Introduction to VB.net

Development Process: (design specification → user interface → set properties → add functionality by code → test & Debug → make .exe → setup)

* use public properties, instead of public fields
* keep as many fields and methods private, make them public only if necessary
* use an _ to denote a private field. If the field or method is private, use camel case (e.g., engineSize, gradeBook, ...)
* public fields, properties or methods should be in Pascal Case. (e.g., EngineSize, GradeBook, ...) (OK)

object = programmable element (e.g., form, button, textbox, ...)
properties = characteristics of objects
method = Action applied on/by object
Events = things that an object can respond to.

Dim variable As type → assigns amount of space (what size? type)

- Dim: (set) reserve space in memory (where? is it)
- variable: assigned to the space (what size?)
- As type: indicates that a datatype is to follow

* Debugging: open watch window | add watch (insert the items to be watched and values. (choose the item and right click)

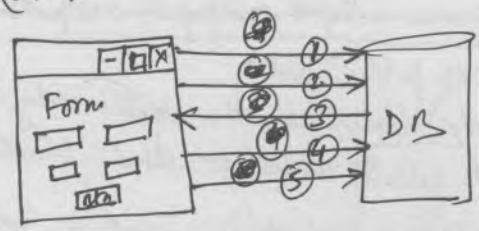
Deploy app (windows):

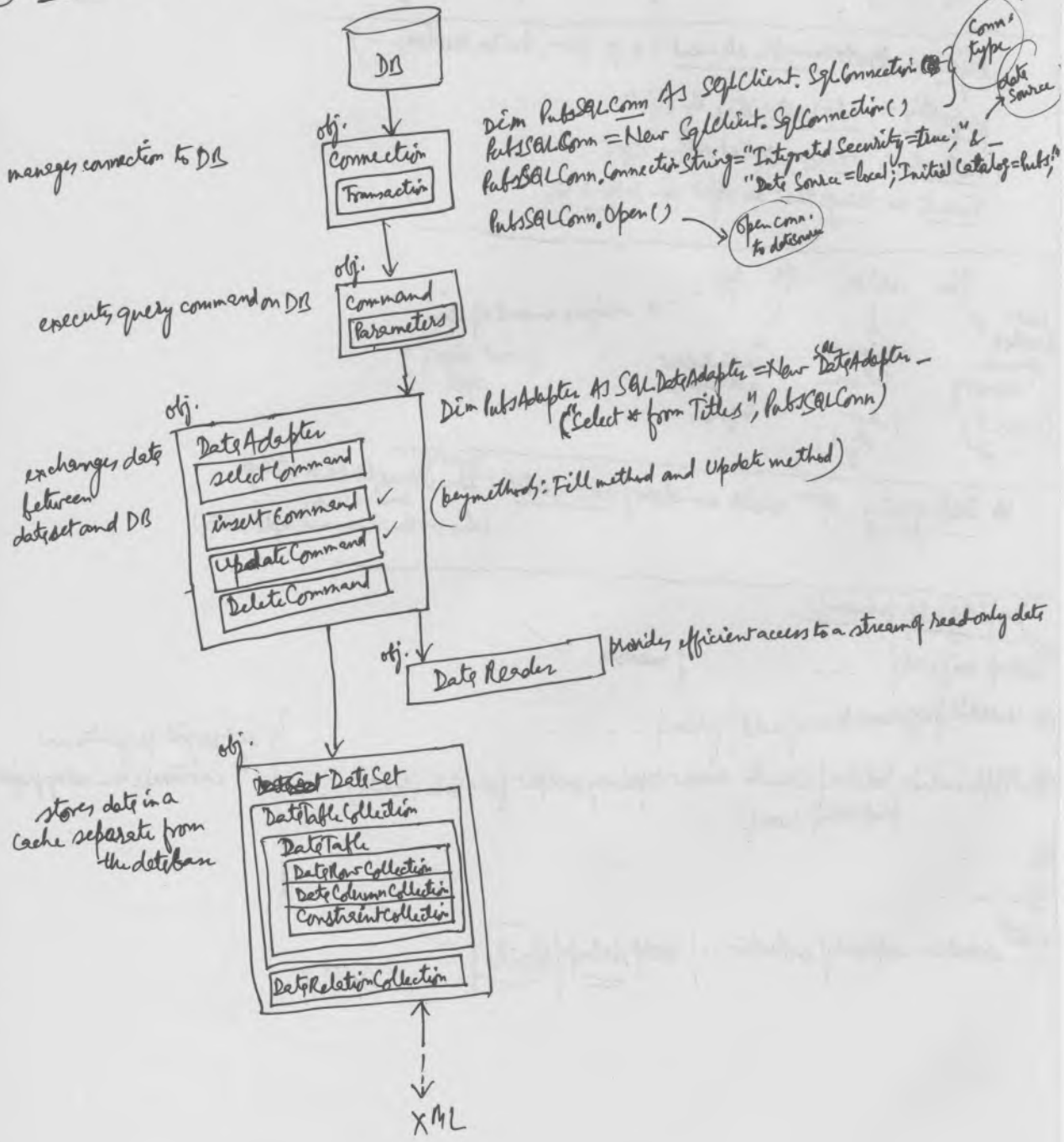
1. Setup wizard | . — — . | users
2. user's programs Menu | add | folder | . . . . .
3. Application Folder | Create shortcut to primary output from Calc (Active) . . . .
   properties | icon | . .
4. . . .
5. — — .
6. solution explorer | solution . . | setup | Build

