msdn® training

Module 9: Developing Components 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:
90 Minutes

This module provides students with the knowledge required to create and use different types of components by using Microsoft® Visual Basic® .NET.

In the first lab, students will create a serviced component that retrieves customer information based on the customer's e-mail address and password. The assembly is installed in the Global Assembly Cache (GAC) to avoid a potential bug in Beta 2. This bug does not allow server-activated applications to be called successfully without being installed in the GAC. Students will register this component with Component Services and set constructor string properties of the component by using the Component Services console. They will then test the serviced component with a simple Windows-based application.

In the second lab, students will create a Web user control that requests logon information from a customer. They will place the user control on a Web Form and use the serviced component that they created in the first lab to retrieve the customer information so that it can be displayed on a welcome screen.

After completing this module, students will be able to:

- Describe the different types of components that they can create in Visual Basic .NET.
- Create components that can be used by managed and unmanaged client applications.
- Create serviced components.
- Create component classes.
- Create Microsoft Windows® Forms controls.
- Create Web Forms user controls.
- Use threading to create multithreaded applications.

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_09.ppt
- Module 9, "Developing Components in Visual Basic .NET"
- Lab 9.1, Creating a Serviced Component
- Lab 9.2, Creating a Web Forms User Control

Preparation Tasks

To prepare for this module:

- Read all of the materials for this module.
- 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 a Serviced Component

↙ To examine the object pooling application

1. Open Microsoft Visual Studio® .NET.
2. Open the ObjectPoolingComponent.sln project in the *install folder*\DemoCode\Mod09\ObjectPoolingComponent folder.
3. View the code for the **Pooling** class, particularly noting the **Imports** statement, class-level attributes, and the purpose of each class member.
4. View the code for the **NoPooling** class, and point out that the class is almost identical to the **Pooling** class, except that it does not use object pooling.
5. View the code for the **Report** class, and point out the **GetReport** method and the **GetSharedProperty** method of the **modCommon** module.
6. View the AssemblyInfo.vb file, pointing out the first three assembly attributes that refer to serviced component applications.

↙ To create the serviced component application

1. Build the project, and then quit Visual Studio .NET.
2. Open Windows Explorer, and then move to the *install folder*\DemoCode\Mod09\ObjectPoolingComponent\bin folder.
3. To avoid a bug in Beta 2, you need to register the assembly in the GAC to create a server-activated application. Open a Command window, type **"%ProgramFiles%\Microsoft.NET\FrameworkSDK\Bin\gacutil.exe" -i** and then drag the objPooling.dll file from Windows Explorer to the command line.
4. Execute the command. This should display a successful message in the Command window.
5. In the Command window, type **"%Windir%\Microsoft.NET\Framework\v1.0.2914\Regsvcs.exe"** and then drag the objPooling.dll file from Windows Explorer to the command line.
6. Execute the command. This should display a successful registration message in the Command window.

↙ To examine the serviced component application

1. Open the Component Services console, and analyze the **Object Pooling** application.
2. View the properties for the **NoPool** and **Pool** components, pointing out the **Object Pooling** settings on the **Activation** tab of each component.

↙ To view the test harness

1. Open Visual Studio .NET.
2. Open the TestPooling.sln project in the *install folder*\DemoCode\Mod09\ ObjectPoolingComponent\TestPooling VB.NET folder.
3. Add a project reference to *install folder*\DemoCode\Mod09\ ObjectPoolingComponent\bin\objPooling.dll.
4. View the form code, examining each method.

↙ To test the component

1. Run the project.
2. Create pooled objects and step through the code, explaining the messages that appear.
3. Create unpooled objects and step through the code, explaining the messages that appear.
4. Quit the application.
5. Run the project again, and show that this time there are no new objects created.
6. Quit the application, and then quit Visual Studio .NET.
7. If you have time, you can also show students the Visual Basic 6.0 test harness in the *install folder*\DemoCode\Mod09\ObjectPoolingComponent\ TestPooling VB6 folder. Note that you must shut down the Component Services application before running the Visual Basic 6.0 test harness to clean up the object pool used by the previous test harness application. You will also notice that the messages displayed while you debug the code may differ from those seen in Visual Basic .NET. This is not an error; Visual Basic 6.0 creates and uses objects differently than Visual Basic .NET.

Important If you have previously run this demonstration on the same machine, you may find that the serviced component is already installed. Remove the **Object Pooling** application from the Component Services console before re-running this demonstration.

Creating a Stopwatch Component

↙ To examine the Stopwatch component class

1. Open Visual Studio .NET.
2. Open the ComponentClasses.sln project in the *install folder*\ DemoCode\Mod09\Stopwatch\Starter folder.
3. View the design window for the **Stopwatch** component class and point out the localTimer control and its properties.
4. View the code for the **Stopwatch** component class, and explain each member of the class. Specifically point out the attributes used in the property definitions.

↙ To create a Toolbox icon for the component

1. Modify the class definition to read as follows:

```
<ToolboxBitmap("")> _
Public Class Stopwatch
```

2. In Solution Explorer, drag the TIMER01.ICO file and place it between the string quotes in the `ToolboxBitmap("")` code. Point out that adding the bitmap as an assembly resource may be a better approach as it will not rely on the icon file being available in the correct location. However, for this demonstration, this approach is acceptable.

↙ To build the component

1. Build the project.
2. Close the project.

↙ To modify the test harness

1. Open the TestComponentClasses.sln project in the *install folder*\DemoCode\Mod09\Stopwatch\Starter\TestStopwatch folder.
2. On the Toolbox, click the **General** tab.
3. On the **Tools** menu, click **Customize Toolbox**, and then click the **.NET Framework Components** tab.
4. Click **Browse** to browse for ComponentClasses.dll in the *install folder*\DemoCode\Mod09\Stopwatch\Starter\bin folder, and then click **Open**. Click the **Stopwatch** component, and then click **OK**.
5. In the design window, open Form1, and then drag the **Stopwatch** component from the Toolbox to the form.
6. In the Properties window for the component, change the name of the component to **sWatch** and set the **EnabledEvents** property to **True**. Point out the property description provided by the **Description** attribute.
7. Examine the code in the form.

↙ To test the component

1. Run the project, ensuring that the Output window is visible in the background.
2. Click **Start Stopwatch**, and point out the events being displayed in the Output window. Click **Tick Events** to turn off the events.
3. Click **Stop Stopwatch** to display the amount of time that has passed since the **Start** method was called on the **Stopwatch** component.
4. Quit the application, and then quit Visual Studio .NET.

Important If you have previously run this demonstration on the same computer, you may find that the **Stopwatch** component is already available in the Toolbox. To ensure that the demonstration functions correctly, reset the Toolbox by using the **Customize Toolbox** dialog box.

Creating an Enhanced TextBox

↙ To view the code

1. Open Visual Studio .NET.
2. Open the MyControls.sln project in the *install folder*\DemoCode\Mod09\UserTextBox folder.
3. View the code for the **MyTextBox** class, and examine all members of the class.
4. Build the project, and then close the project.

↙ To create the test harness

1. Open the TestControl.sln project in the *install folder*\DemoCode\Mod09\UserTextBox\TestControl folder.
2. On the Toolbox, click the **General** tab.
3. On the **Tools** menu, click **Customize Toolbox**. In the **Customize Toolbox** dialog box, click the **.NET Framework Components** tab.
4. Click the **Browse** button to browse for MyControls.dll in the *install folder*\DemoCode\Mod09\UserTextBox\bin folder, and then click **Open**. Select the **MyTextBox** control, and then click **OK**.
5. Display the test form if it is not already displayed.
6. From the Toolbox, drag **MyTextBox** onto the form to create an instance of the **MyTextBox** control.
7. Rename the control **myTB**, and then position it next to the MyTextBox label. Set the **Text** property of the control to **zero**.
8. In the **Undo** button's Click event handler, uncomment the **myTB.Undo** statement.

↙ To test the control

1. Run the project.
2. Sequentially change the text value for each text box to the following values.

Control	Text value
TextBox	**One**
MyTextBox	**One**
TextBox	**Two**
MyTextBox	**Two**
TextBox	**Three**
MyTextBox	**Three**

3. Click the **Undo** button four times, and view the changes in each text box.
4. Close the form and quit Visual Studio .NET.

Creating a Simple Web Forms User Control

↙ To create the ASP .NET Web application

1. Open Visual Studio .NET.
2. Create a new ASP .NET Web application called **SimpleUserControl** in the http://localhost/2373/DemoCode/Mod09/SimpleUserControl folder.

↙ To create the user control

1. On the **Project** menu, click **Add Web User Control**, and rename the control **SimpleControl.ascx**.
2. From the Toolbox, drag a **Label** control and a **TextBox** control to the user control design window.
3. In the Code Editor for the user control, add the following properties, which correspond to the internal control values.

Public property	Corresponding control value
TextValue( ) As String	**TextBox1.Text**
LabelValue( ) As String	**Label1.Text**

4. Save the project.

↙ To use the user control

1. Open WebForm1.aspx in the Design view, and then drag the SimpleControl.ascx file from Solution Explorer to the top left corner of WebForm1.
2. In the HTML view of WebForm1.aspx, locate the following code:

```
<uc1:SimpleControl id="SimpleControl1" runat="server">
```

3. Add the following attribute to the tag to set the **LabelValue** property of the control:

```
LabelValue="Enter a value:"
```

4. Return to the Design view, drag a **Button** control from the Toolbox to the Web Form, and place it well below the **SimpleControl**. Change the following properties of the **Button** control to the following values.

Button property	Value
Text	**Get Value**
(ID)	**btnGet**

5. In the Code Editor for WebForm1, add the following variable declaration immediately after the button declaration:

```
Protected WithEvents SimpleControl1 As SimpleControl
```

6. Create a **Click** event handler for **btnGet** and enter the following code:

```
Response.Write(SimpleControl1.TextValue)
```

7. Save the project.

↙ To test the control

1. Run the project.
2. Type a value into the text box, and then press the **Get Value** button to display the result.
3. Quit the application and Visual Studio .NET.

Using the SyncLock Statement

↙ To examine the object pooling application

1. Open Visual Studio .NET.
2. Open the ThreadingDemo.sln project in the *install folder*\DemoCode\Mod09\ThreadingDemo folder.
3. View the code for the frmThreading form, briefly explaining each member.
4. View the code for the **ThreadObj** class, briefly explaining each member.

↙ To test the application

1. Run the project.
2. Click the **Without SyncLock** button and observe that the results do not match the expected results.
3. Click the **With SyncLock** button, observe that the results correctly match those expected, and then quit the application.
4. Re-examine the **WithSyncLock** code within the **ThreadObj** class, clarifying the use of the **SyncLock** statement that produces the correct results.
5. Quit Visual Studio .NET.

Module Strategy

Use the following strategy to present this module:

- Components Overview

  This lesson provides an overview of creating components with Visual Basic .NET, and of how they can work with both managed and unmanaged clients.

  Many students will be interested in how a Visual Basic 6.0–based client application can use a Visual Basic .NET component, so spend some time on the topic of using components in unmanaged client applications. Strong-named assemblies are covered in Module 10, "Deploying Applications," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease),* but a simple explanation will be required in this lesson.

  The lesson ends with an overview of Microsoft .NET Remoting. Do not discuss this topic in detail. Instead, refer students to the Visual Studio .NET documentation or other to courses, such as Module 13, " Remoting and Web Services," in Course 2349A, *Programming the .NET Framework with C# (Prerelease)*.

- Creating Serviced Components

  This lesson relies on the students' knowledge of MTS or Component Services. Ensure that all students have some awareness of these technologies, and briefly review the overall purpose of serviced components, such as providing predefined functionality for developers to use.

  You will then examine several aspects of serviced components, including transactions, object pooling, constructor strings, and security. Most of these should not be new to MTS or Component Services developers, but the way that the information is specified has changed. Object pooling is a significant enhancement for Visual Basic, and will be new to many students because previous versions of Visual Basic could not take advantage of this technique. Use the demonstration at the end of the lesson to reinforce this concept.

  End the lesson with an overview of other services provided by Component Services and of how to configure an assembly to run as a serviced application. This information will minimize the amount of administration required when installing Visual Basic .NET serviced applications.

  Note that because of a bug with Beta 2, attempting to use a Component Services application as **Server** activation causes an error unless the assembly is installed in the GAC. The demonstrations and labs have been modified to install the assemblies in the GAC to avoid this error.

- Creating Component Classes

  This lesson examines component classes and their uses within a Visual Basic .NET–based application. Students will learn about the architecture of a component class and how to create one. Although this is a small lesson, make sure students recognize the benefits of these types of components.

- Creating Windows Forms Controls

  This lesson relies on students having created Windows user controls in previous versions of Visual Basic. Discuss some of the ways user controls can be created and how attributes can be specified to assist other developers who use the user controls.

  There are other ways to create controls (such as creating controls from scratch) that are not covered because of time constraints. Direct students to the product documentation for further information regarding these topics.

- Creating Web Forms Controls

  This lesson examines how to create Web Forms user controls within an ASP .NET Web application. The lesson is quite short as the creation of these controls is very similar to the creation of Web Forms and Windows Forms user controls.

