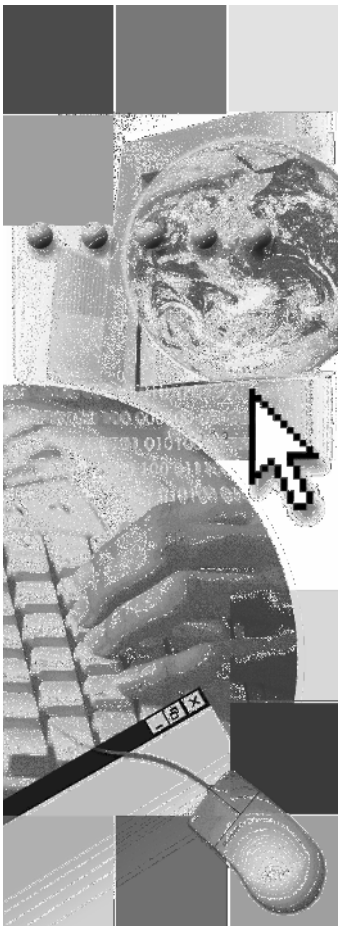
msdn training

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

Contents



This course is based on the prerelease version (Beta 2) of Microsoft® Visual Studio® .NET Enterprise Edition. Content in the final release of the course may be different from the content included in this prerelease version. All labs in the course are to be completed with the Beta 2 version of Visual Studio .NET Enterprise Edition.

Microsoft

Instructor Notes

Presentation:
90 Minutes

Labs:
105 Minutes

This module provides students with the knowledge required to create object-oriented applications that use many of the new features of Microsoft® Visual Basic® .NET, such as inheritance, overloading, shared members, and event handling.

In the first lab, students will create part of the **Customer** class for the Cargo system that they designed in Lab 4.1, Creating Diagrams from Use Cases. They will define the properties, methods, and constructors, based on those shown in Lab 4.1. Finally, they will write the code in a form to test the **Customer** class.

In the second lab, students will create a base class called **Package** and a derived class called **SpecialPackage**. The classes contain some pre-written code, including the properties. Students will add methods to both classes and create the inheritance relationship. They will then complete a pre-written form to test their classes.

After completing this module, students will be able to:

- Define classes.
- Instantiate and use objects in client code.
- Create classes that use inheritance.
- Define interfaces and use polymorphism.
- Create shared members.
- Create class events and handle them from a client application.

Materials and Preparation

This section provides the materials and preparation tasks that you need to teach this module.

Required Materials

To teach this module, you need the following materials:

- Microsoft PowerPoint® file 2373A_05.ppt
- Module 5, "Object-Oriented Programming in Visual Basic .NET"
- Lab 5.1, Creating the Customer Class
- Lab 5.2, Inheriting the Package Class

Preparation Tasks

To prepare for this module, you should:

- Read all of the materials for this module.
- Read the instructor notes and the margin notes for the module.
- Practice the demonstrations.
- Complete the labs.

Demonstrations

This section provides demonstration procedures that will not fit in the margin notes or are not appropriate for the student notes.

Creating Classes

↙ To examine the Employee class

1. Open the Classes.sln solution in the *install folder*\DemoCode\Mod05\Classes folder.
2. View the code for the **Employee** class and point out the private variables, the properties, and the multiple constructors. Specifically point out the **EmployeeId** read-only property.

↙ To test the New Employee code

1. Run the project.
2. Enter values for the **First Name** and the **Last Name**.
3. Click the **New Employee** button on the form. The code will enter break mode at the preset breakpoint.
4. Step through the code, and explain each line as you go. Include the **Dispose** method but not the **Finalize** method. Point out that, in a real situation, the **Dispose** method would be used for saving data and closing a database connection.

↙ To test the Existing Employee code

1. Enter a positive integer value for the **Id** (any number will work), and then click the **Existing** button.
2. Point out that this time the constructor takes the **intEmpId** as a parameter, so it can load the data from a database immediately.
3. Step through the code until the object has been instantiated, and then press F5 to allow the remaining code to run.

↙ To test the Improved New Employee code

1. Click the **Improved New** button on the form, and then step through the code when execution halts at the preset breakpoint.
2. Point out that the constructor takes **strFirstName** and **strLastName** as parameters so that it can create a new **Employee** immediately.
3. Step through the initialization code, and then press F5 to display the form again.
4. Click the **Close** button to close the form and stop the application. Explain that this will cause all remaining objects to be destroyed, and that the **Finalize** methods will execute.

   Remind students that the **Finalize** method should only be used when resources need to be manually reclaimed (such as database connections) because it creates more work for the garbage collector. In this case the **Finalize** method calls the **Dispose** method again to ensure that the resources have been reclaimed in case the class user has forgotten to call the **Dispose** method explicitly. The **Finalize** method is not necessary if the class user has called the **Dispose** method**.** A call to **GC.SuppressFinalize** within the **Dispose** method would have stopped the **Finalize** method from executing and would therefore have improved performance.
5. Quit the Microsoft Visual Studio® .NET integrated development environment (IDE).

Inheritance

↙ To examine the Person class

1. Open the Inheritance.sln solution in the *install folder*\DemoCode\Mod05\Inheritance folder.
2. View the code for the **Person** class, and point out the private variables, the properties, and the methods.
3. View the code for the **Employee** class, and point out that it is a simplified version of the class used in the previous demonstration in that it has only one constructor. Show that it inherits from the **Person** class and only stores the **EmployeeId** value. Also, point out that the **FirstName** and **LastName** properties are not defined in this class.

↙ To test the project

1. Run the project.
2. Type values in the **First Name** and the **Last Name** boxes.
3. Click the **New Person** button on the form. The code will enter break mode at the first preset breakpoint.
4. Step through the code, explaining each line of code as you go. This will include the **Dispose** method and the **Finalize** method when the **GC.Collect** method is executed.
5. Point out that the **Finalize** method also calls the **Dispose** method of the **Person** by means of the **MyClass** object.
6. Enter a positive integer value for the **Id** (any number will work), and click **Existing Emp**.
7. Point out that this time many of the inherited properties of the **Person** class are called rather than those of **Employee**.
8. Step through the code until the form appears again.
9. Close the form and quit the Visual Studio .NET IDE.

Interfaces and Polymorphism

↙ To view the application code

1. Open the Polymorphism.sln solution in the *install folder*\DemoCode\Mod05\Polymorphism folder.
2. View the code for the **IPerson** interface. Point out the property and method definitions and that there is no implementation code.

Delivery Tip
Explain that students can use either public or private methods within the class when implementing the interface.

3. View the code for the **Employee** class, and point out that it now implements the **IPerson** interface and stores the **intEmployeeId**, **strFName**, and **strLName** values. Also point out the **Public EmployeeId** property, the **Private FirstName** and **Private LastName** properties, and the new syntax for the **Implements** keyword in each method signature. Explain that because the properties are marked as private, they will only be visible from the **IPerson** interface. Also explain that marking the properties as private is optional. They could have been marked as public, making them visible from both the **IPerson** interface and the **Employee** class.
4. View the code for the **Student** class, and point out that it implements the **IPerson** interface and stores the **strCourse**, **strFName**, and **strLName** values. Also point out that the **Public Save** method implements the **Save** method of the **IPerson** interface.

↙ To test the application

1. Run the project.
2. Click the **New Employee** button on the form. The Visual Basic .NET process will halt at the first preset breakpoint.
3. Step through the code, explaining each line as you go. Point out that both the **Employee** class and the **IPerson** interface are used to access the various members. The **perPerson.Save** method is called first to show what happens if you use the **IPerson** interface. The **empEmployee.SaveEmployee** method shows that you can use any name that you choose for the implemented method.
4. Click the **New Student** button on the form. The code will enter break mode at the preset breakpoint.
5. Step through the code, explaining each line as you go. Point out the similarity in the calling code for the **IPerson** methods of both the **Student** and **Employee** objects, and explain how this aids code reuse.
6. Close the form and quit the Visual Studio .NET IDE.

Handling Events

↙ To view the code

1. Open the Events.sln solution in the *install folder*\DemoCode\Mod05\Events folder.
2. View the code for the **Employee** class, and point out the **DataChanged** event in the declarations section and its purpose. In the **FirstName** property, point out the raising of the event and the purpose of the code that checks the **blnCancelled** value.
3. View the code for the **frmEvents** form, and point out the module-level variable that uses **WithEvents**.
4. Click the **Class Name** combo box, and then click **empEmployee**. In the **Method Name** combo box, click **DataChanged**. Examine the default event handler code, and point out the **Handles** keyword in the function declaration.

↙ To test the events

1. Run the project.
2. Click the **WithEvents** button on the form. The code will enter break mode at the preset breakpoint.
3. Step through the code, explaining each line of code as you go. This will include the **RaiseEvent** code in the **Employee** class.
4. Click the **AddHandler** button on the form. The code will enter break mode at the preset breakpoint.
5. Explain the **AddHandler** statement, and examine the **EmployeeDataChange** method at the end of the code for the form.
6. Continue debugging as the events are raised to the **EmployeeDataChange** method.
7. Close the form and quit the Visual Studio .NET IDE.

Module Strategy

Use the following strategy to present this module:

- Defining Classes

  This lesson describes how to create classes in Visual Basic .NET. Students will learn how to declare methods and properties, and how to overload class members. When you introduce students to class constructors and destructors, including multiple constructors, point out that the **Finalize** method should only be used when resources need to be manually reclaimed (such as database connections) because this method adds overhead to the disposing of objects.

  Some of this lesson contains simple tasks such as how to create classes and methods. Cover these areas quickly so that more time can be spent on new features, such as the syntax for defining properties and constructors.

- Creating and Destroying Objects

  This lesson describes how to declare, instantiate, and initialize objects. Contrast this approach to the approach used in previous versions of Visual Basic to show the usefulness of constructors.

  Introduce garbage collection as an important change in the way objects are destroyed. Ensure that students understand this process, because many developers will be unaware of the potential dangers. Present the **Dispose** method as a suggested way to handle issues raised by garbage collection. Point out that the notes present two common techniques for creating the **Dispose** method, and that the **IDisposable** interface provides a more consistent approach. If time permits, you may want to show students the "GC Class" topic in the Visual Studio .NET documentation.

  Use the instructor-led demonstration to demonstrate how to create classes that contain multiple constructors. In this demonstration, you will also show how to instantiate and use classes from calling code.

  Have students complete the first lab after the demonstration.

- Inheritance

  This lesson explains how to use the new Visual Basic .NET class inheritance features. Introduce students to member overriding, the **MyBase** keyword, and the **MyClass** keyword. These important features will be new to many students, so be prepared to demonstrate different scenarios to reiterate the concepts.

  Use the instructor-led inheritance demonstration to show how to use the various keywords to create a simple **Person** class that is used as a base class for an **Employee** class.

- Interfaces

  This lesson explains how to define interfaces in Visual Basic .NET and examines the various ways to achieve polymorphism.

  Use the instructor-led polymorphism demonstration to show how to use interfaces to define an **IPerson** interface that is then implemented by **Student** and **Employee** classes.

  This lesson will be a challenge for many students. You will need to gauge the level of understanding and decide how much time to spend on the material and the demonstration. Tell students where they can find additional information about this topic, possibly including books or white papers.

  For more information about interfaces and polymorphism, search for "interfaces and polymorphism" in the Visual Studio .NET documentation and at http://msdn.microsoft.com

- Working with Classes

  This lesson shows how to create shared members, events, and delegates.

  The Event Handing topic mentions the **AddressOf** operator but does not explain it in depth because it is not new to Visual Basic .NET. Some students, however, may require a more detailed explanation.

  Students may also find the concept of delegates to be difficult. A detailed example is provided in the student notes, and you may need to go through this example with the students. This example is also provided in the DemoCode folder, although no instructions accompany this code.

  Use the instructor-led demonstration to show how to define and handle events in a simple class.

Overview

Topic Objective
To provide an overview of the module topics and objectives.

Lead-in
In this module, you will learn how to implement object-oriented programming in Visual Basic .NET.

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

Delivery Tip
This module introduces many important new features of Visual Basic .NET. Ensure that students fully understand these concepts before moving on to the next module.

In this module, you will learn how to implement object-oriented programming in Microsoft® Visual Basic® .NET version 7.0. You will learn how to define classes, their properties, and their methods. You will learn about the life cycle of an object, from creation to destruction. You will also learn how to work with classes by using inheritance, interfaces, polymorphism, shared members, events, and delegates.

After completing this module, you will be able to:

- Define classes.
- Instantiate and use objects in client code.
- Create classes that use inheritance.
- Define interfaces and use polymorphism.
- Create shared members.
- Create class events and handle them from a client application.

◆ Defining Classes

Topic Objective
To provide an overview of the topics covered in this lesson.

Lead-in
This lesson explains how to define classes.