* set project properties and customize the setup project

database programming (tasks):

1. open connect to DB
2. request specific data
3. return data
4. transmit update
5. close the connection



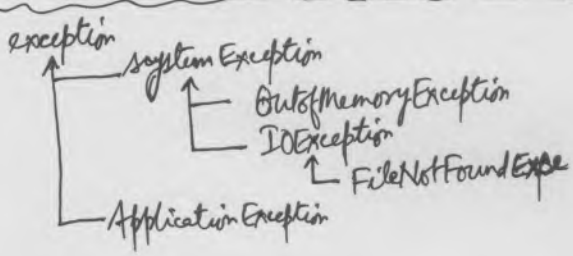


Handling errors and exceptions:

errors: (syntax errors, run-time errors, logic errors)

Debugging windows:
- Autos (view variables in current statement and +3 (after) and −3 (before) statements)
- Call Stack (view history of calls)
- Locals (view, modify local variables)
- watch [Create customer list of variables and expressions to monitor; view and manipulate any watch expressions]

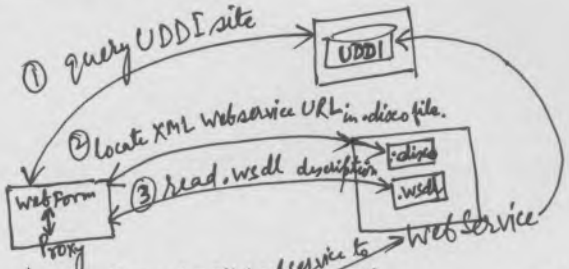
Command Window:
- evaluate expressions: (preface expression with '?') e.g., ?x, ?i, ?tax, ...
- immediate mode: (type 'immed' without >) e.g., immed → used for debugging and evaluating expressions
- command mode: (type >cmd) e.g. >cmd
- command mode (temporary): (type '>' and command name) e.g.; >alias



Try...
throw..
Catch..
Finally

XML Web Service := programmable component for providing particular functionality (of interacting with apps via web)

↓ by SOAP, WSDL (doc.), UDDI (to register)



⑤ Call XML web service from application by proxy:
- create web reference for XML web service
- create instance of XML web service
  e.g., Dim webServiceName As New localhost.service1()
- call web method of XML web service
  e.g., Label1.Text = webServiceName.WebMethod()

