cannot be inherited from other structs
* You cannot define a non-shared constructor that does not take any arguments for a structure. However, you can define a non-shared constructor that takes arguments. This is because every structure has a built-in public constructor without any arguments.
* You cannot override the Finalize method in a structor.
* The declarations of data members in a structure cannot include initializers, the New keyword, set initial sizes for arrays.
* If you declare members of structors with the Dim keyword, the default access of data mmembers in structures is public, not private as in classes and modules.

* Structure members cannot be declared as Protected.
* Structures ~~members~~ are value types rather than reference types. i.e. Assigning a structure instance to another structure causes the entire structure to be copied.

Besides types such as classes and structures VB 2010 offers you another type that serves the similar purpose known as modules. When a VB 2010 module is compiled, it is converted into a class.
Modules are reference types similar to classes, but with some differences. Eg. You can call methods (or variables) in a module without using the module name. The distinctions between modules and classes are following —
* Members of a module are implicitly shared.
* Modules can never be instantiated.
* Modules do not support inheritance.
* Modules can not implement interfaces.
* A module can be declared only inside a namespace
* Modules can not be nested in other types such as a class, a structure or an interface.

Eg. Structure –

```
Module Module1
    Public Structure X
        Public x As Integer
        Public y As Integer
        Public z As Integer
    End Structure

    Sub Main()
        Dim x As X
        ~~x = 1~~
        x.x = 1
        x.y = 2
        x.z = 3
        Console.WriteLine("X = {0}, Y = {1},
        Z = {2}", x.x, x.y, x.z)

        Dim x1 As X = New X()
        Console.WriteLine("X = {0}, Y = {1}, Z = {2}",
        x1.x, x1.y, x1.z)
        Console.ReadKey()
    End Sub
End Module
```

9/9/14

Encapsulation –

Encapsulation is the process of hiding the irrelevant information and showing only the relevant information of a specific object to a user.

In terms of OOP encapsulation is the process of wrapping up data and members of a class. It is an approach of hiding the internal state and behavior of a class or an object from unauthorized access. Encapsulation restricts users from sharing and manipulating data, thereby minimizing the chances of data corruption. The advantages of encapsulation are as follows –

* Provide a way to protect our data from unauthorized access. We can define our data as private (eg. data members, member functions, properties or indexers). The private data can indirectly be operated in two ways, firstly through the accessors and mutator methods and secondly through the named property.

Hiding data for security
disallow access

* Increase the maintainability of the code by showing only the relevant information to a user.
* Prevents data corruption by enabling private member variables of a class to be accessed through specific methods or properties.
* Bind member variables and functions of a class into a single unit. This is also called wrapping up of data members and member functions in a class.

Example of Properties –

```
Module Module2
    Private mFirstName As String
    Private mLastName As String

    Public Property FirstName() As String
        Get
            FirstName = mFirstName
        End Get

        Set(ByVal value As String)
            mFirstName = value
        End Set
    End Property

    Public Property LastName() As String
        Get
            LastName = mLastName
        End Get

        Set(ByVal value As String)
            mLastName = value
        End Set
    End Property

    ReadOnly Property fullName() As String
        Get
            fullName = mLastName & ", " &
        End Get                mFirstName
```

```
    End Property
End Module

Module Module1

    Sub Main()
        Module2.FirstName = "Madhav"
        Module2.LastName = "Dutt"
        Console.WriteLine(Module2.fillName)
        Console.ReadKey()
    End Sub

End Module
```

Accessors (Get) And Mutators (Set)

```
Public Class Employee
    'field Data
    Private empName As String
    Private empID As Integer
    Private curPay As Double
    'constructors

    Public Sub New()
    End Sub

    Public Sub New(ByVal name As String,
      ByVal id As Integer, ByVal pay As Double)

      empName = name
      empID = id
      curPay = pay
    End Sub
    'Members
    Public Sub GiveBonus(ByVal amount
      As Double)
      curPay += amount
    End Sub
    Public Function GetCurPay() As Decimal
      Return curPay
    End Function
```

```
                                        Decimal)
    Public Sub SetCurPay(ByVal pay As
        curPay = pay
    End Sub
    Public Function GetName() As String
        Return empName
    End Function
                                        String)
    Public Sub SetName(ByVal name As
        empName = name
    End Sub
    Public Function GetId() As Integer
        Return empId
    End function
    Public Sub SetId(ByVal id As Integer)
        empID = id
    End Sub
    Public Sub DisplayStats()
        Console.WriteLine("Name : {0}", empName)
        Console.WriteLine("ID: {0}", empID).
        Console.WriteLine("Pay : {0}", curPay)
    End Sub
End Class
```

from page 75

```
Sub Main()
    Dim x As New Employee("Madhav",
            456, 30000D)
    x.DisplayStats()
    x.GiveBonus(10000D)
    x.DisplayStats()
End Sub
    x.SetName("sadhav")
    :
    :
End Sub
```

Inheritance —

The most important reason to use OOP is to promote the redundant code. To reduce the redundant code the object-oriented language supports inheritance. Inheritance is the property through which a class derive the property from another class.

```
Public Class BaseClass
    Public Sub BaseMethod()
        'Sub procedure body
    End Sub
End Class
Public Class DerivedClass Inherits BaseCl
    Public Sub DerivedMethod()
        'Sub procedure body
    End Sub
End Class
```

To declare an object, write the following statement —

```
Dim bc As BaseClass = New DerivedCla
```

In the preceding code, a reference of the BaseClass class holds an object (bc)

type DerivedClass class. In this case we take the object of the DerivedClass class as an object of the BaseClass class. It is possible because we derive the DerivedClass class from the BaseClass class. Therefore we can write:

bc.BaseMethod ();

Inheritance in Visual Basic is of 3 types-

* Single Inheritance - Refers to inheritance in which there is only one derived class and one base class. This means that a derived class inherits properties from single base class.

* Hierarchical Inheritance - Refers to inheritance in which multiple derived classes are inherited from the same base class.

* Multilevel Inheritance - Refers to inheritance in which a child class is derived from a class, which in turn is derived from another class.

Properties
Methods
Events
Type Definitions

Interface –

Interface is a collection of data members and member function however it does not include the implementation of data members and member functions. Interfaces have used to provide with the feature of multiple inheritance to the classes.
The methods defined in an interface do not have implementation, they only specify the parameters that they will take and the types of values they will return. An interface is always implemented in a class. The class that implements the interface needs to implement all the members of the interface. An interface in Visual Basic is the equivalent of an abstract class. An interface can have the same access modifiers as a class such as Public and Private.