  Point out to students that they can also create user controls based on Web Forms simply by copying the controls from the Web Form to the Web user control. There are other types of Web user controls called Custom controls that are not covered in this course. Refer students to the online help documentation for more information.

- Threading

  This lesson examines basic concepts of using threading in a Visual Basic .NET–based application. Most Visual Basic students are not familiar with threads, so be sure to explain the basic concepts first. Discuss the advantages of using threading and how to create and use threads in a Visual Basic .NET–based application.

  The lesson ends by raising some of the issues that students must be aware of when using threads. Be sure to point out that threading is a powerful yet potentially dangerous technique, and that students can choose whether or not to use it in their applications.

Overview

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

Lead-in
In this module, you will learn how to create components in Visual Basic .NET.

- **Components Overview**
- **Creating Serviced Components**
- **Creating Component Classes**
- **Creating Windows Forms Controls**
- **Creating Web Forms User Controls**
- **Threading**

As a Microsoft® Visual Basic® developer, you probably already know how to develop and use components in your applications. In Visual Basic .NET version 7.0, you can use the new design-time features to easily create components and extend their functionality.

After completing this module, you will be able to:

- Describe the different types of components that you can create in Visual Basic .NET.
- Create components that can be used by managed and unmanaged client applications.
- Create serviced components.
- Create component classes.
- Create Microsoft Windows® Forms controls.
- Create Web user controls.
- Use threading to create multithreaded applications.

◆ Components Overview

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

Lead-in
This lesson explains the types of components that you can create in a Visual Basic .NET- based application and how you can make them visible to unmanaged client applications. It also provides an overview of .NET Remoting for component communication.

- **Types of Components**
- **Using Modules As Components**
- **Using Classes As Components**
- **Using Components in Unmanaged Client Applications**
- **.NET Remoting Overview**

In Visual Basic .NET, you can create several types of components that are accessible from both managed client applications (those built on the services of the Microsoft .NET Framework common language runtime) and unmanaged client applications (for example, client applications created in Visual Basic 6.0).

After you complete this lesson, you will be able to:

- Describe the types of components that you can create in Visual Basic .NET.
- Use modules and classes as components.
- Use Visual Basic .NET–based components in unmanaged environments.
- Explain the key concepts of .NET Remoting.

Types of Components

Topic Objective
To explain the different types of components that you can create in Visual Basic .NET.

Lead-in
You can create several types of components in a Visual Basic .NET- based application.

- **Structures**
- **Modules**
- **Classes**
- **Component Classes**
- **Serviced Components**
- **User Controls**
  - Windows Forms user controls
  - Web Forms user controls

Delivery Tip
Point out that this module focuses on how to create and use component classes, serviced components, and user controls. The other component types are mentioned for reference purposes.

In Visual Basic .NET, you can create several different types of components, including:

- Structures
- Modules
- Classes
- Component classes
- Serviced components
- User controls

Structures

You can use structures as components by declaring them as public when you define them. Structures support many features of classes, including properties, methods, and events, but are value types, so memory management is handled more efficiently. Structures do not support inheritance.

Modules

You can use modules as components by declaring them as public when you define them. Declaring modules as public allows you to create code libraries that contain routines that are useful to multiple applications. You can also use modules to create reusable functions that do not apply to a particular component, class, or structure.

If you have used the **GlobalMultiUse** or **GlobalSingleUse** classes in previous versions of Visual Basic, the concept of a code library is not new to you. These classes provide the same functionality in Visual Basic .NET; the client code does not need to qualify these classes by the class name to call the functions.

Classes

You can use classes as components by declaring them as public within an assembly. You can use public classes from any .NET-based client application by adding a reference to the component assembly. You can extend the functionality of classes through mechanisms such as properties, methods, and events. Classes are also extensible through inheritance, which allows applications to reuse existing logic from these components.

Component Classes

A class becomes a component when it conforms to a standard for component interaction. This standard is provided through the **IComponent** interface. Any class that implements the **IComponent** interface is a component. Component classes allow you to open your class in a visual designer, and they allow your class to be sited onto other visual designers.

Serviced Components

Serviced components are derived directly or indirectly from the **System.EnterpriseServices.ServicedComponent** class. Classes configured in this manner are hosted by a Component Services application and can automatically use the services provided by Component Services.

User Controls

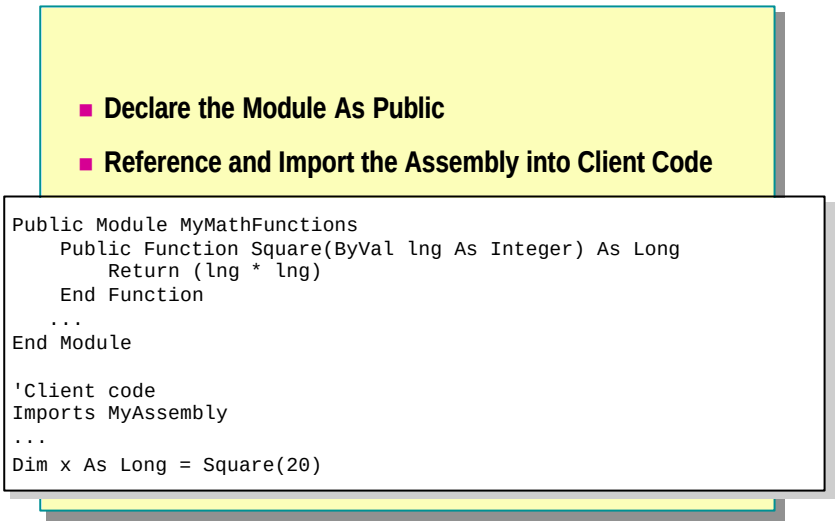
User controls are components that are created by a developer to be contained within Windows Forms or Web Forms. Each user control has its own set of properties, methods, and events that make it suitable for a particular purpose. You can manipulate user controls in the Windows Forms and Web Forms designers and write code to add user controls dynamically at run time, just as you can for the controls provided as part of the .NET Framework.

Note In this module, you will learn how to create and use component classes, serviced components, and user controls. For more information about structures, modules, and classes, see Module 5, "Object-Oriented Programming in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

Using Modules As Components

Topic Objective
To explain how to use modules as components.

Lead-in
In Visual Basic .NET, you can use modules as components outside of the assembly in which they are defined.



In Visual Basic .NET, you can use modules as components outside of the assembly in which they are defined. To make this possible, declare the module as public when you define it. You then need to create a reference in the client assembly to the component assembly and use the **Imports** statement to allow access to the module methods.

The following example shows how to create a public module named *MyMathFunctions* that defines the function **Square**. This module is defined within the *MyAssembly* assembly. The module can then be used as a component in client code, as shown in the second part of the example.

```
Public Module MyMathFunctions
    Public Function Square(ByVal lng As Long) As Long
        Return (lng * lng)
    End Function
   ...
End Module

'Client code
Imports MyAssembly
...
Dim x As Long = Square(20)
```


Note For more information about assemblies, see Module 10, " Deploying Applications," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*. For the purposes of this module, you can think of them as similar to Visual Basic 6.0 Microsoft ActiveX® dynamic-link libraries (DLLs).

Using Classes As Components

Topic Objective
To explain how to use classes as components.

Lead-in
In Visual Basic .NET, you can use classes as components.

- **Declare the Class As Public**
- **Reference and Import the Assembly into Client Code**

```
Public Class Account
    Public Sub Debit(ByVal AccountId As Long, Amount As Double)
        'Perform debit action
    End Sub
    Public Sub Credit(ByVal AccountId As Long, Amount As Double)
        'Perform credit action
    End Sub
End Class

'Client code
Imports MyAssembly
Dim x As New Account( )
x.Debit(1021, 1000)
```

You can use classes as components outside of the assembly in which they are defined by marking the class as public. You then reference the component assembly from the client assembly, and use the **Imports** statement to allow direct access to the class.

The following example shows how to create a public class called **Account** that defines the **Debit** and **Credit** methods. This class is defined in the **MyAssembly** assembly. A separate client assembly then references the assembly, and the class can then be used to created object instances.

```
Public Class Account
  Public Sub Debit(ByVal AccountId As Long, Amount As Double)
      'Perform debit action
  End Sub
  Public Sub Credit(ByVal AccountId As Long, Amount As Double)
      'Perform credit action
  End Sub
End Class

'Client code
Imports MyAssembly
Dim x As New Account( )
x.Debit(1021, 1000)
```

Using Components in Unmanaged Client Applications

Topic Objective
To explain how to create components that can be used by unmanaged client applications, such as Visual Basic 6.0-based clients.

Lead-in
You can use COM to make all Visual Basic .NET components accessible from unmanaged clients, if you follow some simple steps.

- **Setting Assembly Properties**
  - Generate a strong name
  - Select **Register for COM Interop** in **Build** options
- **Exposing Class Members to COM and Component Services**
  - Define and implement interfaces
  - Use the **ClassInterface** attribute with **AutoDual** value
  - Use the **COMClass** attribute

Delivery Tip
Remind students that assemblies and strong names will be covered in Module 10, "Deploying Applications," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

You can create Visual Basic .NET components that can be used by unmanaged client applications. This interoperability allows you to use Component Services features such as object pooling and transactions. In order to expose your components to COM and Component Services, you must set specific assembly properties and create your classes appropriately.

Setting Assembly Properties

You must provide your assembly with a strong name if you want the assembly to be accessible to unmanaged code. To create a strong-named assembly, use a private and public key pair when you build the application, so that the assembly is guaranteed to be unique and cannot be inappropriately altered after you build it.

Naming Your Assembly

You can generate a strong name for your assembly by editing the **Strong Name** section of the **Common Properties** folder in the assembly property pages. The **Generate Key** button creates a key file called KeyFile.snk and links it to your project. Your assembly will then be strong-named the next time you build it. This option will be automatically selected if you choose to **Register for COM Interop** as described in the next paragraph.

Note For more information about creating strong-named assemblies, see Module 10, " Deploying Applications," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

Registering Your Assembly

You can automatically register an assembly for COM interoperability in the **Configuration Properties** section of the assembly property pages. The **Build** section provides a **Register for COM Interop** check box. If you select this check box, your assembly is registered with COM when it is next built. If you subsequently rebuild your assembly after the initial registration, it will first be unregistered before being re-registered. This process ensures that the registry does not contain outdated information.

Exposing Class Members to COM and Component Services

Creating a class that has public properties and methods does not make the class members accessible to COM and Component Services. Unless you expose the class members, the class itself will be accessible, but the methods will not be accessible except through late binding. You can expose the class members and enable early binding by:

- Defining a public interface.
- Using the **ClassInterface** attribute.
- Using the **COMClass** attribute.

Defining a Public Interface

Defining a public interface and implementing it within your public class allows unmanaged client applications to view and bind to the methods of the interface. This approach provides the most consistent and safe way to expose components to COM because use of interfaces prevents many problems associated with versioning.

The following code shows how to create a public interface and then use the interface in a class that will be accessible to unmanaged client applications through COM:

```
Public Interface IVisible
    Sub PerformAction( )
End Interface

Public Class VisibleClass
    Implements IVisible
    Public Sub PerformAction( ) _
                Implements IVisible.PerformAction
      'Perform your action
    End Sub
End Class
```

Using the ClassInterface Attribute

The **System.Runtime.InteropServices** namespace provides the **ClassInterface** attribute. This attribute allows you to create a class with a dual interface so that all members of the class (and base classes) are automatically accessible to unmanaged client applications through COM. The following code shows how to use the **ClassInterface** attribute:

Delivery Tip
Verify that students understand what dual interfaces are, and give a brief explanation if required.

```
Imports System.Runtime.InteropServices
<ClassInterface(ClassInterfaceType.AutoDual)> _
Public Class VisibleClass
    Public Sub PerformAction( )
      'Perform your action
    End Sub
End Class
```

Using the COMClass Attribute

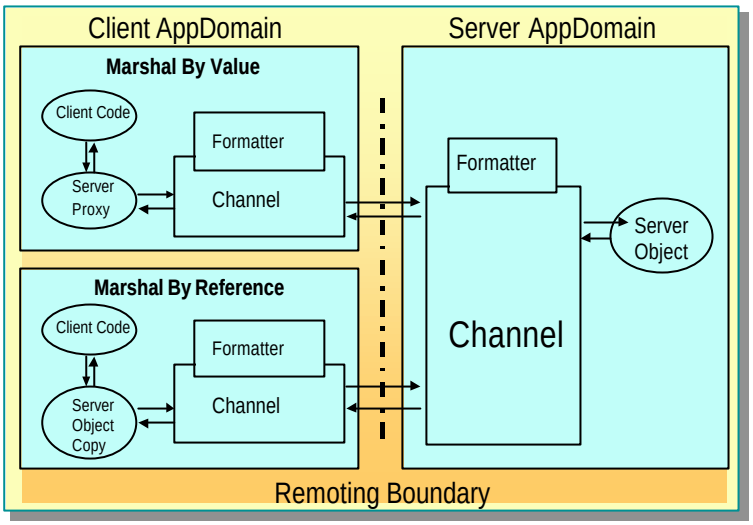
The **Microsoft.VisualBasic** namespace provides the **COMClass** attribute that you can use within a class to expose all of the public class members to COM. Visual Basic .NET provides a class template item called **COM Class** that you can add to any type of project that uses the **COMClass** attribute. Any assembly that contains this type of class will register itself when it is built and subsequently rebuilt.

