msdn training

Module 14: Attributes

Contents

This course is based on the prerelease Beta 1 version of Microsoft® Visual Studio .NET. 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 1 version of Visual Studio .NET.

Microsoft

Information in this document is subject to change without notice. The names of companies, products, people, characters, and/or data mentioned herein are fictitious and are in no way intended to represent any real individual, company, product, or event, unless otherwise noted. Complying with all applicable copyright laws is the responsibility of the user. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation. If, however, your only means of access is electronic, permission to print one copy is hereby granted.

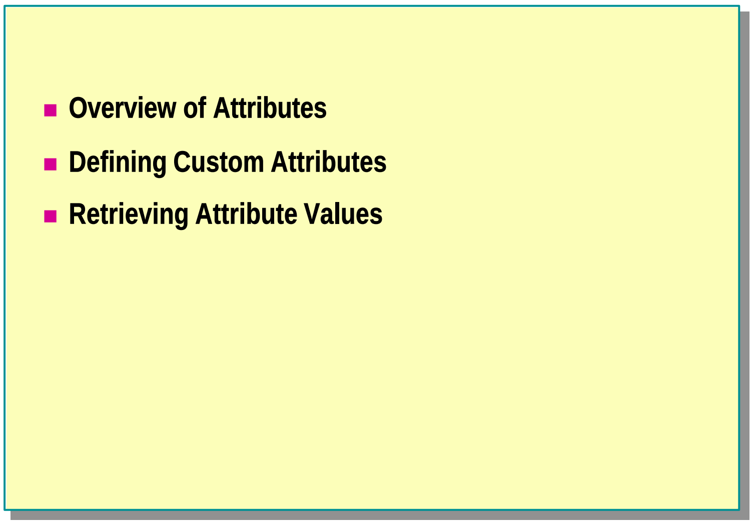
Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2001 Microsoft Corporation. All rights reserved.

Microsoft,ActiveX, BizTalk, IntelliSense, JScript, Microsoft Press, MSDN, PowerPoint, Visual Basic, Visual C++, Visual C#, Visual Studio, Windows, and Windows Media are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names mentioned herein may be the trademarks of their respective owners.

Overview



- Overview of Attributes
- Defining Custom Attributes
- Retrieving Attribute Values

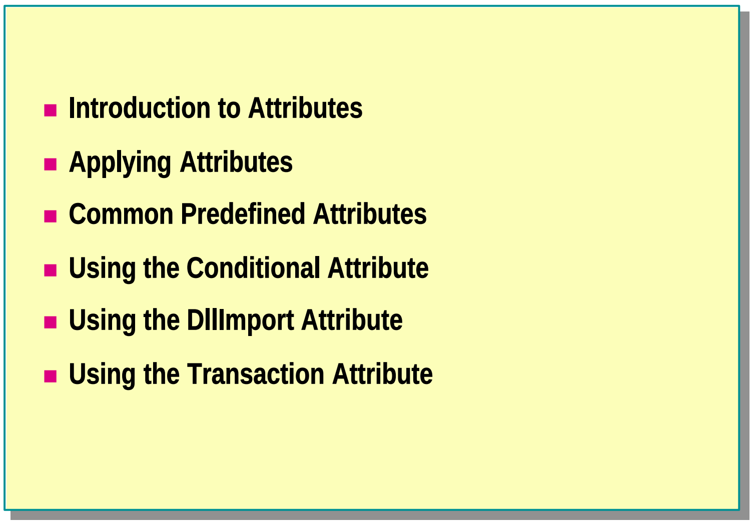
Attributes are a simple technique for adding metadata to classes. They can be useful when you need to build components.

In this module, you will learn the purpose of attributes and the function that they perform in C# applications. You will learn about attribute syntax and how to use some of the predefined attributes in the Microsoft® .NET environment. You will also learn to create custom user-defined attributes. Finally, you will learn how classes and other object types can implement and use these custom attributes to query attribute information at run time.

After completing this module, you will be able to:

- Use common predefined attributes.
- Create simple custom attributes.
- Query attribute information at run time.

◆ Overview of Attributes

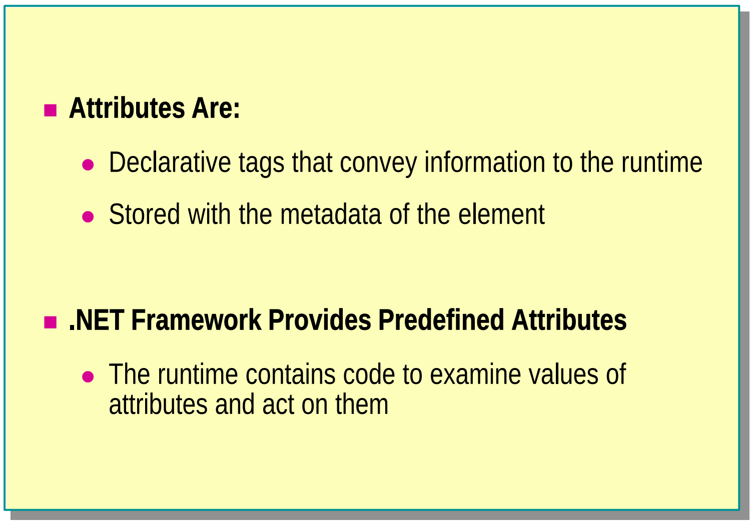


- Introduction to Attributes
- Applying Attributes
- Common Predefined Attributes
- Using the Conditional Attribute
- Using the DllImport Attribute
- Using the Transaction Attribute

With the introduction of attributes, the C# language provides a convenient technique that will help handle tasks such as changing the behavior of the runtime, obtaining transaction information about an object, conveying organizational information to a designer, and handling unmanaged code.

In this section you will learn what attributes are and which tasks you can perform with them. You will learn the syntax for using attributes in your code, and you will be introduced to some of the predefined attributes that are available in the .NET Framework.

Introduction to Attributes



- **Attributes Are:**
  - Declarative tags that convey information to the runtime
  - Stored with the metadata of the element
- **.NET Framework Provides Predefined Attributes**
  - The runtime contains code to examine values of attributes and act on them

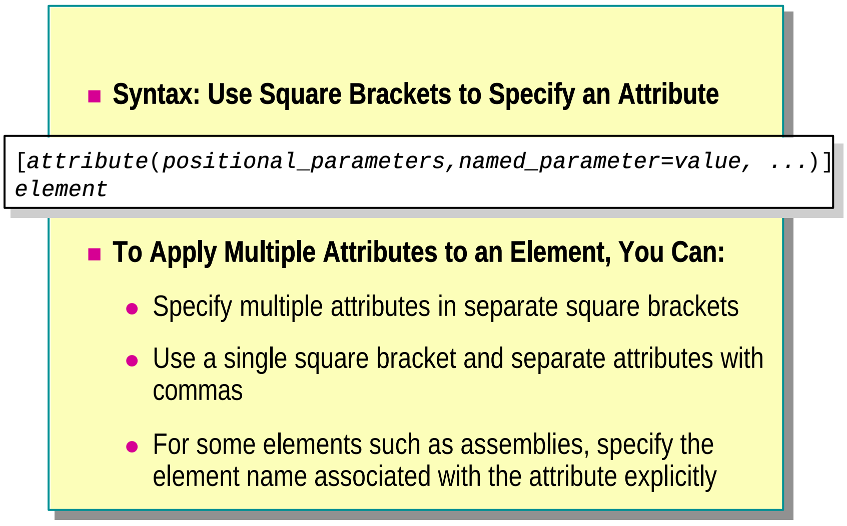
The .NET Framework provides attributes so that you can extend the capabilities of the C# language. An attribute is a declarative tag that you use to convey information to the runtime about the behavior of programmatic elements such as classes, data structures, enumerators, and assemblies.