Syntax: Interface <InterfaceName>
<Abstract Method Declarations>
End Interface

Ex: Interface MyFirstInterface
Sub MyFirstMethod()
End Interface
Interface MySecondInterface
Sub MySecondMethod()
End Interface

Implementation of Interface –

Public Class ImplementInterface
Implements MyFirstInterface, MySecondInterface
Shared Sub Main()
Dim ABC As ImplementInterface = New ImplementInterface()
Dim I1 As MyFirstInterface = CType(ABC, MyFirstInterface)
Dim I2 As MySecondInterface = CType(ABC, MySecondInterface)
I1.MyFirstMethod()
I2.MySecondMethod()
End Sub
Public Sub MyFirstMethod() Implement MyFirstInterface.MyFirstMethod
Console.WriteLine("Call MyFirstMethod()")
End Sub
Public Sub MySecondMethod() Implements MySecondInterface.MySecondMethod
Console.WriteLine("Call MySecondMethod()")
End Sub
End Class

Interface and Inheritance

When a base class implements an interface the derived class of the base class automatically inherits method of the interface. You can initialize an object of the interface by type casting the object of the derived class with the interface itself.

```
Interface MyInterface
    Sub ShowData()
End Interface

Public Class MyBaseClass
    Implements : MyInterface
    Public Sub ShowData()
    'Method body
    End Sub
End Class

Public Class MyDerivedClass
    Inherits MyBaseClass
    Public Shadows Sub ShowData()
    'Method body
    End Sub
End Class
```

```
Public Class MainClass
    Private dc As MyDerivedClass = New
        MyDerivedClass()
    dc.ShowData
    Dim inter As MyInterface = CType(
        dc, MyInterface)
    inter.ShowData()
End Class
```

In the preceding code snippet, an interface MyInterface is declared containing the ShowData() Sub procedure. A base class MyBaseClass implements the MyInterface interface. Next a derived class MyDerived-Class inherits the MyBaseClass. Now in the MainClass class an instance dc of MyDerivedClass class is declared and initilized. Finally the dc object is typecasted to the inter object of the MyInterface interface and used to call ShowData() method.

Inheritance Modifiers –

In Visual Basic 2010 all classes can serve as base classes. However you can use the following two class-level modifiers, called inheritance modifiers, to modify the behaviour –

* **NotInheritable** – Prevents a class from being used as a base class.
* **MustInherit** – Indicates that the class is intended to use as a base class only.
* **Inherits** statements – specifies the base class.

Extension Method –

Extension method is a method that helps you to extend a class without creating a new derived class or modifying the original class. This behaviour of extension methods is similar to that of shared methods. An extension method can either be a Sub procedure or a function. You can not define an extension property, field or event. All extension methods must be marked with the

<Extension()> extension attribute from the System.Runtime.CompilerServices namespace.

```
Imports System.Runtime.CompilerServices
Module MyStringExtensions
  <Extension()> Public Sub Print(ByVal
                    MyString As String)
    'Extension method body.
  End Sub
End Module
```

* Extension methods can be declared only within modules. The module, in which an extension method is defined, is not the same module as the one in which it is called.

```
Imports System.Runtime.CompilerServices
Module MyExt
<Extension()> Public Sub Display(ByVal MyStr As String)
    Console.WriteLine(MyStr)
End Sub
End Module
Module Module2
 Sub Main()
  Dim sample As String = "Madhavendra"
  Sample.Display()
  Sample.ToUpper.Display()      'E.M. enables users to add
 End Sub                        a custom functionality
End Module                      to data types that are
                                already defined without
                                creating a new derived type
```

Polymorphism -

Polymorphism is one of the important features of OOP that is used to exhibit different forms of any particular procedure. The advantages of polymorphism are as follows -

* Allows you to invoke methods of a derived class through base class reference during runtime.
* Provides different implementations of methods in a class that are called through the same name.

Polymorphism can be classified into the following two categories -

* Compile Time Polymorphism (Overloading)
* Run Time Polymorphism (Overriding)

Compile Time Polymorphism (Overloading)

When a compiler compiles a program, it knows the information about the method arguments and accordingly, it binds the appropriate method to an object at the compile time itself. It is of two types -

* Method Overloading
* Operator Overloading

Method Overloading – It is a concept in which a method behaves according to the number and type of parameters passed to it. It allows you to define many methods with the same name but with different signatures. A method signature is the combination of the method's name along with the number and types of parameters (and their order).

```
Public Class Shape
    Public Sub Area(ByVal side As Integer)
        'Calculate Square Area
    End Sub
    Public Sub Area(ByVal Radius As Double)
        'Calculate Circle Area
    End Sub
    Public Sub Area(ByVal Length As Integer,
        ByVal Breadth As Integer)
        'Calculate Rectangle Area
    End Sub
    Public function Area(ByVal Base As Double,
        ByVal Height As Double) As Double
        'Calculate Triangle Area
    End function
End Class
```

```
Public Class MainClass
Dim shape As New Shape
shape.Area (15)
shape.Area (10, 20)
shape.Area (10.5)
shape.Area (15.5, 20.4)
End Class
```

Operator Overloading – All operators have their specified meaning and functionality. However you can change the functionality of an operator. The mechanism of assigning a special meaning to an operator according to the user-defined data type (class, structure) is known as operator overloading. It is not possible to overload all the operators.

Operators	Overloading Status
+, -, Not, ++, --, True, False	Can be overloaded
+, -, *, /, Mod, &, |, ^, <<, >>	"
=, <>, <, >, <=, >=	"
And, Or, Xor, AndAlso, OrElse	Can not be overloaded
IsFalse, IsTrue	"
CType	Can be overloaded
+=, -=, *=, /=, &=, ^=, <<=, >>=	Can not be overloaded but += operator (eg) is evaluated using + operator which can be overloaded
=, ., ?:, New, Is, TypeOf, Like, AddressOf, GetType	Can not be overloaded

```
Public Class Vehicle
    Private strManufacturer As String
    Public Property Manufacturer As String
        Get
            Return strManufacturer
        End Get
        Set(ByVal value As String)
            strManufacturer = value
        End Set
    End Property

    Public Shared Operator = (ByVal objVehicle1
        As Vehicle, ByVal objVehicle2 As Vehicle)
                As Boolean
    If objVehicle1.Manufacturer.Equals(objVehicle2
            Manufacturer) Then
        Return True
    Else
        Return False
    End If
End Operator
Public Shared Operator <> (ByVal obj1 As Vehicle,
    ByVal obj2 As Vehicle) As Boolean
    If Not obj1.Manufacturer.Equals(obj2.
            Manufacturer) Then
        Return True
    Else Return False
End If  End Operator  End Class
```

```
Module Module1
  Sub Main()
  Dim ericsCar As New Vehicle
  ericsCar.Manufacturer = "Volvo"

  Dim sarahsCar As New Vehicle
  sarahsCar.Manufacturer = "Ford"

  If ericsCar = sarahsCar Then
  Console.WriteLine("Both ericsCar and
     sarahsCar shares the SAME object in
     memory")
  Else
    Console.WriteLine("false")
  End If
 End Sub
End Module
```

