How are you Date No.

Basic Concepts (VB.net)

May you enjoy happiness in the coming year Date No.

* language fundamentals:

VB.net takes advantage of English words and syntax to make code as readable as possible.

```
Module HelloWorld
  Public Sub Main()
      Console.Writeline("Hello, World!")
      Console. Readline ()
  End Sub
End Module
```

VB.net (readability)

```
using System;
class HelloWorld
{
   public static void Main()
    {
        Console. Writeline ("Hello, world!");
        Console. Readline();
    }
}
```

C#.net (conciseness)

* Case Insensitivity -
VB.net does not care whether characters are uppercase or lowercase in a program, In addition, the language is also relaxed in regard to the parts of the language that can be understood implicitly.

```
Module  HelloWorld
  Sub  Main
      Console.Writeline ("Hello, world!")
       Console.Readline
  End  Sub
End Module
```

there is no parenthesis after the Sub Main and Console.Readline statements because leaving off the parentheses means that the subroutine has no parameters or arguments as empty parentheses indicates.

* Line Orientation = (lines can not end just anywhere)
There has to be one space before a line continuation because the ~~underscope~~ underscore can also be part of a name.

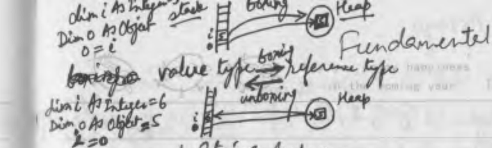
e.g.
```
     Module _
        HelloWorld
         Sub Main _
          - - .

          - .

       End Sub : End module
```
(statement seperator to put more than one statement on a line) (a subroutine or function declaration must ~~be~~ always be first statement on a line)

* Comments = (') single quote character or REM
* Declarations & Names ⇒ Dim x As Integer, ByVal
* forward References ⇒ no need of function prototype declaration in the begining
* Accessibility (access level) = Public, Private, Friend

```
Class Employee
    Public Name As String
    Private Salary As Double
  { Public Sub CompareSalary (ByVal Other As Employee)
  }     Console.write(Name & "makes")
            - - - - - .
    End Sub - -.
End Class
```

```
Sub Main ≡ Function Main (ByValue Args() As String) As Integer
               For Each Arg As String In Args
                   Console.Writeline(Arg)
               Next Arg
               Return 1
           End Function
```

Fundamental types



* char and String types –

– the default value of char data type is chrW(0), null character

– " " " "String " " is Nothing, ""

e.g. dim c As Char

c= "a"c → a string literal with one character in it, ~~follow~~ followed by the ~~$~~ character "c"

* Object data type = (universal type) = reference type

– a variable typed as Object can have values of any type assigned to it

e.g., dim o1, o2, o3 as Object

o1 = 5

o2 = "abc"

o3 = #8/23/70 4:30:24 AM#

* Conversion operators –

CBool (<expr>)	converts <expr> to Boolean
CByte (<expr>)	Converts <expr> to Byte
CInt (<expr>)	" " " Integer
CDbl (<expr>)	" " " Double
CChar (<expr>)	" " " character (Char)
CStr (<expr>)	" " " String
CObj (<expr>)	" " " Object
✓CType (<expr>, <type>)	" " " <type>
DirectCast(<expr>, <type>)	" " " " without lang-specific conversions

Arrays

How are you

1-Dimensional:

dim a() as integer = {1, 2, 9, 4} → to initialize dim a(,) as integer = {{1,2},{3,4}}

Redim a(9) → dimension size = 10 elements

2-Dimensional: dim a(,) as integer

dim x,y as integer

to allocate new array: Redim a(9,9) → dimension sizes =

* the length of a dimension in a ReDim statement is specified in terms of its upper bound.

e.g., ReDim a(9,9) creates an array of two dimensions with ten elements (0 through 9) in each dimension.