You can think of attributes as annotations that your programs can store and use. In most cases, you write the code that retrieves the values of an attribute in addition to the code that performs a change in behavior at run time. In its simplest form, an attribute is an extended way to document your code.

You can apply attributes to many elements of the source code. Information about the attributes is stored with the metadata of the elements they are associated with.

The .NET Framework is equipped with a number of predefined attributes. The code to examine them and act upon the values they contain is also incorporated as a part of the runtime and .NET Framework SDK.

Applying Attributes



- **Syntax: Use Square Brackets to Specify an Attribute**

```
[attribute(positional_parameters,named_parameter=value, ...)]
element
```

- **To Apply Multiple Attributes to an Element, You Can:**
  - Specify multiple attributes in separate square brackets
  - Use a single square bracket and separate attributes with commas
  - For some elements such as assemblies, specify the element name associated with the attribute explicitly

You can apply attributes to different kinds of programming elements. These elements include assemblies, modules, classes, structs, enums, constructors, methods, properties, fields, events, interfaces, parameters, return values, and delegates.

Attribute Syntax

To specify an attribute and associate it with a programming element, use the following general syntax:

```
[attribute(positional_parameters,name_parameter=value, ...)]
element
```

You specify an attribute name and its values within square brackets ([and]) before the programmatic element to which you want to apply the attribute. Most attributes take one or more parameters, which can be either *positional* or *named*.

You specify a positional parameter in a defined position in the parameter list, as you would specify parameters for methods. Any named parameter values follow the positional parameters. Positional parameters are used to specify essential information, whereas named parameters are used to convey optional information in an attribute.

Tip Before using an unfamiliar attribute, it is a good practice to check the documentation for the attribute to find out which parameters are available and whether they should be positional or named.

Example

As an example of using attributes, consider the following code, in which the **DefaultEvent** attribute is applied on a class by using a positional **string** parameter, **ShowResult**:

```
[DefaultEvent("ShowResult")]
public class Calculator: System.WinForms.UserControl
{
   ...
}
```

Applying Multiple Attributes

You can apply more than one attribute to an element. You can enclose each attribute in its own set of square brackets, although you can also enclose multiple attributes, separated with commas, in the same set of square brackets.

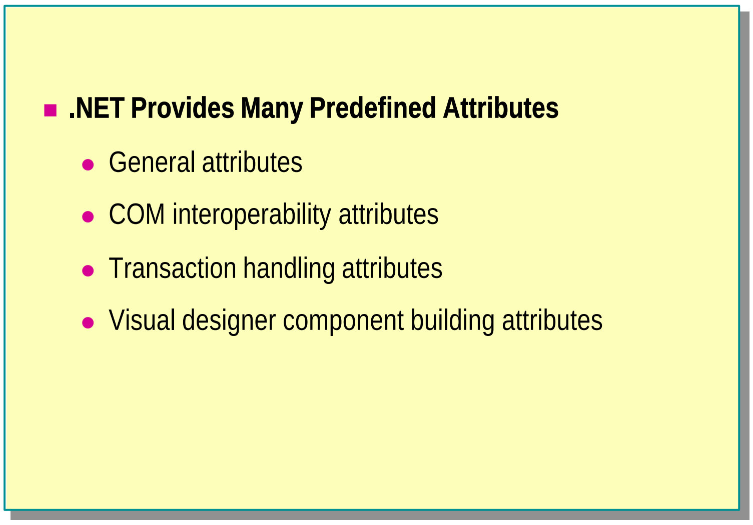
In some circumstances, you must specify exactly which element an attribute is associated with. For example, in the case of assembly attributes, place them after any **using** clauses but before any code, and explicitly specify them as attributes of the assembly.

The following example shows how to use the **CLSCompliant** assembly attribute. This attribute indicates whether or not an assembly strictly conforms to the Common Language Specification.

```
using System;
[assembly:CLSCompliant(true)]

class MyClass
{
   ...
}
```

Common Predefined Attributes



- **.NET Provides Many Predefined Attributes**
  - General attributes
  - COM interoperability attributes
  - Transaction handling attributes
  - Visual designer component building attributes

The capabilities of predefined attributes in the .NET Framework encompass a wide range of areas, from interoperability with COM to compatibility with visual design tools.

This topic describes some of the common predefined attributes that are provided by the .NET Framework. However, it is not intended to be comprehensive. For more information about predefined attributes, refer to the Microsoft Visual Studio.NET Help documents.

General Attributes

The following list summarizes some of the general attributes that are provided by the .NET Framework.

Attribute	Applicable to	Description
Conditional	Method	Tests to see whether a named symbol is defined. If it is defined, any calls to the method are executed normally. If the symbol is not defined, the call is not generated.
DllImport	Method	Indicates that the method is implemented in unmanaged code, in the specified DLL. It causes the DLL to be loaded at run time and the named method to execute.

COM Interoperability Attributes

When using the attributes to provide interoperability with COM, the goal is to ensure that using COM components from the managed .NET environment is as seamless as possible. The .NET Framework has many attributes relating to COM interoperability. Some of these are listed in the following table.

Attribute	Applicable to	Description
ComImport	Class	Indicates that a class or interface definition was imported from a COM type library.
ComRegisterFunction	Assembly	Specifies the method to be called when a .NET assembly is registered for use from COM.
ComUnregisterFunction	Assembly	Specifies the method to be called when a .NET assembly is unregistered for use from COM.
DispId	Method, property	Indicates which dispatch ID is to be used for the method or property.
HasDefaultInterface	Class	Indicates that the class has an explicit default COM interface.
In	Field, parameter	Indicates that the field or parameter is an input parameter.
MarshalAs	Field, parameter	Specifies how data should be marshaled between COM and the managed environment.
ProgId	Class	Specifies which prog ID is to be used for the class.
Out	Field, parameter	Indicates that data should be marshaled out from the callee back to caller.
InterfaceType	Interface	Specifies whether a managed interface is **IDispatch**, **IUnknown**, or dual when it is exposed to COM.

For more information about COM interoperability, search for "Microsoft ComServices" in the .NET Framework SDK Help documents.

Transaction Handling Attributes

Components running in a COM+ environment use transaction management. The attribute you use for this purpose is shown in the following table.

