Module Overview

Module Overview

- Examining Object Metadata
- Creating and Using Custom Attributes
- Generating Managed Code
- Versioning, Signing, and Deploying Assemblies

Overview

Systems often consist of many components, some of which may be shared with other applications in your organization's infrastructure. When you design applications, it is important to think about reuse and consider how the functionality you are implementing might be useful in other applications.

In this module, you will learn how to consume existing assemblies by using reflection and how to add additional metadata to types and type members by using attributes. You will also learn how to generate code at run time by using the Code Document Object Model (CodeDOM) and how to ensure that your assemblies are signed and versioned, and available to other applications, by using the global assembly cache (GAC).

Objectives

After completing this module, you will be able to:

- Use reflection to inspect and execute assemblies.
- Create and consume custom attributes.
- Generate managed code at run time by using CodeDOM.
- Version, sign, and deploy your assemblies to the GAC.

Lesson 1: Examining Object Metadata

Lesson 1: Examining Object Metadata

- What Is Reflection?
- Loading Assemblies by Using Reflection
- Examining Types by Using Reflection
- Invoking Members by Using Reflection
- Demonstration: Inspecting Assemblies

Lesson Overview

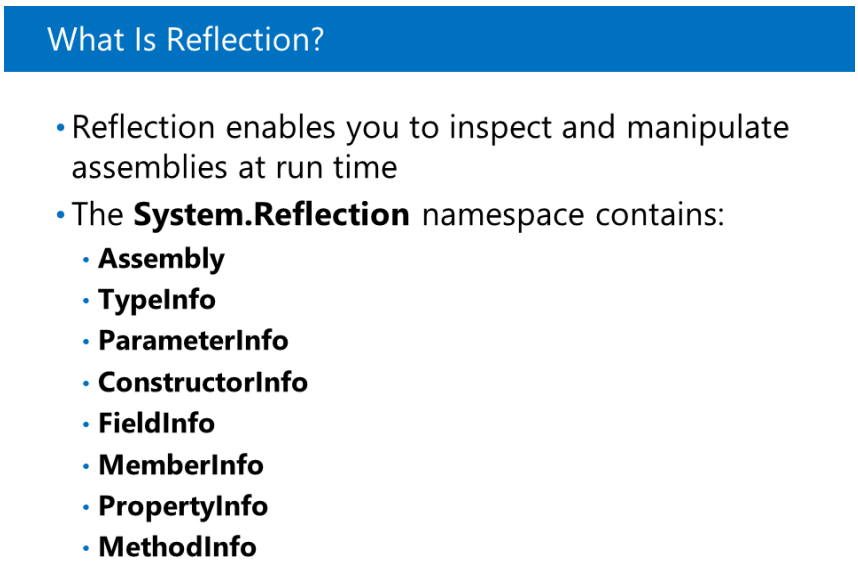
Sometimes applications need to load an existing compiled assembly, view its contents, and execute functionality. For example, a unit testing utility may require the ability to obtain a reference to a type and execute some of the public methods that it exposes. Reflection provides a way for you to examine the types in a compiled assembly.
In this lesson, you will learn how to use reflection to examine the metadata of objects and execute functionality that the object exposes.

OBJECTIVES

After completing this lesson, you will be able to:

- Describe the purpose of reflection.
- Load an assembly.
- Examine the metadata of an existing type.
- Invoke the members that an assembly exposes.

What Is Reflection?



Reflection is a powerful feature that enables you to inspect and perform dynamic manipulation of assemblies, types, and type members at run time.
Reflection is used throughout the Microsoft® .NET Framework. Reflection is used by some of the classes that the base class library provides and some of the utilities that ship with Microsoft Visual Studio®. For example, the serialization classes in the **System.Runtime.Serialization** namespace use reflection to determine which type members should be serialized when serializing types.

Reflection Usage Scenarios

The following table describes some of the possible uses for reflection in your applications.

Use	Scenario
Examining metadata and dependencies of an assembly.	You might choose to do this if you are consuming an unknown assembly in your application and you want to determine whether your application satisfies the unknown assembly's dependencies.
Finding members in a type that have been decorated with a particular attribute.	You might choose to do this if you are implementing a generic storage repository, which will inspect each type and determine which members it needs to persist.
Determining whether a type implements a specific interface.	You might choose to do this if you are creating a pluggable application that enables you to include new assemblies at run time, but you only want your application to load types that implement a specific interface.
Defining and executing a method at run time.	You might choose to do this if you are implementing a virtualized platform that can read types and methods that are implemented in a language such as JavaScript, and then creating managed implementations that you can execute in your .NET Framework application.