* to change the size of the dimensions of an existing array;

e.g. dim a() as integer = {6, 8, 10,

ReDim a(9)

ReDim Preserve a(19)

Erase a = clears one or more arrays, resetting them to their original uninitialized state

* another way to clear an array is to set the array variable to Nothing

May you enjoy happiness
in the coming year Date . . No.

* Arrays of arrays: →

z()(,)(,,)

```
dim a()() as integer
dim x,y as integer
Redim a(3)
For x=0 to 3
  Redim a(x)(x+1)
  For y=0 to x+1
    a(x)(y)= x+y
  Next y
Next x
```

* Enumeration =

dim a as colors
a= Colors.Red

```
Enum Colors
    Red=1
     orange=2
       yellow=4
end enum
```

enum colors as Byte

As integer by default

0

1

2

How are You Date No.

* Operators =

exponentiation	x ^ y
integer division	x \ y
integer remainder	x mod y
concatenation	x & y
inequality	x <> y
type equality	TypeOf x IS y → type e.g. (typeof o is integer)
reference equality	x IS y
String matching	"x" Like "y"
Bitwise AND / logical	x And y
" Not / logical	Not x
- - AndAlso / logical	x AndAlso y ≡ And
- - / logical	x Or y
- - / logical	x OrElse y ≡ Or
- - / logical	x Xor y
return specified type	GetType (x) e.g. dim c as collection t = GetType(collection)

May you enjoy happiness in the coming year

```
class C1
end class
Module test
    Sub main()
    Dim a, b, c As C1
     a = New C1()
     b = a
✓   if a Is b then      → Is (Comparision operators)
    - - - - -
```

* Logical and bitwise operators =

```
✓ if x <> 0 AndAlso y\x = 10 then
    - - - -
    a = Not 143
    b = 312 And 43
    c = 5823 Or 412
    d = 314 Xor 123
```

* When applied to the integer values, the operators Not, And, Or and Xor function as bitwise operators, operating on the binary representation of the values.

e.g. 213 And 57 = 17

213	1	1	0 1 0 1 0 1	AND
57	0	0	1 1 1 0 0 1	
~~And~~ 17	0	0	0 1 0 0 0 1	

Local decl. Statements —

```
dim y        ' → as Object type by default
Dim c, d, e As Double, f, g As Single
```

* type characters are ~~also g~~ also used to declare the type of a variable.

```
Dim  x%        ' type is integer
Dim  ~~x~~ y@        '  "   " decimal
Dim  z$        '  "   " String
Dim  a&        '  "   " long
Dim  b!        '  "   " Single
Dim  c#        '  "   " Double
```

```
Dim x As integer
    x% = 5
```

* initializers =   dim c as integer=5, b as String="Hello"
* Constants = Const ~~const~~ Lower as integer=1
  Const   upper as integer = 10
* Static locals:= a special kind of local variable that retain ~~their~~ their values across calls to the method.

```
sub incrementNumber ()
    static Number As integer=0
    ~~console~~ ~~Number~~ console.writeline(Number)
      Number += 1
end Sub
```

```
sub Main()
    for i as integer=1 to 5
        incrementNumber ()
    Next i
end sub
```

* implicit locals =
  the compiler assumes an implicitly declared local variable with a type of Object.

```
sub test()
dim y as integer
 if y<0 then
  x=y*20
  Console. writeline(x)
 end if
 x=5
end sub
```

* With statement simplifies repeatedly accessing the members of a value.

```
With TextBox1
   TextBox1 = new TextBox()
   .Tabindex=0
   .ForeColor = Color.Red
   .text = "Red"
   .show()
end with
```

* Conditional statements =

```
if - . then
   - - .
elseif - - . then
   - .
else
   - - .
end if
```

* select statement =

Select Case x Case "Red" . . . Case "Green" - - - end select	Selet Case x Case 0 - - . Case 1 To5 - - Case 6 To 10 - - . end Select	Select Case x Case 1,3,5,7 - - . . - - - Case Is <0, Is >10 - - - - - - - - - Case Else - - - - . . . . end select

