

# **Module 5: Object- Oriented Programming in Visual Basic .NET**

# Overview

- Defining Classes
- Creating and Destroying Objects
- Inheritance
- Interfaces
- Working with Classes

# ◆ Defining Classes

- Procedure for Defining a Class
- Using Access Modifiers
- Declaring Methods
- Declaring Properties
- Using Attributes
- Overloading Methods
- Using Constructors
- Using Destructors

# Procedure for Defining a Class

- 1** Add a class to the project
- 2** Provide an appropriate name for the class
- 3** Create constructors as needed
- 4** Create a destructor, if appropriate
- 5** Declare properties
- 6** Declare methods and events

# Using Access Modifiers

- Specify accessibility of variables and procedures

Keyword	Definition
<b>Public</b>	Accessible everywhere.
<b>Private</b>	Accessible only within the type itself.
<b>Friend</b>	Accessible within the type itself and all namespaces and code within the same assembly.
<b>Protected</b>	Only for use on class members. Accessible within the class itself and any derived classes.
<b>Protected Friend</b>	The union of <b>Protected</b> and <b>Friend</b> .

# Declaring Methods

- Same syntax as in Visual Basic 6.0

```
Public Sub TestIt(ByVal x As Integer)
```

```
...
```

```
End Sub
```

```
Public Function GetIt( ) As Integer
```

```
...
```

```
End Function
```

# Declaring Properties

- Syntax differs from that of Visual Basic 6.0

```
Public Property MyData( ) As Integer
    Get
        Return intMyData          'Return local variable value
    End Get
    Set (ByVal Value As Integer)
        intMyData = Value        'Store Value in local variable
    End Set
End Property
```

- ReadOnly, WriteOnly, and Default keywords

```
Public ReadOnly Property MyData( ) As Integer
    Get
        Return intMyData
    End Get
End Property
```

# Using Attributes

- Extra metadata supplied by using "< >" brackets
- Supported for:
  - Assemblies, classes, methods, properties, and more
- Common uses:
  - Assembly versioning, Web Services, components, security, and custom

```
<Obsolete("Please use method M2")> Public Sub M1( )  
    'Results in warning in IDE when used by client code  
End Sub
```

# Overloading Methods

- Methods with the same name can accept different parameters

```
Public Function Display(s As String) As String
    MsgBox("String: " & s)
    Return "String"
End Sub

Public Function Display(i As Integer) As Integer
    MsgBox("Integer: " & i)
    Return 1
End Function
```

- Specified parameters determine which method to call
- The Overloads keyword is optional unless overloading inherited methods

# Using Constructors

- Sub New replaces Class\_Initialize
- Executes code when object is instantiated

```
Public Sub New( )  
    'Perform simple initialization  
    intValue = 1  
End Sub
```

- Can overload, but does not use Overloads keyword

```
Public Sub New(ByVal i As Integer) 'Overloaded without Overloads  
    'Perform more complex initialization  
    intValue = i  
End Sub
```

# Using Destructors

- Sub Finalize replaces Class\_Terminate event
- Use to clean up resources
- Code executed when destroyed by garbage collection
  - Important: destruction may not happen immediately

```
Protected Overrides Sub Finalize( )
```

```
    'Can close connections or other resources
```

```
    conn.Close
```

```
End Sub
```

# ◆ Creating and Destroying Objects

- Instantiating and Initializing Objects
- Garbage Collection
- Using the Dispose Method

# Instantiating and Initializing Objects

- Instantiate and initialize objects in one line of code

```
'Declare but do not instantiate yet
```

```
Dim c1 As TestClass
```

```
'Other code
```

```
c1 = New TestClass( )           'Instantiate now
```

```
'Declare, instantiate & initialize using default constructor
```

```
Dim c2 As TestClass = New TestClass( )
```

```
'Declare, instantiate & initialize using default constructor
```

```
Dim c3 As New TestClass( )
```

```
'Declare, instantiate & initialize using alternative constructor
```

```
Dim c4 As New TestClass(10)
```

```
Dim c5 As TestClass = New TestClass(10)
```

# Garbage Collection

- Background process that cleans up unused variables
- Use  $x = \textit{Nothing}$  to enable garbage collection
- Detects objects or other memory that cannot be reached by any code (even circular references!)
- Allows destruction of object
  - No guarantee of *when* this will happen
  - Potential for resources to be tied up for long periods of time (database connections, files, and so on)
  - You can force collection by using the GC system class