Caution All three approaches can cause versioning problems if public method signatures are altered between versions. For this reason, implementing interfaces is the preferred approach because new interfaces with new method signatures can be created without causing versioning difficulties.

.NET Remoting Overview

Topic Objective
To provide an overview of .NET Remoting.

Lead-in
The .NET Framework provides several services that are used in remoting.



Previous versions of Visual Basic use COM and the distributed version of COM (DCOM) to communicate with components in different processes or on different computers. Visual Basic .NET uses .NET Remoting to allow communication between client and server applications across application domains.

Delivery Tip
Point out that this topic only presents an overview. Module 13, "Remoting and Web Services," in Course 2349A, *Programming the .NET Framework with C# (Prerelease),* covers this topic in more depth.

The .NET Framework provides several services that are used in remoting:

- Communication channels are responsible for transporting messages to and from remote applications by using either a binary format over a Transmission Control Protocol (TCP) channel or Extensible Markup Language (XML) over a Hypertext Transfer Protocol (HTTP) channel.
- Formatters that encode and decode messages before they are transported by the channel.
- Proxies that forward remote method calls to the proper object.
- Remote object activation and lifetime support for marshal-by-reference objects that execute on the server.
- Marshal-by-value objects that are copied by the .NET Framework into the process space on the client to reduce cross-process or cross-computer round trips.

Note For more information about .NET Remoting, see ".NET Remoting Technical Overview" in the Microsoft Visual Studio® .NET documentation.

◆ Creating Serviced Components

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

Lead-in
This lesson examines .NET components that are hosted by Component Services.

- **Hosting Components in Component Services**
- **Using Transactions**
- **Using Object Pooling**
- **Using Constructor Strings**
- **Using Security**
- **Using Other Component Services**
- **Configuring Assemblies for Component Services**

After completing this lesson, you will be able to:

- Describe the requirements for hosting .NET-based components in a Component Services application.
- Enable transaction processing in your components.
- Use object pooling to improve performance for objects that need extra resources.
- Use security attributes to specify how components interact with Component Services security.
- Add constructors to control how a component is initialized.
- Explain how to use other Component Services, such as Just-In-Time activation, from Visual Basic .NET components.
- Set assembly-level attributes to improve the installation of your application.

Hosting Components in Component Services

Topic Objective
To explain the requirements for hosting components in Component Services.

Lead-in
To enable components to be hosted in Component Services, the .NET Framework provides several items that you need to include in your assembly and classes.

- **Add a Reference to System.EnterpriseServices in Your Assembly**
- **The System.EnterpriseServices Namespace Provides:**
  - **ContextUtil** class
  - **ServicedComponent** class
  - Assembly, class, and method attributes

You must add a project reference to the **System.EnterpriseServices** namespace if you want to host a Visual Basic .NET component in a Component Services application. This namespace provides the main classes, interfaces, and attributes for communicating with Component Services.

The **System.EnterpriseServices** namespace provides the following features.

Feature	Usage
ContextUtil class	Use this class to participate in transactions and to interact with security information. The functionality of this class is similar to the functionality of the **ObjectContext** class in Visual Basic 6.0.
ServicedComponent class	All component classes that need to be hosted within a Component Services application must inherit this class. This class defines the base type for all context bound types and implements methods similar to those found in the **IObjectControl** interface used in Visual Basic 6.0–based Component Services applications.
Assembly, class, and method attributes	You can define several assembly attributes for Component Services interrogation in the AssemblyInfo.vb file. These values are used to set the application name and description and other values when the application is installed as a Component Services application. Several class and method attributes are also defined by the **System.EnterpriseServices** namespace, including **TransactionAttribute**, **AutoCompleteAttribute**, **ObjectPoolingAttribute**, and **ConstructionEnablesAttribute**.

Note The " Attribute" part of an attribute name is optional, so, for example, you can use either **AutoComplete** or **AutoCompleteAttribute** in your code.

Using Transactions

Topic Objective
To examine how components can utilize Component Services transactions.

Lead-in
Various objects and attributes enable Visual Basic .NET components to use Component Services transactions.

- **Transaction Attribute Specifies How a Class Participates in Transactions**
- **ContextUtil Class Provides Transaction Voting**
- **AutoComplete Attribute Avoids Using the SetAbort, SetComplete, and ContextUtil Methods**

```
<Transaction(TransactionOption.Required)> Public Class Account
    Inherits ServicedComponent

    Public Sub Debit(...)
        'Perform debit action
        ContextUtil.SetComplete( )
    End Sub

    <AutoComplete( )> Public Sub Credit(...)
        'Perform credit action
        'No SetComplete because AutoComplete is on
    End Sub
End Class
```

Transactions are often required to maintain data integrity and to synchronize updates to data in multiple data sources. You can enable transaction processing in serviced components by including the appropriate attributes and classes in your component code.

Transaction Attribute Options

You use the **Transaction** attribute to specify how a class participates in transactions. You can set transaction support to the one of the following options.

Option	Effect
Disabled	The class instance will not use transactions and will ignore any transactions from parent objects.
NotSupported	The class instance will not be created within the context of a transaction.
Required	The class instance will enlist in an existing transaction that is supplied by the calling object's context. If no transaction exists, one will be created.
RequiresNew	The class instance will always create a new transaction regardless of any transactions already created by calling objects.
Supported	The class instance will enlist in a transaction if provided by the calling object's context but will not create a transaction if one does not already exist.

Using the Transaction Attribute

The following example defines an **Account** class and sets the **Transaction** attribute as **Required**.

```
<Transaction(TransactionOption.Required)> Public Class Account
    Inherits ServicedComponent

    Public Sub Debit(ByVal id As Integer, _
                ByVal amount As Double)
        'Debit code
    End Sub
End Class
```

Transaction Voting Options

You can vote for a transaction outcome by using methods of the **ContextUtil** class, which is supplied by the **System.EnterpriseServices** namespace. This static class provides many methods and properties that will be familiar to you if you have created components that use MTS or Component Services. Several of the common methods are outlined below.

ContextUtil method	Use this method to:
SetAbort	Vote for the failure of a transaction. The transaction can only succeed if all objects involved in the transaction vote unanimously for success. This method also allows the object to be deactivated after the method call is complete.
SetComplete	Vote for the success of a transaction. If all objects involved in the transaction vote for success, then the transaction can be completed. This method also allows the object to be deactivated after the method call is complete.
EnableCommit	Vote for a successful completion of the transaction, while not allowing the object to be deactivated after the method call is complete. This is useful if you want to maintain state across multiple method calls, but you do not need further action to successfully complete the transaction if so requested by the top-level serviced component.
DisableCommit	Vote for an unsuccessful completion of the transaction, while not allowing the object to be deactivated after the method call is complete. This is useful if you want to maintain state across multiple method calls and you need other actions to occur before the transaction can be successfully completed.

Using the ContextUtil Class

The following example shows how to use the **ContextUtil** class to complete or abort transactions in the **Debit** method of the **Account** class, based on any exceptions encountered.

```
Public Sub Debit(ByVal id As Integer, ByVal amount As Double)
    Try
        'Perform update to database
        ...
        ContextUtil.SetComplete( )
    Catch ex As Exception
        ContextUtil.SetAbort( )
        Throw ex
    End Try
End Sub
```

Processing Transactions

To avoid using the **SetAbort** and **SetComplete** methods of **ContextUtil**, you can set the **AutoComplete** attribute of specific methods of the component. If no exceptions occur during the method execution, the object behaves as if **SetComplete** has been called. If exceptions do occur, the object behaves as if **SetAbort** has been called.

Using the AutoComplete Attribute

The following example shows how to use the **AutoComplete** attribute:

```
<AutoComplete( )>Public Sub Credit( _
   ByVal fromAccount As Integer, ByVal amount As Double)
    'Perform update to database
    …
    'No SetComplete or SetAbort is required
End Sub
```

Using Object Pooling

Topic Objective
To examine how components can use object pooling.

Lead-in
Various attributes and interfaces enable Visual Basic .NET components to use object pooling.

- **Object Pooling Allows Objects to Be Created in Advance**
- **ObjectPooling Attribute Specifies MinPoolSize and MaxPoolSize**
- **ServicedComponent Provides CanBePooled Method**

```
<ObjectPooling(Enabled:=True, MinPoolSize:=5, _
               MaxPoolSize:=50)> _
Public Class Account
    Inherits ServicedComponent
    ...
    Public Overrides Function CanBePooled( ) As Boolean
        Return True
    End Function
End Class
```

In Visual Basic .NET, you use the **ObjectPooling** attribute and the **ServicedComponent** base class to create serviced components that use object pooling.

What Is Object Pooling?

Object pooling allows a preset number of objects to be created in advance, so they are ready for use by client requests when the application first starts up. When a client application requests an object, one is taken from the pool of available objects and is used for that request. When the request is finished, the object is placed back in the pool for use by other client requests.

You can use pooling to improve the performance of objects that require significant periods of time to acquire resources and complete an operation. Objects that do not require such resources will not benefit significantly from object pooling.

Enabling Object Pooling

You specify the **ObjectPooling** attribute so that Component Services can place the component in an object pool. You can also specify optional arguments to the attribute that set the **MinPoolSize** and **MaxPoolSize** values of the pool.

- **MinPoolSize**

  To set the minimum number of objects to be created in advance in the pool, use the **MinPoolSize** argument.

- **MaxPoolSize**

  To set the maximum number of objects that can be created in the pool, use the **MaxPoolSize** argument.

  - If no objects are available in the pool when a request is received, the pool can create another object instance if this preset maximum number of objects has not already been reached.
  - If the maximum number of objects have already been created and are currently unavailable, requests will begin queuing for the next available object.

Returning Objects to the Object Pool

Use the **CanBePooled** method to specify whether your component can be returned to the object pool. Objects can only be returned to the pool when they are deactivated. This happens when the **SetComplete** or **SetAbort** methods are called when the object is transactional, or if a **Dispose** method is explicitly called if the object is not transactional.

- **True**

  If your component supports object pooling and can safely be returned to the pool, the **CanBePooled** method should return **True**.

- **False**

  If your component does not support object pooling, or if the current instance cannot be returned to the pool, the **CanBePooled** method should return **False**.

Note If object pooling is disabled for a component, the **CanBePooled** method will not be executed.

Using the CanBePooled Method

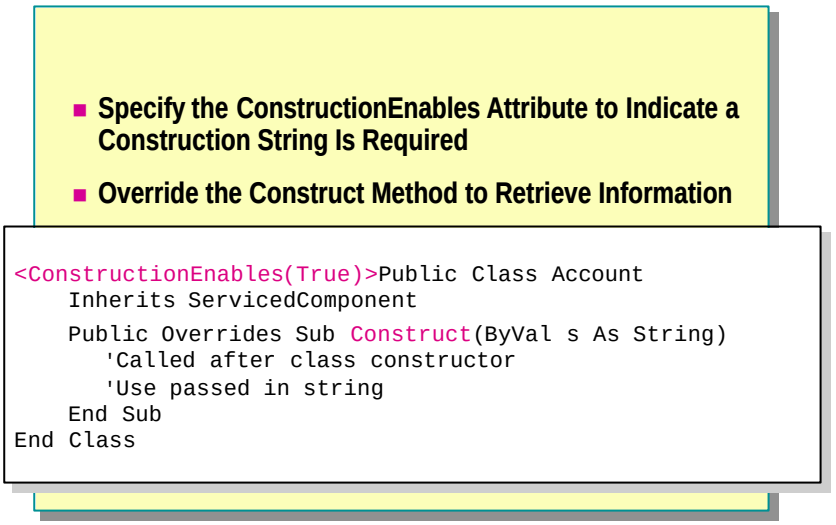
The following example shows how to create an object pool for the **Account** object with a minimum of five objects and a maximum of 50 at any one time. The **CanBePooled** method returns **True** to inform Component Services that the object can be returned to the pool.

```
<ObjectPooling(Enabled:=True, MinPoolSize:=5,MaxPoolSize:=50)>
Public Class Account
    Inherits ServicedComponent

    Public Sub Debit(ByVal id As Integer, _
                     ByVal amount As Double)
    ...
    End Sub

    Public Overrides Function CanBePooled( ) As Boolean
        Return True
    End Function
End Class
```

Using Constructor Strings

Topic Objective
To explain how components can utilize constructor strings.

Lead-in
Component Services provides constructor strings to serviced components that are accessible in Visual Basic .NET components through the .NET Framework.



You can use a constructor string to control how serviced components are initialized. This allows you to specify any initial information the object needs, such as a database connection string, by using the Component Services management console. You can use the **ConstructionEnables** attribute to enable this process in a serviced component. Your Visual Basic .NET component can then receive this constructor information because the inherited **ServicedComponent** class provides the overridable **Construct** method.

Using the ConstructionEnables Attribute

You specify the **ConstructionEnables** attribute at the class level so that a constructor string can be passed to the object during object construction. You can modify this value when the component is installed as a Component Services application using the Component Services management console.

Using the Construct Method

You override the **Construct** method of the **ServicedComponent** base class to receive the string value sent to the component during construction.