* Looping statements =

For x As integer =1 To 10 ' - - - - . Next x	Dim x,y,z As integer For x=1 To 10 For y=1 To 10 For z=1 To 10 - - - . Next z Next y Next x	Dim x As integer For x=10 To 1 Step -1 - - - . Next
✓ For Each z In x - - - . . Next ~~y~~ z	For each z as integer In x - - - Next z	

✓ while x>0 - . End while	Do while x>0 - - - - loop	Do Until x=11 - - - . loop	Do - - . . loop while x>0
			Do - loop untill x=11

* Collection types = - follows design pattern that allows its members to be enumerated.

= ~~any type a type that~~

✓ = a collection type ~~is~~ is any type that implements the interface System.IEnumerable or satisfies the following conditions.

(i) the type contains a method named GetEnumerator that returns a value of some type T.

(ii) the enumerator type T contains a method named MoveNext that returns a Boolean value.

(iii) the enumerator type T contains a read-only or read-write property named Current.

```
Class IntegerCollection
   Class IntegerEnumerator
      Private Collection As IntegerCollection
      Private Index As Integer=-1
      Public sub New (ByVal Collection As IntegerCollection)
         Me.Collection = Collection
      end sub
      public Sub Reset()
         index = -1
      end sub
      Public Function MoveNext() As Boolean    }
         if index < Collection.Length then     }
            index += 1                         }  (ii)
         end if                                }
         return (index = Collection.Length)    }
      end Function
```

```
(iii) { public ReadOnly property Current() As Integer
          Get
              if Index = -1 OrElse index = Collection.length then
              throw New invalidOperationException()
              end if
              Return Collection(index)
          end Get
        end property
end class
   private values() As integer
   public sub New (Byval values() As integer)
          Me. values = values
   end sub
   public Readonly property length() as integer
        Get
          Return values. length
        end get
   end property
(i) { public Function GetEnumerator As integerEnumerator
           return new integerEnumerator(Me)
      end function
end class
```

* Branching statements =

```
x = FetchValue()
-> if x<0 then Goto skipdivision
   y = 1\x
   skipDivision :
   Return y
```

```
for x as integer = 1 to 10
    dim y as integer = x
    while y>0
        y -= 1
        if y=5 then
  -->       exit For
        end if
    end while
next x
```

= Program flow statements =

* class libraries (i.e., DLLs) are not executable. ✓

```
Module test
  sub main()
    dim x as integer
    while true
      x = CInt(Console.Readline())
      if x=0 then
  ---->   End
      end if
    end while
  end sub
end module
```

```
Module Test
  sub main()
    dim x as integer
    While x>0
      x = CInt(Console.Readline())
      if x=-1 then
  --->    stop
      end if
    endwhile
  end sub
end module
```

* Synclock =

```
Imports System.Threading
Module Test
    dim Array(10000) As integer
    dim CurrentIndex as integer = 0
    sub FillArray()
 -->    Synclock Array.SyncRoot
            for Number as integer = 1 to 5000
                Array(CurrentIndex) = Number
                CurrentIndex += 1
            Next number
        end synclock
    end sub
    sub main()
        dim t1 as Thread = New Thread(AddressOf FillArray)
        dim t2 as thread = New thread(AddressOf FillArray)
        t1. start()
        t2. start()
    end sub
end module
```

exceptions

= Common exception types =

System.ApplicationException = application-specific exception occurred.
System.ArgumentException = argument is invalid
System.ArgumentNullException = argument is null /Nothing
System.ArgumentOutOfRangeException = argument was not within its valid range
Sytem.DivideByZeroException = An operation divide by zero
Sytem.DllNotFoundException = The Lib clause of a Declare statement was not found.
Sytem.ExecutionEngineException = .Net framework encountered an internal error
System.InvalidCastException = Conversion from one type to another was not valid
System.NotSupportedException = method is not supported
System.NullReferenceException = program tried to use a Nothing value in an invalid way
Sytem.OutOfMemoryException = program has run out of memory
Sytem.OverflowException = operation overflowed
System.Runtime.InteropServices.COMException = exception occurred while COM object was being called