Executing code by using reflection is much slower than executing static Microsoft Visual C#® code, so you should only use reflection to create and execute code when you really have to and not just because reflection makes it possible.

Reflection in the .NET Framework

The .NET Framework provides the **System.Reflection** namespace, which contains classes that enable you to take advantage of reflection in your applications. The following list describes some of these classes:

- **Assembly**. This class enables you to load and inspect the metadata and types in a physical assembly.
- **TypeInfo**. This class enables you to inspect the characteristics of a type.
- **ParameterInfo**. This class enables you to inspect the characteristics of any parameters that a member accepts.
- **ConstructorInfo**. This class enables you to inspect the constructor of the type.
- **FieldInfo**. This class enables you to inspect the characteristics of fields that are defined within a type.
- **MemberInfo**. This class enables you to inspect members that a type exposes.
- **PropertyInfo**. This class enables you to inspect the characteristics of properties that are defined within a type.
- **MethodInfo**. This class enables you to inspect the characteristics of the methods that are defined within a type.

The **System** namespace includes the **Type** class, which also exposes a selection of members that you will find useful when you use reflection. For example, the **GetFields** instance method enables you to get a list of **FieldInfo** objects, representing the fields that are defined within a type.

Additional Reading: For more information about the **Type** class, refer to the Type Class page at https://aka.ms/moc-20483c-m12-pg1

Additional Reading: For more information about reflection in the .NET Framework class, refer to the Reflection in the .NET Framework page at http://go.microsoft.com/fwlink/?LinkID=267865.

Loading Assemblies by Using Reflection

Loading Assemblies by Using Reflection

- The **Assembly.LoadFrom** method

```
var assemblyPath = "...";
var assembly = Assembly.LoadFrom(assemblyPath);
```

- The **Assembly.ReflectionOnlyLoad** method

```
var assemblyPath = "...";
var rawBytes = File.ReadAllBytes(assemblyPath);
var assembly = Assembly.ReflectionOnlyLoad(rawBytes);
```

- The **Assembly.ReflectionOnlyLoadFrom** method

```
var assemblyPath = "...";
var assembly = Assembly.ReflectionOnlyLoadFrom(assemblyPath);
```

The **System.Reflection** namespace provides the **Assembly** class, which enables you to encapsulate an assembly in your code so that you can inspect any metadata that is associated with that assembly.

The **Assembly** class provides a number of static and instance members that you can use to load and examine the contents of assemblies. There are two ways that you can load an assembly into your application by using reflection:

- Reflection-only context, in which you can view the metadata that is associated with the assembly and not execute code.
- Execution context, in which you can execute the loaded assembly.

Loading an assembly in reflection-only context can improve performance; however, if you do try to execute it, the Common Language Runtime (CLR) will throw an **InvalidOperationException** exception.

To load an assembly, the **Assembly** class provides a number of static methods, which include the following:

- **LoadFrom**. This method enables you to load an assembly in execution context by using an absolute file path to the assembly. The following code example shows how to load the **FourthCoffee.Service.ExceptionHandling** assembly.

```
var assemblyPath =
"C:\\FourthCoffee\\Libs\\FourthCoffee.Service.ExceptionHandling.dll";
var assembly = Assembly.LoadFrom(assemblyPath);
```

- **ReflectionOnlyLoad**. This method enables you to load an assembly in reflection-only context from a binary large object (BLOB) that represents the assembly. The following code example shows how to load the **FourthCoffee.Service.ExceptionHandling** assembly from a byte array.

```
var assemblyPath =
"C:\\FourthCoffee\\Libs\\FourthCoffee.Service.ExceptionHandling.dll";
```

```
var rawBytes = File.ReadAllBytes(assemblyPath);
var assembly = Assembly.ReflectionOnlyLoad(rawBytes);
```

- **ReflectionOnlyLoadFrom**. This method enables you to load an assembly in reflection-only context by using an absolute file path to the assembly. The following code example shows how to load the **FourthCoffee.Service.ExceptionHandling** assembly.

```
var assemblyPath =
"C:\\FourthCoffee\\Libs\\FourthCoffee.Service.ExceptionHandling.dll";
var assembly = Assembly.ReflectionOnlyLoadFrom(assemblyPath);
```

After you have loaded an assembly and have created an instance of the **Assembly** class, you can use any of the instance members to examine the contents of the assembly. The following list describes some of the instance members that the **Assembly** class provides:

- **FullName**. This property enables you to get the full name of the assembly, which includes the assembly version and public key token. The following code example shows the full name of the **File** class in the **System.IO** namespace.

```
mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

- **GetReferencedAssemblies**. This method enables you to get a list of all of the names of any assemblies that the loaded assembly references.
- **GlobalAssemblyCache**. This property enables you to determine whether the assembly was loaded from the GAC.
- **Location**. This property enables you to get the absolute path to the assembly.
- **ReflectionOnly**. This property enables you to determine whether the assembly was loaded in a reflection-only context or in an execution context. If you load an assembly in reflection-only context, you can only examine the code.
- **GetType**. This method enables you to get an instance of the **Type** class that encapsulates a specific type in an assembly, based on the name of the type.
- **GetTypes**. This method enables you to get all of the types in an assembly in an array of type **Type**.

Additional Reading: For more information about the Assembly class, refer to the Assembly Class page at https://aka.ms/moc-20483c-m12-pg2

Examining Types by Using Reflection

Examining Types by Using Reflection

- Get a type by name

```
var assembly = FourthCoffeeServices.GetAssembly();
var type = assembly.GetType("...");
```

- Get all of the constructors

```
var constructors = type.GetConstructors();
```

- Get all of the fields

```
var fields = type.GetFields();
```

- Get all of the properties

```
var properties = type.GetProperties();
```

- Get all of the methods

```
var methods = type.GetMethods();
```

Reflection enables you to examine the definition of any type in an assembly. Depending on whether you are looking for a type with a specific name or you want to iterate through each type in the assembly in sequence, you can use the **GetType** and **GetTypes** methods that the **Assembly** class provides.

The following code example shows how to load a type by using the **GetType** method, passing the fully qualified name of the type as a parameter to the method call.

Load a Type by Using the GetType Method

```
var assembly = FourthCoffeeServices.GetAssembly();
...
var type = assembly.GetType("FourthCoffee.Service.ExceptionHandling.HandleError");
```

If you use the **GetType** method and specify a name of a type that does not exist in the assembly, the **GetType** method returns **null**.

The following code example shows how to iterate through each of the types in an assembly by using the **GetTypes** method. The **GetTypes** method returns an array of **Type** objects.

Iterate All Types in an Assembly by Using the GetTypes Method

```
var assembly = FourthCoffeeServices.GetAssembly();
...
foreach (var type in assembly.GetTypes())
```

```
{
// Code to process each type.
var typeName = type.FullName;
}
```

After you have created an instance of the **Type** class, you can then use any of its members to inspect the type's definition. The following list describes some of the key members that the **Type** class provides:

- **GetConstructors**. This method enables you to get all of the constructors that exist in a type. The **GetConstructors** method returns an array that contains **ConstructorInfo** objects. The following code example show how to get each of the constructors that exists in a type and then get each of the parameters.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var constructors = type.GetConstructors();
foreach (var constructor in constructors)
{
var parameters = constructor.GetParameters();
}
```

- **GetFields**. This method enables you to get all of the fields that exist in the current type. The **GetFields** method returns an array that contains **FieldInfo** objects. The following code example shows how to iterate through each field in a type and then determine whether the field is static or instance by using the **IsStatic** property that the **FieldInfo** class provides.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var fields = type.GetFields();
foreach (var field in fields)
{
var isStatic = field.IsStatic;
}
```

- **GetProperties**. This method enables you to get all of the properties that exist in the current type. The **GetProperties** method returns an array that contains **PropertyInfo** objects. The following code example shows how to iterate through each property and then get the property's name.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var properties = type.GetProperties();
foreach (var property in properties)
{
var propertyName = property.Name;
}
```

- **GetMethods**. This method enables you to get all of the methods that exist in the current type. The **GetMethods** method returns an array that contains **MethodInfo** objects. The following code example shows how to iterate through each method and then get the method's name.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var methods = type.GetMethods();
foreach (var method in methods)
{
var methodName = method.Name;
}
```

- **GetMembers**. This method enables you to get any members that exist in the current type, including properties and methods. The **GetMembers** method returns an array that contains **MemberInfo** objects. The following code example shows how to iterate through each member and then determine whether the member is a property or method.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var members = type.GetMembers();
foreach (var member in members)
{
if (member.MemberType == MemberTypes.Method)
{
// Process the method.
}
if (member.MemberType == MemberTypes.Property)
{
// Process the property.
}
}
```

You can use these members to get a collection of *xxxx***Info** objects that represents the different members that a type exposes. If you loaded the type in execution context, you can then use the *xxxx***Info** objects to execute each member.