Runtime Polymorphism (Overriding)

Overriding is a feature that allows a derived class to provide a specific implementation of a method that is already defined in a base class. There are two conditions which should be satisfied –

* Declare the base class method with the **Overridable** keyword.
* Implement the derived class method by using the **Overrides** keyword.

```
Public Class BaseClass
    Public Overridable Sub showData()
        ' Method body
    End Sub
End Class
Public Class DerivedClass
    Inherits BaseClass
    Public Overrides Sub showData()
        ' Method body
    End Sub
End Class
```

P8
90

When we use the Shadows keyword on a member of a derived class, the member hides the corresponding base class member. As a result, the derived class does not inherit the original version of the member.

Shadowing -

It is a process when two programming element's have the same name then one can hide or shadow the other element. You can shadow a base class member in the derived class by using the Shadows keyword. The method's signature, access level, and return type of the shadowed member can be completely different from that of the base class member.

```
Public Class MyFirstClass
    Public Sub Show()
        'Sub Procedure Body
    End Sub
End Class

Public Class MySecondClass
    Inherits MyFirstClass
    Private Shadows Sub Show()
        'Sub Procedure Body
    End Sub
End Class

Public Class MyThirdClass
    Inherits MySecondClass
    Public Shadows Sub Show()
        'Sub Procedure Body
End Class  End Sub
```

but uses the new version instead. Hiding is similar to overriding, but can be used only with non-overridable methods or properties.


```
Module callShow
  Dim First As New MyfirstClass
  Dim Second As New MySecondClass
  Dim Third As New MyThirdClass
  Public Sub callDisplayProcedures()
    First.Show()  ' Show of MyfirstClass
    Second.Show()  ' Show of MyfirstClass
    Third.Show()  'Show of MyThirdClass
  End Sub
End Module
```

In above code the MySecondClass class shadows the Show subprocedure. When the callShow module calls the Show sub Procedure in the MySecondClass class, the calling code is outside the MySecondClass class and, therefore, it cannot access the private Show Sub procedure. The compiler resolves the reference to the base class's Show Sub Procedure. However, the further derived class MyThirdClass declares the Show sub procedure as Public, so the code in callShow module can access it.

There is one more variant - shadowing variables through scope

1P

Abstract Classes -

Abstract classes are closely related to interfaces. They are classes that cannot be instantiated, and are frequently either partially implemented or not at all implemented. One key difference between abstract classes and interfaces is that a class may implement an unlimited number of interfaces, but may inherit from only one abstract (or any other kind of) class.

An abstract class is denoted in Visual Basic by the keyword MustInherit. An abstract class can have both abstract and non-abstract members.

```
Public MustInherit Class BaseClass
    'Data members and member functions
End Class

Public Class DerivedClass
    Inherits BaseClass
    'Data members and member functions
End Class
Public Class MainClass
    Private dc As DerivedClass
End Class
```

Characteristics of an abstract class —

* Restricts instantiation, implying that you cannot create an object of an abstract class.
* Allows you to define abstract as well as non-abstract members in it.
* Requires at least one abstract method in it
* Restricts the use of NotInheritable keyword in the abstract class.
* Possesses public access specifier; therefore it can be used anywhere in a program.

Abstract Method — To declare a method as abstract, apply the MustOverride keyword before the method name at the time of its declaration.

```
Public MustOverride Sub Area (ByVal Length
    As Integer, ByVal Breadth As Integer)
```

Characteristics of an abstract method —

* Restricts implementation in an abstract class
* Allows implementation in a non-abstract derived class.
* Requires declaration in an abstract class only
* Restricts declaration with the Shared and Overridable keyword.
* Allows to override an Overridable method.

```
Public MustInherit Class Shape
    Public MustOverride Sub Area (ByVal
        Length As Integer, ByVal Breadth As Integer)
    End Sub
End Class

Public Class Rectangle
        Inherits Shape
    Public Overrides Sub Area (ByVal Length
        As Integer, ByVal Breadth As Integer)
        'method body
    End Sub
End Class
```

Dim student(6) As Integer

0	1	2	3	4	5	6

95

Arrays –

An array is a set of values that are logically related to each other, An array allows you ~~to refer~~ to refer to these related values by the same name and to use a number called **Index** for identifying individual values. The individual values of an array are called the elements of the array. These elements are stored in the array with the index values starting from 0 to ~~xxxxxxx~~ the size of the array. To dimension arrays we can use the **Dim** (standard arrays), **ReDim** (dynamic arrays), **Static** (arrays that do not change when between the calls to the procedure they are in), **Private** (arrays private to the form or module they are defined in), **Protected** (arrays restricted to a class or classes derived from that class) and **Public** (arrays global to the whole program) keywords.

Standard Arrays – Dim statement is normally used to

```
Dim Arr(30)
Dim UsingArr( 0 To 20)
Dim Strings (10) As String
Dim TwoDArray (20, 40) As Integer
Dim Bounds (10, 100)

Dim Strings (10) As String
Strings (3) = "It's a string"
System.Console.Write( Strings(3) )
```

We can also initialize the data in an array if we do not give an array an explicit size.

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

Dynamic Arrays — To declare a dynamic array with empty parantheses, we use Dim statement. Dynamic arrays can be dimensioned again as you need by using **ReDim** statement.

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

The Preserve keyword is used to preserve the data in an existing array when we change the size of the last dimen-

```
Dim DynaArr() As String
ReDim DynaArr(10)
DynaArr(0) = "String 0"
ReDim DynaArr(100)
DynaArr(50) = "String 50"
```

Jagged Arrays

An array of which each element is itself an array is called an array of array or a jagged array. Note that having arrays as elements is not the same thing as a multidimensional array, which has more than one index on a single array.