# Using the Dispose Method

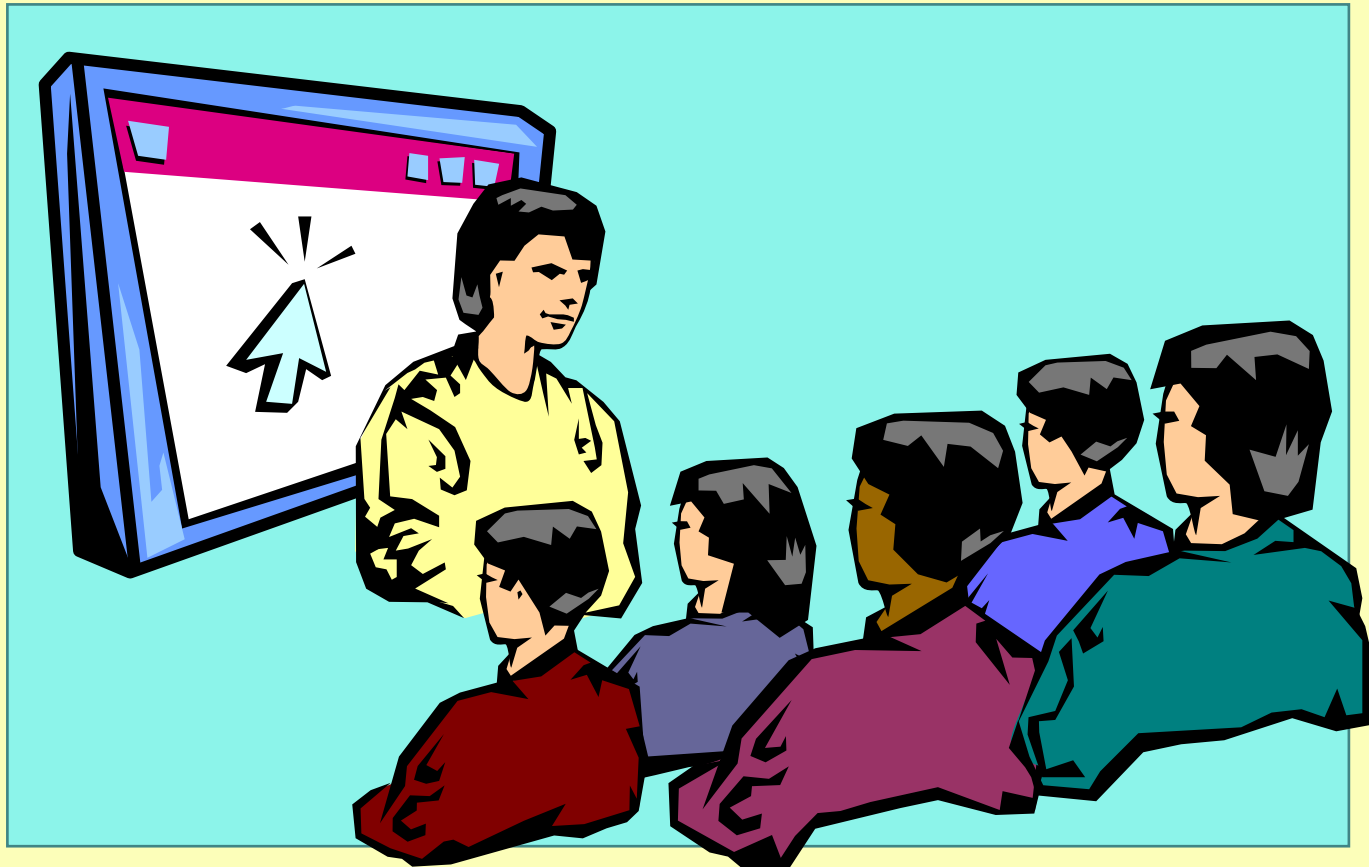
- Create a Dispose method to manually release resources

```
'Class code
Public Sub Dispose( )
    'Check that the connection is still open
    conn.Close          'Close a database connection
End Sub
```

- Call the Dispose method from client code

```
'Client code
Dim x as TestClass = New TestClass( )
...
x.Dispose( )          'Call the object's dispose method
```