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

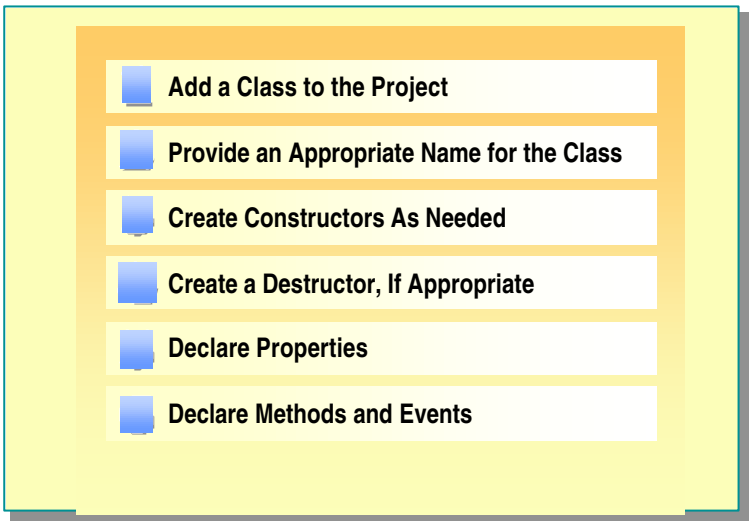
In this lesson, you will learn how to define classes in Visual Basic .NET. After completing this lesson, you will be able to:

- Specify access modifiers (scope) for classes and their procedures.
- Declare methods and properties within a class.
- Use attributes to provide metadata about your code.
- Pass different parameters to one method by using overloading.
- Create and destroy objects.

Procedure for Defining a Class

Topic Objective
To outline the procedure for defining a class.

Lead-in
Defining a class is a simple procedure.



Delivery Tip
Point out that you can define as many classes as you need in a module. You are not limited to one class per file, as you are in previous versions of Visual Basic.

To define a class in Visual Basic .NET, you can follow this general procedure:

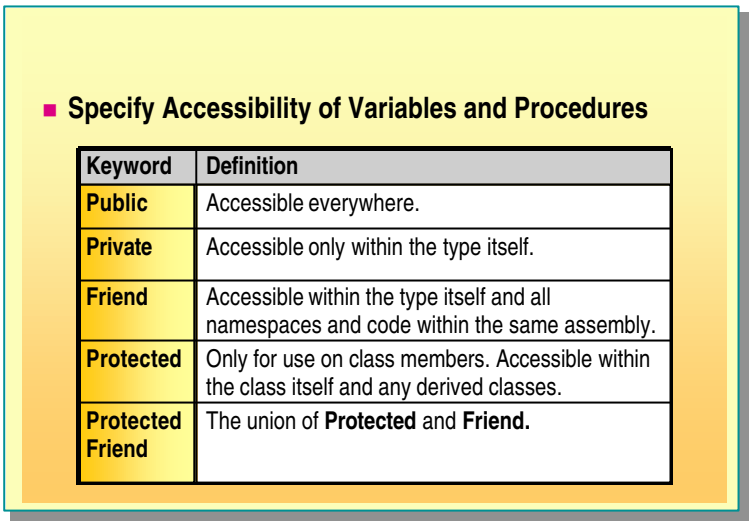
1. Add a class to the project.
2. Provide an appropriate file name for the class when you add it. This will name both the file and the class itself. If you do not change the file name when you add it, you can change the class name at any time by changing the class definition in the code window.
3. Create constructors as needed.
4. Create a destructor if appropriate.
5. Declare properties.
6. Declare methods and events.

Note In Visual Basic .NET, you can define more than one class in a single file. You are not limited to one class per file, as you are in previous versions of Visual Basic, because classes are a block level construct.

Using Access Modifiers

Topic Objective
To describe the access modifiers that are available in Visual Basic .NET.

Lead-in
Before looking at the details for defining classes, you need to understand how access modifiers affect classes.



You can use access modifiers to specify the scope of the variables and procedures in the class that you define. Visual Basic .NET retains three access modifiers that are used in previous versions of Visual Basic and adds two more: **Protected** and **Protected Friend**. The following table defines the five access modifiers available in Visual Basic .NET.

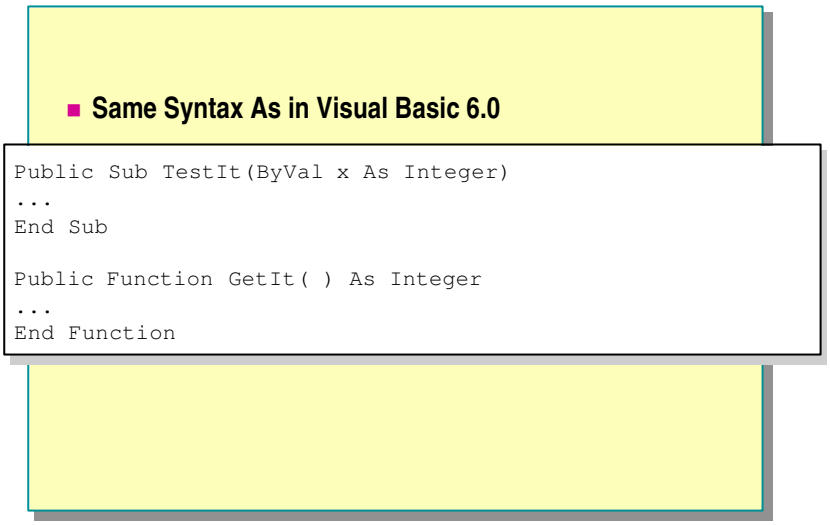
Access modifier	Definition
Public	Accessible everywhere.
Private	Accessible only within the type itself.
Friend	Accessible within the type itself and all namespaces and code within the same assembly.
Protected	Accessible within the class itself and, if other classes are inheriting from the class, within any derived classes. Protected members are available outside of an assembly when inherited by derived classes.
Protected Friend	The union of **Protected** and **Friend**. Accessible to code within the same assembly and to any derived classes regardless of the assembly to which they belong.

Note Inheritance in Visual Basic .NET is described in detail in the Inheritance lesson of this module.

Declaring Methods

Topic Objective
To explain how to declare methods in a class.

Lead-in
The syntax for declaring methods in a class has not changed in Visual Basic .NET.



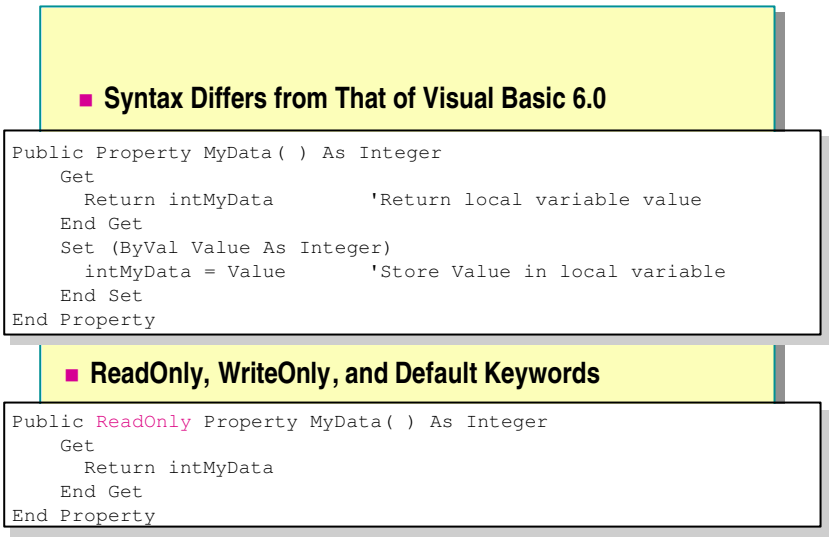
You use the same syntax to declare a method in Visual Basic .NET that you used in Visual Basic 6.0, as shown in the following example:

```
Public Class TestClass

    Public Sub TestIt(ByVal x As Integer)
        ...
    End Sub

    Public Function GetIt( ) As Integer
        ...
    End Function

End Class
```

Declaring Properties

Topic Objective
To explain how to declare properties in a class.

Lead-in
The syntax for declaring class properties has changed significantly in Visual Basic .NET.



Delivery Tip
This is an animated slide. It begins by showing the property syntax. Click the slide to reveal a **ReadOnly** example.

The syntax for declaring class properties has changed significantly in Visual Basic .NET.

Syntax for Declaring Properties

In Visual Basic 6.0, you create properties by declaring two separate procedures: one for the **Get** and one for the **Let** or **Set**. In Visual Basic .NET, you declare your properties by using two code blocks in a single procedure, as follows:

```
[Default|ReadOnly|WriteOnly] Property varname ([parameter
list]) [As typename]
    Get
        [block]
    End Get
    Set (ByVal Value As typename)
        [block]
    End Set
End Property
```

When you create a property declaration, the Visual Basic .NET Code Editor inserts a template for the remainder of the property body.

Example

The following example shows how to declare a property called **MyData** of type **Integer**. The **Get** block returns an unseen local variable called *intMyData* by using a **Return** statement. The **Set** block uses the **Value** parameter to store the passed-in property value to the *intMyData* local variable.

```
Private intMyData As Integer

Public Property MyData( ) As Integer
    Get
        Return intMyData
    End Get
    Set (ByVal Value As Integer)
        intMyData = Value
    End Set
End Property
```

Using Read-Only Properties

You can create read-only properties by using the **ReadOnly** keyword when you declare the property. Read-only properties cannot be used in an assignment statement. The following example shows how to specify a read-only property:

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


You cannot use the **Set** block when defining read-only properties because the property cannot be updated. The compiler will generate an error if you attempt to do this.

Using Write-Only Properties

You can create write-only properties by using the **WriteOnly** keyword when you declare the property. Write-only properties cannot be used to retrieve the value of the property. The following example shows how to create a write-only property:

```
Public WriteOnly Property MyData( ) As Integer
    Set (ByVal Value As Integer)
        intMyData = Value
    End Set
End Property
```

You cannot use the **Get** block when defining write-only properties because the property is not readable. The compiler will generate an error if you attempt to do this.

Using Default Properties

You can create a default property for a class by using the **Default** keyword when you declare the property. You must code the property to take at least one argument, and you must specify **Public**, **Protected**, or **Friend** access.

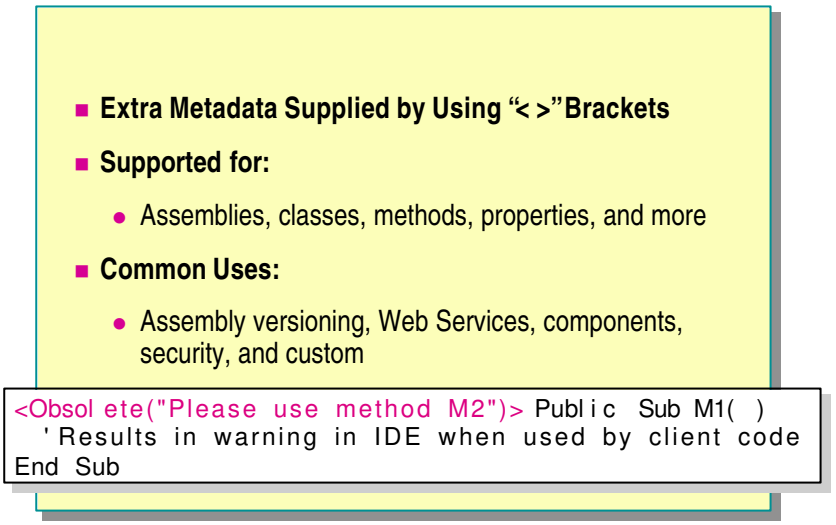
The following example shows how to declare a default property that takes an index as an argument and returns a **Boolean** value:

```
Default Public Property Item(ByVal index As Integer) _
                                As Boolean
  Get
    Return myArray(index) 'Uses a private module-level array
  End Get
  Set(ByVal Value As Boolean)
    myArray(index) = Value
  End Set
End Property
```

Using Attributes

Topic Objective
To explain how to use attributes.

Lead-in
Attributes provide extra information about your code.



You can use attributes to provide extra information or metadata to the developers who read your code. In Visual Basic .NET, you use angular brackets (< and >) to specify an attribute.

Delivery Tip
Discuss some of the common uses for attributes, but point out that we will cover individual attributes throughout the course.

You can apply attributes to many items within your application, including assemblies, modules, classes, methods, properties, parameters, and fields in modules, classes, and structures.

Here are some examples of how to use attributes:

- In assemblies, you can specify metadata including the title, description, and version information of the assembly.
- When creating a Web Service, you can define which methods are accessible as part of the service, in addition to adding descriptions to the methods.
- When designing Windows Forms controls, you can specify information to display in the property browser, or you can set the Toolbox icon.
- For components that use enterprise services, you can set transaction and security settings.

Functionality is provided by the Microsoft .NET Framework to allow you to create your own custom attributes and use them as you want in your applications.

Note For more information about creating custom attributes, see "Writing Custom Attributes" in the Microsoft Visual Studio® .NET documentation.

Example

The following example shows how to use the **Obsolete** attribute to warn developers that a method can no longer be used. An optional message is displayed in the Task List window if a developer attempts to use this method. Using the **Obsolete** method will not create an error when the application is compiled, but will generate a warning as follows:

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

Note All attributes are simply classes that inherit from the **Attribute** class and provide constructors. The **Obsolete** class provides a single constructor that takes a string as the parameter.

Overloading Methods

Topic Objective
To introduce the concept and syntax of overloading in Visual Basic .NET.

Lead-in
Overloading is a powerful object-oriented feature that allows multiple methods to have the same name but accept different parameters.

- **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**

Delivery Tip
Students who are familiar with Microsoft Visual C++ will understand operator overloading. You might want to ask for examples from these students.

Overloading is a powerful object-oriented feature that allows multiple methods to have the same name but accept different parameters. Overloading allows calling code to execute one method name but achieve different actions, depending on the parameters you pass in.