* fundamental difference between VB6/VB.Net —
  — disconnected environment assumed —
    ✓ ByRef – VB6 default
    ✓ ByVal – VB.Net default
  — matured development platform —
    ✓ threading
    ✓ improved error handling

* Namespaces — provide a classification system for classes
  e.g.: classes organize procedures
        namespaces " classes
  — [namespace.class] (avoid conflicts among classes' names)
  — Imports statement

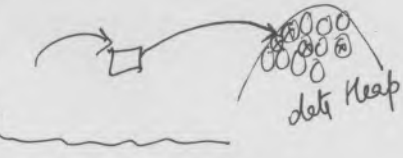
* DLL/s → Assembly/ies { COM environment: code → .dll compilation → dll wrapper → registry to Typelist → CPU
  .net environment: code → assembly (IL intermediate lang. .dll) → CPU/Hard drive disk
  (no registry)
  manifests (meta data of assemblies)

* Type Safe (code accesses only the memory locations it is authorized to access)

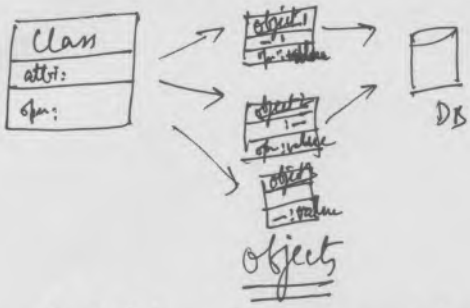
* Types (Value type & reference type)
  ✓ value type = data is stored in the variable
  — VB.net has value type (no inheritance, no ~~overloading~~ overriding base class functionality)
  ✓ Reference type (stores a reference to the data in the variable)

types
Boolean
Byte
Char
Date
Integer
Decimal
Short
Single
String
Double
Long
Object

Option strict disallows
* operations on objects other than =
  <>
  TypeOf...Is
  Is
* omitting as keyword in declarations



class (collection of functionality)
- contains properties, methods, functions, fields



encapsulation : - [encapsulate/enclose code that we dont want to expose to (outside) (consuming the code)]
- (hide the details)

abstraction : - hide implementation from user
- simplify usage
- standardize control

Public class Mark
{ public sub New()
end sub x=53 y="hello" }
end class

Sub main()
dim newobj As New Mark()
Console.WriteLine(newobj.x)
end sub

* private parts are accessed by the members inside of the same class, but it can not be accessed by outside members.

class Mark
...
private sub prmark()
console.writeline("Hello")
end sub
public sub pubmark()
console.writeline("my public part")
prmark()
end sub

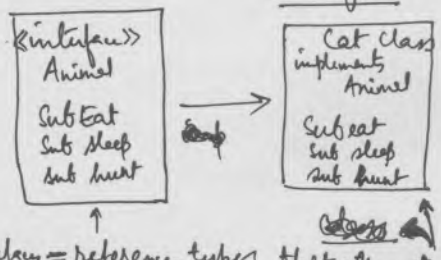
class specifiers

- public class (public access to the class)
- private class (assessible only to it's parent class)
- friend class (" " in the program in which they are defined)
- protected (only accessible within the class itself and its child classes)
- protected friend (combination of protected and friend)

inheritance

public class Janet
→ inherits Mark
end class

— interface —

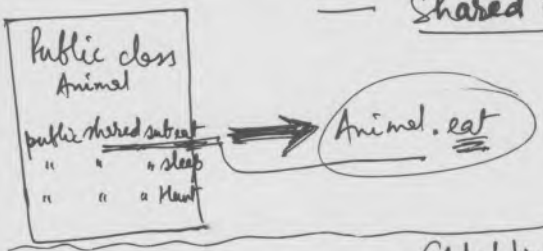


(interface = reference types that other types implement to guarantee that they support certain operations.)