# Demonstration: Creating Classes



# Lab 5.1: Creating the Customer Class



# ◆ Inheritance

- What Is Inheritance?
- Overriding and Overloading
- Inheritance Example
- Shadowing
- Using the MyBase Keyword
- Using the MyClass Keyword

# What Is Inheritance?

- Derived class inherits from a base class
- Properties, methods, data members, events, and event handlers can be inherited (dependent on scope)
- Keywords
  - **Inherits** – inherits from a base class
  - **NotInheritable** – cannot be inherited from
  - **MustInherit** – instances of the class cannot be created; must be inherited from as a base class
  - **Protected** – member scope that allows use only by deriving classes

# Overriding and Overloading

- **Derived class can override an inherited property or method**
  - **Overridable** – can be overridden
  - **MustOverride** – must be overridden in derived class
  - **Overrides** – replaces method from inherited class
  - **NotOverridable** – cannot be overridden (default)
- **Use Overload keyword to overload inherited property or method**

# Inheritance Example

```
Public Class BaseClass
    Public Overridable Sub OverrideMethod( )
        MsgBox("Base OverrideMethod")
    End Sub
    Public Sub Other( )
        MsgBox("Base Other method - not overridable")
    End Sub
End Class
```

```
Public Class DerivedClass
    Inherits BaseClass
    Public Overrides Sub OverrideMethod( )
        MsgBox("Derived OverrideMethod")
    End Sub
End Class
```

```
Dim x As DerivedClass = New DerivedClass( )
x.Other( )           'Displays "Base Other method - not overridable"
x.OverrideMethod( ) 'Displays "Derived OverrideMethod"
```

# Shadowing

- Hides base class members, even if overloaded

```
Class aBase
  Public Sub M1( )      'Non-overridable by default
    ...
  End Sub
End Class

Class aShadowed
  Inherits aBase
  Public Shadows Sub M1(ByVal i As Integer)
    'Clients can only see this method
    ...
  End Sub
End Class
```

```
Dim x As New aShadowed( )
x.M1( )          'Generates an error
x.M1(20)        'No error
```

# Using the MyBase Keyword

- Refers to the immediate base class
- Can only access public, protected, or friend members of base class
- Is not a real object (cannot be stored in a variable)

```
Public Class DerivedClass
    Inherits BaseClass

    Public Overrides Sub OverrideMethod( )
        MsgBox("Derived OverrideMethod")
        MyBase.OverrideMethod( )
    End Sub
End Class
```