The following example shows how to enable a constructor, override the **Construct** method, and pass in a constructor string stored in a local variable.

```
<ConstructionEnables(True)>Public Class Account
    Inherits ServicedComponent

    Private strValue As String

    Public Overrides Sub Construct(ByVal s As String)
        'Called after class constructor
        strValue = s
    End Sub
End Class
```

Using Security

Topic Objective
To explain how Component Services security is accessible in Visual Basic .NET components.

Lead-in
Component Services provide security information that Visual Basic .NET components can use.

- **Security Configuration Attributes Enable Security and Role Configuration**
- **SecurityCallContext Class Provides Role Checking and Caller Information**

```
<ComponentAccessControl(True), SecurityRole("Manager")> _
Public Class Account
    Inherits ServicedComponent
    Public Function GetDetails( ) As String
        With SecurityCallContext.CurrentCall
            If .IsCallerInRole("Manager") Then
                Return .OriginalCaller.AccountName
            End If
        End With
    End Function
End Class
```

When working with serviced components, you can use pre-defined attributes and objects to configure and test security options.

Security Attribute Options

You can set security option by using attributes in your classes. Component Services will use these attributes when configuring your components as described in the following table.

Attribute	Usage
ApplicationAccessControl	Use this assembly-level attribute to explicitly enable or disable application-level access checking.
ComponentAccessControl	Use this component-level attribute to explicitly enable or disable component-level access checking.
SecurityRole	Use this attribute at the assembly level to add a role to the application. Use the attribute at the component level to add a role to the application and link it to the particular component.

Setting Security Options

The following example shows how to set the assembly-level **ApplicationAccessControl** attribute, enable security for the **Account** component, and create the *Manager* role, which will be linked to the **Account** component:

```
<Assembly: ApplicationAccessControl(True)>
<ComponentAccessControl(True), SecurityRole("Manager")> _
Public Class Account
    Inherits ServicedComponent
    ...
End Class
```

Retrieving Security Information

You can discover security information about the caller of a serviced component by using the **SecurityCallContext** class. This class provides information regarding the chain of callers leading up to the current method call. The static **CurrentCall** property of the **SecurityCallContext** class provides access to the following methods and properties.

Method or property	Usage
DirectCaller property	Retrieves information about the last user or application in the caller chain that directly called a method. The property returns an instance of the **SecurityIdentity** class that you can use to determine information about the identity, such as the **AccountName**.
OriginalCaller property	Retrieves information about the first user or application in the caller chain that made the original request for the required action. The property also returns an instance of the **SecurityIdentity** class.
IsCallerInRole method	Tests whether a caller belongs to a particular role; returns a **Boolean** value.
IsUserInRole method	Tests whether the user belongs to a particular role; returns a **Boolean** value

Delivery Tip
Point out that the example in the student notes uses the **IsSecurityEnabled** property of the **ContextUtil** object to avoid any exceptions caused by security being turned off for this component. This is the same technique used previously with MTS and Visual Basic 6.0.

Using the SecurityCallContext Class

The following example shows how use **SecurityCallContext** to determine whether security is enabled, check whether a caller is in the *Manager* role, and return the **AccountName** string from the **OriginalCaller** property, which is a **SecurityIdentity** instance.

```
<ComponentAccessControl(True), SecurityRole("Manager")> _
Public Class Account
    Inherits ServicedComponent

    Public Function GetDetails( ) As String
        If ContextUtil.IsSecurityEnabled Then
            With SecurityCallContext.CurrentCall
                If .IsCallerInRole("Manager") Then
                    Return .OriginalCaller.AccountName
                End If
            End With
        End If
    End Function
End Class
```

Using Other Component Services

Topic Objective
To provide an overview of the remaining services provided by Component Services.

Lead-in
Component Services provides several other services that you can use in Visual Basic .NET components.

- **Other Component Services Include:**
  - Just-in-time activation
  - Queued components
  - Shared properties
  - Synchronization

Component Services provides a series of other services that you can use from Visual Basic .NET components.

Just-in-Time Activation

When just-in-time (JIT) activation is enabled, an object is automatically instantiated when a method is called on a serviced component (activation), and then automatically deactivated when the method is complete (deactivation). When this option is enabled, an object does not maintain state across method calls, and this increases the performance and scalability of the application.

You can override the **Activate** and **Deactivate** methods inherited from the **ServicedComponent** class to perform custom functionality during JIT. If object pooling is enabled, the activation occurs when an existing object has been taken from the pool, and the deactivation occurs when the object is placed back in the pool.

JIT is automatically enabled if a component is transactional, and it cannot be disabled. You can manually enable or disable JIT for non-transactional components by using the **JustInTimeActivation** attribute.

Queued Components

Queued components provide asynchronous communication. This allows client applications to send requests to queued components without waiting for a response. The requests are “recorded” and sent to the server, where they are queued until the application is ready to use the requests. These requests are then “played back” to the application as if they were being sent from a regular client.

You can mark an application for queuing by using the assembly-level **ApplicationQueuing** attribute. Mark individual components with the **InterfaceQueuing** attribute.

Shared Properties

You can use the Shared Property Manger (SPM) components to share information among multiple objects within the same application process. Use the SPM components as you use them from components created in Visual Basic 6.0.

Synchronization

Distributed applications can receive simultaneous calls from multiple clients. Managing these simultaneous requests involves complex program logic to ensure that resources are accessed safely and correctly. Component Services provides this service automatically to components that use transactions. You can also use the **Synchronization** attribute to specify this behavior.

Configuring Assemblies for Component Services

Topic Objective
To explain how to set the assembly-level Component Services attributes and configure the application.

Lead-in
Setting assembly-level Component Services attributes helps define how your application will behave when you deploy it under Component Services.

- **Setting Assembly Attributes**
  - **ApplicationName**
  - **Description**
  - **ApplicationActivation**: library or server application
- **Using Regsvcs to Register and Create Component Services Applications**
  - Regsvcs.exe myApplication.dll
- **Using Lazy Registration**
  - Application registered on first use by client

You can specify some assembly level attributes that provide information when your assembly is installed as a Component Services application. The information is stored in the AssemblyInfo.vb file that is part of your Visual Basic .NET project.

Assembly attribute	Usage
ApplicationName	If you use this attribute to specify the name of the application, a Component Services application with the same name when your assembly is deployed and installed.
Description	Use this attribute to set the Component Services application description value when the assembly is deployed and installed.
ApplicationActivation	Use this attribute to specify whether you want to implement your Component Services application as either a library or a server application. The acceptable values for this attribute are **ActivationOption.Server** or **ActivationOption.Library**.

Setting Assembly Attributes

The following example shows a section of an AssemblyInfo.vb file that specifies the application name, the description, and information about where the application should be activated (that is, in a server or library process).

```
<Assembly: ApplicationName("BankComponent")>
<Assembly: Description("VB .NET Bank Component")>
<Assembly: ApplicationActivation(ActivationOption.Server)>
```

Registering Your Assembly

You can register your assembly with Component Services either manually or automatically.

- Manual registration

  You can use the Regsvcs.exe utility to manually register your assembly. This utility uses the information provided by your assembly attributes so that the Component Services application can be created with the correct default information. The basic syntax for using Regsvcs.exe is shown in the following example:

  ```
  .NET Framework Install path/Regsvcs.exe myApplication.dll
  ```

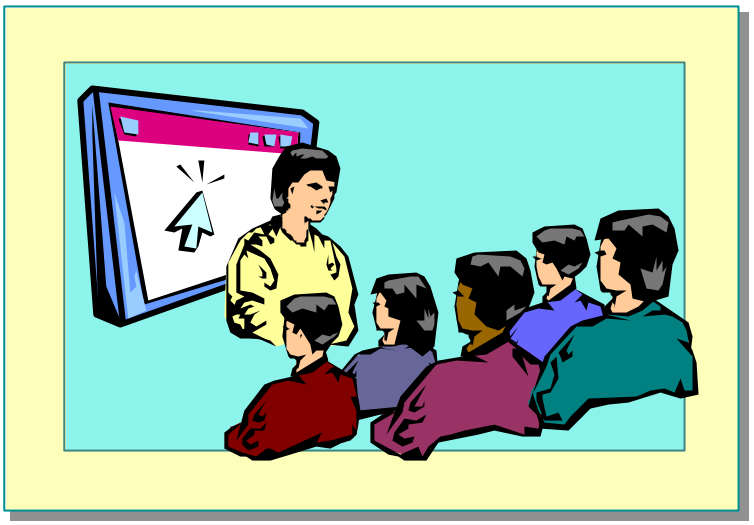
- Automatic registration

  If you do not register your application manually, registration will automatically occur when a client application attempts to create an instance of a managed class that inherits from the **ServicedComponent** class. All of the **ServicedComponent** classes within your assembly will then be registered as part of the Component Services application. This is known as Lazy Registration.

Demonstration: Creating a Serviced Component

Topic Objective
To demonstrate how to create a serviced component.

Lead-in
This demonstration shows how to create a serviced component.



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 a serviced component that uses object pooling and how to call the component from a managed client.

Lab 9.1: Creating a Serviced Component

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will create a serviced component that handles customer logon information.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Create a serviced component.
- Reference a serviced component.

Prerequisites

Before working on this lab, you must be familiar with creating and using components in MTS or Component Services.

Scenario

In this lab, you will create a serviced component based on a preexisting class. The class contains a single method that customers use to logon. You will register this assembly with Component Services and create a test harness application that references and tests your component. The test harness will use a preexisting form that allows you to enter a customer's e-mail address and password to retrieve the customer details by using the component.

Starter and Solution Files

There are starter and solution files associated with this lab. The starter files are in the *install folder*\Labs\Lab091\Starter folder, and the solution files are in the *install folder*\Labs\Lab091\Solution folder.

Estimated time to complete this lab: 60 minutes

Exercise 1
Creating the Serviced Customer Component

In this exercise, you will create a serviced component. The component is based on a prewritten interface called **ICustomer** and a class called **Customer** that implements the interface. You will add a reference to the **EnterpriseServices** assembly and mark the class as a serviced component that requires transactions and a construction string. You will add assembly-level attributes that will be used when you place the component under the control of Component Services.

↙ To open the CustomerComponent project

1. Open Microsoft Visual Studio .NET.
2. On the **File** menu, point to **Open**, and click **Project**.
3. Set the location to *install folder*\Labs\Lab091\Ex01\Starter, click **CustomerComponent.sln**, and then click **Open**.
4. Review the Customer.vb code for the **ICustomer** interface and **Customer** class so that you understand the purpose of the **LogOn** function.

↙ To reference the EnterpriseServices assembly

1. On the **Project** menu, click **Add Reference**.
2. On the **.NET** tab, in the **Component Name** list, click **System.EnterpriseServices**, click **Select**, and then click **OK**.
3. Open the Customer.vb Code Editor. At the start of the code, insert an **Imports** statement that references the **System.EnterpriseServices** namespace.

↙ To mark the Customer class as a serviced component

- Between the **Public Class Customer** definition and the **Implements ICustomer** statement, add an **Inherits ServicedComponent** statement.

↙ To add transactional behavior to the Customer component

1. Modify the class definition to include the **Transactional** class attribute, specifying a value of **TransactionOption.Required**. This is necessary because a "Last_Logon" date-time field is updated each time a customer logs on to the system.
2. In the **Try** block of the **LogOn** method, before the statement that executes **Return datCustomer**, add a call to **ContextUtil.SetComplete**.
3. Within the **Catch** block, before the statement that throws the exception to the calling application, add a call to **ContextUtil.SetAbort**.

↙ To add construction string behavior to the Customer component

1. Modify the class definition to include the **ConstructionEnabled** class attribute, specifying a value of **True**.
2. Override the **Construct** method of the inherited **ServicedComponent** class, and assign the passed in value to the local *connString* variable.

↙ To add the serviced component assembly attributes

1. Open AssemblyInfo.vb.
2. At the beginning of the file, add an **Imports** statement that references the **System.EnterpriseServices** namespace.
3. Add the following assembly attributes.

Assembly attribute	Parameters
ApplicationName	" Customers"
Description	" Customer Component"
ApplicationActivation	ActivationOption.Server

↙ To generate a key file

1. In Solution Explorer, right-click the **CustomerComponent** project, and then click **Properties**.
2. In the **Common Properties** folder of the tree view, click the **Strong Name** node.
3. Select the **Generate strong name using** check box, click **Generate Key** to create and link the key file to the project, and then click **OK**.

↙ To compile the assembly

- On the **Build** menu, click **Build**, and then quit Visual Studio .NET.

Exercise 2
Creating the Serviced Component Application

In this exercise, you will place the component under the control of Component Services and set the construction string for the database connection.

↙ To create the serviced component application

1. Open a command prompt session, and move to *install folder*\Labs\Lab091\Ex01\Starter\bin.
2. Execute the following command to install the assembly into the global assembly cache. (Note that this step is only required for Beta 2 to fix a known bug.)