public interface Test
sub dothat()
end interface

public class implements test
public sub dothat() implements test.dothat
end sub
end class

— Shared member —



public shared sbc As Int16
public shared sub dothis()
end sub

– Global properties –

- generally discourage
- out of control (introduce inaccurate data & errors)

— properties and fields

public class — -

```
private yy as int16
public property y() as int16
    Get
        Return yy
    End get
    Set (byVal value as int16)
        Console.WriteLine("can can go : {0}", y)
    End Set
End property
```

✱ methods

- contain executable statements of a program (e.g., sub / function)

✱ types of methods

- subroutine (sub) = can be used : without arguments, with arguments, optional arguments, default arguments
- function (Function) is a subroutine that ✓ Returns a value

```
Function [Name] As Datatype
-- code --
Return Value
End Function
```

e.g.,

```
public sub walk(Optional byVal x as Int16 = 55)
    Console.WriteLine(x)
end sub
```

```
public function Tanet as int16
    return yy
end function
```

```
public function (byval a as int16) as int16
    return a
end function
```

✱ Overloading = (same name, different signature)

e.g.,

```
public overloads sub test (byval x as string)
public overloads sub test (byval y as int16)
```

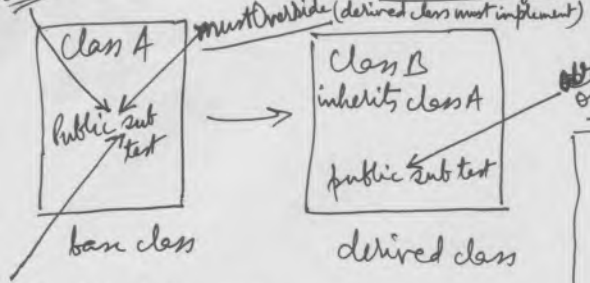
→ signature

```
public Overloads sub test()
end sub
```

overridable in derived class

Overriding (used to change the behavior of a method from a base class)

mustOverride (derived class must implement)



overridable overrides (indicates) this is overriding

notOverridable prevents from being overridden

e.g.

```
public class test
    public overridable sub mark()
        Console.WriteLine("Hello")
    end sub
end class
public class test2
    public overrides sub mark()
        Console.WriteLine("Hello from test2")
    end sub
end class
```

public mustOverride sub mark()

= Shadowing = similar to overloading

where (overloading deals with signatures)

* shadow deals with name

* (replaces / shadows / hides) base class methods with derived class methods so, we can have only derived class methods

e.g.

base class

```
public class test
    public sub mark()
        Console.WriteLine("Hello")
    end sub
end class
```

derived class

```
public class test2
    inherits test
    public shadows sub mark(byVal x As Int32)
        Console.WriteLine(x)
    end sub
    public shadows sub mark()
        Console.WriteLine("Hello from test2")
    end sub
end class
```

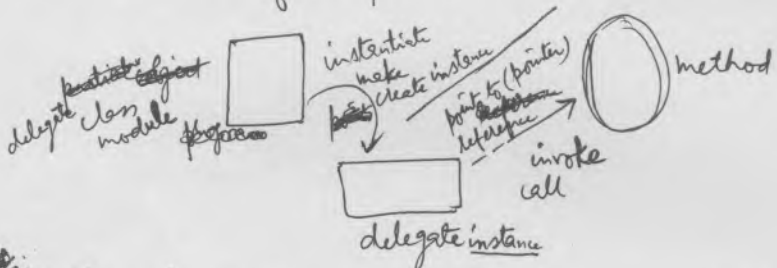
Events

```
Public Class TheEvent
public event test(byval s as string)
  public sub fireEvent()
      raiseEvent Test
  end sub
end class
```

```
private withevents x as New theevent()
private sub btnOK_click() handles
                    btnOK.click
  x.fireEvent
      code goes here
end sub
private sub x_test(byval s as
      string) handles x.test
  code here
end sub
```

delegate (representative)