Additional Reading: For more information about the **Type** class, refer to the Type Class page at https://aka.ms/moc-20483c-m12-pg1

Invoking Members by Using Reflection

Invoking Members by Using Reflection

- Instantiate a type

```
var type = FourthCoffeeServices.GetHandleErrorType();
...
var constructor = type.GetConstructor(new Type[0]));
...
var initializedObject = constructor.Invoke(new object[0]);
```

- Invoke methods on the instance

```
var methodToExecute = type.GetMethod("LogError");
var initializedObject = FourthCoffeeServices.InstantiateHandleErrorType();
...
var response = methodToExecute.Invoke(initializedObject,
  new object[] { "Error message"  }) as string;
```

- Get or set property values on the instance

```
var property = type.GetProperty("LastErrorMessage");
var initializedObject = FourthCoffeeServices.InstantiateHandleErrorType();
...
var lastErrorMessage = property.GetValue(initializedObject) as string;
```

The reflection application programming interface (API) in the .NET Framework enables you to invoke objects and use the functionality that they encapsulate. Invoking an object by using reflection follows the same pattern that you use to invoke an object in Visual C#, which typically involves the following steps:

1. Create an instance of the type.
2. Invoke methods on the instance.
3. Get or set property values on the instance.

When you invoke static members by using reflection, there is no need to explicitly create an instance of the type.

Instantiating a Type

To create an instance of a type by using reflection, you need to get a reference to a constructor that the type exposes and then use the **Invoke** method. To instantiate a type by using the **ConstructorInfo** class, perform the following steps:

1. Create an instance of the **ConstructorInfo** class that represents the constructor you will use to initialize an instance of the type. The following code example shows how to initialize an object by using the default constructor.

```
var type = FourthCoffeeServices.GetHandleErrorType();
...
var constructor = type.GetConstructor(new Type[0]);
```

Note: To get the default constructor, you pass an empty array of type **Type** to the **GetConstructor** method call. If you want to invoke a constructor that accepts parameters, you must pass a **Type** array that contains the types that the constructor accepts.

2. Call the **Invoke** method on the **ConstructorInfo** object to initialize an instance of the type, passing an object array that represents any parameters that the constructor call expects. The following code example shows how to invoke the default constructor.

```
var initializedObject = constructor.Invoke(new object[0]);
```

You now have an instance of the type that you can use to invoke any instance methods that are defined in the type.

Invoking Methods

To invoke an instance method, perform the following steps:

1. Create an instance of the **MethodInfo** class that represents the method you want to invoke. The following code example shows how to get a method named **LogError**.

```
var type = FourthCoffeeServices.GetHandleErrorType();
...
var methodToExecute = type.GetMethod("LogError");
```

2. Call the **Invoke** method on the **MethodInfo** object, passing the initialized object and an object array that represents any parameters that the method call expects. You can then cast the return value of the **Invoke** method to the type of value you were expecting from the method call. The following code example shows how to execute the **LogError** instance method that accepts a *string* parameter and returns a **string** value.

```
var initializedObject = FourthCoffeeServices.InstantiateHandleErrorType();
...
var response =
methodToExecute.Invoke(initializedObject, new object[] { "Error message" })
as string;
```

When you invoke static methods, there is no need to create an instance of the type. Instead, you just create an instance of the **MethodInfo** class that represents the method you want to invoke and then call the **Invoke** method. To invoke a static method, perform the following steps:

1. Create an instance of the **MethodInfo** class that represents the method you want to invoke. The following code example shows how to get a method named **FlushLog**.

```
var type = FourthCoffeeServices.GetHandleErrorType();
...
var methodToExecute = type.GetMethod("FlushLog");
```

2. Call the **Invoke** method on the **MethodInfo** object, passing a **null** value to satisfy the initialized object and an object array that represents any parameters that the method call expects. The following code example shows how to execute the **FlushLog** static method that accepts no parameters and returns a Boolean value.

```
var isFlushed = methodToExecute.Invoke(null, new object[0]) as bool;
```

Getting and Setting Properties Values

To get or set the value of an instance property, you must first perform the following steps:

1. Create an instance of the **PropertyInfo** class that represents the property you want to get or set. The following code example shows how to get a property named **LastErrorMessage**.

```
var type = FourthCoffeeServices.GetHandleErrorType();
...
var property = type.GetProperty("LastErrorMessage");
```

2. If you want to get the value of a property, you must invoke the **GetValue** method on the **PropertyInfo** object. The **GetValue** method accepts an instance of the type as a parameter. The following code example shows how to get the value of the **LastErrorMessage** property.

```
var initializedObject = FourthCoffeeServices.InstantiateHandleErrorType();
...
var lastErrorMessage = property.GetValue(initializedObject) as string;
```