Ex. The following example declares an array variable to hold an array of arrays with elements of the Double Data Type. Each element of the array sales is itself an array that represents a month. Each month array holds values for each day in that month.

```
Dim sales()() As Double = New
                Double(11)() {}
Dim month As Integer
```

```
For month = 0 To 11
    days = DateTime.DaysInMonth(Year(
           Now), month + 1)
    sales(month) = New Double(days - 1){}
Next month
```

The New clause in the declaration of sales sets the array variable to a 12-element array, each element of which is of type Double(), an array of Double elements. The For loop then determines how many days are in each month this year (Year(Now)), and sets the corresponding element of sales to a Double array of the appropriate size.

Ex.

```
Module Module1
    Sub Main()          'create a jagged array
        Dim jagged()() As Integer =
            New Integer(2)() {}
        ' Create temp array and place in index 0
        Dim temp(2) As Integer
        temp(0) = 1
        temp(1) = 2
        temp(2) = 3
        jagged(0) = temp
```

```
'create temp array and place in index1
Dim temp2(0) As Integer
jagged(1) = temp2

'Use array constructor and place in index2
jagged(2) = New Integer() {3,4,5,6}

'Loop through top-level arrays
For i As Integer = 0 To jagged.Length-1
    'Loop through elements in subarrays
    Dim inner As Integer() = jagged(i)
    For a As Integer = 0 To inner.Length-1
        Console.Write(inner(a))
        Console.Write(" ")
    Next
    Console.WriteLine()
Next
End Sub
End Module
```

O/P

```
1 2 3
0
3 4 5 6
```

Few Properties and Methods of Array class -

Property -

Length - Gets the number of elements in all of the dimensions of an array

Instance method -

GetLength(dimension) - Gets the number of elements in the specified dimensions of an array.

GetUpperBound(dimension) - Gets the index of the last elements in the specified dimension of an array.

Static method -

Copy(array1, array2, length) - Copies some or all values in one to another array

BinarySearch(array, value) - Searches a one-dimensional array that is in ascending order for an element with a specified value and returns the index for that element.

Sort(array) - Sorts the elements in a one-dimensional array into ascending order

How to refer to and copy arrays

* Because arrays are created from a class, they are reference type. That means that an array variable contains a reference to an array object and not the actual values in the array. Because of that you can use two or more variables to refer to the same array.
* To copy the elements of one array to another array, you use the Copy method of the Array class.
* When you copy an array, the target array must be the same type as the sending array and it must be large enough to receive all of the elements that are copied to it.

Code that creates a reference to another array

```
Dim inches1() As Double = {1, 2, 3}
Dim inches2() As Double = inches1
inches2(2) = 4          ' inches1(2) also = 4
```

Copy elements of one array to another array

```
Array.Copy(fromarray, toarray, length)
Array.Copy(fromarray, fromindex, toarray, toindex,
                                        length)
```

```
'Copy all the elements of an array
Dim inches() As Double = {1,2,3,4}
Dim cent(3) As Double
Array.Copy(inches, cent, inches.length)
for i As Integer = 0 To cent.length -1
    cent(i) *= 2.54
Next

'Copy some of the elements of an array
Dim names() As String = {"Ram", "Shyam",
Dim last      "Ghanshyam"}
Dim lastTwoNames(1) As String
Array.Copy(names, 1, lastTwoNames, 0, 2)

Return an array from a procedure
Private function RateArray(ByVal count
  As Integer) As Decimal()
  Dim rates(count -1) As Decimal
  for i As Integer = 0 To rates.length-1
      rates(i) = (i+1)/100D
  Next
  Return rates
End function
'call
Dim rates() As Decimal = Me.RateArray(4
```

Procedure that accepts an array argument

```
Private Sub ConvertToCentimeters(ByRef
        measurements() As Double)
    For i As Integer = 0 To measurements.
                                Length - 1
        measurements(i) *= 2.54
    Next
End Sub

'usage
Dim measurements() As Double = {1, 2, 3}
Me.ConvertToCentimeters(measurements)
```

* To return an array from a function procedure, you code a set of parantheses after the return type in the declaration to indicate that the return type is an array.

* To accept an array as a parameter of a procedure, you code a set of parantheses after the parameter name in the declaration to indicate that the parameter is an array.

Collections

Like an array, a collection can hold one or more elements. Unlike arrays, collections don't have a fixed size. Instead, the size of a collection is increased automatically when elements are added to it. In addition, most types of collections provide methods that you can use to change the capacity of a collection. As a result, collections usually work better than arrays when you need to work a varying number of elements. There are mainly five types of collections: lists, sorted lists, queues, stacks, and array lists; apart from other collections provided by .NET Framework.

* A collection is an object that can hold one or more elements.
* The collection classes in the System.Collection.Generic namespace use a feature known as generics to allow you to create typed collections that can only store the specified type.
* The collection classes in the System.Collection namespace allow you to create untyped

collections. With an untyped collection, you can store any type of object in the collection.

* The set of parantheses after a class name indicates that it is a typed collection.

How arrays and collections are similar

+ Both can store multiple elements, which can be value types or reference types.

How arrays and collections are different

* An array is a feature of the Visual Basic language that inherits the Array class. Collections are other classes in the .NET Framework.
* Collection classes provide methods to perform operations that arrays don't provide.
* Arrays are fixed in size. Collections are variable in size.

typed untyped

Commonly used collection classes

.NET 2.0 to 4.0	.NET 1.x	Description
List()	Arraylist	Uses an index to access each element. This class is efficient for accessing elements sequentially, but inefficient for inserting elements into a list

SortedList()	SortedList	Uses a key to access a value, which can be any type of object. This class can be inefficient for accecing elements sequentially, but it is efficient for inserting elements into a list.
Queue()	Queue	Uses special methods to add and remove elements
Stack()	Stack	Uses special methods to add and remove elements

Example of Untyped collections

```
Dim numbers As New ArrayList
numbers.Add(3)
numbers.Add(70)
numbers.Add("Test")  ' will compile but causes an exception
Dim sum As Integer = 0
Dim number As Integer
For i As Integer = 0 To numbers.Count-1
    number = CInt(numbers(i))  'cast is requir
    sum += number
Next
```

Example of Typed collections

```
Dim numbers As New List(Of Integers)
numbers.Add(3)
numbers.Add(70)
'numbers.Add("Test") ' won't compile - prevents runtime error
Dim sum As Integer = 0
Dim number As Integer
for i As Integer = 0 To numbers.Count - 1
    number = numbers(i)  ' no cast is required
    sum += number
Next
```

* Typed collections have two advantages over untyped collections. First, they check the type of each element at compile-time and prevent runtime errors from occuring. Second, they reduce the amount of casting that's needed when retrieving objects.
* Untyped collections are part of the System.Collections namespace, while typed collections are part of the System.Collections.Generic namespace.