- is Object
  - * reference to a method
  - * similar to function pointers



- public delegate sub dosomething(byval x as string) → define delegate
- dim delEx as Dosomething = new dosomething(Addressof somemethod) → make the delegate instance of delegate that we will call
- delEx.invoke("Hello") → invoke the delegate

```
Public sub someMethod(ByVal x as string)
```

e.g.

method (function)
```
public sub printThis(byval x as string)
    Console.WriteLine(x)
end sub
```
→ method to called by our delegate

definition ← `public delegate sub delegateA(byval y as string)` → define our delegate

```
sub main()
    dim delex as delegateA
    delex = Addressof printThis
    delex.invoke("Hello")
end sub
```

pointer / pointing to method/function → delex = Addressof printThis

invoke delegate → delex.invoke("Hello")

→ make the delegate instance and point to/reference to function/method the

types of delegates

- single
- multicast

(multi-cast) e.g.:

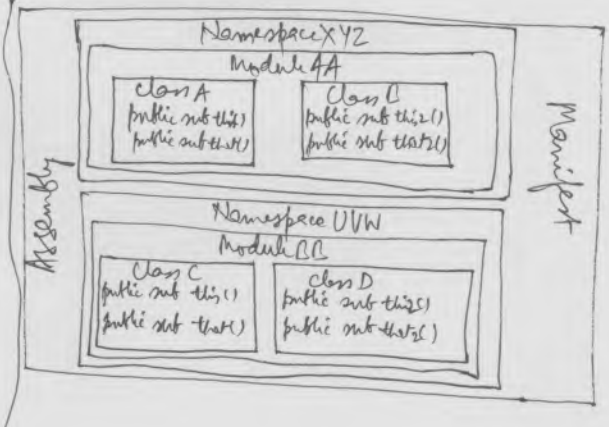
```
public delegate sub exampledelegate(byval x as string)
dim a as exampledelegate
dim b as exampledelegate
a = New exampledelegate(AddressOf dothis)
b = new exampledelegate(AddressOf dothat)
dim c as exampledelegate
c = exampledelegate.combine(a,b)
```

Module

- reference type similar to classes
- differs from classes:
  - all members are implicitly SHARED
  - modules can not be instantiated
  - do not support interfaces
  - do not support inheritance
  - can not implement interfaces

e.g.

```
module [Module]
    public sub main()
    end sub
    public sub test()
    end sub
    public class classtest()
        public sub dosomething()
        end sub
    end class
  sub main()
    dim a as new module2.classtest()
        a.dosomethingelse()
    end sub
end module
module module2
    public sub main()
    end sub
    public class classtest()
        public sub dosomethingelse()
        end sub
    end class
end module2
```



structure

- UDT (user-defined type)
- class-light
= acts like a class but value typed
- structures can utilize:
  - variables
  - subs
  - functions

```
structure mark
    public a as integer
    public sub dothis()
    end sub
    public function dothat()
    end function
end structure
sub main()
    dim x as new mark()
    x.a = 5
```

2.)

value types & reference types

```
dim a as int16 = 5
dim b as int16 = a        => value type
a = 10
console.writeline(a)
console.writeline(b)

dim x as new mark()
dim xx as mark = x        => reference type
x.aa = 3
console.writeline("x.aa is: {0}", x.aa)
console.writeline("xx.aa is: {0}", xx.aa)

xx.aa = 5
console.writeline("x.aa is: {0}", x.aa)
console.writeline("xx.aa is: {0}", xx.aa)

public class mark()
    public aa as integer
end class
```

---

* thread (smallest executable section of code)

(simultaneous execution of many code-lines by switching among each other)

CPU

* spawning thread

Remoting (allows objects in one appt to talk to another)
within any two computers

① channels (common link)
② application domain (
③ application context

System.Runtime.Remoting

web services:

(ASP.net ~~web services~~ → System.Web namespace)