For overloading to occur, the method signature must be unique. You can achieve this by changing the number of parameters in the signature or by changing the data types of the parameters. Changing the way a parameter is passed (that is, by value or by reference) does not make a signature unique, nor does changing a function return data type.

You can optionally specify a method as overloaded with the **Overloads** keyword. If you do not use the keyword, the compiler assumes it by default when you declare multiple methods that have the same name. However, when overloading a method from an inherited class, you must use the **Overloads** keyword.

Note Overloading a method from an inherited class will be discussed in the Inheritance lesson of this module.

The following example shows how to overload a method. This code allows different types of information (string, integers, and so on) to be displayed by calling the **Display** method of a class and passing in 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
```

When you call the **Display** method, the parameters you specify determine which overloaded method will be called.

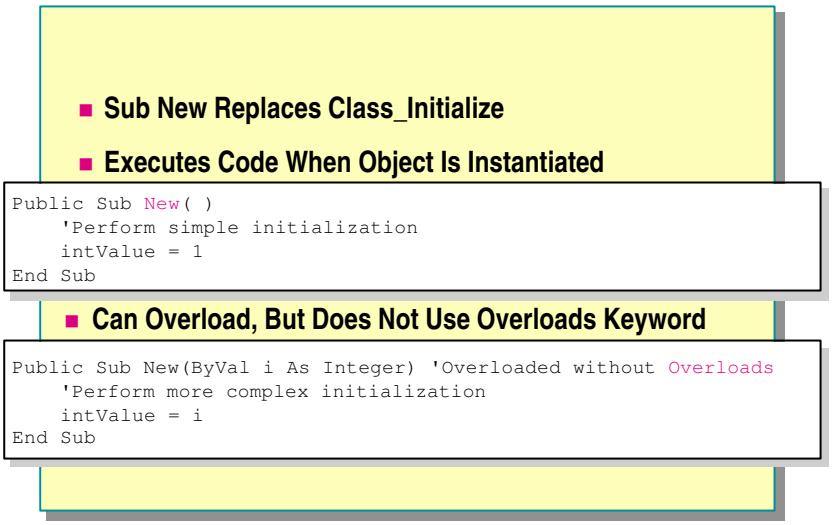
Without using overloading, you need two different methods, such as **DisplayString** and **DisplayInteger**, to accept the different types of parameters in the preceding example.

Note If the **Option Strict** compiler option is on, you must explicitly declare values as specific types when passing them to the overloaded methods as parameters, and the compiler can identify which instance of the method to call. If **Option Strict** is off and a generic variable (such as **Object**) is passed as a parameter, the decision of which instance of the method to call is left until run time. For more information about overload resolution, see "Procedure Overloading" in the Visual Studio .NET documentation.

Using Constructors

Topic Objective
To explain the concept and syntax of class constructors.

Lead-in
Visual Basic .NET allows you to create class constructors.



Delivery Tip
This is an animated slide. It begins by showing the first example. Click the slide to reveal an example of overloading.

In Visual Basic 6.0, you place initialization code in the **Class_Initialize** event of your class. This code is executed when the object is instantiated, and you can use it to set initial values of local variables, to open resources, or to instantiate other objects.

In Visual Basic .NET, you control the initialization of new objects by using procedures called *constructors*. The **Sub New** constructor replaces the **Class_Initialize** event and has the following features:

- The code in the **Sub New** block will always run before any other code in a class.
- Unlike **Class_Initialize**, the **Sub New** constructor will only run once: when an object is created.
- **Sub New** can only be called explicitly in the first line of code of another constructor, from either the same class or a derived class using the **MyBase** keyword.

Using Sub New

The following example shows how to use the **Sub New** constructor:

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

The change in Visual Basic .NET from the **Class_Initialize** event to the **Sub New** constructor means you can overload the **New** subroutine and create as many class constructors as you require. This is useful if you want to initialize your object when you instantiate it. To do this in Visual Basic 6.0, you must call methods or properties after the object is created.

Overloading Constructors

You can overload constructors just as you can overload any other method in a class. However, you cannot use the **Overloads** keyword when overloading constructors. The following example shows how to overload the **New** subroutine and create multiple class constructors:

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

Public Sub New(ByVal i As Integer)
    'Perform more complex initialization
    intValue = i
End Sub

Public Sub New(ByVal i As Integer, _
      ByVal s As String)
    intValue = i
    strValue = s
End Sub
```

Using Destructors

Topic Objective
To explain destructors and their syntax in Visual Basic .NET.

Lead-in
Destructors are used to destroy objects in Visual Basic .NET.

- **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
```

In Visual Basic .NET, you can control what happens during the destruction of objects by using procedures called *destructors*.

Delivery Tip
Point out that the Finalize method should only be used where necessary so it doesn't cause unnecessary processing during object clean-up.

The new **Finalize** destructor replaces the **Class_Terminate** event found in previous versions of Visual Basic. This subroutine is executed when your object is destroyed, and you can use it to clean up open resources, such as database connections, or to release other objects in an object model hierarchy.

The following example shows how to use the **Finalize** destructor:

```
Protected Overrides Sub Finalize( )
    'Can close connections of other resources
    conn.Close
End Sub
```


Note You will learn about the **Overrides** keyword in the Inheritance lesson of this module.

Delivery Tip
Point out that garbage collection is covered in more depth in the next lesson.

In Visual Basic 6.0, the **Class_Terminate** event runs when an object is no longer being referenced by any variables. You use the **Set x = Nothing** statement to release a particular reference. When all the references are gone, the event executes and resources can be cleaned up.

In Visual Basic .NET, when you set an object reference to **Nothing**, you still release variables. However, the object may not be destroyed until a later stage due to the introduction of garbage collection.

◆ Creating and Destroying Objects

Topic Objective
To provide an overview of the topics covered in this lesson.

Lead-in
In this lesson, you will learn about creating and destroying objects.

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

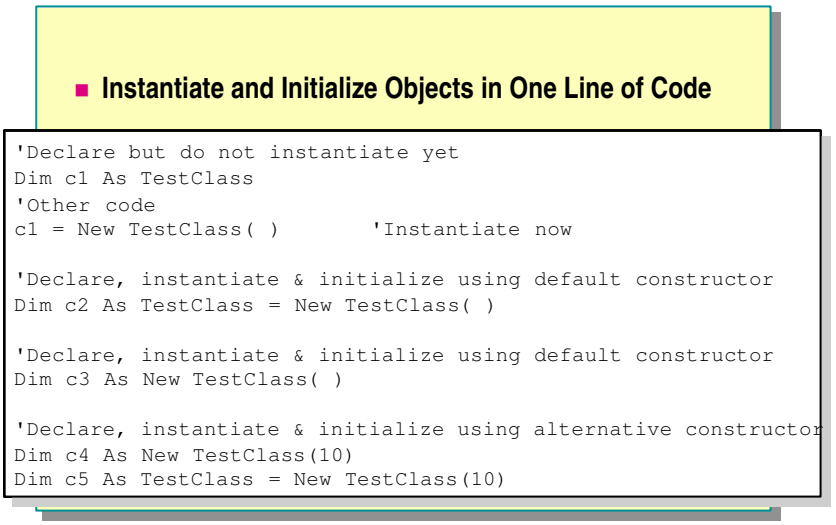
In this lesson, you will learn about creating and destroying objects. After completing this lesson, you will be able to:

- Instantiate and initialize objects.
- Explain the role that garbage collection plays in the object life cycle.
- Use the **Dispose** method to destroy an object and safely clean up its resources.

Instantiating and Initializing Objects

Topic Objective
To explain how to instantiate and initialize objects.

Lead-in
The method for instantiating and initializing objects has changed in Visual Basic .NET.



You can now instantiate and initialize objects in one line of code. This means you can write simpler and clearer code that can call different class constructors for multiple variables.

Delivery Tip
This is an animated slide. It begins by showing the first part of the example. Click the slide to reveal the following sections:
1. c2 example
2. c3 example
3. c4 and c5 example

Example 1

The following example shows how to declare a variable in one line and instantiate it in a following line. Remember that the **Set** keyword is no longer needed.

```
'Declare but do not instantiate yet
Dim c1 As TestClass
'Other code
c1 = New TestClass( )        'Instantiate now
```


Example 2

The following example shows how to declare, instantiate, and initialize an object in one statement. The default constructor for the class will be executed.

Delivery Tip
Ensure that students are aware of the differences between the behavior of the following statement in Visual Basic .NET as opposed to previous versions of Visual Basic:
Dim x As New TestClass

```
'Declare, instantiate & initialize using default constructor
Dim c2 As TestClass = New TestClass( )
```

Example 3

The following example performs the same functionality as Example 2. It looks similar to code from previous versions of Visual Basic, but behaves quite differently.

```
'Declare, instantiate & initialize using default constructor
Dim c3 As New TestClass
```

Visual Basic 6.0

In Visual Basic 6.0, the preceding code creates the object when the object is first used. If you destroy the variable by assigning the **Nothing** keyword, it will automatically be recreated when it is next referenced.

Visual Basic .NET

In Visual Basic .NET, the preceding code declares and instantiates the object variables immediately. If you destroy the variable by assigning the **Nothing** keyword, it will not automatically be recreated when it is next referenced.

Example 4

The following examples show how to declare, instantiate, and initialize objects in single statements. Both statements call alternative constructors for the class.

```
'Declare, instantiate & initialize using alternate constructor
Dim c4 As New TestClass(10)
Dim c5 As TestClass = New TestClass(10)
```

Garbage Collection

Topic Objective
To explain garbage collection and how it affects object lifetime.

Lead-in
Garbage collection significantly alters how objects are destroyed.

- **Background Process That Cleans Up Unused Variables**
- **Use *x = Nothing* to Enable Garbage Collection**
- **Detects Objects or Other Memory That Cannot Be Reached by Any Code (Even Circular References!)**
- **Calls Destructor 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

In previous versions of Visual Basic, object destruction is based on a reference count. References are removed when you set the object to **Nothing** or when the variable goes out of scope. When all references to an object have been removed, the object is destroyed. This is effective in most situations, but some objects, such as those left orphaned in a circular reference relationship, may not be destroyed.

In Visual Studio .NET, setting an object reference to **Nothing** or allowing it to go out of scope removes the link from the variable to the object and allows garbage collection to take place. This background process traces object references and destroys those that cannot be reached by executing code, including objects that are not referenced. The garbage collector executes the object destructor (the **Finalize** method discussed earlier in this module.)

Garbage collection provides several performance advantages:

- It cleans up circular references and improves code performance because objects do not need to keep a reference count.
- Because reference counting is no longer required, the time taken to instantiate an object is reduced.
- The time taken to release an object variable reference is reduced.
- No storage is required for the reference count, which means that the amount of memory that an object uses is also reduced.

It is important to note that garbage collection introduces a time delay between when the last reference to an object is removed and when the collector destroys the object and reclaims the memory. This time delay can be quite significant if the object is holding open resources that may affect the scalability or performance of the application, such as database connections.

It is possible to attempt to force the collection of unused objects by using the **GC** (Garbage Collector) system class' **Collect** method as follows:

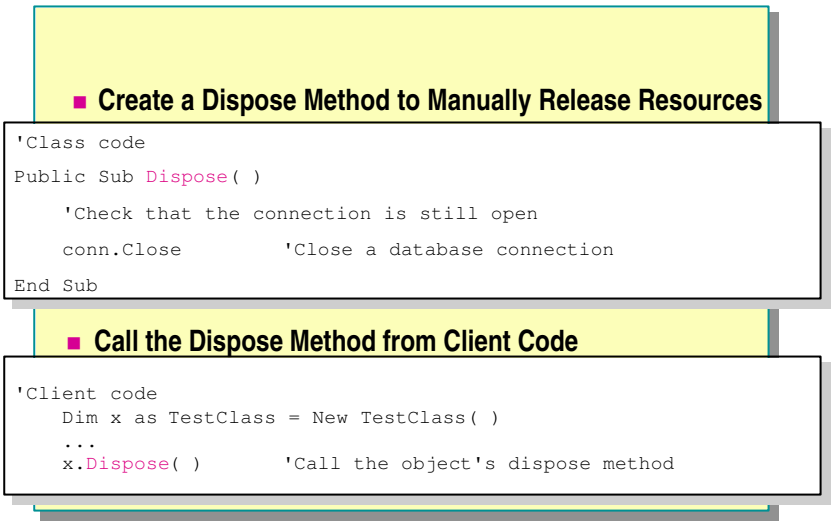
```
GC.Collect()
```

However, using the **Collect** method is not recommended because forcing garbage collection may result in poor performance because of the unnecessary collection of other unused objects. You should instead use a standard method such as **Dispose**, which is discussed in the next topic.

Using the Dispose Method

Topic Objective
To explain how to use the **Dispose** method to aid resource management.

Lead-in
One way to overcome the time delay issue of garbage collection is to use a standard method such as the **Dispose** method.



Delivery Tip
This is an animated slide. It begins by showing the **Dispose** example. Click the slide to reveal the client code example.

Point out that you must be cautious when executing code in the **Finalize** method if resources have been cleaned up in a **Dispose** method.

Because of the potential time delay created by garbage collection, you may want to create a standard method called **Dispose** for your class. Many Visual Studio .NET objects use this method to clean up resources.

When client code has no further need for an object's resources, it can directly call code placed in the **Dispose** method of the object. If the client code does not call the **Dispose** method explicitly before garbage collection occurs, the **Finalize** method of the class can also call the **Dispose** method. However, note that you may cause an exception if you attempt to run the **Dispose** code twice. An exception can occur if you have already closed or released an object and do not test to determine whether it still exists in the second execution of the **Dispose** code.

Example

The following simple example shows how to 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

Protected Overrides Sub Finalize( )
    Dispose( )   'Optional call to Dispose
End Sub

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

The IDisposable Interface

Visual Basic .NET provides an interface called **IDisposable** to improve accuracy and consistency among objects. This interface provides one method, **Dispose**, which does not take any arguments. By implementing this interface in all of your classes, you will consistently provide a **Dispose** method that can be easily called by client code.

Note You will learn how to implement interfaces in the Interfaces lesson of this module.

If you completely clean up your object in a **Dispose** method (whether you use **IDisposable** or not), garbage collection does not need to execute your **Finalize** method. You can disable the execution of the **Finalize** method by calling the **SuppressFinalize** method on the **GC** object, as shown in the following example. This method accepts a single argument that is a reference to the object that should not have its **Finalize** method called. In Visual Basic .NET, this is done with the **Me** keyword.

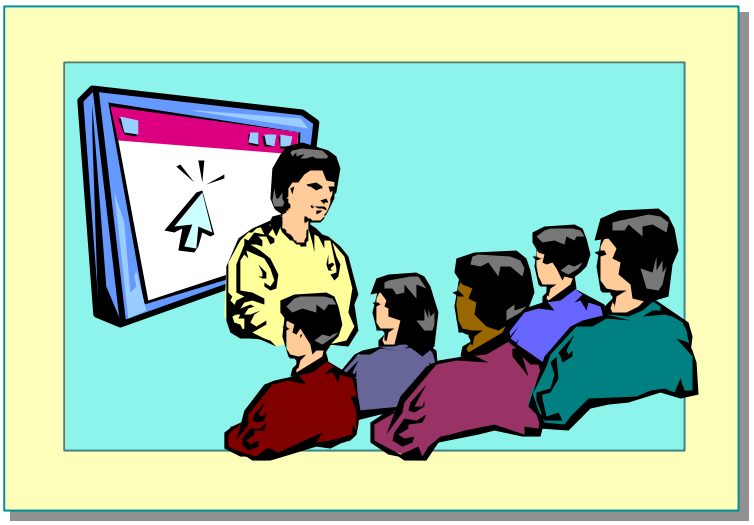
In the following example, if the client code called the **Dispose** method directly, the connection would be closed and the **Finalize** method would not be called by garbage collection. If the client did not call the **Dispose** method, the **Finalize** method will still execute when garbage collection destroys the object.

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

Protected Overrides Sub Finalize( )
    Dispose( )  'Optional call to Dispose
End Sub
```