* more general exception types should come last.

```
try
  ....
catch...
catch...
  ⁝ ...
(error)
finally
  .....
end try
```

```
Try
  ...
catch e As Exception
  ....
Finally
  .....
End Try
```

```
Try ...
  ...
Catch e As Exception
 Console.Writeline(--- . )
 Throw e
End Try
```

* catch block may have conditional statements attached to them to provide additional conditions for handling an exception.
e.g, Catch e AS Exception When count<10

= Module & Namespace =

```
~~[illegible]~~ Imports- ...
Namespace ---
        Module -.
          ---
        End Module
        Namespace --
            Module...
              ---  ~~Class~~ .
            ~~End Module~~  ---
            ~~Module~~  ~~End Class~~
               Sub --
               end sub   enum .. end enum ✓
               Function -- (Byval ---, ByRef ---) As --
                  return ...
               End Function
            End Module  sub main ---- end sub ✓
            class ---
               Class --
               End class
            End class
            ~~End Module~~
        end Namespace
End Namespace
```

= **Preprocessing** = statements processed before code compilation
- not considered part of the code
- begin with # sign

```
# Const DEBUG = True
  Module --
     Sub Main()
   #If DEBUG Then
      - - - -
   #ElseIf RETAIL Then
      - - - -
   #Else
      - - -
   #End If
      - - -
   End Sub
  End Module
 # Const DEBUG = False
```

(no effect on compilation)

(must enclose entire blocks)

```
Module - - -
#Region "- - -"
   Sub Main()
     - - -
   end sub
#End Region
End Module
```

= **Classes and Structures** :=

```
Structure Customer
   Public Name As String
   [Public Sub New(ByVal x As Integer, ...)
     Me.x = x
    End Sub
End Structure
- - - - -
Dim x As Customer   'Valuetype
```

```
Class Customer
      Inherits Person
      Implements IDisposable
   Public Name As String
 - Public Shared PrintPrefix As Boolean = True
 - [Public Sub New(ByVal Name As String, ...)
    ~~End Sub~~ MyBase.New(Name)
      Me.Name = Name
      ...
    End Sub
End Class
  Shared Sub New()..
    End Sub
- - ..
Dim x As Customer
   x = New Customer()   'Reference type
Dim x As New Customer
```

* **Overloading methods** —

```
Sub Print (ByVal i As Integer)
  - - -
End Sub
Sub Print (ByVal d As double)
  - - -
End Sub
```

ByVal

different types parameters

```
Function GetValue(ByVal i As Integer) As Integer
  - - - .
End Function
Function GetValue(ByVal d As Double) As ~~Boolean~~ Double
  - - - .
End Function
```

~~Array~~ Return Type

* **Declare statement and Alias**:

```
Declare Function GetWindowsDirectory Lib "Kernel32" (ByVal Buffer As String, ...) As Integer
                                      ^ Alias "GetWindowsDirectoryA"
```

* **Fields and Properties** =

```
Class Order
     Private _Cost As Double
     Public Property Cost() As Double
                  { Get
   ReadOnly         Return _Cost
  (immutable)     { End Get
                  { Set (ByVal value As Double)
   WriteOnly          - - -
                      _Cost = value
                  { End Set
     End Property
     Public Quantity As Integer
End Class
```

```
Module Test
   Public ReadOnly myAddress As Address = _
                       New Address(- - - )
     - - .
End Module
```

```
Structure Address
   Public ReadOnly Street, ... As String
   - - .
   - -
```

* **Index Properties** : —

```
Class OrderCollection
    Private _Orders(1) As Order
    Public Properties Orders (ByVal Index As Integer) As Order
        Get
           If _Orders(Index) Is Nothing Then
              _Orders (Index) = New Order ()
           End If
           Return _Orders(Index)
        End Get
        Set (Value As Order)
           _Orders(Index) = Value
        End Set
    End Property
End class
```

```
Module Test
   Sub Main()
     - - -
     OrderCollection.Orders(5).Cost
                           = 10.34
     - - - .
```

* put the call to Dispose method in a Finally block

* Shared constructor will be run before anything that could depend on it can be accessed.