# Using the MyClass Keyword

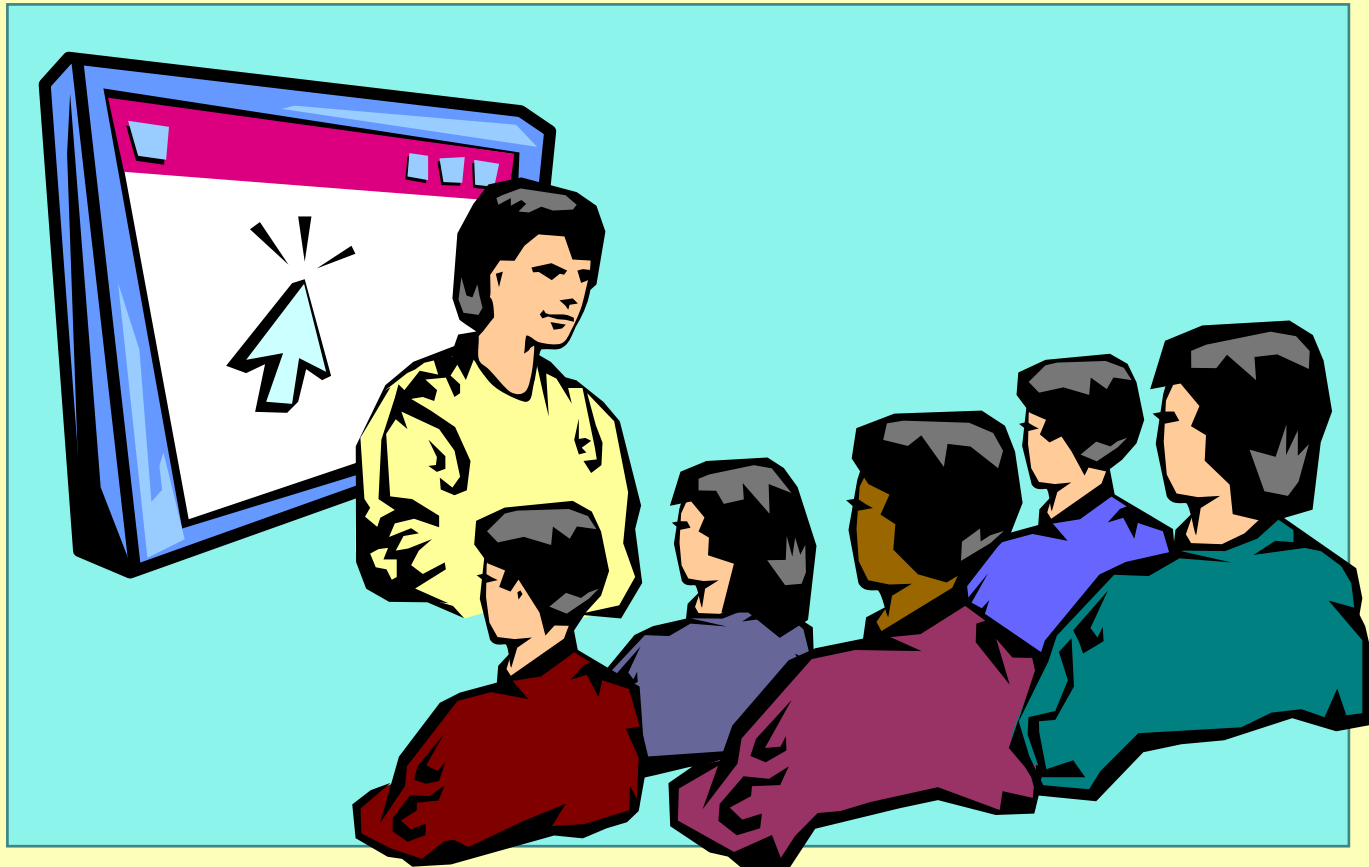
- Ensures that base class gets called, not derived class

```
Public Class BaseClass
    Public Overridable Sub OverrideMethod( )
        MsgBox("Base OverrideMethod")
    End Sub

    Public Sub Other( )
        MyClass.OverrideMethod( ) 'Will call above method
        OverrideMethod( )         'Will call derived method
    End Sub
End Class
```

```
Dim x As DerivedClass = New DerivedClass( )
x.Other( )
```

# Demonstration: Inheritance



# ◆ Interfaces

- Defining Interfaces
- Achieving Polymorphism

# Defining Interfaces

- Interfaces define public procedure, property, and event signatures
- Use the Interface keyword to define an interface module
- Overload members as for classes

```
Interface IMyInterface
    Function Method1(ByRef s As String) As Boolean
    Sub Method2( )
    Sub Method2(ByVal i As Integer)
End Interface
```