3. If you want to set the value of a property, you must invoke **SetValue** method on the **PropertyInfo** object. The **SetValue** method accepts an instance of the type and the value you want to set the property to as parameters. The following code example shows how to set the **LastErrorMessage** property to the text **Database connection error**.

```
var initializedObject = FourthCoffeeServices.InstantiateHandleErrorType();
...
property.SetValue(initializedObject, "Database connection error");
```

When you get or set a static property, there is no need to create an instance of the type. Instead, you just create an instance of the **PropertyInfo** class and then call either the **GetValue** or **SetValue** method. To get or set the value of a static property, perform the following steps:

1. Create an instance of the **PropertyInfo** class that represents the property you want to get or set. The following code example shows how to get a property named **LastErrorMessage**.

```
var type = FourthCoffeeServices.GetHandleErrorType();
...
var property = type.GetProperty("LastErrorMessage");
```

2. If you want to get the value of a property, you must invoke the **GetValue** method on the **PropertyInfo** object, passing a **null** value to satisfy the initialized object parameter. The following code example shows how to get the value of the **LastErrorMessage** static property.

```
var lastErrorMessage = property.GetValue(null) as string;
```

3. If you want to set the value of a property, you must invoke the **SetValue** method on the **PropertyInfo** object, passing a **null** value to satisfy the initialized object parameter, and the value you want to set the property too. The following code example shows how to set the **LastErrorMessage** static property to the text **Database connection error**.

```
property.SetValue(null, "Database connection error");
```

Demonstration: Inspecting Assemblies

Demonstration: Inspecting Assemblies

In this demonstration, you will create a tool that you can use to inspect the contents of an assembly

In this demonstration, you will create a tool that you can use to inspect the contents of an assembly.
The application will use the **System.Reflection** classes to load an assembly, get all of the types in that assembly, and then for each type, get all of the properties and methods.

Demonstration Steps

You will find the steps in the **Demonstration: Inspecting Assemblies** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD12_DEMO.md.

Lesson 2: Creating and Using Custom Attributes

Lesson 2: Creating and Using Custom Attributes

- What Are Attributes?
- Creating and Using Custom Attributes
- Processing Attributes by Using Reflection
- Demonstration: Consuming Custom Attributes by Using Reflection

Lesson Overview

The .NET Framework uses attributes to provide additional metadata about a type or type member. The .NET Framework provides many attributes out of the box that you can use in your applications.
In this lesson, you will learn how to create your own custom attributes and read the metadata that is encapsulated in custom attributes at run time by using reflection.

OBJECTIVES

After completing this lesson, you will be able to:

- Describe the purpose of attributes.
- Create and use custom attributes.

- Process custom attributes by using reflection

What Are Attributes?

- Use attributes to provide additional metadata about an element
- Use attributes to alter run-time behavior

```
[DataContract(Name = "SalesPersonContract", IsReference=false)]
public class SalesPerson
{
  [Obsolete("This property will be removed in the next release.")]
  [DataMember]
  public string Name { get; set; }

  ...
}
```

Attributes provide a mechanism that uses the declarative programming model to include additional metadata about elements, such as types, properties, and methods, in your application. Your application can use the additional information that attributes provide to control application behavior and how data is processed at run time. You can also use the information that attributes encapsulate in case tools and other utilities that aid the development process, such as unit test frameworks.

The .NET Framework makes extensive use of attributes throughout the base class library. The following list describes some of the attributes that the .NET Framework provides:

- The **Obsolete** attribute in the **System** namespace, which you can use to indicate that a type or a type member has been superseded and is only there to ensure backward compatibility.
- The **Serializable** attribute in the **System** namespace, which you can use to indicate that an **IFormatter** implementation can serialize and deserialize a type.
- The **NonSerialized** attribute in the **System** namespace, which you can use to indicate that an **IFormatter** implementation should not serialize or deserialize a member in a type.
- The **DataContract** attribute in the **System.Runtime.Serialization** namespace, which you can use to indicate that a **DataContractSerializer** object can serialize and deserialize a type.
- The **QueryInterceptor** attribute in the **System.Data.Services** namespace, which you can use to control access to an entity in Window Communication Foundation (WCF) Data Services.
- The **ConfigurationProperty** attribute in the **System.Configuration** namespace, which you can use to map a property member to a section in an application configuration file.

All attributes in the .NET Framework derive either directly from the abstract **Attribute** base class in the **System** namespace or from another attribute.