List

* A list is a collection that automatically adjusts its capacity to accommodate new elements.
* The default capacity of a list is 16 elements, but you can specify a different capacity when you create a list. When the number of elements in a list exceeds its capacity, the capacity is automatically doubled.

```
'create a list of String elements
Dim titles As New List(Of String)

'create list of Decimal elements
Dim prices As New List(Of Decimal)

'create a list of strings with a capacity of 3
Dim lastNames As New List(Of String)(3)
```

Common properties and methods of the List() class -

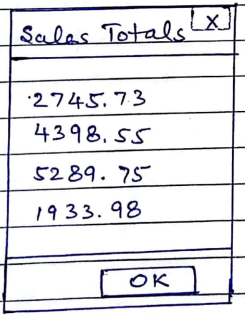
Property	Description
Item(index)	Gets or sets the element at the specified index. The index for the first item in a list is 0. Since Item is default property, its name can be omitted.

Capacity	Gets or sets the number of elements the list can hold.
Count	Gets the number of elements in the list.

Method	Description
Add(Object)	Adds an element to the end of a list and returns the element's index.
Clear()	Removes all elements from the list and sets its Count property to zero.
Contains(Object)	Returns a Boolean value that indicates if the list contains the specified object.
Insert(index, object)	Inserts an element into a list at the specified index.
Remove(Object)	Removes the first object occurrence of the specified object
RemoveAt(index)	Removes the element at the specified index of a list.
BinarySearch(Object)	Searches a list for a specified object and returns the index for that object.
Sort()	Sorts the elements in a list into ascending order.

```
' code that causes the size of a list of
' names to be increased.

Dim lastNames As New List (Of String) (3)
lastNames.Add ("Ram")
LastNames.Add ("Shyam")
lastNames.Add ("Mohan")
lastNames.Add("Sita")  ' capacity is doubled to 6 elements
lastNames.Add("Geeta")
lastNames.Add("Reeta")
lastNames.Add("Hari")  ' capacity is doubled
                          to 12 elements

' Syntax for retrieving a value from a list
  listName[.Item](index)

'Create a list that holds Decimal values using
 collection initializer to assign values to list
Dim salesTotal As New List (Of Decimal) From
    {3275.68D, 4398.55D, 5289.75D, 1933.98D}

'Retrieve the first value from the list
Dim sales1 As Decimal = salesTotals(0)
```

```
'Insert and remove an element from the list
salesTotals.Insert(0, 2745.73D) 'insert a new element
sales1 = salesTotals(0)                    '2745.73
Dim sales2 As Decimal = salesTotals(1)     '3275.68
salesTotals.RemoveAt(1)                    'remove 2nd element
sales2 = salesTotals(1)                    '4398.55

'Display the list in a message box
Dim salesTotalString As String = ""
For Each d As Decimal In salesTotals
    salesTotalString &= d.ToString & vbCrLf
Next
MessageBox.Show(salesTotalString, "Sales Totals")
```



```
'Check for an element in the list and remove it if
 it exists

Dim x As Decimal = 2745.73D
If salesTotals.Contains(x) Then
    salesTotals.Remove(x)
End If

'Sort and search the list
                        'In Array sort is static method
salesTotals.Sort()      'Sort is instance method
Dim sales2Index As Integer = salesTotals.BinarySearch _
                             (sale2)
```

Sorted List

We can implement a collection by using the SortedList() class. A sorted list is useful when you need to look up values in the list based on a key value. Eg: A sorted list consists of item numbers and unit prices, the keys are the item numbers. Then, the list can be used to look up the unit price for any item number.

Each item in a sorted list is actually a KeyValuePair structure that consists of two properties: Key and Value. Here, the Value property can store value types or reference types.

Like a list, you can set the initial capacity of a sorted list by specifying the number of elements in the parantheses when the list is created. Then, if the number of elements in the list exceeds the capacity as the program executes, the capacity is doubled.

Since using a sorted list makes it easy to look up a key and return its corresponding value, this is the right type of collection to use when you need to do that type of lookup. A sorted list is also the right choice when you need to keep the elements in sequence key.

Common properties and methods of the SortedList() class

Property	Description
Item(key)	Gets or sets the value of the element with the specified key. Since Item is the default property, its name can be omitted.
Keys	Gets a collection that contains the keys in the list.
Values	Gets a collection that contains the values in the list.
Capacity	Gets or sets the number of elements the list can hold.
Count	Gets the number of elements in the list.

Method	Description
Add(key, value)	Adds an element with the specified key and value to the sorted list.
Clear()	Removes all elements from the sorted list.
ContainsKey(key)	Returns a Boolean value that indicates whether or not the sorted list contains the specified key.
ContainsValue(value)	Returns a Boolean value that indicates whether or not the sorted list contains the specified value.
Remove(key)	Removes the element with the specified key from the sorted list.
RemoveAt(index)	Removed the element at the specified index from the sorted list.

Properties of the KeyValuePair structure

Property	Description
Key	The key for the SortedList item.
Value	The value associated with the key.

```
'Create and Load a sorted list
Dim salesList As New SortedList(Of String, Decimal)
salesList.Add("Adams", 3274.68D)
salesList.Add("Finkle", 4398.55D)
salesList.Add("Lewis", 5289.75D)
salesList.Add("Potter", 1933.97D)

'Look up a value in the sorted list based on a key
Dim employeeKey As String = "Lewis"
Dim salesTotal As Decimal = salesList(employeeKey)

'Convert the sorted list to a tab-delimited string
Dim salesTableString As String = ""
For Each employeeSalesEntry As KeyValuePair(Of String, Decimal)
                In salesList
    salesTableString &= employeeSalesEntry.Key & vbTab &
        employeeSalesEntry.Value & vbCrLf
Next
MessageBox.Show(salesTableString, "Sorted List Totals")
```

Queues and Stack

A queue is first-in, first-out (FIFO) collection because its items are retrieved in the same order in which they were added.

A stack is last-in, first-out (LIFO) collection because its items are retrieved in the reverse order from the order in which they were added.