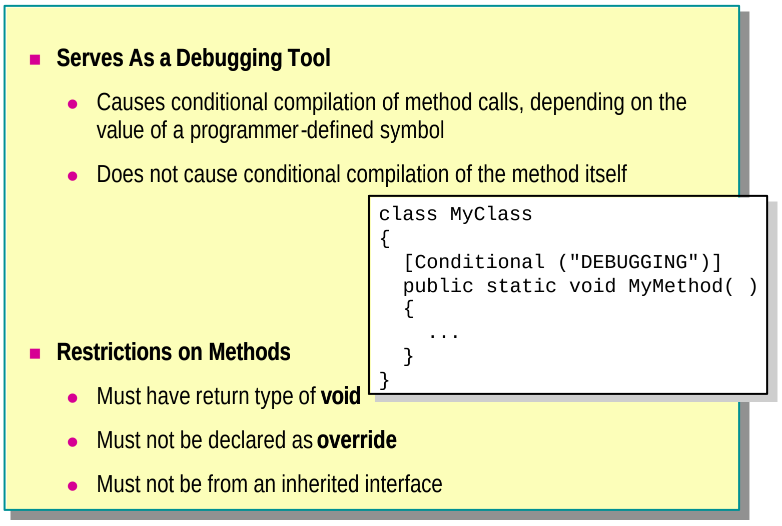
Attribute	Applicable to	Description
Transaction	Class	Specifies whether the component supports transactions, requires a transaction, should be invoked in the context of a new transaction, or whether transactions are ignored or unsupported.

Visual Designer Component-Building Attributes

Developers who build components for a visual designer use the attributes listed in the following table.

Attribute	Applicable to	Description
Bindable	Property	Specifies whether the property can be data-bound.
DefaultProperty	Class	Specifies the default property for the component.
DefaultValue	Property	Indicates that the property is the default value for the component.
Localizable	Property	Specifies that this property should be persisted to the resources file when forms are localized.
Persistable	Property	Indicates whether the property should be persisted and how it should be persisted.
DefaultEvent	Class	Specifies the default event for the component.
Browseable	Property, event	Indicates whether the property or event should be displayed in the property window of the visual designer.
Category	Property, event	Specifies the category into which the visual designer should place this property or event in the property window.
Description	Property, event	Defines a brief piece of text to be displayed at the bottom of the property window in the visual designer when this property or event is selected.

Using the Conditional Attribute



- **Serves As a Debugging Tool**
  - Causes conditional compilation of method calls, depending on the value of a programmer-defined symbol
  - Does not cause conditional compilation of the method itself

```
class MyClass
{
  [Conditional ("DEBUGGING")]
  public static void MyMethod( )
  {
    ...
  }
}
```

- **Restrictions on Methods**
  - Must have return type of **void**
  - Must not be declared as **override**
  - Must not be from an inherited interface

You can use the **Conditional** attribute as a debugging aid in your C# code. This attribute causes conditional compilation of method calls, depending on the value of a symbol that you define. It lets you invoke methods that, for example, display the values of variables, while you test and debug code. After you have debugged your program, you can “undefine” the symbol and recompile your code without changing anything else. (Or you can simply remove the symbol from the command line, and not change anything.)

Example

The following example shows how to use the **Conditional** attribute. In this example, the **MyMethod** method in **MyClass** is tagged with the **Conditional** attribute by the symbol DEBUGGING:

```
class MyClass
{
  [Conditional ("DEBUGGING")]
  public static void MyMethod( )
  {
    ...
  }
}
```

The symbol DEBUGGING is defined as follows:

```
#define DEBUGGING

class AnotherClass
{
  public static void Test( )
  {
    MyClass.MyMethod( );
  }
}
```

As long as the symbol DEBUGGING remains defined when the method call is compiled, the method call will operate normally. When DEBUGGING is undefined, the compiler will omit calls to the method. Therefore, when you run the program, it will be treated as though that line of code does not exist.

You can define the symbol in one of two ways. You can either add a **#define** directive to the code as shown in the preceding example, or define the symbol from the command line when you compile your program.

Restrictions on Methods

The methods to which you can apply a **Conditional** attribute are subject to a number of restrictions. In particular, they must have a return type of **void**, they must not be marked as **override**, and they must not be the implementation of a method from an inherited interface.

Note The **Conditional** attribute does not cause conditional compilation of the method itself. The attribute only determines the action that will occur when the method is called. If you require conditional compilation of a method, then you must use the **#if** and **#endif** directives in your code.

Using the DllImport Attribute

- **With the DllImport Attribute, You Can:**
  - Invoke unmanaged code in DLLs from a C# environment
  - Tag an external method to show that it resides in an unmanaged DLL

```
[DllImport("MyDLL.dll", EntryPoint="MessageBox")]
public static extern int MyFunction(string param1);

public class MyClass( )
{
  ...
  int result = MyFunction("Hello Unmanaged Code");
  ...
}
```

You can use the **DllImport** attribute to invoke unmanaged code in your C# programs. *Unmanaged code* is the term used for code that has been developed outside the .NET environment (that is, standard C compiled into DLL files). By using the **DllImport** attribute, you can invoke unmanaged code residing in dynamic-link libraries (DLLs) from your managed C# environment.

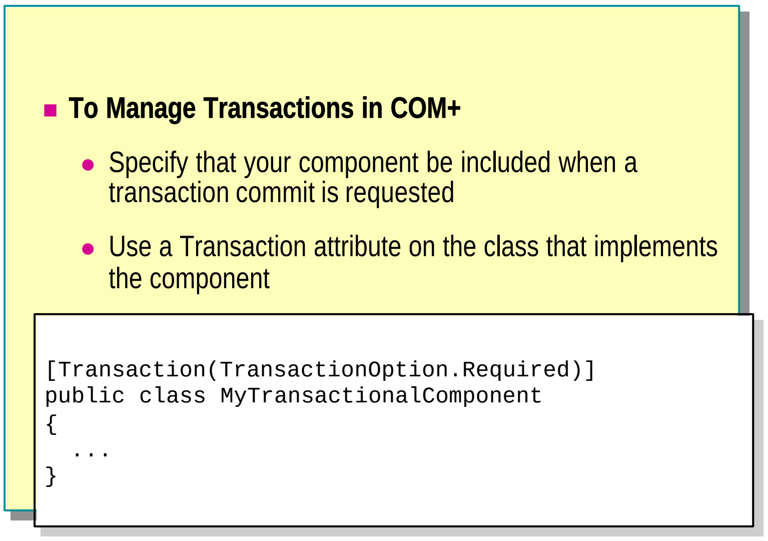
Invoking Unmanaged Code

The **DllImport** attribute allows you to tag an **extern** method as residing in an unmanaged DLL. When your code calls this method, the Common Language Runtime locates the DLL, loads it into the memory of your process, marshals parameters as necessary, and transfers control to the address at the beginning of the unmanaged code. This is unlike a normal program, which does not have direct access to the memory that is allocated to it. The following code provides an example of how to invoke unmanaged code:

```
[DllImport("MyDLL.dll", EntryPoint="MessageBox")]
public static extern int MyFunction(string param1);

public class MyClass( )
{
  ...
  int result = MyFunction("Hello Unmanaged Code");
  ...
}
```

Using the Transaction Attribute



- **To Manage Transactions in COM+**
  - Specify that your component be included when a transaction commit is requested
  - Use a Transaction attribute on the class that implements the component

```
[Transaction(TransactionOption.Required)]
public class MyTransactionalComponent
{
  ...
}
```

It is likely that, as a Microsoft Visual Basic® or C++ developer working in a Microsoft environment, you are familiar with technologies such as COM+. An important feature of COM+ is that it allows you to develop components that can participate in distributed transactions, which are transactions that can span multiple databases, machines, and components.