- Use the Inherits keyword in an interface to inherit from other interfaces

# Achieving Polymorphism

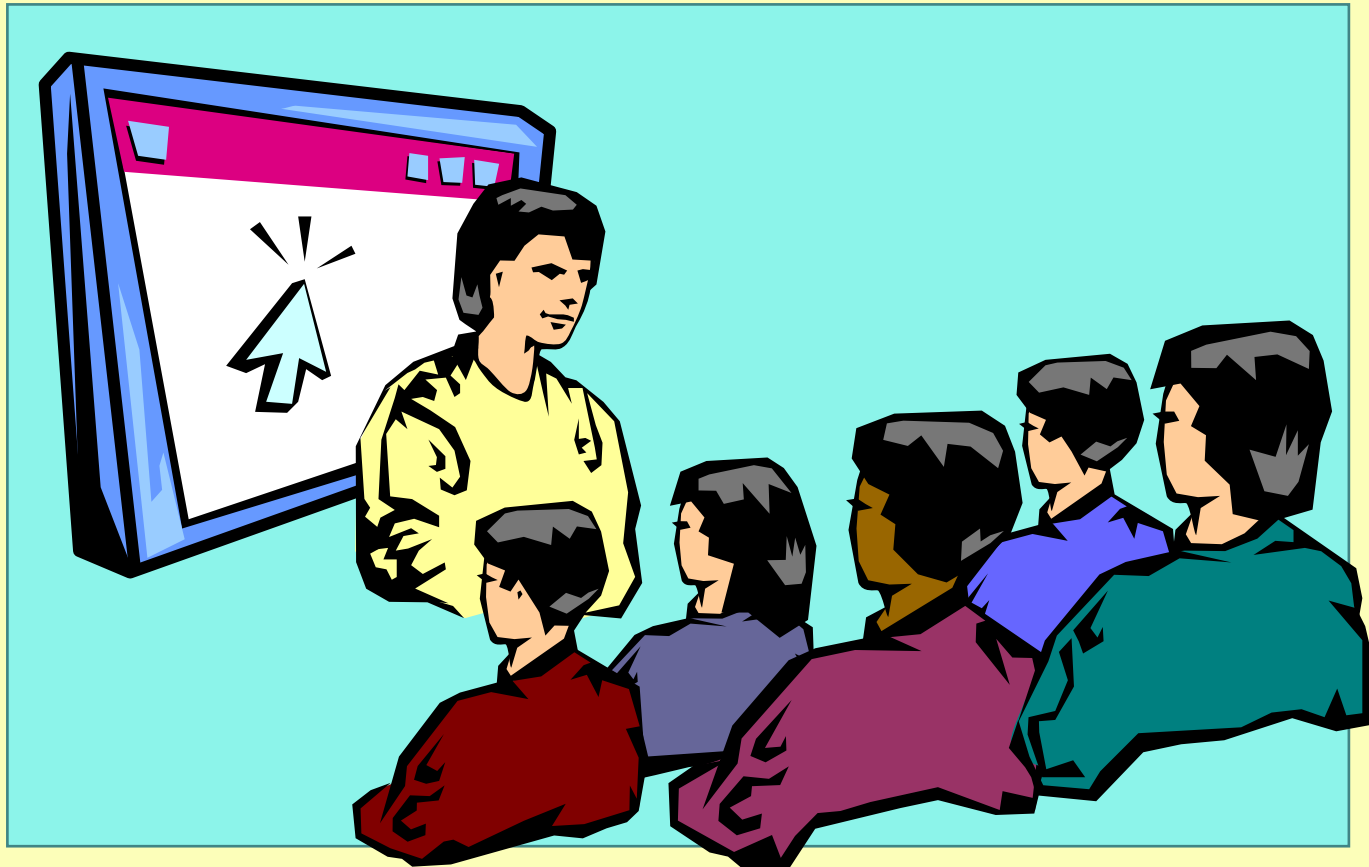
## ■ Polymorphism

- Many classes provide the same property or method
- A caller does not need to know the type of class the object is based on

## ■ Two approaches

- Interfaces
  - Class implements members of interface
  - Same approach as in Visual Basic 6.0
- Inheritance
  - Derived class overrides members of base class

# Demonstration: Interfaces and Polymorphism



# ◆ Working with Classes

- Using Shared Data Members
- Using Shared Procedure Members
- Event Handling
- What Are Delegates?
- Using Delegates
- Comparing Classes to Structures

# Using Shared Data Members

- Allow multiple class instances to refer to a single class-level variable instance

```
Class SavingsAccount
  Public Shared InterestRate As Double
  Public Name As String, Balance As Double
  Sub New(ByVal strName As String, ByVal dblAmount As Double)
    Name = strName
    Balance = dblAmount
  End Sub
  Public Function CalculateInterest( ) As Double
    Return Balance * InterestRate
  End Function
End Class
```

```
SavingsAccount.InterestRate = 0.003
Dim acct1 As New SavingsAccount("Joe Howard", 10000)
MsgBox(acct1.CalculateInterest, , "Interest for " & acct1.Name)
```

# Using Shared Procedure Members

- Share procedures without declaring a class instance
- Similar functionality to Visual Basic 6.0 “global” classes
- Can only access shared data

```
'TestClass code  
Public Shared Function GetComputerName( ) As String  
    ...  
End Function
```

```
'Client code  
MsgBox(TestClass.GetComputerName( ))
```