Properties and methods of the Queue() class

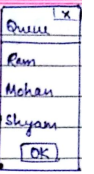
Property	Description
Count	Gets the number of items in the queue.

Method	Description
Enqueue(object)	Adds the specified object to the end of the queue
Dequeue()	Gets the object at the front of the queue and removes it from the queue.
Clear()	Removes all items from the queue.
Peek()	Retrieves the next item in the queue without deleting it.

Example of Queue

```
Dim nameQueue As New Queue (of String)
nameQueue.Enqueue("Ram")
nameQueue.Enqueue("Mohan")
nameQueue.Enqueue("Shyam")
```

```
Dim nameQueueString As String = ""
Do While nameQueue.Count > 0
    nameQueueString &= nameQueue.Dequeue & vbCrLf
Loop
MessageBox.Show(nameQueueString, "Queue")
```

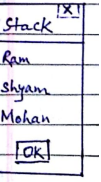


Properties and methods of the Stack() class

Property	Description
Count	Gets the number of items in the stack.

Method	Description
Push(Object)	Adds the specified object to the top of the stack.
Pop()	Gets the object at the top of the stack and removes it from the stack.
Clear()	Removes all items from the stack.
Peek()	Retrieves the next item in the stack without deleting it.

```
'Example of stack
Dim nameStack As New Stack(Of String)
nameStack.Push("Ram")
nameStack.Push("Shyam")
nameStack.Push("Mohan")
Dim nameStackString As String = ""
Do While nameStack.Count > 0
    nameStackString &= nameStack.Pop & vbCrLf
Loop
MessageBox.Show(nameStackString, "Stack")
```



Array List

Array List is the most common untyped collection. An array list works like a list. However, since an array list defines an untyped collection, there are few differences.

When you declare an Arraylist class, you don't define the type within parantheses. Instead, each element in the array list is stored as an Object type. As a result, any value type that you store in the array list must be converted to a reference type. To do that, an object is created and the value is stored in that object. The process of putting a value in an object is known as **boxing**, and it's done automatically whenever a value type needs to be converted to a reference type.

When you retrieve an element from an array, you must cast the Object type to the appropriate data type. The process of getting a value out of the Object type is known as **unboxing**.

```
'Create an array list that holds decimal values
.Dim salesTotals As New Arraylist From
    {3275.68D, 4398.55D, 5289.75D, 1933.98D}
'Retrieve the first value from the array list
Dim sales1 As Decimal = CDec(salesTotals(0))
```

```
' Insert and remove an element from the array list
salesTotals.Insert(0, 2745.73D)
sales1 = CDec(salesTotals(0))
Dim sales2 As Decimal = CDec(salesTotals(1))
salesTotals.RemoveAt(1)
sales2 = CDec(salesTotals(1))

' Display the array list
Dim salesTotalsString As String = ""
For Each d As Decimal In salesTotals
    salesTotalsString &= d.ToString & vbCrLf
Next
MessageBox.Show(salesTotalsString, "Sales Total")

' Check for an element in the array list and removes it if it exists
Dim x As Decimal = 2745.73D
If salesTotals.Contains(x) Then
    salesTotals.Remove(x)
End If

' Sort and search the array list
salesTotals.Sort()
Dim salesIndex As Integer = salesTotals.BinarySearch(sales2)
```

Me - The Me keyword provides a way to refer to the specific instance of a class or structure in which the code is currently executing. Me behaves like either an object variable or a structure variable referring to the current instance. Using Me is particularly useful for passing information about the currently executing instance of a class or structure to a procedure in another class, structure or module.

```
Sub ChangeColor (FormName As Form)
    Randomize()
    FormName.BackColor = Color.FromArgb
    (Rnd()*256, Rnd()*256, Rnd()*256))
End Sub
```

Usage

```
ChangeColor (Me)
```

My - The My feature provides easy and intuitive access to a number of .NET Framework classes, enabling the Visual Basic user to interact with the computer, application, settings, resources and so on.

MyBase - The MyBase keyword behaves like an object variable referring to the base class of the current instance of a class. MyBase is commonly used to access base class members that are overriden or shadowed in a derived class. MyBase.New is used to explicitly call a base class constructor from a derived class constructor.

MyClass - The MyClass keyword behaves like an object variable reffering to the current instance of a class as originally implemented. MyClass is similar to Me but all method calls on it are treated as if the method were NotOverridable.

Ex.
```
Public Class ~~Top~~ ~~Bottom~~ Top
    Public Overridable Sub TestProc()
        Console.WriteLine("Top TestProc")
    End Sub
End Class
```

```
Public Class Mid
    Inherits Top
    Public Overrides Sub TestProc()
        Console.WriteLine("Mid TestProc()")
    End Sub
End Class
Public Class
    Public Sub Test()
        MyBase.TestProc()
        MyClass.TestProc()
        Me.TestProc()
        TestProc()
    End Sub
End Class

Public Class Bottom
    Inherits Mid
    Public Overrides Sub TestProc()
        Console.WriteLine("Bottom TestProc()")
End Sub
End Class

Public Sub Main()
    Dim x As New Bottom
    x.Test()
End Sub
```

Difference between Shadowing and Overriding

When you define a class that inherits from a base class, you sometimes want to redefine one or more of the base class elements in the derived class. Shadowing and overriding are both available for this purpose.

Shadowing and overriding are both used when a derived class inherits from a base class and both redefine one declared element with another. But there are significant difference between the two.

* Shadowing protects against a subsequent base-class modification that introduces a member you have already defined in your derived class.

→ Overriding achieves polymorphism by defining a different implementation of a procedure or property with the same calling sequence.

* Base class element can not enforce or prohibit shadowing.

→ Base class element can specify MustOverride, NotOverridable or Overridable

* Shadows keyword recommended in derived class; Shadows assumed if neither Shadows nor Overrides specified. (Compiler issues warning)

→ Overridable or MustOverride required in base class; Overrides required in derived class.

* Shadows can change readability and writability of redefining element.
→ Can not change readability or writability of overridden property.

* Shadows can change access level of redefining element.
→ Can not change access level of overriden element.

* Any declared element type can be redefined.
→ Only a procedure (function, Sub, or Operator) or property can be redefined.

* Redefining can be done for any element type.
→ In overriding only a procedure or property with the identical calling sequence.

Files and Data Streams

System.IO – The System.IO namespace provides a variety of classes for working with files and for managing directories, files, and paths.