Managing Transactions in COM+

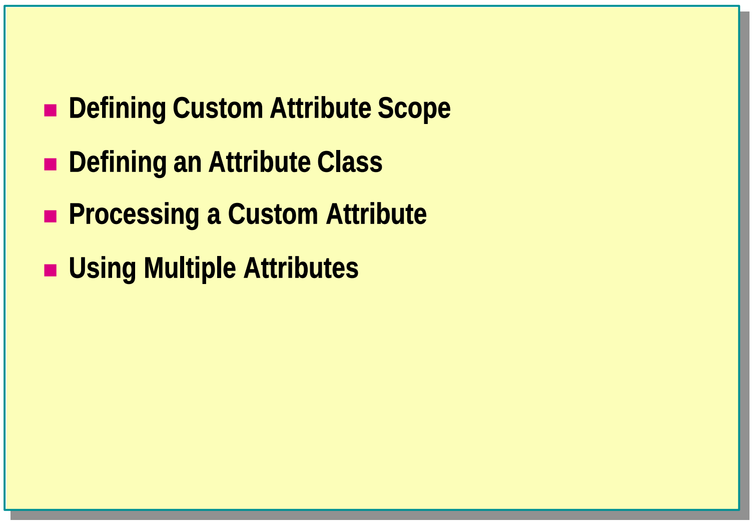
Writing code to guarantee a correct transaction commit in a distributed environment is difficult. However, if you use COM+, it takes care of managing the transactional integrity of the system and coordinating events on the network.

In this case, you only need to specify that your component be included when an application that uses your component requests a transaction commit. To make this specification, you can use a **Transaction** attribute on the class that implements the component, as follows:

```
[Transaction(TransactionOption.Required)]
public class MyTransactionalComponent
{
  ...
}
```

The **Transaction** attribute is one of the predefined .NET Framework attributes that the .NET runtime interprets automatically.

◆ Defining Custom Attributes



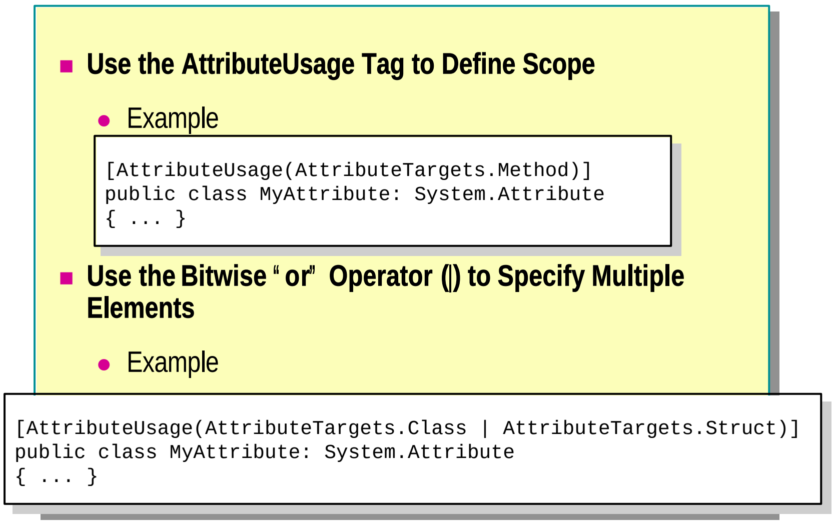
- Defining Custom Attribute Scope
- Defining an Attribute Class
- Processing a Custom Attribute
- Using Multiple Attributes

When you encounter a situation in which none of the predefined .NET Framework attributes satisfy your requirements, you can create your own attribute. Such a custom attribute will provide properties that allow you to store and retrieve information from the attribute.

Like predefined attributes, custom attributes are objects that are associated with one or more programmatic elements. They are stored with the metadata of their associated elements, and they provide mechanisms for a program to retrieve their values.

In this section, you will learn how to define and use your own custom attributes.

Defining Custom Attribute Scope



- **Use the AttributeUsage Tag to Define Scope**
  - Example

```
[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute: System.Attribute
{ ... }
```

- **Use the Bitwise "or" Operator (|) to Specify Multiple Elements**
  - Example

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttribute: System.Attribute
{ ... }
```

As with some predefined attributes, you must explicitly specify the programming element to which you want to apply a custom attribute. To do so, you annotate your custom attribute with an **AttributeUsage** tag as shown in the following example:

```
[AttributeUsage(target_elements)]
public class MyAttribute: System.Attribute
{ ... }
```

Defining Attribute Scope

The parameter to **AttributeUsage** contains values from the **System.AttributeTargets** enumeration to specify how the custom attribute can be used. The members of this enumeration are summarized in the following table.

Member name	Attribute can be applied to
Class	class
Constructor	constructor
Delegate	delegate
Enum	enum
Event	event
Field	field
Interface	interface
Method	method
Module	module

(*continued*)

Member name	**Attribute can be applied to**
Parameter	parameter
Property	property
ReturnValue	return value
Struct	struct
Assembly	assembly
ClassMembers	class, struct, enum, constructor, method, property, field, event, delegate, interface
All	Any element

Example of Using Custom Attributes

To specify that the **MyAttribute** custom attribute can be applied only to methods, use the following code:

```
[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute: System.Attribute
{
...
}
```

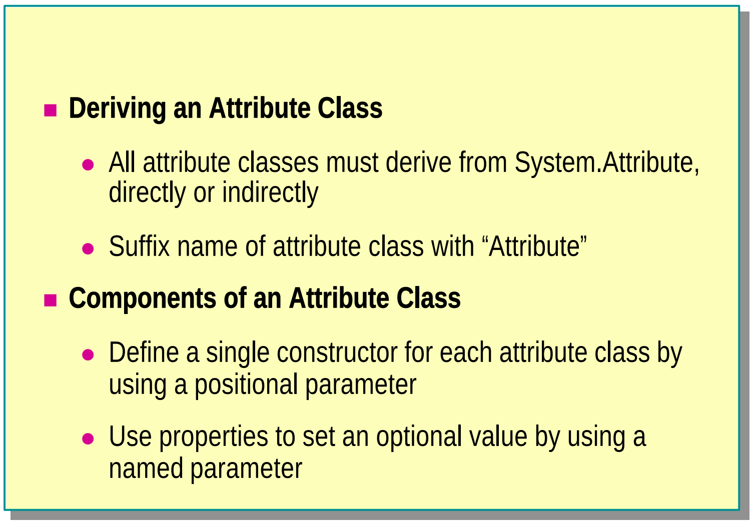
Specifying Multiple Elements

If the attribute can be applied to more than one element type, use the bitwise “or” operator (|) to specify multiple target types. For example, if **MyAttribute** can also be applied to constructors, the earlier code will be modified as follows:

```
[AttributeUsage(AttributeTargets.Method |
➥AttributeTargets.Constructor)]
public class MyAttribute: System.Attribute
{
...
}
```

If a developer attempts to use the **MyAttribute** in a context different from that which is defined by **AttributeUsage**, the developer’s code will not compile.

Defining an Attribute Class



- **Deriving an Attribute Class**
  - All attribute classes must derive from System.Attribute, directly or indirectly
  - Suffix name of attribute class with “Attribute”
- **Components of an Attribute Class**
  - Define a single constructor for each attribute class by using a positional parameter
  - Use properties to set an optional value by using a named parameter

After you define the scope of a custom attribute, you need to specify the way you want the custom attribute to behave. For this purpose, you must define an attribute class. Such a class will define the name of the attribute, how it can be created, and the information that it will store.

The .NET Framework SDK provides a base class, **System.Attribute**, that you must use to derive custom attribute classes and to access the values held in custom attributes.

Deriving an Attribute Class

All custom attribute classes must derive from **System.Attribute**, either directly or indirectly. The following code provides an example:

```
public class DeveloperInfoAttribute: System.Attribute
{
    ...
    public DeveloperInfoAtribute(string developer, string date)
    public(string Date)
    {
         get { ... }
         set { ... }
    }
}
```

It is a good practice to append the name of a custom attribute class with the suffix “ Attribute,” as in **DeveloperInfoAttribute**. This makes it easier to distinguish the attribute classes from the non-attribute classes.

Components of an Attribute Class

All attribute classes must have a constructor. For example, if the **DeveloperInfo** attribute expects the name of the developer as a string parameter, it must have a constructor that accepts a string parameter.

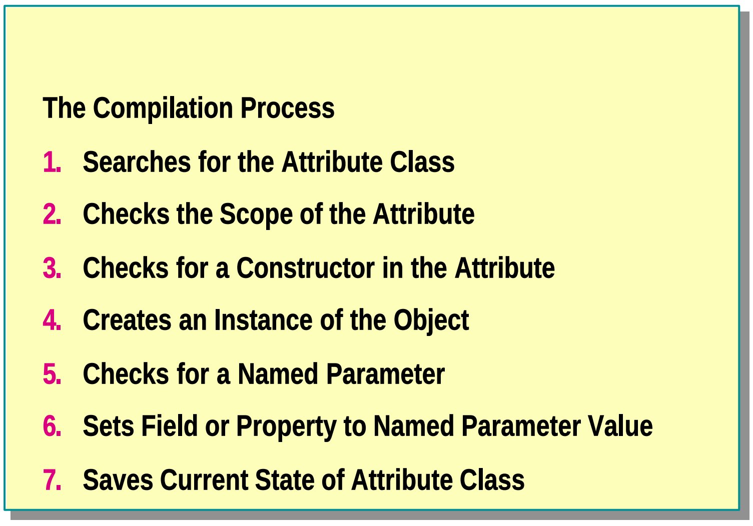
A custom attribute must define a single constructor that sets the mandatory information. The positional parameter or parameters of the attribute pass this information to the constructor. If an attribute has optional data, then it is attempting to overload the constructor. This is not a good practice to adopt. Use named parameters to provide optional data.

An attribute class can, however, provide properties to get and set data. Therefore, you must use properties to set optional values, if required. Then a developer can specify the optional values as named parameters when using the attribute.

For example, the **DeveloperInfoAttribute** provides a **Date** property. You can call the **set** method of the **Date** property to set the named parameter: *Date*. The developer name, *Bert*, for example, is the positional parameter that is passed to the constructor of the attribute:

```
[DeveloperInfoAttribute("Bert", Date="11-11-2000")]
public class MyClass
{
   ...
}
```

Processing a Custom Attribute



The Compilation Process

1. **Searches for the Attribute Class**
2. **Checks the Scope of the Attribute**
3. **Checks for a Constructor in the Attribute**
4. **Creates an Instance of the Object**
5. **Checks for a Named Parameter**
6. **Sets Field or Property to Named Parameter Value**
7. **Saves Current State of Attribute Class**

When the compiler encounters an attribute on a programming element, the compiler uses the following process to determine how to apply the attribute:

1. Searches for the attribute class
2. Checks the scope of the attribute
3. Checks for a constructor in the attribute
4. Creates an instance of the object
5. Checks for a named parameter
6. Sets the field or property to a named parameter value
7. Saves the current state of the attribute class

To be completely accurate, the compiler actually verifies that it *could* apply the attribute, and then stores the information to do so in the metadata. The compiler does not create attribute instances at compile time.

Example

To learn more about how the compiler handles attributes, consider the following example:

```
[AttributeUsage(AttributeTargets.Class)]
public class DeveloperInfoAttribute: System.Attribute
{
   ...
}
.....
{
.....
}

[DeveloperInfo("Bert", Date="11-11-2000")]
public class MyClass
{
...
}
```

Note As is mentioned in the previous topic, it is a good practice to add the suffix “ Attribute” to the name of an attribute class. Strictly speaking, it is not necessary to do so. Even if you omit the Attribute suffix as shown in the example, your code will still compile correctly. However, without the Attribute suffix there are potential issues concerning how the compiler searches for classes. Always use the Attribute suffix.

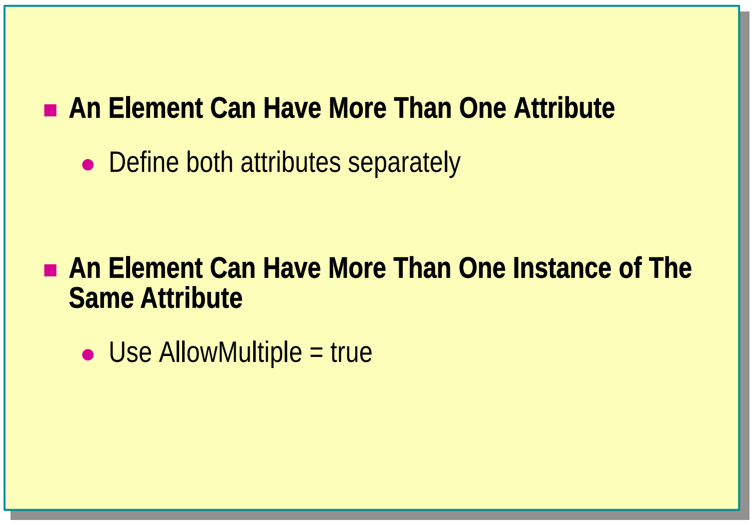
The Compilation Process

In the preceding example, when **MyClass** is compiled, the compiler will search for an attribute class called **DevloperInfoAttribute**. If the class cannot be located, the compiler will then search for **DeveloperInfo**.

After it finds **DeveloperInfo**, the compiler will check whether the attribute is allowed on a class. Then it will check for a constructor that matches the parameters specified in the attribute use. If it finds one, it creates an instance of the object by calling the constructor with the specified values.

If there is a named parameter, the compiler matches the name of the parameter with a field or property in the attribute class, and then sets the field or property to the specified value. Then the current state of the attribute class is saved to the metadata for the program element on which it is applied.

Using Multiple Attributes



- **An Element Can Have More Than One Attribute**
  - Define both attributes separately
- **An Element Can Have More Than One Instance of The Same Attribute**
  - Use AllowMultiple = true

You can apply more than one attribute to a programming element, and you can use multiple instances of the same attribute in an application.

Using Multiple Attributes

You can apply more than one attribute to a programming element. For example, the following code shows how you can tag the **FinancialComponent** class, a Microsoft Windows® control, with two attributes: **Transaction** and **DefaultProperty**:

```
[Transaction(TransactionOption.Required)]
[DefaultProperty("Balance")]
public class FinancialComponent: System.WinForms.UserControl
{
  ...
  public long Balance
  {
    ...
  }
}
```

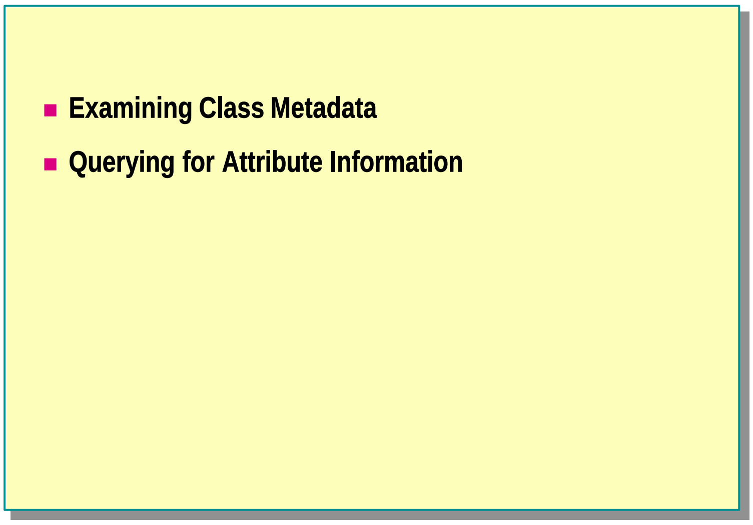
Using the Same Attribute Multiple Times

The default behavior of a custom attribute does not permit multiple instances of the attribute. However, under some circumstances it might make sense to allow an attribute to be used on the same element more than once.

An example of this is the custom attribute **DeveloperInfo**. This attribute allows you to record the name of the developer that wrote a class. If more than one developer was involved in the development, you need to use the **DeveloperInfo** attribute more than once. For an attribute to permit this, you must mark it as **AllowMultiple** in the **AttributeUsage** attribute, as follows:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
public class DeveloperInfoAttribute: System.Attribute
{
   ...
}
```

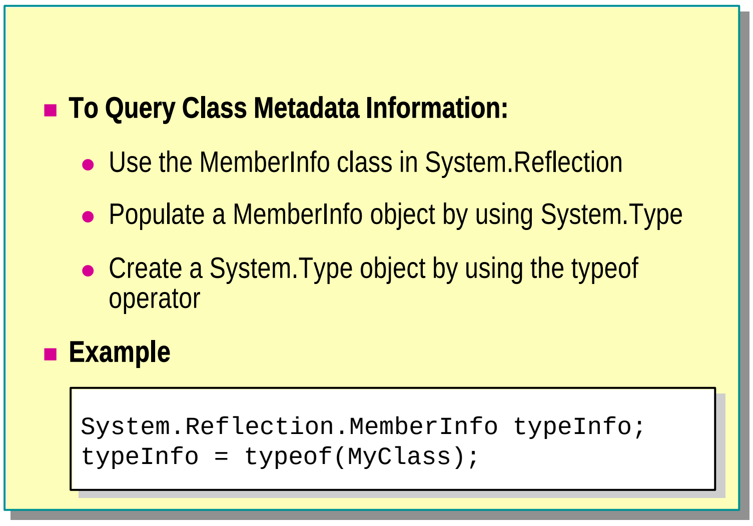
◆ Retrieving Attribute Values



- Examining Class Metadata
- Querying for Attribute Information

After you have applied attributes to programming elements in your code, it is useful to be able to determine the values of the attributes. In this section, you will learn how to use reflection to examine the attribute metadata of a class and query classes for attribute information.

Examining Class Metadata



- **To Query Class Metadata Information:**
  - Use the MemberInfo class in System.Reflection
  - Populate a MemberInfo object by using System.Type
  - Create a System.Type object by using the typeof operator
- **Example**

```
System.Reflection.MemberInfo typeInfo;
typeInfo = typeof(MyClass);
```

The .NET runtime supplies a mechanism called *reflection* that allows you to query information held in metadata. Metadata is where attribute information is stored.

Using the MemberInfo Class

The .NET Framework provides a namespace named **System.Reflection**, which contains classes that you can use for examining metadata. One particular class in this namespace—the **MemberInfo** class—is very useful if you need to find out about the attributes of a class.

To populate a **MemberInfo** array, you can use the **GetMembers** method of the **System.Type** object. To create this object, you use the **typeof** operator with a class or any other element, as shown in the following code:

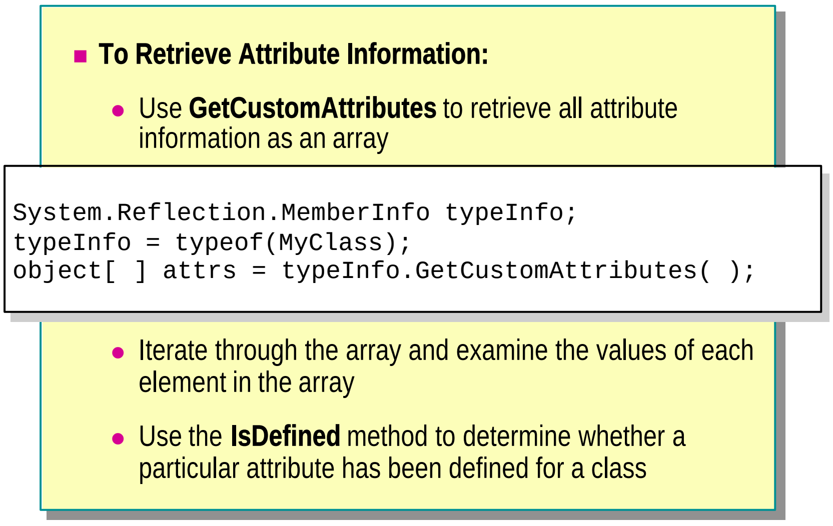
```
System.Reflection.MemberInfo[ ] memberInfoArray;
memberInfoArray = typeof(MyClass).GetMembers( );
...
```

Once created, the *typeInfo* variable can be queried for metadata information about the class **MyClass**.

Tip If you need more detailed information, for example, if you want to discover the values of attributes that a method has, you can use a **MethodInfo** object. In addition, there are other "Info" classes: **ConstructorInfo**, **EventInfo**, **FieldInfo**, **ParameterInfo**, and **PropertyInfo**. Detailed information about how to use these classes is beyond the scope of this course, but you can find out more by searching for "System.Reflection namespace" in the .NET Framework SDK Help documents.

Note **MemberInfo** is actually the abstract base class of the other "Info" types.

Querying for Attribute Information



- **To Retrieve Attribute Information:**
  - Use **GetCustomAttributes** to retrieve all attribute information as an array

```
System.Reflection.MemberInfo typeInfo;
typeInfo = typeof(MyClass);
object[ ] attrs = typeInfo.GetCustomAttributes( );
```

  - Iterate through the array and examine the values of each element in the array
  - Use the **IsDefined** method to determine whether a particular attribute has been defined for a class

After you create the *typeInfo* variable, you can query it to get information about the attributes applied to its associated class.

Retrieving Attribute Information

The **MemberInfo** object has a method called **GetCustomAttributes**. This method retrieves the information about all attributes of a class and stores it in an array, as shown in the following code:

```
object [ ] attrs = typeInfo.GetCustomAttributes( );
```

You can then iterate through the array to find the values of the attributes that you are interested in.