# Event Handling

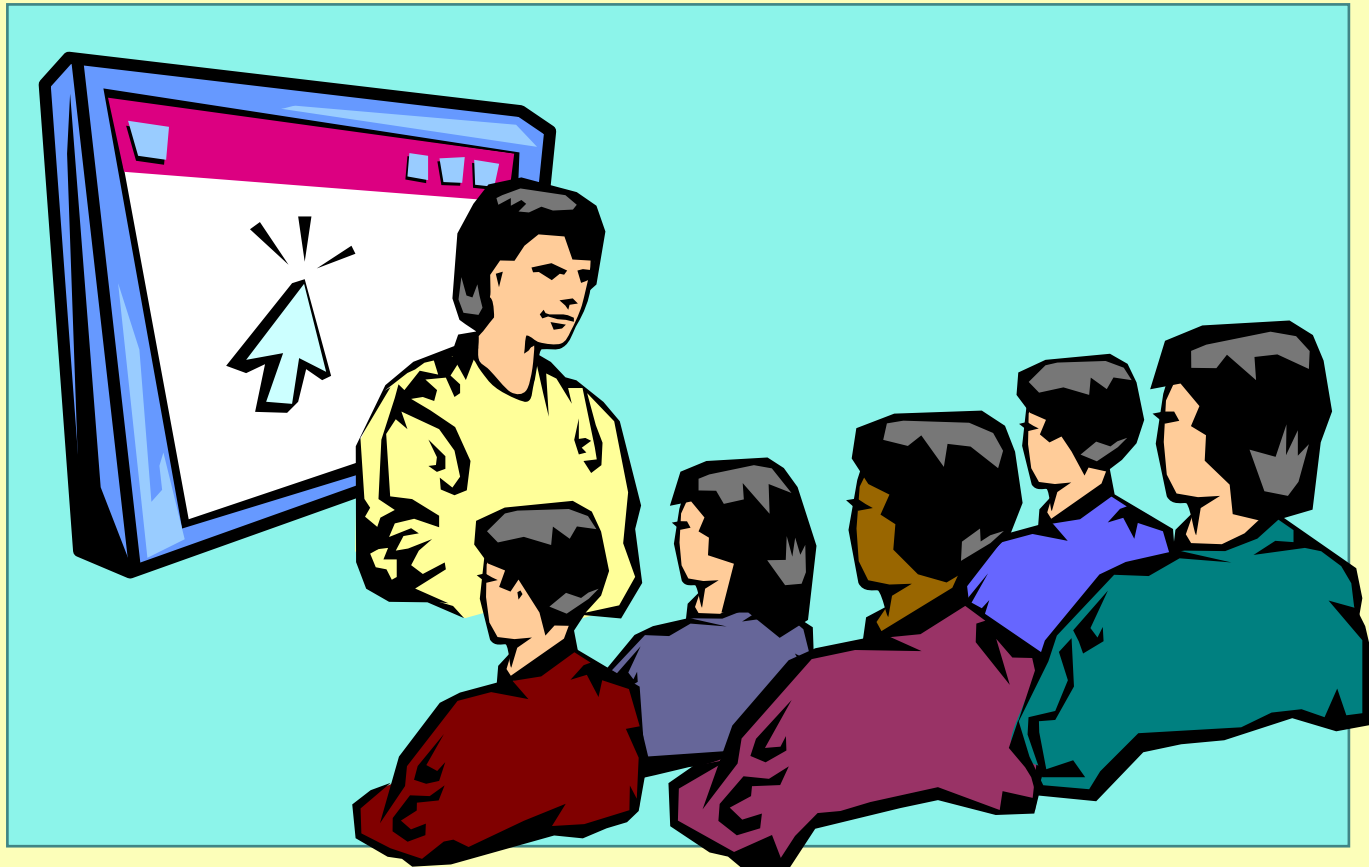
- Defining and raising events: same as Visual Basic 6.0
- WithEvents keyword: handles events as in Visual Basic 6.0
  - In Visual Basic .NET, works with **Handles** keyword to specify method used to handle event
- AddHandler keyword: allows dynamic connection to events

```
Dim x As New TestClass( ), y As New TestClass( )
AddHandler x.anEvent, AddressOf HandleEvent
AddHandler y.anEvent, AddressOf HandleEvent
...

Sub HandleEvent(ByVal i As Integer)
    ...
End Sub
```

- RemoveHandler keyword: disconnects from event source

# Demonstration: Handling Events



# What Are Delegates?

- Objects that call the methods of other objects
- Similar to function pointers in Visual C++
- Reference type based on the System.Delegate class
- Type-safe, secure, managed objects
- Example:
  - Useful as an intermediary between a calling procedure and the procedure being called