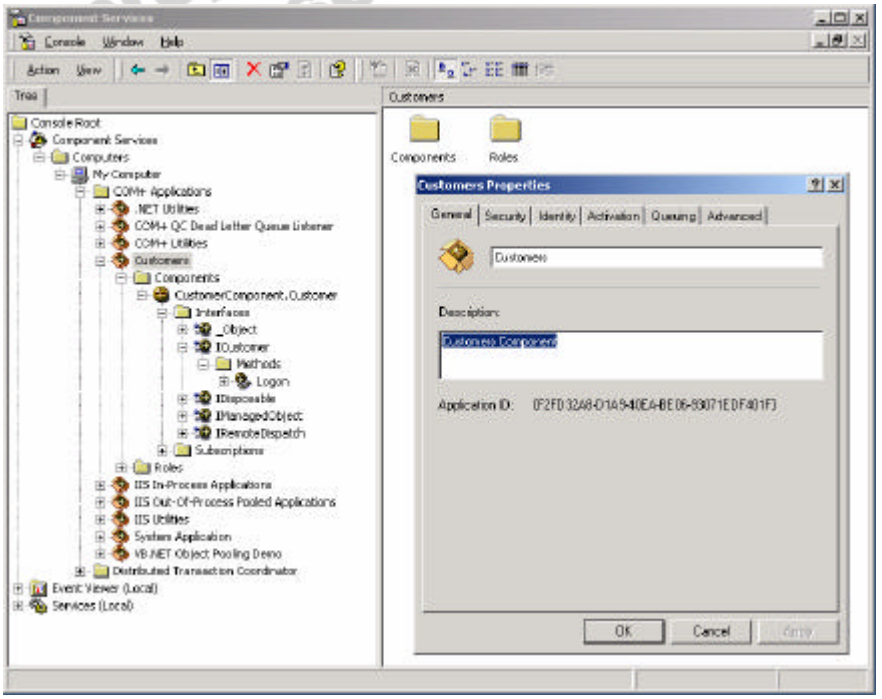
```
"%ProgramFiles%\Microsoft.NET\FrameworkSDK\Bin\gacutil.exe"
-i CustomerComponent.dll
```

3. Execute the following command to register the assembly and create the serviced component application:

```
"%Windir%\Microsoft.NET\Framework\v1.0.2914\Regsvcs.exe"
CustomerComponent.dll
```

↙ To confirm that the assembly is now a serviced component application

1. On the **Start** menu, point to **Programs**, point to **Administrative Tools**, and then click **Component Services**.
2. Locate the COM+ applications installed on the local computer.
3. Right-click the **Customers** application, and then click **Properties**. Your screen should appear similar to the following screen shot:



4. Confirm that the assembly-level attributes that you specified in your project have been set in the application.
5. Close the **Customers Properties** dialog box.

↙ To set properties for the Customer component

1. Expand the **Customers** application, and locate **CustomerComponent.Customer** within the list of components.
2. Right-click the **CustomerComponent.Customer** component, and then click **Properties**.
3. Click the **Transactions** tab to view the transactional setting for the class.
4. Click the **Activation** tab, set the constructor string to the following value, and then click **OK**:

   ```
   Data Source=LocalHost;Initial Catalog=Cargo;Integrated Security=True;
   ```

5. Close the Properties and Component Services windows.

Exercise 3
Testing the Serviced Customer Component

In this exercise, you will modify a prewritten test harness application to reference the serviced **Customer** component. You will then test the application.

↙ To open the test harness project

1. Open Visual Studio .NET.
2. On the **File** menu, point to **Open**, and then click **Project**.
3. Set the location to *install folder*\Labs\Lab091\Ex03\Starter, click **TestHarness.sln**, and then click **Open**.

↙ To set a reference to the serviced component assembly

1. On the **Project** menu, click **Add Reference**.
2. In the **Add Reference** dialog box, click **Browse**, and then locate the *install folder*\Labs\Lab091\Ex01\Starter\bin folder.
3. Click **CustomerComponent.dll**, and then click **Open**.
4. From the existing list of .NET components, click **System.EnterpriseServices**, and then click **Select**.
5. Click **OK** to close the **Add Reference** dialog box.

↙ To call the Customer object

1. In the frmTestCustomer code window, add an **Imports CustomerComponent** statement.
2. Locate the **btnLogon_Click** method. Within the **Try** block, declare an **ICustomer** variable called **cust**, and instantiate it by creating a new **Customer** object. Your code should look as follows:

```
Dim cust As ICustomer = New Customer( )
```

3. Call the **LogOn** method of the **cust** object, passing in the following values.

Parameter	Value
Email	**txtEmail.Text**
Password	**txtPassword.Text**

4. Use the **ds** Dataset object to store the value returned from the **LogOn** method.

↙ To test the application

1. On the **Debug** menu, click **Start**.
2. Enter the following values.

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

3. Click **Logon**, and confirm that a record is successfully retrieved from the component.
4. Click **Close** to quit the test harness application.
5. Quit Visual Studio .NET.

◆ Creating Component Classes

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

Lead-in
This lesson examines component classes.

- **Architecture of a Component Class**
- **Creating a Component Class**

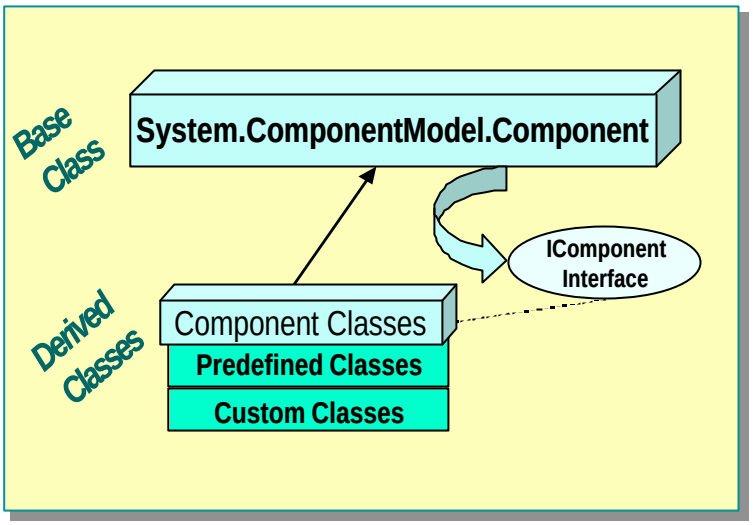
After completing this lesson, you will be able to:

- Describe the architecture of a component class.
- Create a component class.

Architecture of a Component Class

Topic Objective
To describe the architecture of a component class.

Lead-in
Component classes offer several features not included in standard Visual Basic .NET classes.



In addition to supporting classes and structures, the **System** namespace provides a library of components designed to make component development easy. When you create a component class based on the **ComponentModel.Component** base class, you automatically inherit the basic architecture for your class.

IComponent Interface

The **IComponent** interface allows you to create custom components or to configure existing components such as the **MessageQueue** or **Timer** components within the visual designer for your component. After you place any existing components on your component (*siting*), you can access them in your component code in the same way as you can when they are placed in the component tray of a Windows Form.

ComponentModel.Component Base Class

The **ComponentModel.Component** base class automatically implements the **IComponent** interface and provides all of the necessary code for handling the siting of components. This is useful because implementing the **IComponent** interface directly would require you to manually create the functionality for handling sited components in addition to the functionality for your component to be sited on another component.

Enhanced Design-Time Features

The **IComponent** interface provides enhanced design-time features. You can add your component class to the Toolbox and the component tray of a Windows Form, a Web Form, or any other item that implements the **IContainer** interface, including another component class. Developers using your component can then use the Properties window to set properties of the component in the same way that they would for .NET Framework components.

To add a compiled component class to the Toolbox, perform the following steps:

1. On the **Tools** menu, click **Customize Toolbox**.
2. In the **Customize Toolbox** dialog box, click the **.NET Framework Components** tab.
3. Browse for the component assembly that you want to add.
4. Select the component from the displayed list of compiled components to add it to the Toolbox.

Creating a Component Class

Topic Objective
To explain how to create a component class.

Lead-in
Creating a component class is similar to creating a standard class item, but there are a few extra steps.

1. **Inherit the System.ComponentModel.Component**
   - Perform any initialization in constructor
   - Override **Dispose** method
2. **Add Any Sited Components**
   - Use Server Explorer or Toolbox items
3. **Create Required Functionality**
   - Properties, methods, and events
4. **Build the Assembly**

The procedure for creating a component class with Visual Basic .NET is similar to the procedure for creating standard classes, but there are a few extra steps.

Delivery Tip
Point out the sample code in the student notes, but explain that a demonstration immediately follows this topic.

1. Inherit the **System.ComponentModel.Component** class.

   The **Component Class** template item contains the required code to inherit the **System.ComponentModel.Component** class, including the constructor code required to add your component class to a container. Add any initialization code for your component class as part of the construction process by placing code in the prewritten **Sub New** method.

   You can override the **Dispose** method of the inherited **Component** class to free any resources before the instance of your component is destroyed.

2. Add any sited components.

   If your component class requires other components in order to fulfill its purpose, you can add them within the Design view by dragging them from the Toolbox or Server Explorer to your component class. These components can then be programmatically accessed from within the code for your component class.

3. Create required functionality.

   Your component class can provide public properties, methods, and events to allow the user of your component to interact with it at both design time and run time.

4. Build the assembly.

   Building the assembly enables other managed clients to make a reference to your component.

The following example shows how to create a component class that is derived from the **System.ComponentModel.Component** class. It extends the functionality of the standard **Timer** class by defining additional properties and events.

Delivery Tip
Point out that inheriting from the Timer class would also produce a similar component.

```
Public Class Hourglass
  Inherits System.ComponentModel.Component

  Public Event Finished( )
  Private WithEvents localTimer As System.Timers.Timer

  Public Sub New( )
     MyBase.New( )

     'This call is required by the Component Designer.
     InitializeComponent( )

      'Initialize the timer for 1 minute (60000 milliseconds)
      localTimer = New System.Timers.Timer( )
      localTimer.Enabled = False
      localTimer.Interval = 60000
  End Sub

  Public Property Enabled( ) As Boolean
     Get
        Return localTimer.Enabled
     End Get
     Set(ByVal Value As Boolean)
        localTimer.Enabled = Value
     End Set
  End Property

  Private Sub localTimer_Tick(...) Handles localTimer.Tick
     'Raise the finished event after localtimer_Tick is raised
     RaiseEvent Finished( )
  End Sub

  Public Overloads Overrides Sub Dispose( )
     'Disable the localTimer object
     localTimer.Enabled = False
     localTimer.Dispose( )
     MyBase.Dispose( )
  End Sub
End Class
```

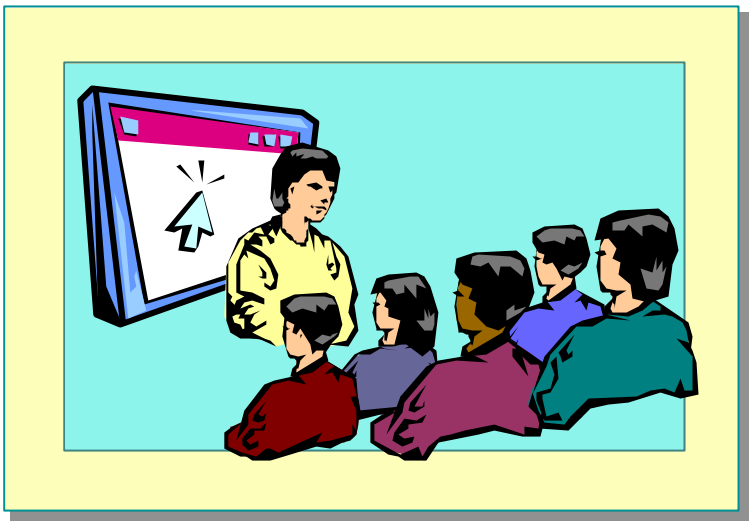

When examining the code, note the following:

- The component behaves as an hourglass that raises a **Finished** event one minute after it is enabled.
- The component can be turned on by using the **Enabled** property at design time or run time.
- The **localTimer** is initialized as part of the **Sub New** constructor and set for a timer interval of 60,000 milliseconds, or one minute.
- The **Dispose** method is overridden to ensure that the **localTimer** object is safely disposed of.

Demonstration: Creating a Stopwatch Component

Topic Objective
To demonstrate how to create and use a component class.

Lead-in
This demonstration shows how to create a stopwatch component class and use it from another application.



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 a component class that can be used by another assembly.

◆ Creating Windows Forms Controls

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

Lead-in
This lesson examines how to create Windows Forms controls in Visual Basic .NET.

- **Inheriting from the UserControl Class**
- **Inheriting from a Windows Forms Control**
- **Providing Control Attributes**

In previous versions of Visual Basic, you can create ActiveX controls that can be reused by different client applications. In Visual Basic. NET, you can also use inheritance to create controls.

After completing this lesson, you will be able to:

- Create a control based on the **System.Windows.Forms.UserControl** class.
- Create a control based on an existing Windows Forms control.
- Add attributes to your controls that enable advanced design-time functionality.

Inheriting from the UserControl Class

Topic Objective
To explain how to create a control that inherits from the **UserControl** class.

Lead-in
In Visual Basic .NET, you can inherit from the **UserControl** class to create the same type of user controls that you can create in Visual Basic 6.0.