* Shared members are shared by all instances of the class or structure i.e., if one instance of a class or structure changes the value of a shared member, all other instances of the class or structure will see new value.

How are you
Date. No.

Inheritance =

```
Class Person
    Public Name As String
    - - .
End class
Class Employee
    Inherits Person
        .
End class
Module Test
 Sub main()
   Dim p As Person = New Employee()
       p.Name = "John"
       - ..
End module
```

Overriding: (to change the implementation of derived methods)

```
Class Person
    Public Name As String
    - - - - - -
    Overridable Sub Print()
        Console.writeline(Name)
    End sub
End Class

Class Employee
    Inherits Person
    Overrides Sub Print()
        Console.writeline(Name)     ] ≡ MyBase.Print()
        Console.writeline("Salary = & Salary)
    End Sub
    Public Salary As Integer
End Class
```

* Abstract Classes & Methods:

- can never directly be created.
- may have constructors to initialize methods or pass values along to base class contructors.

```
MustInherit Class Person
      Public Name As String
      MustOverride Sub PrintName()
      Sub Print()
          PrintName()  <-
      End sub  ...
End Class
Class Customer
    Inherits Person
    Overrides Sub PrintName()
        - - - :
    End Sub
End Class
```

d_hcomp.h

May you enjoy happiness in the coming year Date. No.

```
class hCompress {
public: hCompress(const string& fname, bool v = false);
        void setFile(const string& fname);
        void compress();
        double compressionRatio() const;
        int size() const;
        void displayTree() const;
private: fstream source;
         fstream dest;
        vector<int> charFreq, charLoc;
        int numberLeaves;
        short treeSize;
        vector<huffNode> tree;
        bool verbose;
        long fileSize;
        long totalBits;
        bool oneChar;
        bool fileOpen;
        void freqAnalysis();
        void buildTree();
        void generateCodes();
        void writeCompressedData();
        void treeData();
};
```

```
hCompress hc("demo.dat", true);
   hc. compress();
```

```
void hCompress :: writeCompressedData () {
  bitVector  compressedData(totalBits);
    int  bitPos, i, j ;
   unsigned  char ch;
  source. clear (0) ;
   source.seekg(0, ios :: beg);
    bitPos =0;
   while ( true ) {  ch = source.get();
                  if(! source)
                         break;
                  i = charLoc [ch];
              for(j=0 ;j< tree [i].numberOfBits; j++)
             { if(tree[i]. bits. bit(j) ==1)
             compressedData.set( bitPos);
             bitPos ++;
               }
          }
        compressedData. write (dest);
    }
```

* **Interface** :— defines a set of methods, properties, and events that make up a particular capability
  - a contract that a type fulfills when it implements the interface.
  - allows public access level (always)
  - must supply an implementation for all the members of interface

```
Interface ISizable
         Inherits IDrawable, IMovable
  ReadOnly Property Height() As Integer
  - - - - - - - - - -
  Sub Resize (ByVal - -, - - -)
  Event Resized (ByVal - - -, - - -)
End Interface
```

```
Class Square
    Implements ISizable, IComparable, - -
  - - -
    Private _Height, - - -   As - -
    Public ReadOnly Property - - - - - -  _
                  Implements ISizable.Height
      Get
        Return _Height
      End Get
    End Property
    - - - -
    Public Sub SquareResize( - - ) - - Implements - -
      - - - -
      RaiseEvent SquareRe- - ( - - -, - - )
    End Sub
    Public Event Squa- - ( - - - ) Implements - - -
      - -
End Class
```

```
Interface IClickable
    Event Click (ByVal sender As Object, ByVal e As EventArgs)
    Event Release(  "                              "          )
End Interface

class Square
  Implements IClickable

  Event click ( - - -, - - - ) Implements IClickable.Click, IClickable.Release
End class
```