# Using Delegates

- Delegate keyword declares a delegate and defines parameter and return types

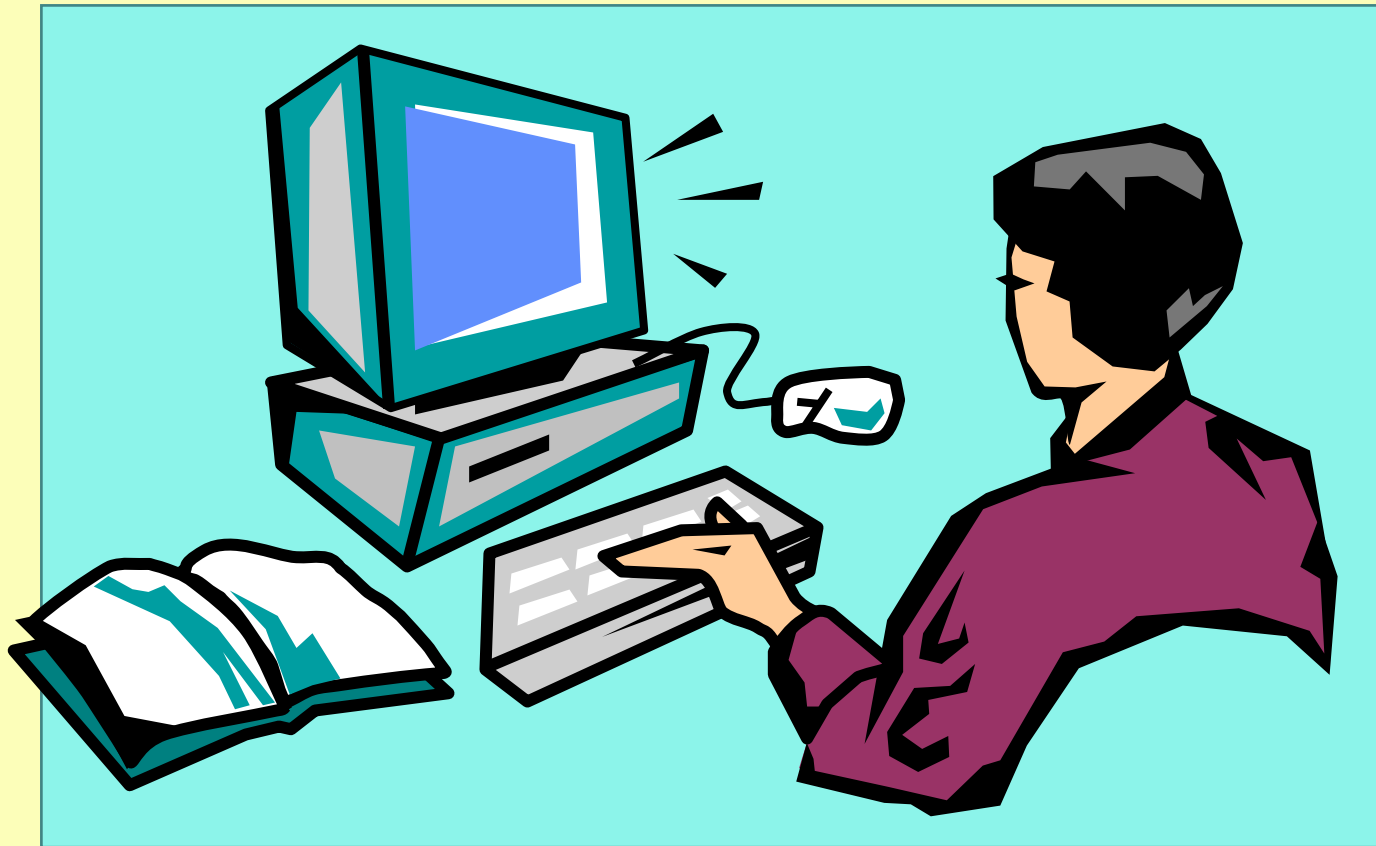
```
Delegate Function CompareFunc( _  
    ByVal x As Integer, ByVal y As Integer) As Boolean
```

- Methods must have the same function parameter and return types
- Use Invoke method of delegate to call methods

# Comparing Classes to Structures

Classes	Structures
Can define data members, properties, and methods	Can define data members, properties, and methods
Support constructors and member initialization	No default constructor or member initialization
Support <b>Finalize</b> method	Do not support <b>Finalize</b> method
Extensible by inheritance	Do not support inheritance
Reference type	Value type

## Lab 5.2: Inheriting the Package Class



# Review

- Defining Classes
- Creating and Destroying Objects
- Inheritance
- Interfaces
- Working with Classes