Demonstration: Creating Classes

Topic Objective
To demonstrate how to create and use simple classes.

Lead-in
This demonstration will show you how to define classes and instantiate and destroy objects.



Delivery Tip
The step-by-step instructions for this demonstration are in the instructor notes for this module.

In this demonstration, you will learn how to define a simple class that uses multiple constructors. You will also learn how to instantiate and use the class from within client code.

Lab 5.1: Creating the Customer Class

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will define the **Customer** class for the Cargo system.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Create classes.
- Instantiate, initialize, and use classes from calling code.

Prerequisites

Before working on this lab, you should be familiar with creating classes in Visual Basic .NET.

Scenario

In this lab, you will begin creating the Cargo system. You will create the **Customer** class and a test application to instantiate, initialize, and test the class.

Starter and Solution Files

There are starter and solution files associated with this lab. The starter files are in the *install folder*\Labs\Lab051\Starter folder, and the solution files are in the *install folder*\Labs\Lab051\Solution folder.

Estimated time to complete this lab: 45 minutes

Exercise 1
Defining the Customer Class

In this exercise, you will define the **Customer** class. The starter project contains several forms that you will use to test your **Customer** class.

↙ To open the starter project

1. Open Visual Studio .NET.
2. On the **File** menu, point to **Open**, and then click **Project**. Set the folder location to *install folder*\Labs\Lab051\Starter, click **Lab051.sln**, and then click **Open**.

↙ To create the Customer class

1. On the **Project** menu, click **Add Class**.
2. In the **Add New Item** dialog box, change the name of the class file to **Customer.vb**, and click **Open**.

↙ To define the class properties

1. Add the following private variables to the class definition.

Variable name	Data type
intCustomerID	**Integer**
strFName	**String**
strLName	**String**
strEmail	**String**
strPassword	**String**
strAddress	**String**
strCompany	**String**

2. Add the following public properties, and use these to access the private variables created in the previous step.

Property name	Read/Write access	Data type
CustomerID	Read-only	**Integer**
FirstName	Read-write	**String**
LastName	Read-write	**String**
Email	Read-write	**String**
Password	Read-write	**String**
Address	Read-write	**String**
Company	Read-write	**String**

3. Save the project.

To define the class methods

1. Add the following methods to the class definition.

Method name	Type	Parameters
LogOn	Public Sub	ByVal strEmail As String ByVal strPassword As String
AddCustomer	Public Function	ByVal strEmail As String ByVal strPassword As String ByVal strFName As String ByVal strLName As String ByVal strCompany As String ByVal strAddress As String <RETURN VALUE> As Integer
New	Public Sub	<None>
New	Public Sub	ByVal intID As Integer

2. On the **File** menu, point to **Open**, and click **File**. In the **Files of type** list, click **Text Files**. Click **Code.txt**, and then click **Open**.
3. Locate the LogOn code in Code.txt. Copy the procedure code to the **LogOn** method of the **Customer** class.
4. Locate the AddCustomer code in Code.txt. Copy the procedure code to the **AddCustomer** method of the **Customer** class.

To complete the class constructors

1. In the **Customer** class, locate the default constructor definition (the `Sub New` without parameters), and initialize the *intCustomerID* variable to **-1**.
2. Locate the alternative constructor code in Code.txt. Copy the procedure code to the parameterized constructor of the **Customer** class.
3. Save the project.

Exercise 2
Testing the LogOn Procedure

In this exercise, you will test the **LogOn** procedure from a simple form.

↙ To create the Logon button code

1. Open frmLogOn in the Code Editor and locate the **btnLogOn_Click** event procedure.
2. Declare and instantiate a **Customer** variable called *cusCustomer.*
3. Call the **LogOn** method of the **cusCustomer** object, passing in the text properties of txtEmail and txtPassword as parameters.
4. Display the properties of the **cusCustomer** object in the appropriate text boxes. Use the information in the following table:

Text box	Property of cusCustomer
txtID	**CStr(CustomerID)**
txtFName	**FirstName**
txtLName	**LastName**
txtAddress	**Address**
txtCompany	**Company**

5. Set cusCustomer to **Nothing**.
6. Save the project.

↙ To test the LogOn code

1. Set a breakpoint on the first line of the **btnLogOn_Click** procedure.
2. On the **Debug** menu, click **Start**. On the menu form, click **Test 'Logon'** to display the test form, and then type the following values in the appropriate text boxes.

Text box	Value
E-mail	**karen@wingtiptoys.msn.com**
Password	**password**

3. Click the **Logon** button, and step through the procedure.
4. Confirm that your code retrieves the customer information and displays it correctly in the text boxes. Close the form.
5. Reopen the LogOn form and enter the following incorrect values in the appropriate text boxes.

Textbox	Value
E-mail	**john@tailspintoys.msn.com**
Password	**john**

6. Click the **Logon** button, and step through the procedure.
7. Confirm that your code causes an exception to be generated and handled by the form.
8. Click the **Close** button to quit the application. Remove the breakpoint on **btnLogOn_Click**.

Exercise 3
Testing Customer Retrieval

In this exercise, you will test the parameterized constructor that retrieves the customer details from a simple form. A sales agent who needs full access to the customer's information could use this type of form.

↙ To create the Retrieve button code

1. Open frmRetrieve in the Code Editor, and locate the **btnRetrieve_Click** event procedure.
2. Declare and instantiate a **Customer** variable called *cusCustomer*. Use the parameterized constructor to pass in the existing customer ID from the txtID text box. (Use the **CInt** function to convert it to an integer value.)
3. Display the properties of the **cusCustomer** object in the appropriate text boxes. Use the information in the following table:

Textbox	Property of cusCustomer
txtEmail	**Email**
txtPassword	**Password**
txtFName	**FirstName**
txtLName	**LastName**
txtAddress	**Address**
txtCompany	**Company**

4. Save the project.

↙ To test the Retrieve code

1. Set a breakpoint on the first line of the **btnRetrieve_Click** procedure.
2. On the **Debug** menu, click **Start**. On the menu form, click **Test 'Get Details'** to display the test form, and then type the value **1119** in the **CustomerID** text box.
3. Click the **Retrieve** button, and step through the procedure.
4. Confirm that your code retrieves the customer information and displays it correctly in the text boxes.
5. Click the **Clear Data** button to reset the information, and type the value **1100** in the **CustomerID** text box.
6. Click the **Retrieve** button, and step through the procedure.
7. Confirm that your code causes an exception to be generated and handled by the form.
8. Click the **Close** button to quit the application. Remove the breakpoint on **btnRetrieve_Click**.

Exercise 4
Testing the AddCustomer Procedure

In this exercise, you will test the **AddCustomer** procedure from a simple form.

↙ To create the Add Customer button code

1. Open frmNew in the Code Editor, and locate the **btnNew_Click** event procedure.
2. Declare and instantiate a **Customer** variable called *cusCustomer*.
3. Call the **AddCustomer** function of the **cusCustomer** object, passing in the appropriate values and displaying the return value in a message box. Use the **CStr** function to convert the integer value to a string. Use the information in the following table:

Parameter name	Value
strEmail	**txtEmail.Text**
strPassword	**txtPassword.Text**
strFName	**txtFName.Text**
strLName	**txtLName.Text**
strCompany	**txtCompany.Text**
strAddress	**txtAddress.Text**

4. Save the project.

↙ To test the Add Customer code

1. Set a breakpoint on the first line of the **btnNew_Click** procedure.
2. On the **Debug** menu, click **Start**. On the menu form, click **Test 'New Customer'** to display the test form.
3. Enter values in all text boxes.
4. Click the **New Customer** button, and step through the procedure.
5. Confirm that your code passes the information to the procedure correctly, and that a new ID is returned.
6. Click the **Close** button and quit the application. Remove the breakpoint on **btnNew_Click**.

◆ Inheritance

Topic Objective
To provide an ov erview of the topics covered in this lesson.

Lead-in
In this lesson, you will learn about inheritance.

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

Delivery Tip
Inform students that although inheritance is an important new feature of Visual Basic .NET, you can continue to create classes without inheritance, as in previous versions of Visual Basic.

In this lesson, you will learn how to implement class inheritance. After completing this lesson, you will be able to:

- Inherit from an existing class.
- Override and overload some base class methods in a derived class.
- Use the **MyBase** keyword to access the base class from a derived class.
- Use the **MyClass** keyword to ensure that you call the correct class.

What Is Inheritance?

Topic Objective
To explain the keywords and syntax for class inheritance.

Lead-in
Inheriting a class allows you to use the features of a base class without rewriting the code.

- **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

---

Delivery Tip
Point out that inheritance can be used to create highly reusable code. However, many applications do not need inheritance and can perform perfectly well without it. Make clear that if inheritance is used, students should not create complex hierarchies that will become unmanageable.

In Visual Basic .NET, you can use inheritance to derive a class from an existing base class. The derived class can inherit all the base class properties, methods, data members, events, and event handlers, making it easy to reuse the base class code throughout an application.

The Inherits Keyword

The following example shows how to use the **Inherits** keyword to define a derived class that will inherit from an existing base class:

```
Public Class DerivedClass
    Inherits BaseClass
...
End Class
```

The NotInheritable Keyword

The following example shows how to use the **NotInheritable** keyword to define a class that cannot be used as a base class for inheritance. A compiler error is generated if another class attempts to inherit from this class.

```
Public NotInheritable Class TestClass
...
End Class
Public Class DerivedClass
    Inherits TestClass   'Generates a compiler error
...
End Class
```

The MustInherit Keyword

You use the **MustInherit** keyword to define classes that are not intended to be used directly as instantiated objects. The resulting class must be inherited as a base class for use in an instantiated derived class object. If the client code attempts to instantiate an object based on this type of class, a compiler error is generated, as shown in the following example:

```
Public MustInherit Class BaseClass
...
End Class
...
'Client code
Dim x As New BaseClass( ) 'Generates a compiler error
```

The Protected Keyword

You use **Protected** access to limit the scope of a property, method, data member, event, or event handler to the defining class and any derived class based on that base class. Following is an example:

```
Public Class BaseClass
    Public intCounter As Integer   'Accessible anywhere

   'Accessible only in this class or a derived class
    Protected strName As String
...
End Class
```

Note The derived class is also known as a *subclass*, and the base class is known as a *superclass* in Unified Modeling Language (UML) terminology.

Overriding and Overloading

Topic Objective
To explain how to override and overload methods and properties in a derived class.

Lead-in
You may want to implement your own logic in a derived class rather than use that of the base class. This is called *overriding*. You can also extend members of the base class with the **Overload** keyword.

- **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**

Delivery Tip
Point out that examples are shown in the student notes but that a fuller example will be discussed on the next slide.