- **Inherit from System.Windows.Forms.UserControl**
- **Add Required Controls to Designer**
- **Add Properties and Methods That Correspond to Those of Constituent Controls**
- **Add Any Additional Properties and Methods**
- **No InitProperties, ReadProperties, or WriteProperties**
  - Property storage is automatic

In previous versions of Visual Basic, you can create a unique new control by placing one or more existing controls onto a UserControl designer. You can then create custom properties, methods, and events to set and retrieve values for the contained controls. This type of control is useful when several forms require the same layout of controls, such as forms for addresses or contact details.

Adding Required Controls

In Visual Basic .NET, you can create the same type of user controls by inheriting your control from the **System.Windows.Forms.UserControl** class, which is automatic if you create a control using the **User Control** template item. You can inherit from this base class to use a designer similar to the one used in previous versions of Visual Basic. By using this method, you can:

- Place as many controls on the designer as you need to in order to create your own user control.
- Access these controls within your user control class, because they are declared as private variables.
- Add your own properties and methods that correspond to the properties and methods of the constituent controls.
- Add public properties, methods, and events in exactly the same way that you do for a regular class.

Adding Properties and Methods

In previous versions of Visual Basic, you persist the properties to a **PropertyBag** object, so the control retains its settings between design time and run time. To do this, you write code in the **ReadProperties** and **WriteProperties** events of the **UserControl** class.

In Visual Basic .NET, this persisting of information is automatic and requires no extra code.

Example

The following example shows how to create a simple user control that contains a label and a text box:

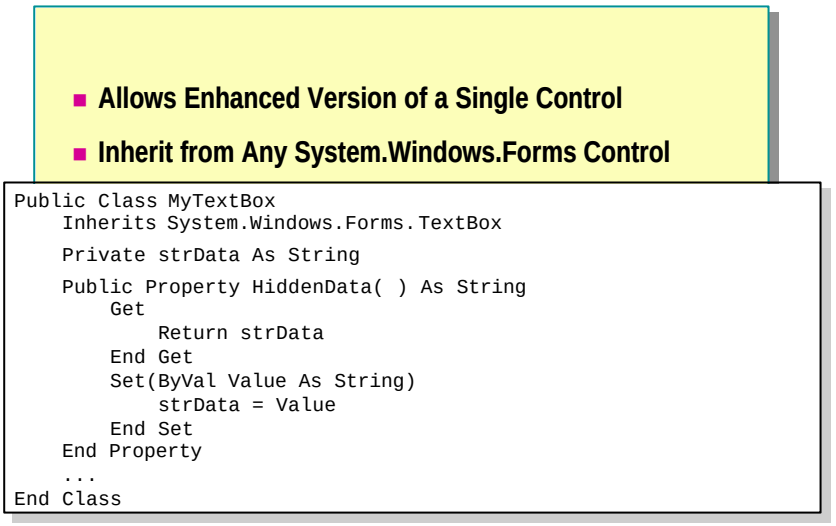
```
Public Class LabelAndTextControl
    Inherits System.Windows.Forms.UserControl

    Public Property TextBoxText( ) As String
        Get
            Return TextBox1.Text
        End Get
        Set(ByVal Value As String)
            TextBox1.Text = Value
        End Set
    End Property

    Public Property LabelText( ) As String
        Get
            Return Label1.Text
        End Get
        Set(ByVal Value As String)
            Label1.Text = Value
        End Set
    End Property
    ... 'Windows Form Designer generated code
End Class
```

The TextBox1 and Label1 controls are privately declared variables within the user control that are only accessible using the public properties **TextBoxText** and **LabelText**.

Inheriting from a Windows Forms Control

Topic Objective
To explain how to inherit from a Windows Forms control.

Lead-in
Inheritance makes it easy for you to enhance an existing control in Visual Basic .NET.



In previous versions of Visual Basic, you can create enhanced versions of an existing control by placing an instance of the control on the UserControl designer. You can then create public properties, methods, and events that correspond to the equivalent items of the constituent control, adding any custom items to create your enhanced behavior.

In Visual Basic .NET, you can create a control that inherits from any **System.Windows.Forms** class, such as the **TextBox** or **Label** class. Because this approach uses inheritance, there is no need to create public properties, methods, and events that map to the constituent control. This greatly reduces the amount of code required. You only need to create any extra functionality, as described for user controls in the previous topic.

The following example shows how to create a control that inherits from **SystemWindows.Forms** and adds a public property:

```
Public Class MyTextBox
    Inherits System.Windows.Forms.TextBox

    Private strData As String

    Public Property HiddenData( ) As String
        Get
            Return strData
        End Get
        Set(ByVal Value As String)
            strData = Value
        End Set
    End Property
    ...
End Class
```

This code creates a new control that inherits all of the **TextBox** class functionality and adds a property called **HiddenData**.

Note For some existing controls, you can create a new graphical front end by overriding the **OnPaint** method of the base class. However, some controls, such as the **TextBox** control, are painted directly by Windows and cannot be overridden.

Providing Control Attributes

Topic Objective
To explain how to use control attributes.

Lead-in
Control attributes can be used to supply extra information about the control and its properties, methods, and events.

- **System.ComponentModel Provides Control Attributes**
- **Class Level - DefaultProperty, DefaultEvent, ToolboxBitmap**
- **Property Level - Category, Description, DefaultValue**

```
Imports System.ComponentModel
<ToolboxBitmap("C:\TXTICON.BMP"), DefaultEvent("Click")> _
Public Class MyTextBox
    Inherits System.Windows.Forms.UserControl
    <Category("Appearance"), _
     Description("Stores extra data"), _
     DefaultValue("Empty")> _
    Public Property HiddenData( ) As String
        ...
    End Property
    ...
End Class
```

In previous versions of Visual Basic, you can use the **Procedure Attributes** dialog box to set control attributes, such as property descriptions and their categories, which can be viewed in the Object Browser. You can supply similar information in Visual Basic .NET by using the attributes provided by the **System.ComponentModel** namespace.

Setting Class-level Attributes

You can specify several attributes for the control, including **DefaultProperty**, **DefaultEvent**, and **ToolboxBitmap**. The following example shows how to set the **ToolboxBitmap** and **DefaultEvent** attributes for the **MyTextBox** class:

```
<ToolboxBitmap("C:\TXTICON.BMP"), DefaultEvent("Click")> _
Public Class MyTextBox
    Inherits System.Windows.Forms.UserControl
...
End Sub
```

Setting Property-level Attributes

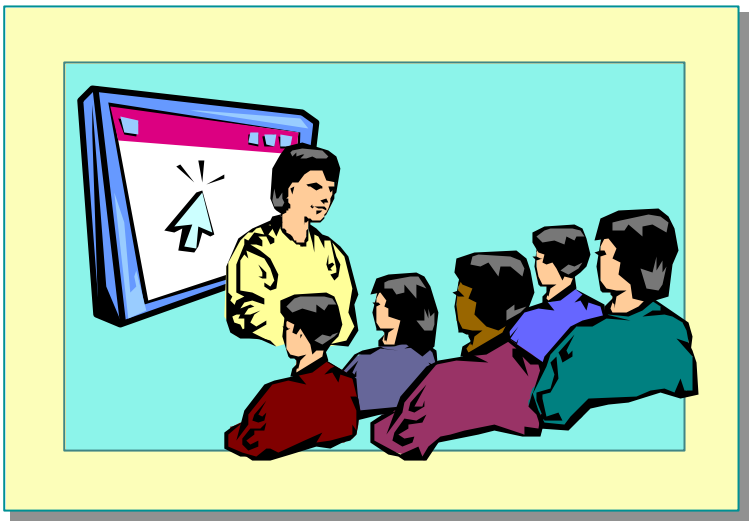
You can specify property-level attributes for any public properties, including the **Category**, **Description**, and **DefaultValue** attributes. The following example shows how to set these attributes for the **HiddenData** property:

```
Imports System.ComponentModel

Public Class MyTextBox
    Inherits System.Windows.Forms.UserControl

    <Category("Appearance"), _
     Description("Stores extra data"), _
     DefaultValue("Empty")> _
    Public Property HiddenData( ) As String
        ...
    End Property
    ...
End Class
```

Demonstration: Creating an Enhanced TextBox

Topic Objective
To demonstrate how to create a control based on an existing Windows Forms control.

Lead-in
In this demonstration, you will learn how to create a control based on the Windows Forms **TextBox** control.



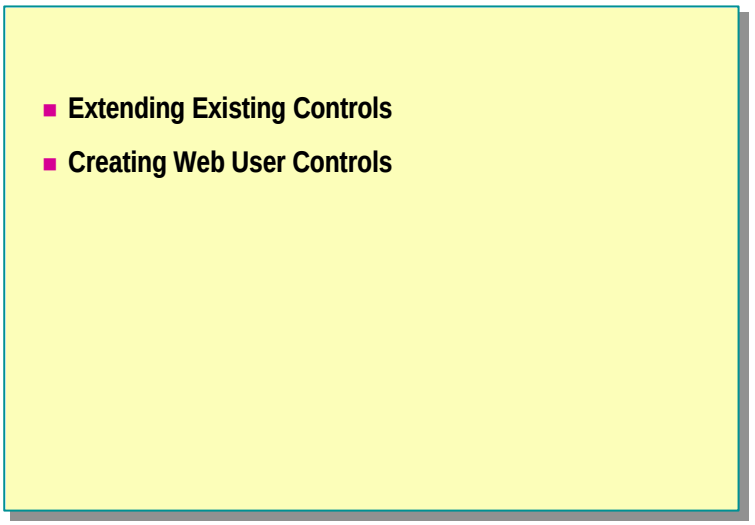
Delivery Tip
The step-by-step instructions for this demonstration are in the instructor notes for this module.

In this demonstration, you will learn how to create a Windows Forms user control based on the existing **TextBox**.

◆ Creating Web Forms User Controls

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

Lead-in
This lesson examines how to create Web Forms user controls in Visual Basic .NET.



In Visual Basic .NET, you can create controls for use within ASP .NET Web Forms.

After completing this lesson, you will be able to:

- Create a Web Forms user control based on other controls in the **System.Web.UI.UserControl** class.
- Use a Web Forms user control within a Web Form.

Extending Existing Controls

Topic Objective
To explain how to create a Web user control by extending existing Web server controls.

Lead-in
You can create your own Web user controls by extending the existing Web server controls provided by ASP .NET.

1. **Add a Web User Control to an ASP.NET Web Project**
2. **Use the Toolbox to Drag Existing Controls to the Web User Control Designer**
3. **Add Properties and Methods**
4. **Save the .ascx File**
5. **Drag the .ascx File from Solution Explorer to the Web Forms Designer**
6. **Create Web Form Code as Usual**

Creating your own Web user control allows you to extend the controls provided with ASP .NET. You can extend a single control with added features or create a new control that is a combination of existing controls.

To create your own Web user control:

1. Add a Web user control to your ASP .NET Web project.
2. Use the Toolbox to drag-and-drop existing Web server controls to the Web user control designer.
3. Add properties and methods in the code-behind file.
4. Save the .ascx Web user control file.

To use your Web user control:

1. Open your Web Form.
2. Drag the .ascx file from Solution Explorer to the Web Forms Designer.
3. Create any Web Form code that accesses the Web user control, as you would for existing Web server controls.
4. Test your control by running your application and displaying the Web Form.

Creating Web User Controls

Topic Objective
To explain how to create a simple Web user control.

Lead-in
The code for creating a Web user control is very similar to that of a Web Form.

```
<%@ Control Language="vb" AutoEventWireup="false"
  Codebehind="SimpleControl.ascx.vb"
  Inherits="MyApp.SimpleControl"%>
<asp:TextBox id="TextBox1" runat="server"></asp:TextBox>
```

```
Public MustInherit Class SimpleControl
 Inherits System.Web.UI.UserControl
 Protected WithEvents TextBox1 As System.Web.UI.WebControls.TextBox
 Public Property TextValue( ) As String
     Get
         Return TextBox1.Text
     End Get
     Set(ByVal Value As String)
         TextBox1.Text = Value
     End Set
 End Property
End Class
```

To create a Web user control, you need to create:

1. The graphical layout of the controls in the .ascx file.
2. The code that executes in the.ascx.vb code-behind file.

The following example shows how to create a Web user control based on the existing **TextBox** control while inheriting from the **UserControl** class. It also provides a custom property for setting the **TextBox1.Text** value.

The following code is located in the Web user control .ascx file:

```
<%@ Control Language="vb" AutoEventWireup="false"
  Codebehind="SimpleControl.ascx.vb"
  Inherits="MyApp.SimpleControl"%>
<asp:TextBox id="TextBox1" runat="server"></asp:TextBox>
```

The example code shows the similarity between Web Forms and Web user control code, the main difference being the **@ Control** directive and the lack of any <html>, <body>, or <form> tags.

The following code is located in the .ascx.vb code-behind file.