Iterating Through Attributes

You can iterate through the array of attributes and examine the value of each one in turn. In the following code, the only attribute of interest is **DeveloperInfoAttribute**, and all the others are ignored. For each **DeveloperInfoAttribute** found, the values of the **Developer** and **Date** properties are displayed as follows:

```
...
object [ ] attrs = typeInfo.GetCustomAttributes( );
foreach(Attribute atr in attrs) {
  if (atr is DeveloperInfoAttribute) {
    DeveloperInfoAttribute dia = (DeveloperInfoAttribute)atr;
    Console.WriteLine("{0}  {1}", dia.Developer, dia.Date);
  }
}
...
```

Tip **GetCustomAttributes** is an overloaded method. If you only want values for that one attribute type, you can invoke this method by passing the type of the custom attribute you are looking for through it, as shown in the following code:

```
object [ ] attrs =
typeInfo.GetCustomAttributes(typeof(DeveloperInfoAttribute));
```

Using the IsDefined Method

If there are no matching attributes for a class, **GetCustomAttributes** returns a **null** object reference. However, to find out whether a particular attribute has been defined for a class, you can use the **IsDefined** method of **MemberInfo** as follows:

```
Type devInfoAttrType = typeof(DeveloperInfoAttribute);
if (typeInfo.IsDefined(devInfoAttrType) {
  Object [ ] attrs =
        typeInfo.GetCustomAttributes(devInfoAttrType);
  ...
}
```

Note You can use Intermediate Language Disassembler (ILDASM) to see these attributes inside the assembly.

Lab 14: Defining and Using Attributes

Objectives

After completing this lab, you will be able to:

- Use the predefined **Conditional** attribute.
- Create a custom attribute.
- Add a custom attribute value to a class.
- Use reflection to query attribute values.

Prerequisites

Before working on this lab, you should be familiar with the following:

- Creating classes in C#
- Defining constructors and methods
- Using the **typeof** operator
- Using properties and indexers in C#

Estimated time to complete this lab: 45 minutes

Exercise 1
Using the Conditional Attribute

In this exercise, you will use the predefined **Conditional** attribute to conditionally execute your code.

Conditional execution is a useful technique if you want to incorporate testing or debugging code into a project but do not want to edit the project and remove the debugging code after the system is complete and functioning correctly.

During this exercise, you will add a method called **DumpToScreen** to the **BankAccount** class (which was created in earlier labs). This method will display the details of the account. You will use the **Conditional** attribute to execute this method depending on the value of a symbol called **DEBUG_ACCOUNT**.

✓ To apply the Conditional attribute

1. Open the Audit.sln project in the *install folder*\Labs\Lab14\Starter\Bank folder.
2. In the **BankAccount** class, add a public void method called **DumpToScreen** that takes no parameters.

   The method must display the contents of the account: account number, account holder, account type, and account balance. The following code shows a possible example of the method:

   ```
   public void DumpToScreen( )
   {
     Console.WriteLine("Debugging account {0}. Holder is {1}.
   ➥Type is {2}. Balance is {3}",
        this.accNo, this.holder, this.accType, this.accBal);
   }
   ```

3. Make use of the method’s dependence on the **DEBUG_ACCOUNT** symbol.

   Add the following **Conditional** attribute before the method as follows:

   ```
   [conditional("DEBUG_ACCOUNT")]
   ```

4. Compile your code and correct any errors.

To test the Conditional attribute

1. Open the TestHarness.sln project in the *install folder*\ Labs\Lab14\Starter\TestHarness folder.
2. Add a reference to the **Bank** library.
   a. In Solution Explorer, expand the **TestHarness** tree.
   b. Right-click **References**, and then click **Add Reference**.
   c. Click **Browse**, and then navigate to *install folder*\ Labs\Lab14\Starter\Bank\Bin\Debug.
   d. Click **Bank.dll**, click **Open**, and then click **OK**.
3. Review the **Main** method of the **CreateAccount** class. Notice that it creates a new bank account.
4. Add the following line of code to **Main** to call the **DumpToScreen** method of **myAccount**:

   ```
   myAccount.DumpToScreen( );
   ```

5. Save your work, compile the project, and correct any errors.
6. Run the test harness.

   Notice that nothing happens. This is because the **DumpToScreen** method has not been called.
7. Use the ILDASM utility (**ildasm**) from the command line to examine *install folder*\Labs\Lab14\Starter\Bank\Bin\Debug\Bank.dll.

   You will see that the **DumpToScreen** method is present in the **BankAccount** class.
8. Double-click the **DumpToScreen** method to display the Microsoft intermediate language (MSIL) code.

   You will see the **Conditional** attribute at the beginning of the method. The problem is in the test harness. Because of the **Conditional** attribute on **DumpToScreen**, the runtime will effectively ignore calls made to that method if the **DEBUG_ACCOUNT** symbol is not defined when the calling program is compiled. The call is made, but because **DEBUG_ACCOUNT** is not defined, the runtime finishes the call immediately.
9. Close ILDASM.
10. Return to the test harness. At the top of the CreateAccount.cs file, before the first **using** directive, add the following code:

    ```
    #define DEBUG_ACCOUNT
    ```

    This defines the **DEBUG_ACCOUNT** symbol.
11. Save and compile the test harness, correcting any errors.
12. Run the test harness.

    Notice that the **DumpToScreen** method displays the information from **myAccount**.

Exercise 2
Defining and Using a Custom Attribute

In this exercise, you will create a custom attribute called **DeveloperInfoAttribute**. This attribute will allow the name of the developer and, optionally, the creation date of a class to be stored in the metadata of that class. This attribute will permit multiple use because more than one developer might be involved in the coding of a class.

You will then write a method that retrieves and displays all of the **DevloperInfoAttribute** values for a class.

✓ To define a custom attribute class

1. Using Visual Studio.NET, create a new Microsoft Visual C#™ project, using the information shown in the following table.

Element	Value
Project Type	Visual C# Projects
Template	Class Library
Name	CustomAttribute
Location	*install folder*\Labs\Lab14\Starter

2. Change the name and file name of class **Class1** to **DeveloperInfoAttribute**.

   Make sure that you also change the name of the constructor.

3. Specify that the **DeveloperInfoAttribute** class is derived from **System.Attribute**.

   This attribute will be applicable to classes, enums, and structs only. It will also be allowed to occur more than once when it is used.

4. Add the following **AttributesUsage** attribute before the class definition:

```
[AttributeUsage(AttributeTargets.Class |
➥AttributeTargets.Enum | AttributeTargets.Struct,
➥AllowMultiple=true)]
```

5. Document your attribute with a meaningful summary (between the <summary> tags). Use the exercise description to help you.

6. The **AttributesUsage** attribute requires the name of the developer of the class as a mandatory parameter and takes the date that the class was written as an optional string parameter. Add private instance variables to hold this information, as follows:

```
private string developerName;
private string dateCreated;
```

7. Modify the constructor so that it takes a single string parameter that is also called **developerName**, and add a line of code to the constructor that assigns this parameter to **this.developerName**.

8. Add a public **string** read-only property called **Developer** that can be used to **get** the value of **developerName**. Do not write a **set** method.

9. Add another public **string** property that is called **Date**. This property should have a **get** method that reads **dateCreated** and a **set** method that writes **dateCreated**.
10. Compile the class and correct any errors.