When a derived class inherits from a base class, it inherits all the functions, subroutines, and properties of the base class, including any implementation in the methods. Occasionally you may want to create implementation code specific to your derived class rather than using the inherited methods. This is known as *overriding*. You can also overload methods defined in the base class with the **Overloads** keyword.

Overriding

Use the following keywords to create your own implementation code within a derived class:

- **Overridable**

  To create your own special implementation of the derived class, specify the **Overridable** keyword in a base class member definition for a function, subroutine, or property, as shown in the following example:

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

- **MustOverride**

  To create a base class member that must be overridden in all derived classes, define the member with the **MustOverride** keyword. Only the member prototype can be created in the base class, with no implementation code. You can only use this keyword in a base class that is marked as **MustInherit**. The following example shows how to define a method that must be overridden:

```
Public MustOverride Sub PerformAction( )
```

  MustOverride methods are useful in base classes because they allow you to define baseline functionality without locking in implementation details that can make them difficult to extend.

- **Overrides**

  To specify that a derived class method overrides the implementation of the base class method, use the **Overrides** keyword. If the base class method that is being overridden is not marked as **Overridable**, a compile-time error will occur. The method signature must exactly match the method being overridden, except for the parameter names. The following example shows how to declare a derived class method that overrides the base class implementation:

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

Note You can override methods by selecting (Overrides) in the Class Name drop-down list in the IDE, and then selecting the method you want to override.

- **NotOverridable**

  Base class members without the **Overridable** keyword are, by default, not overridable. However, if a base class member is marked as overridable, then the member will be overridable in any derived classes based on the immediate deriving class. To prevent this behavior, mark the overridden method in the derived class as **NotOverridable**. This will stop subsequent inheritance from overriding the method.

  The following example shows how to declare a derived class method that overrides the base class implementation but does not allow any further overriding:

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


Overloading

You can create a method in a derived class that overloads a method defined in a base class by using the **Overloads** keyword. Just as for overloading methods within the same class, the method signatures must include different parameters or parameter types. The following example shows how to overload a method from a base class:

```
Public Overloads Sub Other(ByVal i As Integer)
    MsgBox("Overloaded CannotOverride")
End Sub
```

Note that the base class method does not need to be marked as **Overridable** to be overloaded.

Inheritance Example

Topic Objective
To provide an example of inheritance.

Lead-in
Let's look at a full example of inheritance.

```
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"
```

Delivery Tip
This is an animated slide. It begins by showing an example of code for base class. Click the slide to reveal the following sections:
1. Derived class example
2. Client code example

Explain the example from the slide, and then look at the examples in the notes.

There are three parts to this inheritance example:

- Code for the base class
- Code for the derived class
- Code for the calling client

Base Class Code

The base class in the following example is specified as **MustInherit**. This means that the class must be inherited from because it cannot be instantiated directly.

```
Public MustInherit Class BaseClass
    Public MustOverride Sub PerformAction( )

    Public Overridable Sub OverrideMethod( )
        MsgBox("Base OverrideMethod")
    End Sub

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

The following table explains the methods used in the preceding code.

Method	Declared as	Description
PerformAction	**MustOverride**	Any implementation for this method must be created in the deriving class.
OverrideMethod	**Overridable**	Any implementation for this method can be overridden as a derived class.
Other	**NotOverridable** (by default)	Any implementation for this method cannot be overridden in a derived class. **NotOverridable** is the default for any method.

Derived Class Code

The derived class in the following example inherits from the base class. This means that the class inherits all of the methods and properties of the base class.

```
Public Class DerivedClass
    Inherits BaseClass

    Public NotOverridable Overrides Sub PerformAction( )
        MsgBox("Derived PerformAction")
    End Sub

    Public Overrides Sub OverrideMethod( )
        MsgBox("Derived OverrideMethod")
    End Sub

    Public Overloads Sub Other(ByVal i As Integer)
        MsgBox("Overloaded Other")
    End Sub
End Class
```


Because the **PerformAction** method was marked as **MustOverride** in the base class, it must be overridden in this derived class. This derived class also marks the method as **NotOverridable** so that no other class can override this method if **DerivedClass** is used as a base class for inheritance.

The method **OverrideMethod** is overridden in this derived class. Any calls to **OverrideMethod** will result in the derived class implementation being executed rather than the base class implementation.

The **Other** method cannot be overridden, but can be overloaded by the derived class using the **Overloads** keyword.

Calling Code

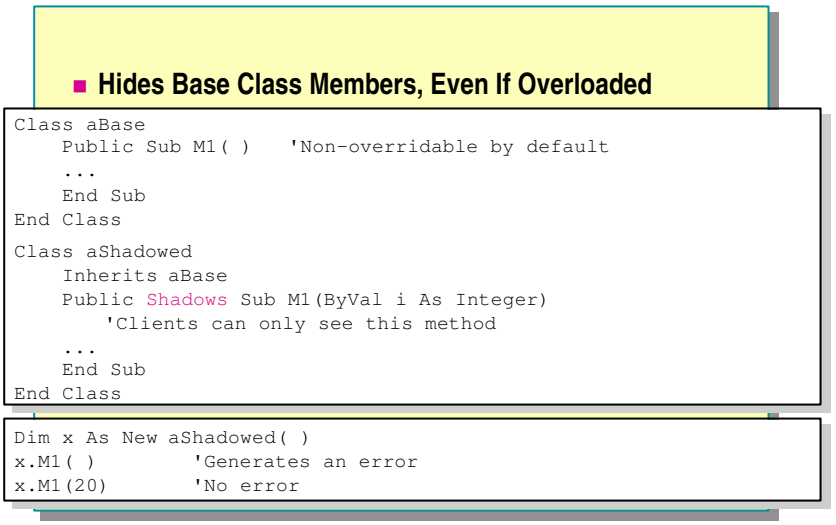
The preceding example defines and instantiates a **DerivedClass** variable. The following example shows how to call all the individual methods for the derived class. The results are shown as comments in the code.

```
Dim x As DerivedClass = New DerivedClass( )
x.Other( )      'Displays "Base Other method – not overridable"
x.Other(20)         'Displays "Overloaded Other"
x.OverrideMethod( )  'Displays "Derived OverrideMethod"
x.PerformAction( )   'Displays "Derived PerformAction"
```

Shadowing

Topic Objective
To describe the concept of shadowing and explain how to shadow a method in a derived class.

Lead-in
Visual Basic .NET allows you to shadow a method in a derived class.



When a derived class inherits from a base class, it can either override a method on the base class or shadow it. Overriding replaces the existing method based on the method name and signature. Shadowing effectively hides the method in the base class, based solely on the method name. This means shadowing a method also hides any overloaded methods within the base class. You can shadow a method regardless of whether the base method is specified as overridable.

To learn how shadowing works, consider an example of a derived class that shadows a method from the base class. The method in the base class has not been specified as overridable.

The following example shows a base class that defines a single method called **M1**. The derived class declares an **M1** method that automatically shadows the base class method and accepts a single argument. The client code can only access the shadowed method that accepts the argument, and an error will be generated if it attempts to access the base class method.

```
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

'Client Code
Dim x As New aShadowed( )
x.M1( )   'Generates an error because method is hidden
x.M1(20)  'No error
```

Using the MyBase Keyword

Topic Objective
To explain how to use the **MyBase** keyword.

Lead-in
Sometimes you need to access the base class from a derived class.

- **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
```

You can use the **MyBase** keyword to access the immediate base class from which a derived class is inheriting. When using **MyBase**, you should be aware of some limitations:

Delivery Tip
Calling the constructor of a base class is a common use of the **MyBase** keyword.

- It refers only to the immediate base class in the hierarchy. You cannot use **MyBase** to gain access to classes higher in the hierarchy.
- It allows access to all of the public, protected, or friend members of the base class.
- It is not a real object, so you cannot assign **MyBase** to a variable.

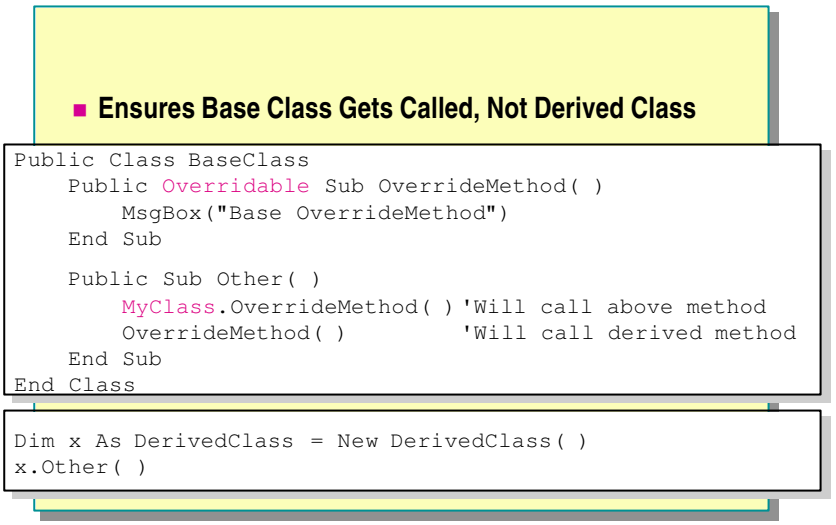
If a derived class is overriding a method from a base class but you still want to execute the code in the overridden method, you can use **MyBase**. This is a common practice for constructors and destructors. The following example shows how to use the **MyBase** keyword to execute a method as implemented in the base class:

```
Public Class DerivedClass
    Inherits BaseClass
    Public Sub New()
        MyBase.New()'Call the constructor of the base class
        intValue = 1
    End Sub
    Public Overrides Sub OverrideMethod()
        MsgBox("Derived OverrideMethod")
        MyBase.OverrideMethod() 'Call the original method
    End Sub
End Class
```

Using the MyClass Keyword

Topic Objective
To explain how to use the **MyClass** keyword.

Lead-in
It is important that you call the correct class in inheritance. The **MyClass** keyword can help you call the right one.



Delivery Tip
Point out the important note in the student notes regarding the differences between the **MyClass** and **Me** keywords.

You can use the **MyClass** keyword to ensure that a base class implementation of an overridable method is called, rather than that of a derived class. When using **MyClass**, you should be aware of some limitations:

- It allows access to all of the public, protected, or friend members of the deriving class.
- It is not a real object, so you cannot assign **MyClass** to a variable.

Example

The following example shows how to call a base class method from a derived class by using the **MyClass** keyword:

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

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

Output

The output from the execution of the preceding code is as follows:

```
Base OverrideMethod
```

Flow of Execution

The code in the example is executed as follows:

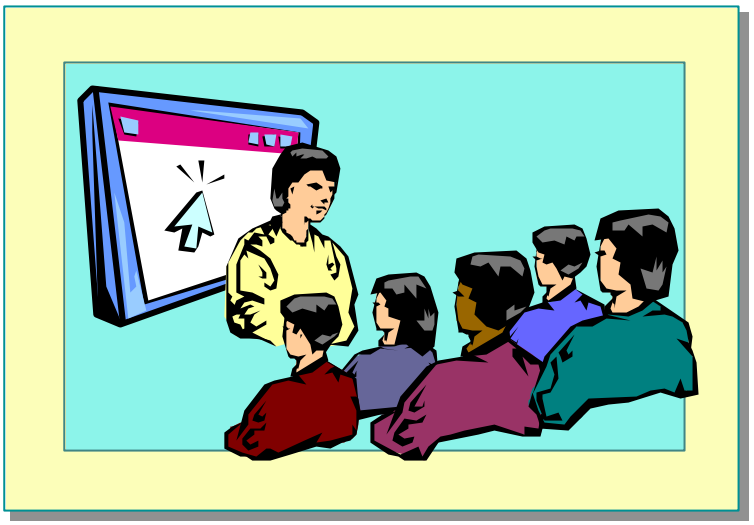
1. The **Other** method is called on the **DerivedClass** object, but because there is no implemented code in the derived class, the base class code is executed.
2. When the **MyClass.OverrideMethod** call is executed, the **OverrideMethod** subroutine is implemented in the base class.
3. Execution returns to the client code.

Important The **Me** keyword in previous versions of Visual Basic is not the same as the **MyClass** keyword. The **Me** keyword has the same effect as if no keyword were used. In the example in this topic, the derived class implementation would be executed even if **Me** were used within the base class.

Demonstration: Inheritance

Topic Objective
To demonstrate class inheritance.

Lead-in
This demonstration examines how to create and use class inheritance.



Delivery Tip
The step-by-step instructions for this demonstration are in the instructor notes for this module.

In this demonstration, you will see how to define a base class with a mixture of overridable and non-overridable methods. You will see how to derive a class that inherits from the base class, and how to use the **MyBase** keyword.

Interfaces

Topic Objective
To provide an overview of the topics covered in this lesson.

Lead-in
In this lesson, you will learn about interfaces and polymorphism.

- **Defining Interfaces**
- **Achieving Polymorphism**

Delivery Tip
This lesson assumes some knowledge about interfaces and polymorphism.

The topic is not covered extensively because interfaces and polymorphism are not new to Visual Basic .NET. You should ensure that students are aware of additional appropriate information.