Classes for managing directories, files and paths –

* The classes for managing directories, files, and paths are stored in the System.IO namespace.
* To use the classes in the System.IO namespace, you should include an Imports statement. Otherwise, you have to qualify the references to its classes with System.IO.
* All of the methods of the Directory, File and Path classes are shared methods.

System.IO classes used to work with drives and directories –

Class	Description
Directory	Used to create, edit, delete or get information on directories.
File	Used to create, edit, delete or get information on files.
Path	Used to get path information from a variety of plateforms.

Anadaram

PAGE No.
DATE

Common methods of Directory class -

Method	Description
Exists(path)	Returns a Boolean value indicating whether a directory exists.
CreateDirectory(path)	Creates the directories in a specified path
Delete(path)	Deletes the directory at the specified path. The directory must be empty.
Delete(path, recursive)	Deletes the directory at the specified path. If True is specified for the recursive argument, any subdirectories and files in the directory are deleted. If False is specified, the directory must be empty

Common methods of File class -

Method	Description
Exists(path)	Returns a Boolean value indicating whether a file exists.
Delete(path)	Deletes a file.
Copy(source, dst)	Copies a file from a source path to a destination path.
Move(source, dst)	Moves a file from a source path to a destination path.

Sundaram

PAGE No.
DATE

Ex.1
```
Dim dir As String = "C:\VB 2010 Files\"
If Not Directory.Exists(dir) Then
    Directory.CreateDirectory(dir)
End If
```

Ex.2
```
Dim path As string = dir & "Products.txt"
If File.Exists(path) Then
    File.Delete(path)
End If
```

How files and streams work

* An input file is a file that is read by a program; an output file is a file that is written by a program. Input and output operations are often referred to as I/O operations or file I/O.
* A stream is the flow of data from one location to another. To write data, you use an output stream. To read data, you use an input stream. A single stream can also be used for both input and output.
* To read and write text files, you use text streams. To read and write binary files, you use binary streams.

Fundamentals

PAGE No.
DATE

Two types of files

Type	Description
Text	A file that contains text (string) characters. The fields in each record are typically delimited by special characters like tab or pipe characters, and records are typically delimited by new line character.
Binary	A file that can contain a variety of data types.

Two types of streams

Stream	Description
Text	Used to transfer text data.
Binary	Used to transfer binary data.

System.IO classes used to work with files and streams

Class	Description
FileStream	Provides access to input and output files.
StreamReader	Used to read a stream of characters.
StreamWriter	Used to write a stream of characters.
BinaryReader	Used to read a stream of binary data
BinaryWriter	Used to write a stream of binary data.

How to use the FileStream class

Syntax for creating a FileStream object

```
New FileStream(path, mode [, access [, share]])
```

Members in the FileMode enumeration

Member	Description
Append	Opens the file if it exists and seeks to the end of the file. If the file doesn't exist, it's created. This member can only be used with Write file access.
Create	Creates a new file. If the file already exists, it's overwritten.
CreateNew	Creates a new file. If the file already exists, an exception is thrown.
Open	Opens an existing file. If the file doesn't exist, an exception is thrown.
OpenOrCreate	Opens a file if it exists, or creates a new file if it doesn't exist.
Truncate	Opens an existing file and truncates it so its size is zero bytes.

Members in the FileAccess enumeration

Member	Description
Read	Data can be read from the file but not written to it.
ReadWrite	Data can be read from and written to the file. This is the default.
Write	Data can be written to the file but not read from it.

Members in the FileShare enumeration

Member	Description
None	The file cannot be opened by other applications.
Read	Allows other applications to open the file for reading only. This is the default.
ReadWrite	Allows other applications to open the file for both reading and writing.
Write	Allows other applications to open the file for writing only.

Common method of the FileStream class

Method	Description
Close()	closes the file stream and releases any resources associated with it.

Ex.1- Code to create a FileStream object for writing.

```
Dim path As String = "C:\VB 2010\Files\Products.txt"
Dim fs As New FileStream (path, FileMode.Create,
                          FileAccess.Write)
```

Ex.2- Code to create a new FileStream object for reading

```
Dim path As String = "C:\VB2010\Files\Products.txt"
Dim fs As New FileStream (path, FileMode.Open,
                          FileAccess.Read)
```

How to use the exception classes for file I/O –

* To catch any I/O exception, you can use the IOException class.
* To catch specific I/O exceptions, you can use the exception classes that inherit the IOException class such as DirectoryNotFoundException, FileNotFoundException, EndOfStreamException.

The exception classes for file I/O

Class	Description
IOException	- The base class for exceptions that are thrown during the processing of a stream, file or directory.
DirectoryNotFoundException	- Occurs when part of a directory or file path can't be found.
FileNotFoundException	- Occurs when a file can't be found.
EndOfStreamException	- Occurs when an application attempts to read beyond the end of a stream.

Ex.
```
Dim dirPath As String = "C:\VB 2010\Files\"
Dim filePath As String = dirPath & "Products.txt"
Dim fs As FileStream
Try
    fs = New FileStream(filePath, FileMode.Open)
    ' # code that uses the file stream
    ' # to read and write data from the file
```

```
Catch ex As FileNotFoundException
    MessageBox.Show(filePath & " not found.", "File
        Not found")
Catch ex As DirectoryNotFoundException
    MessageBox.Show(dirPath & " not found.", "DirectoryNotFound")
Catch ex As IOException
    MessageBox.Show(ex.Message, "IOException")
Finally
    If fs IsNot Nothing Then
        fs.Close()
    End If
End Try
```

How to work with binary files

To read and write data in a binary file, you use the BinaryReader and BinaryWriter classes.

How to write a binary file –

First of all we create a BinaryWriter object using the following syntax:

```
New BinaryWriter(stream)
```

To do that we must supply a FileStream object as the argument for the constructor of the BinaryWriter class. This links the stream to the

BinaryWriter object so it can be used to write to the file.
After creating BinaryWriter object, we can use its Write method to write all types of data. This method begins by figuring out what type of data has been passed to it. Then it writes that type of data to the file.

Common methods of the BinaryWriter class

Method	Description
Write(data)	Writes the specified data to the output stream.
Close()	closes the BinaryWriter object and the associated FileStream object.