    Because the class is in a class library, the compilation process will produce a DLL (CustomAttribute.dll) rather than a stand-alone executable program. The complete code for the **DeveloperInfoAttribute** class follows:

```
namespace CustomAttribute
{
  using System;
  /// <summary>
  ///  This class is a custom attribute that allows
  ///  the name of the developer of a class to be stored
  ///  with the metadata of that class.
  /// </summary>
  [AttributeUsage(AttributeTargets.Class |
  ➥AttributeTargets.Enum | AttributeTargets.Struct,
  ➥AllowMultiple=true)]
  public class DeveloperInfoAttribute: System.Attribute

  {
    private string developerName;
    private string dateCreated;

    // Constructor. Developer name is the only
    // mandatory parameter for this attribute.
    public DeveloperInfoAttribute(string developerName)
    {
      this.developerName = developerName;
    }
    public string Developer
    {
      get
      {
        return developerName;
      }
    }

    // Optional parameter
    public string Date
    {
      get
      {
        return dateCreated;
      }
      set
      {
        dateCreated = value;
      }
    }
  }
}
```

⇙ To add a custom attribute to a class

1. You will now use the **DeveloperInfo** attribute to record the name of the developer of the **Rational** number class. (This class was c reated in an earlier lab, but it is provided here for your convenience.) Open the Rational.sln project in the *install folder*\Labs\Lab14\Starter\Rational folder.
2. Perform the following steps to add a reference to the **CustomAttribute** library that you created earlier:
   a. In Solution Explorer, expand the **Rational** tree.
   b. Right-click **References**, and then click **Add Reference**.
   c. In the **Add Reference** dialog box, click **Browse**.
   d. Navigate to the *install folder*\Labs\Lab14\Starter\CustomAttribute\Bin\Debug folder, and click **CustomAttribute.dll**.
   e. Click **Open**, and then click **OK**.
3. Add a **CustomAttribute.DeveloperInfo** attribute to the **Rational** class, specifying your name as the developer and the current date as the optional date parameter, as follows:

```
[CustomAttribute.DeveloperInfo("Your Name", ➥Date="Today")]
```

4. Add a second developer to the **Rational** class.
5. Compile the **Rational** project and correct any errors.
6. Open a Command window and navigate to the *install folder*\Labs\Lab14\Starter\Rational\Bin\Debug folder.

   This folder should contain your Rational.exe executable.
7. Run ILDASM and open Rational.exe.
8. Expand the **Rational** namespace in the tree view.
9. Expand the **Rational** class.
10. Near the top of the class, notice your custom attribute and the values that you supplied.
11. Close ILDASM.

To use reflection to query attribute values

Using ILDASM is only one way to examine attribute values. You can also use reflection in C# programs. Return to Visual Studio, and edit the **TestRational** class in the **Rational** project.

1. In the **Main** method, create a variable called *attrInfo* of type **System.Reflection.MemberInfo**, as shown in the following code:

```
public static void Main( )
{
  System.Reflection.MemberInfo attrInfo;
...
```

2. You can use a **MemberInfo** object to hold information about the members of a class. Assign the **Rational** type to the **MemberInfo** object by using the **typeof** operator, as follows:

```
  attrInfo = typeof(Rational);
```

3. The attributes of a class are held as part of the class information. You can retrieve the attribute values by using the **GetCustomAttributes** method. Create an object array called **attrs**, and use the **GetCustomAttributes** method of **attrInfo** to find all of the custom attributes used by the **Rational** class, as shown in the following code:

```
  object[ ] attrs = attrInfo.GetCustomAttributes( );
```

4. Now you need to extract the attribute information that is stored in the **attrs** array and print it. Create a variable called *developerAttr* of type **CustomAttribute**.**DeveloperInfo**, and assign it the first element in the **attrs** array, casting as appropriate, as shown in the following code:

```
  CustomAttribute.DeveloperInfoAttribute developerAttr;
  developerAttr =
    ➥(CustomAttribute.DeveloperInfoAttribute)attrs[0];
```

Note In production code, you would use reflection rather than a cast to determine the type of the attribute.

5. Use the **get** methods of the **DeveloperInfoAttribute** attribute to retrieve the **Developer** and **Date** attributes and print them out as follows:

```
  Console.WriteLine("Developer: {0}\tDate: {1}",
    ➥developerAttr.Developer, developerAttr.Date);
```

6. Repeat steps 4 and 5 for element 1 of the **attrs** array.

   You can use a loop if you want to be able to retrieve the values of more than two attributes.

7. Compile the project and correct any errors.

   The completed code for the **Main** method is shown in the following code:

```
namespace Rational
{
using System;

// Test harness
public class TestRational
{
  public static void Main( )
  {
    System.Reflection.MemberInfo attrInfo;
    attrInfo = typeof(Rational);
    object [ ] attrs = attrInfo.GetCustomAttributes( );
    CustomAttribute.DeveloperInfoAttribute developerAttr;
    developerAttr =
      ➥(CustomAttribute.DeveloperInfoAttribute)attrs[0];
    Console.WriteLine("Developer: {0}\tDate: {1}",
      ➥developerAttr.Developer, developerAttr.Date);
    developerAttr =
      ➥(CustomAttribute.DeveloperInfoAttribute)attrs[1];
    Console.WriteLine("Developer: {0}\tDate: {1}",
      ➥developerAttr.Developer, developerAttr.Date);
  }
}
}
```

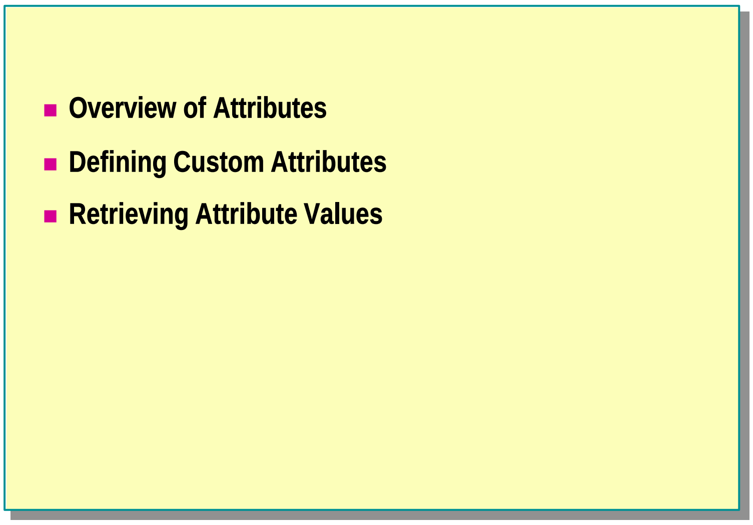
   Here is an alternative **Main** that uses a `foreach` loop:

```
  public static void Main( )
  {
    System.Reflection.MemberInfo attrInfo;
    attrInfo = typeof(Rational);
    object[ ] attrs = attrInfo.GetCustomAttributes( );

    foreach (CustomAttribute.DeveloperInfoAttribute
      ➥ devAttr in attrs)
    {
      Console.WriteLine("Developer: {0}\tDate: {1}",
        ➥devAttr.Developer, devAttr.Date);
    }
  }
```

8. When you run this program, it will display the names and dates that you supplied as **DeveloperInfoAttribute** information to the **Rational** class.

Review



- Overview of Attributes
- Defining Custom Attributes
- Retrieving Attribute Values

1. Can you tag individual objects by using attributes?

2. Where are attribute values stored?

3. What mechanism is used to determine the value of an attribute at run time?

4. Define an attribute class called **CodeTestAttributes** that is applicable only to classes. It should have no positional parameters and two named parameters called **Reviewed** and **HasTestSuite**. These parameters should be of type **bool** and should be implemented by using read/write properties.

5. Define a class called **Widget**, and use **CodeTestAttributes** from the previous question to mark that **Widget** has been reviewed but has no test suite.

6. Suppose that **Widget** from the previous question had a method called **LogBug**. Could **CodeTestAttributes** be used to mark only this method?