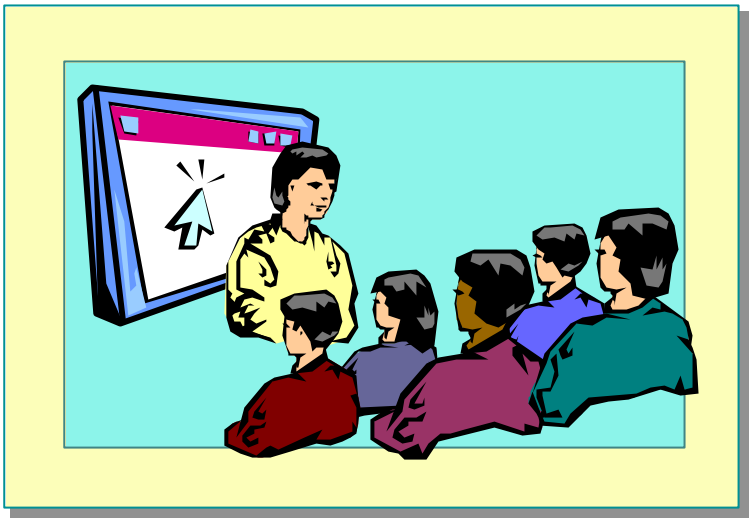
```
Public MustInherit Class SimpleControl
  Inherits System.Web.UI.UserControl
  Protected WithEvents TextBox1 _
            As System.Web.UI.WebControls.TextBox

    Public Property TextValue( ) As String
        Get
            Return TextBox1.Text
        End Get
        Set(ByVal Value As String)
            TextBox1.Text = Value
        End Set
    End Property
End Class
```

The **SimpleControl** class is similar to most classes in that it allows public access to private members of the class. However, note that it is through inheriting the **UserControl** class that the Web user control functionality is provided.

Demonstration: Creating a Simple Web Forms User Control

Topic Objective
To demonstrate how to create a Web Forms user control based on multiple constituent controls.

Lead-in
This demonstration shows how to create a simple Web Forms user control that uses a **Label** and a **TextBox** as its constituent controls.



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 a simple Web Forms user control that contains a **Label** and a **TextBox** as its constituent controls.

Lab 9.2: Creating a Web Forms User Control

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will create a Web Forms user control that creates a logon screen for customers.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Create a Web Forms user control.
- Use a Web Forms user control on a Web Form.

Prerequisites

Before working on this lab, you must be familiar with creating Web Form applications.

Scenario

In this lab, you will create a Web Forms user control that requests logon information for a customer. The control will retrieve the customer information by means of the serviced component that you created in the previous lab. You will then use this control on a Web Form and test the control.

Starter and Solution Files

There are starter and solution files associated with this lab. The starter files are in the *install folder*\Labs\Lab092\Starter folder, and the solution files are in the *install folder*\Labs\Lab092\Solution folder.

Estimated time to complete this lab: 30 minutes

Exercise 1
Creating the LogOn Web Forms User Control

In this exercise, you will open a preexisting Web Forms application that allows you to logon as a customer of the system. You will create a LogOn Web Forms user control that uses text boxes and validation controls. This user control allows users to enter their e-mail address and password and then click a **Submit** button.

↙ To open the existing Web Forms application

1. Open Visual Studio .NET.
2. On the **File** menu, point to **Open**, and then click **Project**.
3. Set the location to **Error! Hyperlink reference not valid.***install folder*\Labs\Lab092\Ex01\Starter, click **LogonControl.sln**, and then click **Open**.

↙ To create the Web user control interface

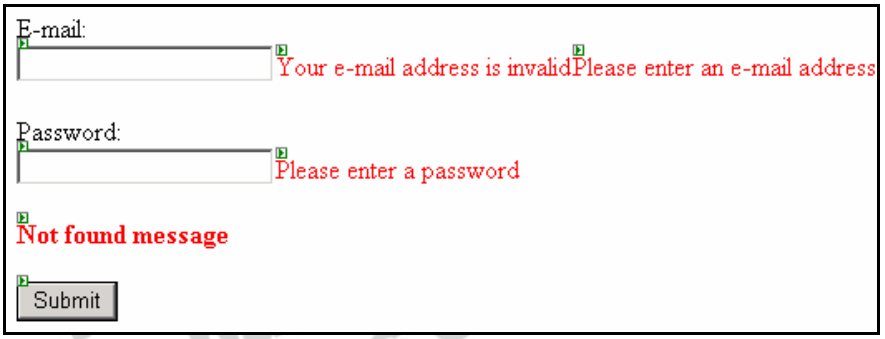
1. On the **Project** menu, click **Add Web User Control**. Rename the item to **Logon.ascx**, and then click **Open**.
2. From the **Web Forms** tab of the Toolbox, insert the following controls, and set their property values as shown.

Control	Property name	Property value
Label	**(ID)**	**lblEmail**
	Text	**E-mail:**
TextBox	**(ID)**	**txtEmail**
RegularExpressionValidator	**(ID)**	**revEmail**
	ErrorMessage	**Your e-mail address is invalid**
	ControlToValidate	**txtEmail**
	ValidationExpression	**Browse and select Internet E-mail Address**
	Display	**Dynamic**
RequiredFieldValidator	**(ID)**	**rfvEmail**
	ErrorMessage	**Please enter an e-mail address**
	ControlToValidate	**txtEmail**
	Display	**Dynamic**
Label	**(ID)**	**lblPassword**
	Text	**Password:**
TextBox	**(ID)**	**txtPassword**
	TextMode	**Password**

(*continued*)

Control	Property name	Property value
RequiredFieldValidator	**(ID)**	**rfvPassword**
	ErrorMessage	**Please enter a password**
	ControlToValidate	**txtPassword**
	Display	**Dynamic**
Label	**(ID)**	**lblNotFound**
	Text	**Not found message**
	ForeColor	**Red**
	Visible	**False**
Button	**(ID)**	**btnSubmit**
	Text	**Submit**

3. Arrange your controls as shown in the following screen shot:



↙ To create the Web user control code

1. View the Code Editor for **Logon.ascx**.
2. Declare an event with the following signature:

```
Public Event SubmitPressed(ByVal Email As String, _
                           ByVal Password As String)
```

3. Create a **Click** event handler for the **btnSubmit** event. In this method, set the **Visible** property of the lblNotFound label to **False**, and raise the **SubmitPressed** event, passing the following parameters:

Parameter	Value
Email	**txtEmail.Text**
Password	**txtPassword.Text**

4. Create a **DisplayMessage** subroutine that accepts a single string argument called **Message**. Within the subroutine, set the following values for the lblNotFound label.

Control property	Value
Text	**Message**
Visible	**True**

5. Save your project.

Exercise 2
Testing the LogOn Web Forms User Control

In this exercise, you will create a simple Web Form that uses the **Logon** user control to get customer logon information from the user. This information will then be passed to the serviced customer component for validation and information retrieval that you created in an earlier exercise. You will then redirect the browser to a preexisting Web Form that displays a welcome message with the customer's first name.

In case you have not completed the previous exercise, a starter project has been provided.

↙ To open the starter project

1. On the **File** menu, point to **Open**, and then click **Project**.
2. Set the location to *install folder*\Labs\Lab092\Ex02\Starter, click **LogonControl.sln**, and then click **Open**.

↙ To set a reference to the serviced component assembly

1. On the **Project** menu, click **Add Reference**.
2. In the **Add Reference** dialog box, click **Browse**, and then locate the *install folder*\Labs\Lab091\Ex01\Starter\bin folder.
3. Click **CustomerComponent.dll**, and then click **Open**.
4. From the existing list of .NET components, click **System.EnterpriseServices**, and then click **Select**.
5. Click **OK** to close the **Add Reference** dialog box, and click **OK** when asked if you want to add the reference path to the project.

↙ To create the logon page

1. On the **Project** menu, click **Add Web Form**, and rename the file **LogonPage.aspx**.
2. Drag Logon.ascx from Solution Explorer to the LogonPage Web Forms Designer to create an instance of the control on the Web Form.
3. In the LogonPage code window, add an **Imports CustomerComponent** statement.
4. Add the following variable declaration after the **Inherits System.Web.UI.Page** statement:

```
Protected WithEvents Logon1 As Logon
```

5. Create an event handler procedure for the **SubmitPressed** event of Logon1, and add the following code:

```
Dim ds As DataSet, dr As DataRow

Dim cust As ICustomer = New Customer( )
ds = cust.Logon(Email, Password)

If ds.Tables(0).Rows.Count = 0 Then
    Logon1.DisplayMessage _
     ("No match was found. Please reenter your details.")
Else
    dr = ds.Tables(0).Rows(0)
    Session("FirstName") = dr("FirstName")
    Response.Redirect("Welcome.aspx")
End If
```

6. Save the project.

↙ To test the application

1. In Solution Explorer, right-click **LogonPage.aspx**, and then click **Set As Start Page**.
2. On the **Debug** menu, click **Start**.
3. Click **Submit** without entering any values in the text boxes to test the validation controls.
4. Enter the following deliberately incorrect values in the text boxes, and then click **Submit**.

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

5. Confirm that an error message is displayed by the user control.
6. Enter the same e-mail address as in step 4, but use the correct password of **password**, and then click **Submit**. Confirm that the welcome message is displayed and that the customer has been recognized.
7. Quit Microsoft Internet Explorer and Visual Studio .NET.

◆ Threading

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

Lead-in
Visual Basic .NET allows developers to use the power of threading in a way not previously available in Visual Basic.

- **What Is a Thread?**
- **Advantages of Multithreading**
- **Creating Threads**
- **Using Threading**
- **When to Use Threading**

Previous versions of Visual Basic have limited threading support. Visual Basic .NET allows developers to use the full power of threads when necessary. When you use threading correctly, you can enhance the performance of your application and make it more interactive.

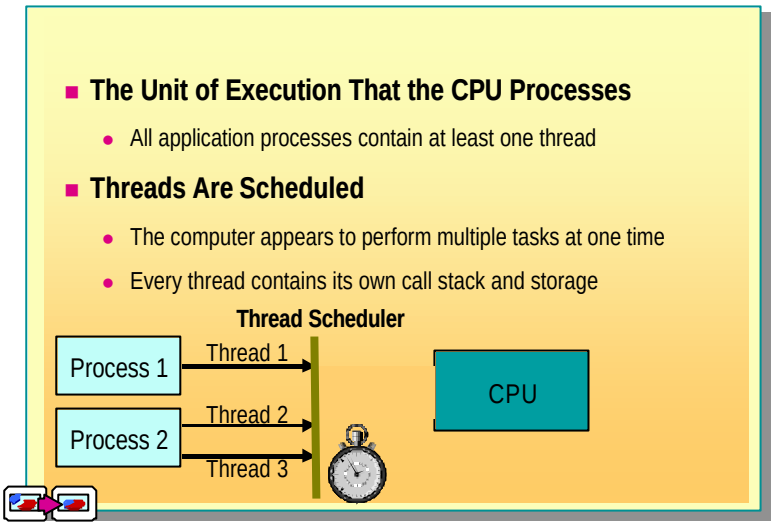
After you complete this lesson, you will be able to:

- Explain the basic concepts of threading.
- List the advantages of incorporating multithreading into your applications.
- Create and use threads by using the **System.Threading** namespace.
- Avoid some potential problems in your multithreaded applications.

What Is a Thread?

Topic Objective
To explain the basic concepts of threading.

Lead-in
Before examining how Visual Basic .NET enables threading, it is important to understand the basic concepts of threading.



Delivery Tip
The slide associated with this topic is an animated slide. Click the slide to reveal the following lessons, showing the iterative process of the thread scheduler:
1. Thread 1
2. Thread 2
3. Thread 3
4. Thread 1
5. Thread 2
6. Thread 3

An application running on a computer is known as a process. Each process gets work done by using one or more *threads*. The thread is the unit of execution that is processed by the CPU of the computer.

Threading Process

A CPU can only execute a single thread at any one instant, so a thread scheduler allocates a certain amount of CPU time for each thread to get as much work done as possible before allowing another thread to access the CPU. This scheduling makes a computer appear to perform multiple tasks at once. In reality, the following is what happens:

1. Every thread contains its own call stack and storage for local variables. This information is kept with the thread and passed to the CPU whenever the thread is scheduled for processing.
2. When the time is up, the thread scheduler removes the thread from the CPU and stores the call stack and variable information.

The more threads that are running on the system, the less frequently a thread is scheduled to run in the CPU. This is why a computer can appear to be running slowly when you have multiple applications open and functioning at the same time.

Threading Types

Different programming languages support different types of threading:

- Previous versions of Visual Basic support the apartment threading model.

  This model places some restrictions on the types of applications that these versions are best suited for creating. One of these restrictions is that an object is tied to the thread that it is created on, and cannot be used for object pooling in Component Services. However, this model makes development easy because you do not need to be involved with more complex issues such as synchronization.

- Visual Basic .NET supports the free threading model.

  This model allows you to use multithreading and features such as object pooling or to continue using single threads as you have in applications created with previous versions of Visual Basic.

Advantages of Multithreading

Topic Objective
To explain the advantages of multithreading and free threading.

Lead-in
Multithreading can provide many benefits to your applications.

- **Improved User Interface Responsiveness**
  - Example: a status bar
- **No Blocking**
- **Asynchronous Communication**
- **No Thread Affinity**
  - Objects are not tied to one thread

A multithreaded application has several advantages over a single-threaded application.

Improved User Interface Responsiveness

You can use multiple threads in a single process to improve the responsiveness of the user interface. The following is an example:

- Use threads for lengthy processing operations, such as using the spelling checker or reformatting pages. These extra threads can then raise events to the main user interface thread to update items such as a status bar.
- Assign each thread a priority level so that particular threads can run as a higher priority than other lower priority threads. In an application that relies heavily on user interaction, you should run the user interface thread as a higher priority thread.