Applying Attributes

To use an attribute in your code, perform the following steps:

1. Bring the namespace that contains the attribute you want to use into scope.
2. Apply the attribute to the code element, satisfying any parameters that the constructor expects.
3. Optionally set any of the named parameters that the attribute exposes.

The following code example shows how to apply the **DataContract** attribute to the **SalesPerson** class definition and set the Name and IsReference named parameters.

Applying the DataContract Attribute

```
[DataContract(Name = "SalesPersonContract", IsReference=false)]
public class SalesPerson
{
...
}
```

You can apply multiple attributes to a single element to create a hierarchy of metadata that describes the element. The following code example shows how to apply the **Obsolete** and **DataMember** attributes to the **Name** property in the **SalesPerson** class, to indicate that the property should be serialized but will be removed from the type definition in the next release.

Applying the Obsolete and DataContract Attributes

```
[DataContract(Name = "SalesPersonContract", IsReference=false)]
public class SalesPerson
{
[Obsolete("This property will be removed in the next release. Use the FirstName and LastName properties
instead.")]
[DataMember]
public string Name { get; set; }
}
```

Additional Reading: For more information about the Attribute class, refer to the Attribute Class page at https://aka.ms/moc-20483c-m12-pg4

Creating and Using Custom Attributes

Creating and Using Custom Attributes

Derive from the **Attribute** class or another attribute

```
[AttributeUsage(AttributeTargets.All)]
public class DeveloperInfoAttribute : Attribute
{
  private string _emailAddress;
  private int _revision;

  public DeveloperInfo(string emailAddress, int revision)
  {
    this._emailAddress = emailAddress;
    this._revision = revision;
  }
}
```

```
[DeveloperInfo("holly@fourthcoffee.com", 3)]
public class SalePerson
{
  ...
}
```

The .NET Framework provides an extensive set of attributes that you can use in your applications. However, there will be a time when you need an attribute that the .NET Framework does not provide. For example, maybe you want to include information about the developer who authored the source code for an application, or maybe you need some additional data for a custom testing framework you are using to test the application.

To create an attribute, perform the following steps:

1. Create a type that derives from the **Attribute** base class or another existing attribute type.
2. Apply the **AttributeUsage** attribute to your custom attribute class to describe which elements you can apply this attribute to.

Note: If you apply an attribute to an element that conflicts with the value of the AttributeUsage attribute, the compiler will throw an error at build time.

3. Define a constructor to initialize the custom attribute.
4. Define any properties that you want to enable users of the attribute to optionally provide information. Any properties that you define that have a **get** accessor will be exposed through the attribute as a named parameter.

When creating attributes, the convention is to end the name with an attribute suffix. When the type is used as an attribute, the compiler removes that suffix.

The code example shows how to create an attribute that encapsulates metadata about the developer who creates the element.

Creating a Custom Attribute

```
[AttributeUsage(AttributeTargets.All)]
public class DeveloperInfoAttribute : Attribute
{
private string _emailAddress;
private int _revision;
public DeveloperInfo(string emailAddress, int revision)
{
this._emailAddress = emailAddress;
this._revision = revision;
}
public string EmailAddress
{
get { return this._emailAddress; }
}
public int Revision
{
get { return this._revision; }
```

```
}
}
```

Using a custom attribute is no different from using an attribute that the .NET Framework provides. You simply apply the attribute to an element and ensure that you pass the required information to the constructor.
The following code example shows how to apply the **DeveloperInfo** attribute to a type definition.

Applying a Custom Attribute

```
[DeveloperInfo("holly@fourthcoffee.com", 3)]
public class SalePerson
{
...
[DeveloperInfo("linda@fourthcoffee.com", 1)]
public IEnumerable<Sale> GetAllSales()
{
...
}
}
```

Additional Reading: For more information about how to create custom attributes, refer to the Creating Custom Attributes (C# and Visual Basic) page at https://aka.ms/moc-20483c-m12-pg5.