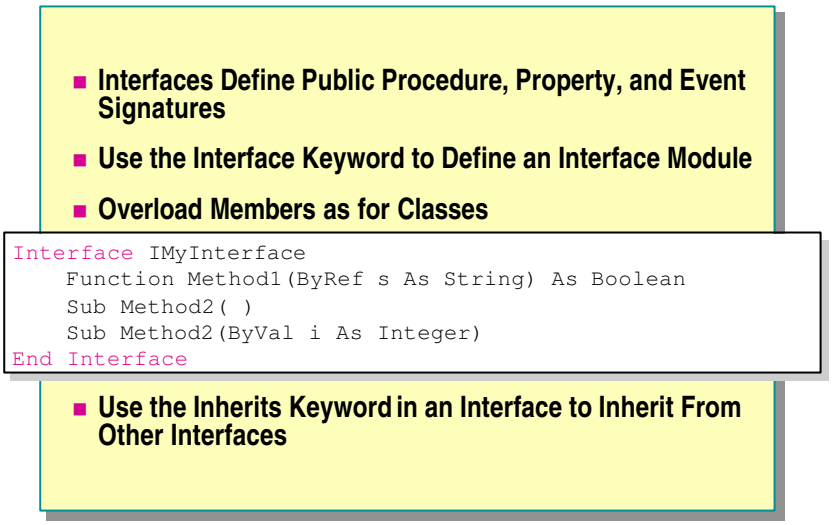
In this lesson, you will learn how to use interfaces to achieve polymorphism. After completing this lesson, you will be able to:

- Define an interface by using the **Interface** keyword.
- Add member signatures that define the properties, methods, and events that your interface supports.
- Implement an interface by using the **Implements** keyword.

Defining Interfaces

Topic Objective
To explain how to create interfaces in Visual Basic .NET.

Lead-in
Interfaces are a new and powerful feature of Visual Basic .NET.



An interface defines signatures for procedures, properties, and events but contains no implementation code. These signatures define the names of the members, the parameter details, and the return values. Interfaces form a binding contract between clients and a server. This contract enables a client application to ensure that a class will always support particular member signatures or functionality, and this aids in the versioning of components.

Delivery Tip
Check student understanding of the concept of interfaces. Give a brief review if necessary, with a simple example of interfaces (such as the **IPerson**, **Student**, **Employee** example used in the notes).

In Visual Basic 6.0, interfaces are created automatically whenever a public class is compiled as part of a COM component. This functionality works fine in most situations; however, you need to create a class in order to define an interface, which is not always necessary from a developer's perspective.

Visual Basic .NET introduces the **Interface** keyword, which allows you to explicitly create an interface without creating a class. Interfaces can be defined in the **Declarations** section of any module. This new approach creates a visible distinction between a class and an interface, and this makes the concept clearer for the developer.

You can use overloading when you define interfaces—just as you use it to define classes—to create multiple versions of a member signature with different parameters.

Example

The following example shows how to define an interface that includes three method signatures, two of which are overloaded:

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

An interface can also inherit another interface if you use the **Inherits** keyword before any member signatures are defined. If an interface is inherited from the above example, it will contain all of the base interface signatures, in addition to those defined in the new interface definition.

Achieving Polymorphism

Topic Objective
To explain how to use polymorphism in Visual Basic .NET.

Lead-in
Visual Basic .NET combines a traditional use of interfaces—to create polymorphism—with the new class inheritance features.

- **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

Polymorphism is achieved when multiple classes provide the same properties or methods and the calling code does not need to know what type of class the object is based on. This creates a form of reuse because you can write generic client code to handle multiple types of classes without knowing about the methods of each individual class. You can use two different approaches to achieve polymorphism in Visual Basic .NET: interfaces and inheritance.

Delivery Tip
The code shown in the student notes is demonstrated immediately after this topic.

Interfaces

In Visual Basic 6.0, you can achieve polymorphism by creating an abstract class—with the sole purpose of defining an interface—and then implementing that interface in other classes by using the **Implements** keyword. This approach allows multiple classes to share the same interface and allows classes to have multiple interfaces.

In Visual Basic .NET, you do not need abstract classes to achieve polymorphism. You can create interfaces explicitly by using a combination of the **Interface** and **Implements** keywords.

Example

The following example shows how to implement polymorphism in Visual Basic .NET. As you examine this code, note the following:

- The **IPerson** interface defines two member signatures: **LastName** and **Display**.
- The **Employee** class implements the **IPerson** interface and both of its members.
- By using the **Implements** keyword for each individual method, you can specify your own name for the method and it will still be executed if a client application requests the original name of the interface.

```
Interface IPerson              'IPerson interface definition
    Property LastName( ) As String
    Sub Display( )
End Interface

Class Employee                 'Employee class definition
    Implements IPerson         'Implements IPerson interface
    Private strName As String
    Private strCompany As String

    'This method is public but also implements IPerson.Display
    Public Sub Display( ) Implements IPerson.Display
        MsgBox(LastName & " " & Company,, "Employee")
    End Sub

    'This property is private but implements IPerson.LastName
    Private Property LastName( ) As String _
      Implements IPerson.LastName
        Get
            Return strName
        End Get
        Set (ByVal Value As String)
        ...
    End Property

    Public Property Company( ) As String
    ...
    End Property
End Class
```

Client Code Example

Delivery Tip
Point out that the **p.Display( )** and **emp.Display( )** methods are actually the same call to the **Employee** object, but by means of different interfaces.

The following code shows how the **Employee** class and **IPerson** interface could be used in a client application or within the same assembly as the class and interface definitions.

- An **Employee** object is instantiated and details specific to the employee, such as the **Company** property, are assigned.
- An **IPerson** interface variable is then assigned to the **Employee** object to access methods that can only be accessed through the **IPerson** interface, such as the **LastName** property.
- Both calls to the **Display** methods actually call the same code within the **Employee** class, as shown in the previous example.

```
Dim perPerson As IPerson, empEmployee As New Employee( )
empEmployee.Company = "Microsoft"
perPerson = empEmployee
perPerson.LastName = "Jones"
perPerson.Display( )'Call the display method on the interface
empEmployee.Display( ) 'Display method is defined as public
```

Inheritance

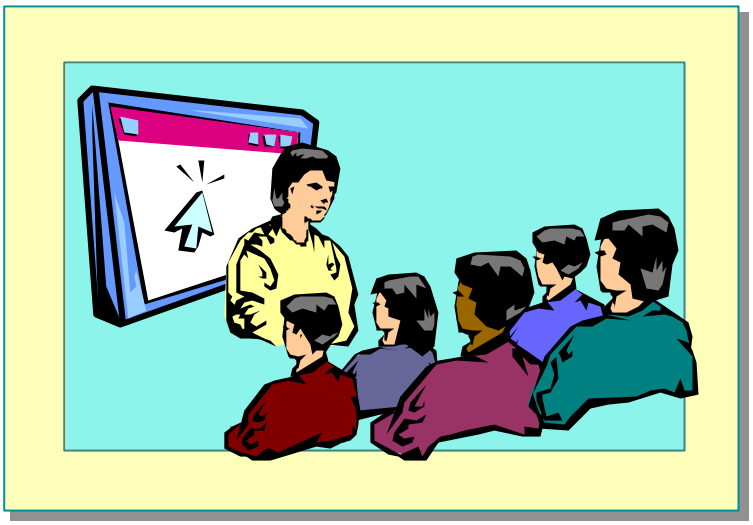
Another way to achieve polymorphism with Visual Basic .NET is to use class inheritance. A base class can be created that contains member signatures and that optionally contains implementation code that can be inherited in a derived class. The derived class must then override the individual methods with its own implementation code, achieving unique functionality while retaining a common method signature.

Note For more information about polymorphism, search for "Polymorphism" in the Visual Studio .NET documentation.

Demonstration: Interfaces and Polymorphism

Topic Objective
To demonstrate how to use interfaces to achieve polymorphism.

Lead-in
This demonstration shows how to use interfaces to achieve polymorphism.



Delivery Tip
The step-by-step instructions for this demonstration are in the instructor notes for this module.

In this demonstration, you will learn how to create an interface and implement it to achieve polymorphism in two separate classes.

◆ Working with Classes

Topic Objective
To provide an overview of the topics covered in this lesson.

Lead-in
This lesson examines how to work with classes in Visual Basic .NET.

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

In this lesson, you will learn how to work with classes. After completing this lesson, you will be able to:

- Use shared data members to share data across class instances.
- Use shared procedure members.
- Define and handle class events.
- Use delegates in event handling.
- Describe how structures differ from classes.

Using Shared Data Members

Topic Objective
To explain how to use shared data members to share data across class instances.

Lead-in
Shared data members allow data sharing for all instances of a class.

- **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 Functi on
End Class
```

```
SavingsAccount . InterestRate = 0.003
Di m acct 1 As New SavingsAccount ("Joe Howard", 10000)
MsgBox(acct1.CalculateInterest, , "Interest for " & acct1.Name)
```

Delivery Tip
This is an animated slide. It begins by showing the **Shared** example code. Click the slide to reveal an example of client code.

In Visual Basic 6.0, you can share data among objects by using a module file and a global variable. This approach works, but there is no direct link between the objects and the data in the module file, and the data is available for anyone to access.

In Visual Basic .NET, you can use shared data members to allow multiple instances of a class to refer to a single instance of a class-level variable. You can use shared members for counters or for any common data that is required by all instances of a class.

An advantage of shared data members is that they are directly linked to the class and can be declared as public or private. If you declare data members as public, they are accessible to any code that can access the class. If you declare the data members as private, provide public shared properties to access the private shared property.

The following example shows how you can use a shared data member to maintain interest rates for a savings account. The **InterestRate** data member of the **SavingsAccount** class can be set globally regardless of how many instances of the class are in use. This value is then used to calculate the interest on the current balance.

```
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
```

The following code shows how a client application can use the **SavingsAccount** class in the previous example. As you examine this code, note the following:

- The **InterestRate** can be set before and after any instances of the **SavingsAccount** object are created.
- Any changes to the **InterestRate** will be seen by all instances of the **SavingsAccount** class.


```
Sub Test( )
    SavingsAccount.InterestRate = 0.003

    Dim acct1 As New SavingsAccount("Joe Howard", 10000)
    Dim acct2 As New SavingsAccount("Arlene Huff", 5000)

    MsgBox(acct1.CalculateInterest, , "Interest for " & _
           acct1.Name)
    MsgBox(acct2.CalculateInterest, , "Interest for " & _
           acct2.Name)
End Sub
```

The following example shows how to implement a public shared property for a private shared data member:

```
Class SavingsAccount
    Private Shared InterestRate As Double

    Shared Property Rate( )
        Get
            Return InterestRate
        End Get
        Set(ByVal Value)
            InterestRate = Value
        End Set
    End Property
```

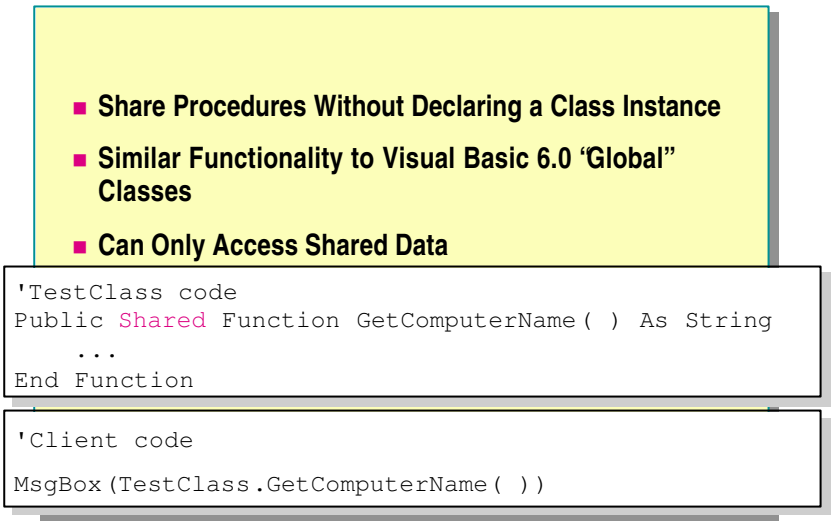
The following code shows how a client application can use the shared property in the previous example:

```
SavingsAccount.Rate = 0.003
```

Using Shared Procedure Members

Topic Objective
To explain shared procedure members.

Lead-in
You can create shared procedure members that allow you to call the method without creating an instance of the class.



Delivery Tip
Point out that this feature is often used for library routines.

You can use shared procedure members to design functions that can be called without creating an instance of the class. Shared procedures are particularly useful for creating library routines that other applications can easily access. This concept is similar to the **GlobalMultiUse** and **GlobalSingleUse** classes used in Visual Basic 6.0.

As described in the previous topic, shared members can only access data that is marked as **Shared**. For example, a shared method cannot access a module level variable that is marked as either **Dim**, **Private**, or **Public**.

Example

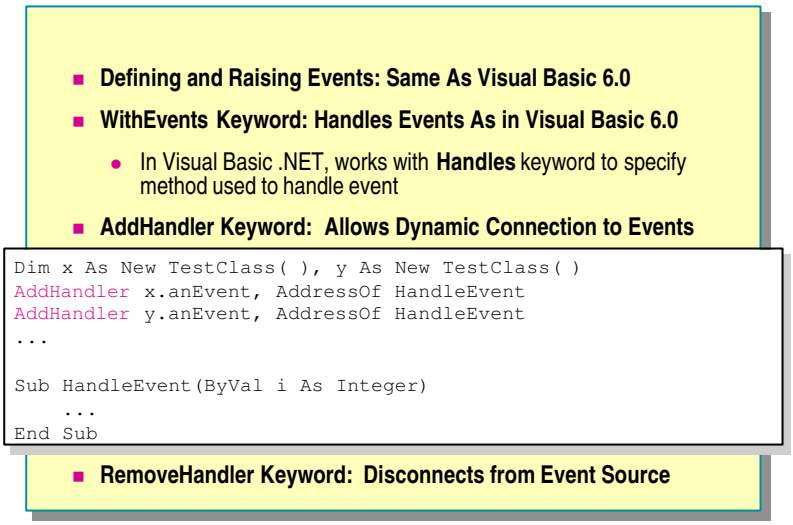
The following example shows how a commonly used function, such as **GetComputerName**, can be created as a shared procedure member so that a client application can easily use it. The client only needs to reference the method prefixed by the class name because no instance of the class is required.

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

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

Event Handling

Topic Objective
To introduce event-handling options that are new in Visual Basic .NET.

Lead-in
Visual Basic .NET provides some powerful new event-handling enhancements.



As a Visual Basic developer, you are familiar with creating events. However, Visual Basic .NET provides powerful new event handling features with the addition of the **Handles**, **AddHandler** and **RemoveHandler** keywords.

Defining and Raising Events

In Visual Basic .NET, you can define and raise events in the same way you do in Visual Basic 6.0, by using the **Event** and **RaiseEvent** keywords.

Example

The following example shows how to define and raise an event:

```
' TestClass code
Public Event anEvent(ByVal i As Integer)

Public Sub DoAction( )
    RaiseEvent anEvent(10)
End Sub
```


The WithEvents Keyword

Delivery Tip
The **Handles** keyword is shown in the **WithEvents** example in the student notes.

You can use the **WithEvents** keyword in the same way that you used it in Visual Basic 6.0. However, in Visual Basic .NET you also use the **Handles** keyword to specify which method will be used to handle an event. You can link an event to any handler, whether it is the default handler or your own method. This approach allows you to link multiple events with a single method handler, as long as the parameters match those of the events.

Example

The following example shows how you can use **WithEvents** in conjunction with the new **Handles** keyword to link an event with a handler.

```
'Client code
Dim WithEvents x As TestClass
Dim WithEvents y As TestClass

Private Sub Button1_Click(...) Handles Button1.Click
    x = New TestClass( )
    y = New TestClass( )
    x.DoAction( )
    y.DoAction( )
End Sub

Private Sub HandleEvent(ByVal x As Integer) _
  Handles x.anEvent, y.anEvent
...
End Sub
```

The AddHandler Keyword

The new **AddHandler** keyword allows you to dynamically connect to the events of an object and handle them in any chosen method. This has some advantages over the **WithEvents** keyword: the variable does not need to be declared at the module level, and you can point multiple events from the same object to a single handler method. You can also point events from multiple objects to the same handler method by using the **AddHandler** keyword.

Syntax

The syntax for **AddHandler** is shown below.

```
AddHandler object.EventName, AddressOf methodName
```

Example

The following example shows a single method handler called **HandleEvent** being used for two instances of **TestClass**:

```
Dim x As New TestClass( ), y As New TestClass( )

AddHandler x.anEvent, AddressOf HandleEvent
AddHandler y.anEvent, AddressOf HandleEvent
```

The RemoveHandler Keyword

The new **RemoveHandler** keyword disconnects your event handler from the object's events.

Syntax

The syntax for **RemoveHandler** is shown below.

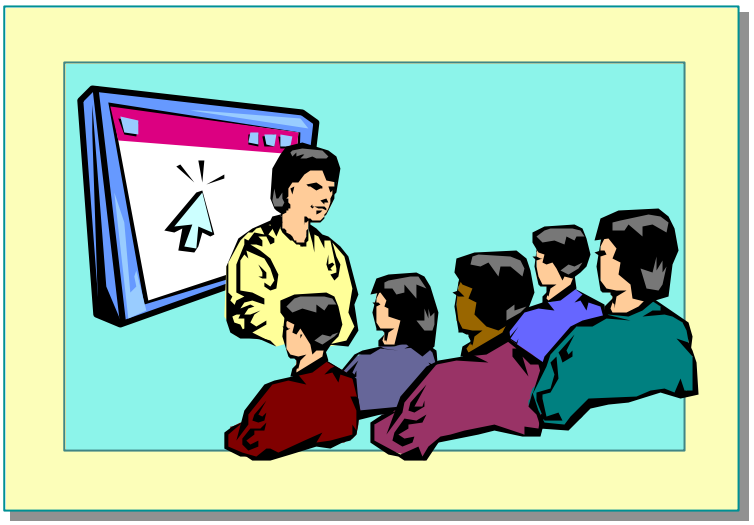
```
RemoveHandler object.EventName, AddressOf methodName
```

Note **AddressOf** creates a reference to a procedure that can be passed to appropriate methods. It was introduced in previous versions of Visual Basic. For more information about **AddressOf**, search for "AddressOf" in the Visual Studio .NET documentation.

Demonstration: Handling Events

Topic Objective
To demonstrate how to define, raise, and handle events.

Lead-in
Using events is a common requirement for class development.



Delivery Tip
The step-by-step instructions for this demonstration are in the instructor notes for this module.

In this demonstration, you will learn how to define and raise events in a class and how to handle them in client code.

What Are Delegates?

Topic Objective
To provide an introduction to the concept and use of delegates.

Lead-in
Delegates are a powerful new feature of Visual Basic .NET.

- **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

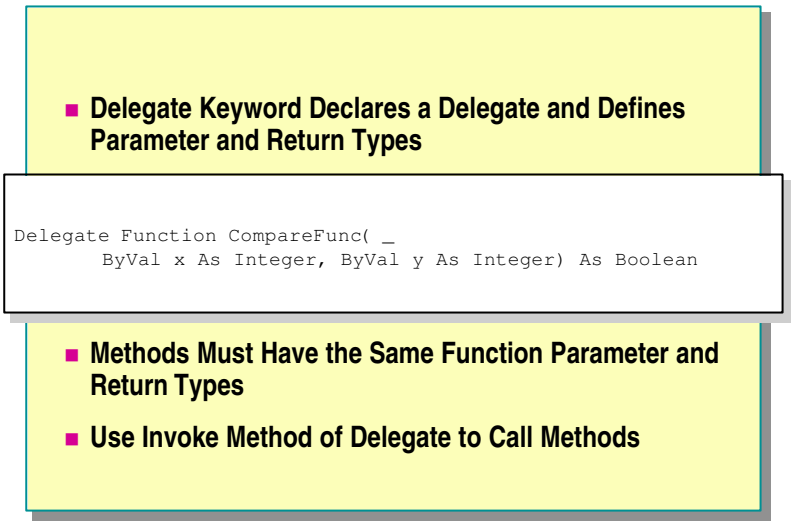
The common language runtime supports objects called *delegates* that can call the methods of other objects dynamically. Delegates are sometimes described as *type-safe function pointers* because they are similar to the function pointers used in other programming languages. Unlike function pointers, Visual Basic .NET delegates are a reference type based on the class **System.Delegate** and can reference both shared methods (methods that can be called without a specific instance of a class) and instance methods. Delegates provide the same flexibility as function pointers in Microsoft Visual C++® without the risk of corrupted memory because they are type-safe, secure, managed objects.

Delegates are useful when you need an intermediary between a calling procedure and the procedure being called. For example, you might want an object that raises events to be able to call different event handlers under different circumstances. Unfortunately, the object raising events cannot know ahead of time which event handler is handling a specific event. Visual Basic .NET allows you to dynamically associate event handlers with events by creating a delegate for you when you use the **AddHandler** statement. At run time, the delegate forwards calls to the appropriate event handler.

Using Delegates

Topic Objective
To provide an example of how to use delegates in event handling.

Lead-in
Let's look at an example of how you can use delegates in event handling.



You use the **Delegate** keyword to declare a delegate function signature that defines the parameter and return types. Only methods that have the same function parameter and return types can be used with a particular delegate object.

Example

To learn how delegates work, consider an example that shows how to declare a delegate function signature, create methods to accept the parameter types you have defined, and call the functions by using the delegate object. The final part of this example shows how to use the delegate to perform a bubble sort.

Declaring a Delegate Function Signature

Delivery Tip
This is an advanced feature of Visual Basic .NET. A simple example is provided in the notes. It is followed by a more complex example.

Recommend that the students look at the advanced example if they are interested in this topic. The example is also included in the Demo Code folder.

The following code shows how to create a delegate function called **CompareFunc**, which takes two **Integer** parameters and returns a **Boolean** value.

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

Creating Methods

After you create a delegate, you can then create methods that accept the same parameter types, as follows:

```
Function CompareAscending( _
        ByVal x As Integer, ByVal y As Integer) As Boolean
    Return (y > x)
End Function
Function CompareDescending( _
        ByVal x As Integer, ByVal y As Integer) As Boolean
    Return (x > y)
End Function
```

Calling Methods

After you create the necessary functions, you can write a procedure to call these two functions by using a delegate object as follows:

```
Sub SimpleTest( )
    Dim delDelegate As CompareFunc

    delDelegate = New CompareFunc(AddressOf CompareAscending)
    MsgBox(delDelegate.Invoke(1, 2))

    delDelegate = New CompareFunc(AddressOf CompareDescending)
    MsgBox(delDelegate.Invoke(1, 2))
End Sub
```

Performing a Bubble Sort by Using Delegates

Now that you have created a delegate and defined its methods, you can start using the delegate. A bubble sort routine is a good example of how you might use delegates. This type of sort routine starts at the top of a list and compares each item, moving it up the list if appropriate (bubbling it up), until the complete list is sorted. The following method takes a *sortType* parameter that will specify whether the sort should be ascending or descending. It also takes an array of **Integer** values to be sorted. The appropriate delegate object is created, depending on the order of the sort.

```
Sub BubbleSort(ByVal sortType As Integer, _
                    ByVal intArray( ) As Integer)
   Dim I, J, Value, Temp As Integer
   Dim delDelegate As CompareFunc

   If sortType = 1 Then    'Create the appropriate delegate
     delDelegate = New CompareFunc(AddressOf CompareAscending)
   Else
     delDelegate = New CompareFunc(AddressOf _
                                        CompareDescending)
   End If

   For I = 0 To Ubound(intArray)
       Value = intArray(I)
       For J = I + 1 To Ubound(intArray)
           If delDelegate.Invoke(intArray(J), Value) Then
               intArray(I) = intArray(J)
               intArray(J) = Value
               Value = intArray(I)
           End If
       Next J
   Next I
End Sub
```

The following code shows how to call the bubble sort procedure:

```
Sub TestSort( )
    Dim a( ) As Integer = {4, 2, 5, 1, 3}

    BubbleSort(1, a)     'Sort using 1 as ascending order
    MsgBox(a(0) & a(1) & a(2) & a(3) & a(4), , "Ascending")