No Blocking

Blocking occurs because a call to a single-threaded application must wait until any previous call by another client application has been fully satisfied before executing any other code. In server-based applications, blocking will occur if multiple clients make simultaneous requests of a process and only a single thread is available.

Multithreaded applications are able to perform actions on different threads simultaneously (through thread scheduling) without waiting for other threads to finish their current execution. This allows multiple clients to be handled by different threads without any blocking in a server-based application.

Asynchronous Communication

Asynchronous communication is possible in a multithreaded application because one thread can make a request to another thread. The calling thread can continue with other processing because the request executes on a separate thread. An event can be raised when the second thread finishes executing the requested functionality, informing the first thread that it has completed its work.

No Thread Affinity

Visual Basic .NET uses the free threading model. This model does not restrict you to using an object only on the thread where it was initially created. You can create an object on one thread and then pass it to another thread without difficulty. This improves scalability when used in conjunction with Component Services and object pooling.

Creating Threads

Topic Objective
To explain how to create and use threads.

Lead-in
The .NET Framework provides the **System.Threading.Thread** class, which allows you to create multiple threads.

- **Use the System.Threading.Thread Class**
  - Constructor specifies delegate method
  - Methods provide control of thread processing
  - Properties provide state and priority information
- **Use a Class If Parameters Are Required**
  - Allow public access to class variables
  - Raise an event when finished

The .NET Framework provides a simple way to create and work with multiple threads.

Using the System.Threading.Thread Class

Delivery Tip
The next topic, Using Threading, provides a full example of threading.

Use the **Thread** class to create multiple threads within a Visual Basic .NET–based application.

Constructing the Thread

When a **Thread** instance is created, the **AddressOf** operator passes the constructor a delegate representing the method to be executed, as shown in the following example:

```
Dim th As New Threading.Thread(AddressOf PerformTask)
...
Sub PerformTask( )
    ...
End Sub
```

Threading Methods

The **Thread** class also provides several methods to control the processing of a thread.

Method	Purpose
Start	Begins execution of the method delegate declared in the thread constructor.
Abort	Explicitly terminates an executing thread.
Sleep	Pauses a thread. Specifies the number of milliseconds as the only parameter. If you pass zero as the parameter, the thread gives up the remainder of its current time slice. This is similar to **DoEvents** in previous versions of Visual Basic.
Suspend	Temporarily halts execution of a thread.
Resume	Reactivates a suspended thread.

Threading Properties

The **Thread** class provides properties to retrieve information about the thread state and to manipulate the thread priority.

Property	Purpose
ThreadState	Use the **ThreadState** property to determine the current state of a thread, such as **Running**, **Suspended**, or **Aborted**.
Priority	Modify the priority of a thread by setting its **Priority** property by using the **ThreadPriority** enumeration. The enumeration provides the following values: **AboveNormal**, **BelowNormal**, **Highest**, **Lowest**, and **Normal**.

Warning If you set thread priorities to a value of **Highest**, this may affect vital system processes by depriving them of CPU cycles. Use this setting with caution.

Creating and Testing Threads

The following example shows how to create a thread, test the state of the thread, and change its priority:

```
Dim th As New Threading.Thread(AddressOf PerformTask)
th.Start( )
If th.ThreadState = ThreadState.Running Then
    th.Priority = ThreadPriority.AboveNormal
End If
```

Using Classes to Supply Parameters

Delivery Tip
The next topic, Using Threading, provides an example of this approach.

You cannot specify a method delegate that accepts arguments in the thread constructor. If your procedure requires information to perform its required action, you can:

- Use classes to provide methods that perform operations on local data.
- Use public properties or variables to supply the local data.

To use classes to supply parameters, you must create an instance of the class before calling the thread constructor. Use the **AddressOf** operator to pass a reference to the method of the class as the constructor parameter. You can then use the properties or public variables to supply any data required by the method. When the worker method finishes its execution, you can raise an event to inform the calling thread that the operation is completed.

Using Threading

Topic Objective
To explain a simple example of threading.

Lead-in
Let's take a look at a simple threading example.

```
Class Calculate
  Public iValue As Integer
  Public Event Complete(ByVal Result As Integer)
  Public Sub LongCalculation( )
    'Perform a long calculation based on iValue
    ...
    RaiseEvent Complete(iResult)     'Raise event to signal finish
  End Sub
End Class
```

```
Sub Test( )
  Dim calc As New Calculate( )
  Dim th As New Threading.Thread(AddressOf calc.LongCalculation)
  calc.iValue = 10
  AddHandler calc.Complete, AddressOf CalcResult
  th.Start( )
End Sub
Sub CalcResult(ByVal Result As Integer)
  ...
End Sub
```

This topic shows how to prepare a class for threading, create a thread, start the thread, and perform calculations on the new thread.

Preparing a Class for Threading

The following example shows how to create a **Calculate** class and prepare it for threading by using the **Complete** event:


```
Class Calculate
   Public iValue As Integer
   Public Event Complete(ByVal Result As Integer)
   Public Sub LongCalculation( )
    'Perform a long calculation based on iValue
     ...
    RaiseEvent Complete(iResult)'Raise event to signal finish
   End Sub
End Class
```

When examining the previous code, note the following:

- The class provides a **LongCalculation** worker function, which will be executed on a separate thread.
- The worker function uses information stored in the public *iValue* integer variable to calculate its result.
- The **Calculate** class provides a **Complete** event to notify the calling thread that the calculation is finished.

Creating and Using a Thread

The following example shows how to create a thread and use threading to perform calculations:

```
Sub Test( )
   Dim calc As New Calculate( )
   Dim th As New Threading.Thread(_
                AddressOf calc.LongCalculation)
   calc.iValue = 10
   AddHandler calc.Complete, AddressOf CalcResult
   th.Start( )
End Sub

Sub CalcResult(ByVal Result As Integer)
   'Perform appropriate action when calculation is finished
   ...
End Sub
```

When examining this code, note the following:

- The **Test** subroutine instantiates a **Calculate** object and specifies the **LongCalculation** delegate in the **Thread** constructor.
- A value is assigned to the *iValue* variable for use by the worker function.
- An event handler is created to detect completion of the calculation.
- The **Start** method is called on the separate thread to begin the processing of the calculation.

When to Use Threading

Topic Objective
To explain some of the potential problems caused by multithreading.

Lead-in
Using multiple threads requires you to think carefully about resources.

- **Use Threads Carefully**
  - Using more threads requires more system resources
- **Synchronize Access to Shared Resources**
  - Prevent two threads from accessing shared data simultaneously
  - Use **SyncLock** statement to block sections of code

```
Sub Worker( )
    SyncLock(theData)         'Lock this object variable
        theData.id = iValue
        'Perform some lengthy action
        iValue = theData.id
    End SyncLock              'Unlock the object variable
End Sub
```

Delivery Tip
Point out that incorrect use of threads can have serious consequences.

Using multiple threads is a useful programming concept in enterprise development; however, improper use of threads can cause performance problems, create inconsistent data, and cause other errors.

System Resources

Threads consume memory and other valuable resources, such as CPU processing time. If your application creates multiple threads, it may do so at the expense of other applications or other threads within your own process. The more threads you create, the longer the delay between CPU time slices for each thread. If all applications created an excessive number of threads and used them constantly, the system would spend most of its time swapping threads in and out of the CPU, since the thread scheduler itself requires the CPU to perform the swapping logic.

Shared Resources

If multiple threads need to access the same information at the same time, a concurrency problem may arise. Two threads accessing a shared global resource may get inconsistent results back from the resource if other threads have altered the data.

The following is an example of a situation in which this can occur:

- Thread A updates a value on a shared resource such as an integer, setting the value to 10 before performing some lengthy action.
- Thread B updates the same integer value to 15 during the delay of thread A's lengthy action.
- When this action is completed, thread A may read the integer value back from the resource whose value is now 15.

Synchronizing Shared Resources

You can avoid inconsistent results by locking the resource between the time that the value is initially set and the time that it is read back. You can use the **SyncLock** statement to lock a reference type such as a class, interface, module, array, or delegate.

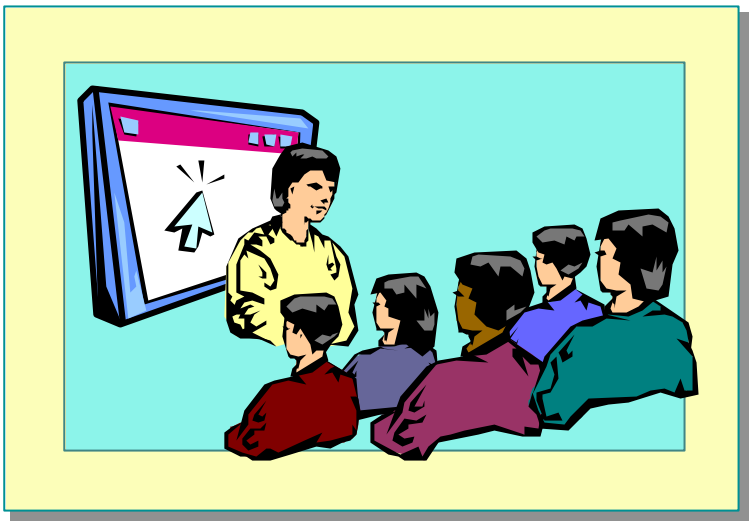
The following example defines a shared resource called **SharedReference** that exposes an integer variable. The **ThreadObj** class defines the method that will be executed by different threads. This method uses the **SyncLock** statement to lock the shared resource object while it is in use. The module code shows how you can test this behavior by creating two threads and two worker objects, and then starting both threads consecutively.

```
Imports System.Threading

'Shared data
Public Class SharedReference
    Public Id As Integer
End Class

'Class for running on other threads
Public Class ThreadObj
    Private sr As SharedReference
    Private Count As Integer

    'Constructor with reference and Id
    Public Sub New(ByRef sharedRef As SharedReference, _
                   ByVal ID As Integer)
        sr = sharedRef
        Count = ID
    End Sub

    'Actual worker method
    Public Sub RunMethod( )
        SyncLock (sr)   'Lock sr object
            sr.Id = Count

            'Execute lengthy code
            'sr.Id could have changed without SyncLock

            Count = sr.Id
        End SyncLock   'Release sr object lock
    End Sub
End Class

Module MainModule
    Sub Main( )
        'Create shared data object
        Dim sr As New SharedReference( )

        'Create two worker objects
        Dim worker1 As New ThreadObj(sr, 1)
        Dim worker2 As New ThreadObj(sr, 2)

        'Create two threads
        Dim t1 As New Thread(AddressOf worker1.RunMethod)
        Dim t2 As New Thread(AddressOf worker2.RunMethod)

        'Start both threads
        t1.Start( )
        t2.Start( )
    End Sub
End Module
```

Demonstration: Using the SyncLock Statement

Topic Objective
To demonstrate how to synchronize shared resources by using the **SyncLock** statement.

Lead-in
This demonstration shows how to use the **SyncLock** statement to synchronize a shared resource.



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 use the **SyncLock** statement when using multiple threads in an application created in Visual Basic .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.

- Components Overview
- Creating Serviced Components
- Creating Component Classes
- Creating Windows Forms Controls
- Creating Web Forms User Controls
- Threading

---

1. An unmanaged client application uses a class created in Visual Basic .NET but cannot access any methods of the class. What is the likely cause of this problem, and how would you fix it?

   The class may have public methods defined without using an interface or any class-level attributes. To solve this problem, create and implement methods in interfaces rather than classes, use the ClassInterface attribute, or use the COMClass attribute.

2. Modify the following code to use auto completion of transactions rather than the explicit **SetAbort** and **SetComplete** methods.

```
<Transaction(TransactionOption.Required)> _
Public Class TestClass
    Public Sub MySub( )
        Try
            'Perform action
            ContextUtil.SetComplete( )
        Catch ex As Exception
            ContextUtil.SetAbort( )
            Throw ex
        End Try
    End Sub
End Class
```

```
<Transaction(TransactionOption.Required)> _
Public Class TestClass
    <AutoComplete( )>Public Sub MySub( )
            'Perform action
    End Sub
End Class
```


3. Create assembly attributes so Component Services can automatically create an application named "TestComponents" that runs as server activation.

   <Assembly: ApplicationName("TestComponents")>

   <Assembly: ApplicationActivation(ActivationOption.Server)>

4. Why would you use the **IComponent** interface?

   The interface enables component classes to site other components and enables the component class to be sited on other components.

5. The following code causes a compilation error. Explain what is causing the error and how it could be fixed.

```
Sub Main( )
    Dim t As New Thread(AddressOf MySub)
    t.Start(10)
End Sub

Sub MySub(ByVal x As Integer)
    ...
End Sub
```

The MySub procedure cannot be called directly because it expects an argument and the Start method of a Thread cannot accept parameters. To fix this error, you could create the following code:

```
Sub Main( )
    Dim obj As New ThreadObj( )
    Dim t As New Thread(AddressOf obj.MySub)
    obj.x = 10
    t.Start( )
End Sub


Class ThreadObj
    Public x As Integer
    Sub MySub( )
        ...
    End Sub
End Class
```