Processing Attributes by Using Reflection

Processing Attributes by Using Reflection

Use reflection to access the metadata that is encapsulated in custom attributes

```
var type = FourthCoffee.GetSalesPersonType();

var attributes = type.GetCustomAttributes(typeof(DeveloperInfo),
false);

foreach (var attribute in attributes)
{
  var developerEmailAddress = attribute.EmailAddress;
  var codeRevision = attribute.Revision;
}
```

If you use the existing attributes that the .NET Framework provides, they normally have a purpose other than just to exist in your code. For example, the **IFormatter** serializers use the **Serializable** attribute during the serializing and deserializing processes.
Similarly, you can use attributes to encapsulate metadata about an element in your code. For example, if you create an attribute that provides information about the developer who authored the element, you may want a way to extract this information so that you can produce a document listing all of the developers who were involved in developing the application.
Similar to accessing other type and member information, you can also use reflection to process attributes. The **System.Reflection** namespace provides a collection of extension methods that you can use to access attributes and the metadata they encapsulate. The following list describes some of these methods:

- **GetCustomAttribute**. This method enables you to get a specific attribute that was used on an element. The following code example shows how to get a **DeveloperInfo** attribute that has been applied to a type.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var getInheritedAttributes = false;
var attribute =
type.GetCustomAttribute(typeof(DeveloperInfo), getInheritedAttributes);
```

Note: The *getInheritedAttributes* parameter instructs the **GetCustomAttribute** method call to return either only attributes that have been explicitly applied to the current type, or attributes that have been either explicitly applied or inherited from any parent type.

- **GetCustomAttribute**. This generic method also enables you to get a specific attribute that was used on an element. This is a generic method, so you can specify the type of attribute you want the **GetCustomAttribute** method to return. This means that you do not have to write conditional logic to filter an array of objects to determine the type of an attribute and perform any casting. The following code example shows how to get a **DeveloperInfo** attribute that has been applied to a type.

```
var type = FourthCoffeeServices.GetHandleErrorType();
```

```
var getInheritedAttributes = false;
var attribute = type.GetCustomAttribute<DeveloperInfo>(getInheritedAttributes);
```

- **GetCustomAttributes**. This method enables you to get a list of specific attributes that were used on an element. Typically, you would use this method if more than one attribute of the same type has been applied to an element, in which case this method call would return all instances. The following code example shows how to get all of the **DeveloperInfo** attributes that were applied to a type.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var getInheritedAttributes = false;
var attributes =
type.GetCustomAttributes(typeof(DeveloperInfo), getInheritedAttributes);
```

- **GetCustomAttributes**. This generic method enables you to get a list of specific attributes that were used on an element. Similar to the **GetCustomAttribute** generic method, because this is a generic method, the method call will only return attributes that match the generic type. The following code example shows how to get all of the **DeveloperInfo** attributes that were applied to a type.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var getInheritedAttributes = false;
var attributes =
type.GetCustomAttributes<DeveloperInfo>(getInheritedAttributes);
```

The following code example shows how to iterate through each of the custom attributes that have been applied to a type definition and then access the data that any **DeveloperInfo** attributes encapsulate.

Iterating Custom Attributes That Have Been Applied to a Type

```
var type = FourthCoffee.GetSalesPersonType();
var attributes = type.GetCustomAttributes(typeof(DeveloperInfo), false);
foreach (var attribute in attributes)
{
var developerEmailAddress = attribute.EmailAddress;
var codeRevision = attribute.Revision;
}
```

Note: The **false** value that is passed to the **GetCustomAttributes** method instructs the method to only get custom attributes that have been explicitly applied to the current type, not attributes that have been inherited.

To access custom attributes that have been applied to a member, you must create an *xxxx***Info** object that represents the member and then invoke the **GetCustomAttributes** method.

The following code example shows how to iterate through each of the custom attributes that have been applied to a method called **GetAllSales** and then access the data that any **DeveloperInfo** attributes have encapsulated.

Iterate Custom Attributes That Have Been Applied to a Method