DCOM over internet (calling objects over HTTP) serialization / [similar to components] - black box functionality - no user interface / - stateless architecture - asynchronous

imports System.Web.Services

public class Form1

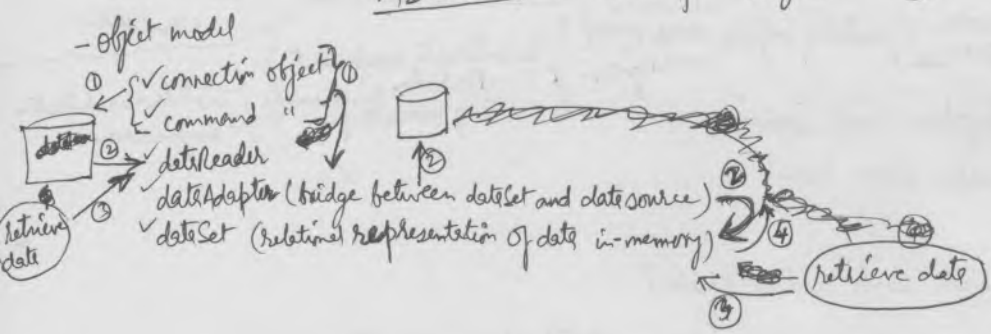
private sub Button1_click(- - - -)

dim a as new myreference.Service1()

~~a. mywebservice~~

msgbox(a.mywebreference)

ADO.net → namespace (System.Data)



VB.net

= introduction (Form Controls)

- size / position: 1. easiest to adjust graphically
  - click and drag the border to move
  - " " " the handles to resize
  2. properties window
     - size properties controls width and height
     - location properties controls x and y
  3. adjusting multiple controls
     - format menu
     - layout toolbar
  4. code is being written by Visual studio

- event Handling code
- event handlers
  1. create project and add a form or forms
  2. Add controls to the forms
  3. set the properties of controls at design time using visual interfaces
  4. Add event handlers (modify properties at run time)

- form menu :- ~~form control~~ (form components)
  form controls

Create console Application :

Key Terms :- Namespace (hierarchical organization of types within code assemblies)
[there are 4 kinds of types (classes, interfaces, structures, enumerations)]

- Assembly (a unit of executable code)
  (in the form of DLL and EXE files)
- Manifest (data structure that contains information or metadata about an assembly)
  (uses notepad and .Net framework)

Timer → for window form : Me.Opacity -= 0.01 → Opacity = opaque/transparent (0 or 1) (transparency) ⇒ 1 is fully opaque, 0 " transparent

= variables and data types =

* variables can be declared outside of a procedure : public, private, protected, friend, protected friend

* Data type :
  - System.Object
  - System.char
  - System.Int16
  - ...

Convert data types :

conversion function : CBool(), CByte(), CChar(), CDate(), CDbl(), CDec(), CInt(), CLng(), CObj(), CShort(), CSng(), CStr()

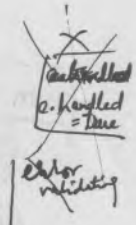
CType(expression, typename)

* Operators = (not equal) <> <= > >= +/- (exp)^ * / \ < And Or Not += -= AndAlso OrElse & *= /=

Delegate :
* special object that registers the procedure to be called
* provide defined procedure signatures
* event handlers ensure correct parameters / return values

Event Handler = public Delegate Sub EventHandler(_
ByVal sender As Object,_
ByVal e As EventArgs)

multicast delegate



* Inheritance chain :

System.Object
System.MarshalByRefObject
System.ComponentModel.Component
System.Windows.Forms.ScrollableControl
System.Windows.Forms.ContainerControl
System.Windows.Forms.Form

* = Class & Object =

class = definition of object type
↳ properties, methods, events
(A mechanism for creating objects)
– method of defining allocation and use of computer memory and ~~processing~~ processing.