Ex.1
```
Dim binaryOut As New BinaryWriter(
    New FileStream(path, FileMode.Create, FileAccess.Write))
For Each product As Product In products
    binaryOut.Write(product.Code)
    binaryOut.Write(product.Description)
    binaryOut.Write(product.Price)
Next
binaryOut.Close()
```

In the above code, a binary writer is created for a file stream that specifies a file that has write-only access. Since the mode argument has been set to

Create, this will overwrite the file if it exists, and it will create the file if it doesn't exist. Then, a for each loop is used to write the elements in a List() collection named products to the file. Since each element in the List() collection is an object of the Product class, each property of the Product object is written to the file seperately using the Write method. After all of the elements in the List() collection have been written to the file, the Close method is used to close both the BinaryWriter and the FileStream objects.

Ex.2

```
Imports System.IO
Module Module1
    Sub Main()
        Dim arr() As Int32 = {2, 6, 9, 11, 45, 87, 777, 23,
                              266, 44, 83, 94}
        Using writer As BinaryWriter = New
            BinaryWriter(File.Open("file.bin", FileMode.Create))
            For Each value As Int32 In arr
                writer.Write(value)
            Next
        End Using
    End Sub
End Module
```

How to read a binary file -

Like the BinaryWriter class, the argument that we pass to the BinaryReader is the name of the FileStream object that connects the stream to a file.

In a binary file, there's no termination character to indicate where one record ends and another begins. Because of that, we can't read an entire record at once. Instead, you have to read one character or one field at a time. To do that, you use the Read methods of the BinaryReader class. You must use the appropriate method for the data type of the field that you want to read.

Before you read the next ~~next~~ character or field, you want to be sure that you aren't at the end of the file. To do that you use the PeekChar method. Then, if there's at least one more character to be read, this method returns that character without advancing the cursor to the next position in the file. If there isn't another character, the PeekRead method returns a value of -1. Then you can use the Close method to close the binary reader and the associated file stream.

Syntax for creating a BinaryReader object

```
New BinaryReader(stream)
```

Common methods of the BinaryReader class

Method	Description
PeekChar()	Returns the next available character in the input stream without advancing to the next position. If no more characters are available, this method returns -1.
Read()	Returns the next available character from the input stream and advances to the next position in the file.
ReadBoolean()	Returns a Boolean value from the input stream and advances the current position of the stream by one byte.
ReadByte()	Returns a byte from the input stream and advances the current position of the stream accordingly.
ReadChar()	Returns a character from the input stream and advances the current position of the stream accordingly.
ReadDecimal()	Returns a decimal value from the input stream and advances the current position of the stream by 16 bytes.
ReadInt32()	Returns a 4-byte signed integer from the input stream and advances the current position of the stream by 4 bytes.
ReadString()	Returns a string from the input stream and advances the current position of

the stream by the number of characters in the string.

Close() Closes the BinaryReader object and the associated FileStream object.

Ex.1

```
Dim binaryIn As New BinaryReader(New
    FileStream(path, FileMode.OpenOrCreate, FileAccess.Read))
Dim products As New List(Of Product)
Do While binaryIn.PeekChar <> -1
    Dim product As New Product
    product.Code = binaryIn.ReadString
    product.Description = binaryIn.ReadString
    product.Price = binaryIn.ReadDecimal
    products.Add(product)
Loop
binaryIn.Close()
```

In the above code, a FileStream object is created for a file that will have read-only access. Since the mode argument for the file stream specifies OpenOrCreate, this opens an existing file if one exists or creates a new file that's empty and opens it. Then, a new BinaryReader object is created for that file stream. Finally, the Do loop that follows is executed until the PeekChar method returns a value of -1, which means the end of the file has been reached.

Within the Do loop, the three fields in each record are read and assigned to the properties of the Product object. Because the first two fields in each record contain string data, the ReadString method is used to retrieve its contents. Then, the Product object is added to the List() collection. When the Do loop ends, the Close method of the BinaryReader object is used to close both the BinaryReader and the FileStream objects.

Ex.2

```
Module Module1
    Sub Main()
        Using reader As New BinaryReader(File.Open(
            "file.bin", FileMode.Open))
            'Dim pos As Integer = 0
            'Dim length As Integer = reader.BaseStream.Length
            While reader.PeekChar <> -1
                Dim value As Integer = reader.ReadInt32
                Console.WriteLine(value)
                'pos += 4
            End While
        End Using
    End Sub
End Module
```

Sundaram

PAGE No.

How to work with text files

To read and write characters in a text file, you use the StreamReader and StreamWriter classes. When working with text files, you often need to use string, date and numeric data.

How to write text file —

Create a StreamWriter object which takes a FileStream object as its argument as follows:

```
New StreamWriter (stream)
```

After creating StreamWriter object we write any data to a text file by using the Write, and WriteLine methods. When we use WriteLine method, a line terminator is automatically added. Typically, a line terminator is used to end each record. The fields in a record are typically separated by special characters, such as pipe character, by adding those characters through code.

Both the Write and WriteLine methods of StreamWriter class are overloaded to accept any type of data. As a result if we pass a non-string data type to

* Dim sw As New StreamWriter (New FileStream(path, FileMode.Create, FileAccess.Write))

either of these methods, the method converts the data type to a string that represents the data type and then it writes that string to the stream. To do that, these methods automatically call the ToString method of the data type.

Common methods of the StreamWriter class-

Method	Description
Write(data)	Writes the data to the output stream
WriteLine(data)	Writes the data to the output stream and appends a line terminator (usually a carriage return and a line feed).
Close()	Closes the StreamWriter object and the associated FileStream object.

Ex.-
```
'Get the directories currently in the C drive
Dim cDirs As DirectoryInfo() = New
    DirectoryInfo("C:\").GetDirectories()
* Using sw As StreamWriter = New StreamWriter
    ("CDriveDirs.txt")
    For Each Dir As DirectoryInfo In cDirs
        sw.WriteLine(Dir.Name)
    Next
End Using
```

How to read a text file —

Create a StreamReader object, you can use a FileStream object as the Argument.

Basic syntax for creating StreamReader object —
New StreamReader(stream)

Common methods of StreamReader class —

Method	Description
Peek()	Returns the next available character in the input stream without advancing to the next position. If no more characters are available, this method returns -1.
Read()	Reads the next character from the input stream.
ReadLine()	Reads the next line of characters from the input stream and returns it as a string
ReadToEnd()	Reads the data from the current position in the input stream to the end of the stream and returns it as a string. This is typically used to read the contents of an entire file.
Close()	Closes both the StreamReader and the associated FileStream object.

Ex. -

```
Dim line As String = ""
Using sr As StreamReader = New StreamReader(
    "CDriveDirs.txt"
    Do
        line = sr.ReadLine()
        Console.WriteLine(line)
    Loop Until line Is Nothing
End Using
```