    BubbleSort(2, a)     'Sort using 2 as descending order
    MsgBox(a(0) & a(1) & a(2) & a(3) & a(4), , "Descending")
End Sub
```

Comparing Classes to Structures

Topic Objective
To explain the differences between classes and structures.

Lead-in
Classes and structures have similar functionality, but classes provide a more powerful basis for object-oriented development.

Classes	Structures
Can define data members, properties, and methods	Can define data members, properties, and methods
Supports constructors and member initialization	No default constructor or member initialization
Support **Finalize** method	Do not support **Finalize** method; implement **IDisposable**
Extensible by inheritance	Do not support inheritance
Reference type	Value type

Classes and structures are similar in several ways: both can define data members, properties, and methods. However, classes provide some advanced features that developers can use.

	Classes	Structures
Initialization	Supports constructors and member initialization.	No default constructor and no initialization of members.
Finalize method	Support **Finalize** method.	Do not support **Finalize** method. You must manually release resources.. Structures can use interfaces, so they can use the **IDispose** interface to implement the **Dispose** method to release resources.
Inheritance	Extensible by inheritance.	Do not support inheritance.
Data type	Reference data type.	Value data type.
	When an object variable is passed to a function, the address reference of the data is passed rather than the data itself.	When a structure variable is passed to a function, the actual data must be copied to the function.
	Assigning one class variable to another points both variables to the same object. Any updates to either variable will therefore affect the other.	Assigning one structure variable to another creates an actual copy of the structure. Updates to one of the variables will therefore not affect the other.
		The difference in data type has a significant effect on application performance. A class with a lot of internal data will perform better than a large data structure under these conditions.

Lab 5.2: Inheriting the Package Class

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will create a base class and a class that derives from it.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Create base classes.
- Create derived classes that use inheritance.
- Use inherited classes from calling code.

Prerequisites

Before working on this lab, you should be familiar with inheritance in Visual Basic .NET.

Scenario

In this lab, you will continue creating the Cargo system. You will create the **Package** base class, the **SpecialPackage** derived class, and the test application. Some of the code has been created for you.

Starter and Solution Files

There are starter and solution files associated with this lab. The starter files are in the *install folder*\Labs\Lab052\Starter folder, and the solution files are in the *install folder*\Labs\Lab052\Solution folder.

Estimated time to complete this lab: 60 minutes

Exercise 1
Completing the SpecialPackage Class

In this exercise, you will examine the pre-written **Package** class and complete the partially written **SpecialPackage** class. You will inherit from the **Package** class, and override some of its methods.

↙ To open the starter project

1. Open Visual Studio .NET.
2. On the **File** menu, point to **Open**, and click **Project**. Set the folder location to *install folder*\Labs\Lab052\Starter, click **Lab052.sln**, and then click **Open**.

↙ To examine the Package class

1. Open the Package.vb class.
2. Examine the existing properties and methods.

   The **Package** class retrieves and stores information about a single package that will be delivered to a customer. It contains information about the package, including a description, size dimensions, instructions, weight, and value. These properties have been created for you.

   The **Package** class provides methods to simulate the creation, retrieval, and deletion of package information. These methods are marked as overridable for inheriting classes. Note that the **IsSpecialPackage** method is marked as shared so that it can be accessed without instantiating object variables. These methods have been created for you.

↙ To examine the SpecialPackage class

1. Open the SpecialPackage.vb class.
2. Examine the existing properties.

↙ To inherit from the Package class

1. At the top of the **SpecialPackage** class, locate the list of private variables. Insert the following line immediately before the variable declarations:

   ```
   Inherits Package
   ```

   This will create the relationship between the **Package** base class and the derived class **SpecialPackage**.

2. Add the following methods to the **SpecialPackage** class definition:

Method name	Type	Parameters
GetDetails	Public Overrides Sub	ByVal intID As Integer
CreatePackage	Public Overloads Function	ByVal intDeliveryID As Integer ByVal strDescription As String ByVal strDimensions As String ByVal strInstructions As String ByVal strWeight As String ByVal dblValue As Double ByVal blnOxygen As Boolean ByVal strTemperature As String ByVal strTimeLimit As String ByVal strExtra As String <RETURN VALUE> As Integer
DeletePackage	Public Overrides Sub	ByVal intID As Integer

↙ To implement the GetDetails method

1. Locate the **GetDetails** method declaration and add code to call the **MyBase.GetDetails** method, passing the *intID* as the parameter to retrieve the simulated **Package** details.
2. After the call to **MyBase.GetDetails**, assign the following values to the **SpecialPackage** properties to simulate a returned record from the database:

Property	Value
OxygenRequired	**True**
Temperature	**80**
TimeLimit	**5 hours**
ExtraInstructions	**Feed if time limit exceeded**

↙ To implement the CreatePackage method

1. Locate the **CreatePackage** method declaration, and add code to call the **MyBase.CreatePackage** method, passing in the following values as the parameters to create the simulated **Package** record.

Parameter Name	Value
intDeliveryID	*intDeliveryID*
strDescription	*strDescription*
strDimensions	*strDimensions*
strInstructions	*strInstructions*
strWeight	*strWeight*
dblValue	*dblValue*

2. After the call to **MyBase.CreatePackage**, assign the following values to the **SpecialPackage** properties to simulate the update to the database.

Property	Value
OxygenRequired	**blnOxygen**
Temperature	**strTemperature**
TimeLimit	**strTimeLimit**
ExtraInstructions	**strExtra**

3. After the property value assignments, use the **MsgBox** function to display the message "Special instructions added".
4. Return the PackageID as the return value of the **CreatePackage** method.

↙ To implement the DeletePackage method

1. Locate the **DeletePackage** method declaration, and insert code to use the **MsgBox** function to display the message "Deleting special package details" to simulate the deletion of the **SpecialPackage** database record.
2. After the displaying the message, call the **MyBase.DeletePackage** method, passing *intID* as the parameter, to simulate the deletion of the **Package** record.
3. Save the project.

Exercise 2
Retrieving Packages

In this exercise, you will write the calling code for the **Retrieve** button that calls either a **Package** or a **SpecialPackage** object. You will then test your code by entering some values into the Package form.

↙ To create the Retrieve button code

1. Open frmPackage in the Code Editor, and locate the **btnRetrieve_Click** event procedure.
2. Create an **If** statement that calls the **Package.IsSpecialPackage** shared function, passing in the **Text** property of txtID as the parameter. (Use the **CInt** function to convert the text value into an **Integer**.)

↙ To use a SpecialPackage object

1. In the true part of the **If** statement, declare and instantiate a **SpecialPackage** variable called **aSpecial**.
2. Set the **Checked** property of chkSpecial to **True**.
3. Call the **GetDetails** method of the **aSpecial** object, passing in the **Text** property of txtID as the parameter. (Use the **CInt** function to convert the text value into an **Integer**.)
4. Display the properties of the **aSpecial** object in the appropriate text boxes. Use the information in the following table to assign the text box values to the properties of the **aSpecial** object.

Control	Property of aSpecial
txtDeliveryID.Text	**DeliveryID**
txtDescription.Text	**Description**
txtDimensions.Text	**Dimensions**
txtInstructions.Text	**Instructions**
txtValue.Text	**Value**
txtWeight.Text	**Weight**
txtExtra.Text	**ExtraInstructions**
txtTemperature.Text	**Temperature**
txtTimeLimit.Text	**TimeLimit**
chkOxygen.Checked	**OxygenRequired**

↙ To use the Package object

1. In the false block of the **If** statement, set the **Checked** property of chkSpecial to **False**, and declare and instantiate a **Package** variable called **aPackage**.
2. Call the **GetDetails** method of the **aPackage** object, passing in the **Text** property of txtID as the parameter. (Use the **CInt** function to convert the text value into an **Integer**.)
3. Display the properties of the **aPackage** object in the appropriate textboxes. Use the information in the following table to assign the text box values to the properties of the **aPackage** object.

Control	Property of aPackage
txtDeliveryID.Text	**DeliveryID**
txtDescriptio n.Text	**Description**
txtDimensions.Text	**Dimensions**
txtInstructions.Text	**Instructions**
txtValue.Text	**Value**
txtWeight.Text	**Weight**
txtExtra.Text	**“ ”**
txtTemperature.Text	**“ ”**
txtTimeLimit.Text	**“ ”**
chkOxygen.Checked	**False**

4. Save the project.

↙ To test the Re trieve button code

1. Set a breakpoint on the first line of the **btnRetrieve_Click** procedure. From the **Debug** menu, click **Start**.
2. Enter the value **18** in the **PackageID** box, click the **Retrieve** button, and then step through the procedure.
3. Confirm that your code retrieves the package information and displays it correctly in the text boxes.
4. Click the **Clear Data** button to reset the information.
5. Enter the value **33** in the **PackageID** box, click the **Retrieve** button, and step through the procedure.
6. Confirm that your code retrieves the special package information and displays it correctly in the text boxes.
7. Click the **Close** button to quit the application. Remove the breakpoint on **btnRetrieve_Click**.

Exercise 3
Creating Packages

In this exercise, you will write the calling code for the **New** button that creates either a **Package** or **SpecialPackage** object. You will then test your code by entering some values into the Package form.

↙ To create the New Package button code

1. Locate the **btnNew_Click** event procedure.
2. Create an **If** statement that checks the **Checked** property of the chkSpecial check box.

↙ To create a Package object

1. In the false part of the **If** statement, declare and instantiate a **Package** variable called **aPackage**.
2. Call the **CreatePackage** method of the **aPackage** object, passing in the following values as parameters.

Parameter	TextBox
iDelivery	CInt(txtDeliveryID.Text)
strDescription	txtDescription.Text
strDimensions	txtDimensions.Text
strInstructions	txtInstructions.Text
strWeight	txtWeight.Text
dblValue	CDbl(txtValue.Text)

3. Store the return of the **CreatePackage** method in the **Text** property of the txtID box. (Use the **CStr** function to convert the **Integer** to a **String**.)

↙ To create a SpecialPackage object

1. In the true part of the **If** statement, declare and instantiate a **SpecialPackage** variable called **aPackage**.
2. Call the overloaded **CreatePackage** method of the **aPackage** object, passing in the following values as parameters.

Parameter	Value
iDelivery	Cint(txtDeliveryID.Text)
strDescription	txtDescription.Text
strDimensions	txtDimensions.Text
strInstructions	txtInstructions.Text
strWeight	txtWeight.Text
dblValue	CDbl(txtValue.Text)
blnOxygen	chkOxygen.Checked
strTemperature	txtTemperature.Text
strTimeLimit	txtTimeLimit.Text
strExtra	txtExtra.Text

3. Store the return of the overloaded **CreatePackage** method in the **Text** property of the txtID box. (Use the **CStr** function to convert the **Integer** to a **String**.)
4. Save the project.

↙ To test the standard Package code

1. Set a breakpoint on the first line of the **btnNew_Click** procedure.
2. On the **Debug** menu, click **Start**.
3. Enter the following values.

Control	Value
DeliveryID	**11**
Description	**Software**
Instructions	**None**
Dimensions	**NA**
Weight	**NA**
Value	**50**

4. Click the **New** button, and step through the procedure.
5. Confirm that the code correctly passes the values to the package class.
6. Click the **Clear Data** button.

↙ To test the SpecialPackage code

1. Enter the following values.

Control	Value
DeliveryID	**43**
Description	**Heart Transplant**
Instructions	**Deliver to Joe Howard**
Dimensions	**NA**
Weight	**NA**
Value	**0**
SpecialPackage checkbox	**Checked**
ExtraInstructions	**Speed is essential**
OxygenRequired checkbox	**Unchecked**
Temperature	**20**
TimeLimit	**2 hours**

2. Click the **New** button, and debug the procedure.
3. Confirm that the code passes the values to the **SpecialPackage** class correctly.
4. Click the **Close** button to quit the application, and remove the breakpoint on **btnNew_Click**.

Exercise 4
Deleting Packages

In this exercise, you will write the calling code for the **Delete** button that deletes either a **Package** or **SpecialPackage** object. You will then test your code by entering some values into the Package form.

↙ To create the Delete button code

1. Locate the **btnDelete_Click** event procedure.
2. Create an **If** statement that calls the **Package.IsSpecialPackage** shared function, passing in the **Text** property of txtID as the parameter. (Use the **CInt** function to convert the text value into an **Integer**.)

↙ To delete a SpecialPackage object

1. In the true part of the **If** statement, declare and instantiate a **SpecialPackage** variable called **aSpecial**.
2. Call the **DeletePackage** method of the **aSpecial** object, passing in the **Text** property of txtID as the parameter. (Use the **CInt** function to convert the text value into an **Integer**.)

↙ To delete a Package object

1. In the false part of the **If** statement, declare and instantiate a **Package** variable called **aPackage**.
2. Call the **DeletePackage** method of the **aPackage** object, passing in the **Text** property of txtID as the parameter. (Use the **CInt** function to convert the text value into an **Integer**.)
3. Save the project.

↙ To test the Delete button code

1. Set a breakpoint on the first line of the **btnDelete_Click** procedure, and, on the **Debug** menu, click **Start**.
2. Enter the value **18** in the PackageID text box, and click the **Delete** button to debug the procedure.
3. Confirm that your code simulates the deletion of the **Package** object.
4. Click the **Clear Data** button to reset the information.
5. Enter the value **33** in the PackageID text box, and click the **Delete** button to debug the procedure.
6. Confirm that your code simulates the deletion of the **SpecialPackage** object.
7. Click the **Close** button to quit the application, and remove the breakpoint on **btnDelete_Click**.
8. Close Visual Studio .NET.

Review

Topic Objective
To reinforce module objectives by reviewing key points.

Lead-in
The review questions cover some of the key concepts taught in the module.

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

---

1. Create code that defines multiple constructors for a **Person** class. The first constructor will not take any arguments. The second will take two string values: **FirstName** and **LastName**.

```
Class Person
    Sub New( )
        'Default constructor
    End Sub
    Sub New(ByVal FirstName As String, _
        ByVal LastName As String)
        'Second constructor
    End Sub
End Class
```


2. Garbage collection occurs immediately after all references to an object are removed. True or false? If false, explain why.

   False. Garbage collection may happen at any time after all object references have been removed.

3. Describe the functionality of the **MyBase** keyword.

   MyBase is used in a derived class to access methods and properties in the immediate base class.

4. What is a potential problem that may result from the following class code sample? How can you rewrite the code to resolve the problem?

```
Class Person
    Private Sub Save( )
        'Save the local data in a database
    End Sub

    Sub Dispose( )
        Save( )
    End Sub

    Protected Overrides Sub Finalize( )
        Dispose( )
        MyBase.Finalize( )
    End Sub
End Class
```

The Dispose method can be called directly from a client and might be called again when the object is destroyed by garbage collection. This would result in the Save method being called twice, which may create data inconsistencies.

To avoid this, use the SuppressFinalize method of the GC class to stop the Finalize method being called after Dispose. Add the line "GC.SuppressFinalize()" in the Dispose method after the Save line as follows):

```
Sub Dispose()
    Save()
    GC.SuppressFinalize()
End Sub
```

5. You can create an interface explicitly in Visual Basic .NET. True or false? If false, explain why.

   True. You can create an interface explicitly by using the Interface…End Interface statement block.

6. Will the following code compile correctly? If not, why?

```
Class Person
    Event NameChanged( )
    Private strName As String

    Sub ChangeName(ByVal strNewName As String)
        strName = strNewName
        RaiseEvent NameChanged( )
    End Sub
End Class

Module TestCode
 Sub Main( )
    Dim x As New Person( )
    AddHandler x.NameChanged, AddressOf HandleIt
    x.ChangeName("Jeff")
 End Sub

 Sub HandleIt(ByVal strValue As String)
    MsgBox(strValue)
 End Sub
End Module
```

   The code will not compile correctly because the signature of the event does not match the signature of the delegate in the AddHandler statement.