* CLR Types : – classes
– structures (grouping of values, can have properties/methods)
~~– enum~~
– enumeration (name/value pairs to choose from)
– delegate (type-safe function pointer)

* Object creation : – object lives in computer memory
– properties defined in the class are written to and/or read from an object.
– methods " " " " are called in objects.

Class libraries
Constructors
properties (set/get)
Method : – Dispose = to clean up resource (memory occupied)
– will automatically call finalize, but do not rely on this

add Reference for many projects
imports classes

Constructor : public Sub new()
end Sub

~~private myProperty() As String~~
public property myProperty() As String
Get
end Get
Set (byval value As String)
end set
end property

Readonly ⇒ get()
no set()

public Event Greeting(- - -)

End Event

must Event ①

public Function SayHello(- - -) As - - -

RaiseEvent Greeting(- - -)

return - - -

end - function

private WithEvents Tim (obj) As - - -

private Sub btnRaiseEvent_click (- — -)

end sub

(shift + F2) → definition of - -

indeterminate = disabled

String class — strings are immutable (unchanged contents)

= " " objects

(" " "Nothing" until you assign a value)

- string values are greater than Nothing
  (you can compare strings to Nothing)
- strings are indexed.

DateTime represents a single instant (number of ticks / year, month and day / y-m-d-hour-min-sec / parse method)

TimeSpan " elapsed time (number of ticks : "[±] d.hh:mm:ss.ff) / parse method / From method

(value type)

DateTime calculations don't modify the original value

(return new DateTime/TimeSpan value)

Exception class for .Net error handling

- linked lists of errors that were raised to trigger this error
- you can inherit from the Exception class
- nest with Try/Catch blocks (makes it easy to push/pop error handlers)

= create exception object for passing (ToString method / message / InnerException object)

Try - → throw (optional) put anywhere in between the two blocks

Catch - →

Finally - → (don't need both blocks i.e., at least one block should be these)

Exception Object:
- the exception class provides useful properties/methods
  (* ToString – converts to name and stack dump)
  (* Message – returns error message)
  (* InnerException – " another object) exception

* Catching specific Exceptions: (e.g. FileNotFoundException) provided by .net documentation lists
- you can add as many catch blocks as necessary.
- " " use specific exceptions to handle different errors differently.
- If none match, use base Exception, if included

Throw – to pass specific error back to your caller

Finally – guaranteed runs (no matter what) (unconditionally)
code – only one block

```
Try
  - - -
  {throw} - - -
Catch e As FileNotFoundException          } specific exceptions
    strMsg = " - - - - - "
Catch e As UnAuthorizedAccessException    }
    strMsg = " - - - - - "                } generic exceptions
Catch - -                                 } common
  {throw}
  -
Finally
  - - -
End Try
```

System.Object
  System.Exception
    System.SystemException
      System.IO.IOException

* inheritance – ability of a class to take on all the functionality of another class
– may add new capabilities & functionality

Base class (super class / parent class) →
Derived class (sub class / child class) → overloads, overrides

* single inheritance → (VB.net)

* Overriding :– creating method/property of same name as in base class
– use Overridable method/property
Override " "

* (Overridable method/property) → method Virtual
(method/property based on object type)

(Like override but not marked as "Override") → method Shadow
not virtual
based on variable type

* Protected : members can be accessed by derived classes also.

polymorphism = – using inheritance / interfaces to allow code to work with many different types of objects

– inheritance-based polymorphism :– objects can be interchanged (due to, derived from common ancestor)
– interface-based polymorphism :– objects can be interchanged (due to, they implement common interface)

Command window :

? (variable/object/...

watch window

output window

* Using interfaces = a specified set of properties/and methods
a class can have many or interfaces

* Directory / file class : - path class handles parsing file names
- requires different objects to work with files, folders, and content
- System.IO provides { File & FileInfo
directory & DirectoryInfo

* if you need one "access", use File or Directory class
* if you want multiple "accesses", use FileInfo or DirectoryInfo class