* Events and Delegates =

— Defining and Raising Events =

```
class Button
    Private X,Y As Integer
✓   Public Event click()
    Public Event Moved(- - -;- -)

End class

    Sub Move(- - -;- - -)
        Me.X = X
        - - -
        RaiseEvent Moved(X,Y)
    End Sub
```

Declarative Event Handling =

```
class Form1
    Public WithEvents Button1 As Button
    - - - - -
    Public Sub Button_Click() Handles Button1.click
    End Sub - - -
End class
```

* Handling Events Dynamically :—

```
class Form1
    Public Buttons As Arraylist = New ArrayList()
    Public Sub CreateButton()
        Dim NewButton As Button = New Button()
        AddHandler NewButton.click, AddressOf Me.Button_click
        Button.add(NewButton)
    End Sub
    Public Sub DeleteAllButtons()
        For Each Button As Button in Buttons
            RemoveHandler Button.Click, AddressOf Me.Button_click
        Next Button
        Buttons.Clear()
    End sub
    Public sub Button_click()
        MsgBox("Button1 was clicked!")
    End sub
End class
```

* Events are built on top of delegates.

Delegates = are types that represents references to methods.

```
Delegate Sub SubroutineDelegate (ByVal x as Integer, ByVal y as Integer)
Delegate Function FunctionDelegate () As Integer
 class Button
    Public Sub Move (ByVal x As Integer, ByVal y as Integer)
    End Sub
End class
Module Test
    Sub Main()
        Dim s As SubroutineDelegate
        Dim b As Button = New Button()
        s = New SubroutineDelegate(AddressOf b.Move)
    End Sub
End Module
```

exact same set of parameters

refer to method Button.move

✓ * Reference to any method in any type that has the exact same set of parameters and the same Return type.

```
Module Test
    Delegate Function ModifyDelegate (ByVal Value As Integer) As Integer
    Sub ModifyArray (ByVal a() As Integer, ByVal Modify As ModifyDelegate)
        For Index As Integer = 0 To a.Length()-1
            a(Index) = Modify(a(Index))
        Next Index
    End Sub
    Function AddOne (ByVal i As Integer) As Integer
        Return i+1
    End Function
    Function DivideByTwo (ByVal i As Integer) As Integer
        return i/2
    End Function
    Sub Main()
        Dim a(9) As Integer
        ModifyArray(a, AddressOf AddOne)
        ModifyArray(a, AddressOf DivideByTwo)
    End Sub
End Module
```

Attributes :-

How are You Date. No.

- allows to define new kinds of information that can be specified on declarations without requiring changes in the language.

* attributes are classes that can be attached to declarations, just like ~~[illegible]~~ modifiers.

* an Instance of an attribute class is applied to a ~~ded~~ declaration by enclosing a constructor call in (<>).

```
class Test
  <ThreadStatic> Public Shared x As Integer
     ↑
  (System.ThreadStaticAttribute)
```

```
<System.AttributeUsage (AttributeTargets.All)> Class FirstAttribute
                                                  Inherits Attribute
                                               End class
<AttributeUsage(AttributeTargets.All, AllowMultiple := True, Inherited := False)> _
                                               Class SecondAttribute
                                                 Inherits Attribute
                                               End Class
```

= Versoning =

May you enjoy happiness in the coming year Date. No.

```
Class Base
  -- Sub Print()
End class

Class Derived
  Inherits Base
  Shadows Sub Print()
  End Sub
End class
```

(hides)
the method shadows the base member by the same name