```
var type = FourthCoffee.GetSalesPersonType();
var methodToInspect = type.GetMethod("GetAllSales");
var attributes = methodToInspect.type.GetCustomAttributes(typeof(DeveloperInfo), false);
foreach (var attribute in attributes)
{
var developerEmailAddress = attribute.EmailAddress;
var codeRevision = attribute.Revision;
}
```

Additional Reading: For more information about how to process custom attributes, refer to the Accessing Custom Attributes page at http://go.microsoft.com/fwlink/?LinkID=267869.

Demonstration: Consuming Custom Attributes by Using Reflection

Demonstration: Consuming Custom Attributes by Using Reflection

In this demonstration, you will use reflection to read the **DeveloperInfo** attributes that have been used to provide additional metadata on types and type members

In this demonstration, you will use reflection to read the **DeveloperInfo** attributes that have been used to provide additional metadata on types and type members.
The application uses the **GetCustomAttribute** generic method that the **Type** and **MemberInfo** classes provide to extract any additional information that a custom **DeveloperInfo** attribute encapsulates.

Demonstration Steps

You will find the steps in the **Demonstration: Consuming Custom Attributes by Using Reflection** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD12_DEMO.md.

Lesson 3: Generating Managed Code

- What Is CodeDOM?
- Defining a Type and Type Members
- Compiling a CodeDOM Model
- Compiling Source Code into an Assembly

Lesson Overview

You can use Visual Studio to write managed code when you have clearly defined requirements upon which to base your implementation. However, sometimes you may want to generate code at run time based on a varying set of requirements. In this scenario, you need a framework that enables you to define instructions that a process can translate into executable code. The .NET Framework provides the CodeDOM feature for this very purpose.
In this lesson, you will learn how to generate managed code at run time by using CodeDOM.

OBJECTIVES

After completing this lesson, you will be able to:

- Describe the purpose of CodeDOM.
- Define a type by using CodeDOM.
- Compile a CodeDOM model into source code files.
- Compile source code files into an assembly.

What Is CodeDOM?

What Is CodeDOM?

- Define a model that represents your code by using:
  - The **CodeCompileUnit** class
  - The **CodeNamespace** class
  - The **CodeTypeDeclaration** class
  - The **CodeMemberMethod** class
- Generate source code from the model:
  - Visual C# by using the **CSharpCodeProvider** class
  - JScript by using the **JScriptCodeProvider** class
  - Visual Basic by using the **VBCodeProvider** class
- Generate a .dll or a .exe that contains your code

CodeDOM is a feature of the .NET Framework that enables you to generate code at run time. CodeDOM enables you to build a model that represents the source code that you want to create and then generate the source code by using one of the code generators that are included in CodeDOM. By default, CodeDOM includes code generators for Visual C#, Microsoft Visual Basic®, and Microsoft JScript®.

CodeDOM is a powerful feature that you can use for generating template source files that contain boilerplate code or even generating source code files that serve as a proxy between your application and a remote entity. To use CodeDOM in your application, use the following namespaces:

- **System.CodeDom**. This namespace contains types that enable you to define a model that represents the source code you want to generate.
- **System.CodeDom.Compiler**. This namespace contains types that enable you to generate and manage compiled code.
- **Microsoft.CSharp.CSharpCodeProvider**. This namespace contains the Visual C# code compiler and generator.
- **Microsoft.JScript.JScriptCodeProvider**. This namespace contains the JScript code compiler and generator.
- **Microsoft.VisualBasic.VBCodeProvider**. This namespace contains the Visual Basic code compiler and generator.

You can use the classes in the **System.CodeDom** namespace to create a model that represents the code you want to create. The model can include anything from a complex class hierarchy to a single class with some members. For example, you can use the **CodeNamespace** class to represent a namespace, and you can use the **CodeMemberMethod** class to represent a method.

The following table describes some of the classes you can use to create your model.

Class	Description
CodeCompileUnit	Enables you to encapsulate a collection of types that ultimately will compile into an assembly.
CodeNamespace	Enables you to define a namespace that you can use to organize your class hierarchy.
CodeTypeDeclaration	Enables you to define a class, structure, interface, or enumeration in your model.
CodeMemberMethod	Enables you to define a method in your model and add it to a type, such as a class or an interface.
CodeMemberField	Enables you to define a field, such as an **int** variable, and add it to a type, such as a class or struct.
CodeMemberProperty	Enables you to define a property with **get** and **set** accessors and add it to a type, such as a class or struct.
CodeConstructor	Enables you to define a constructor so that you can create an instance type in your model.
CodeTypeConstructor	Enables you to define a static constructor so that you can create a singleton type in your model.

CodeEntryPoint	Enables you to define an entry point in your type, which is typically a static method with the name **Main**.
CodeMethodInvokeExpression	Enables you to create a set of instructions that represents an expression that you want to execute.
CodeMethodReferenceExpression	Enables you to create a set of instructions that detail a method in a particular type that you want to execute. Typically, you would use this class with the **CodeMethodInvokeExpression** class when you implement the body of method in a model.
CodeTypeReferenceExpression	Enables you to represent a reference type that you want to use as part of an expression in your model. Typically, you would use this class with the **CodeMethodInvokeExpression** class and the **CodeTypeReferenceExpression** class when you implement the body of method in a model.
CodePrimitiveExpression	Enables you to define an expression value, which you may want to pass as a parameter to a method or store in a variable.

After you have defined your model by using the classes in the **System.CodeDom** namespace, you can then use a code generator provider, such as the **CSharpCodeProvider** class in the **Microsoft.CSharp.CSharpCodeProvider** namespace, to compile your model and generate your code.

Additional Reading: For more information about CodeDOM, refer to the Dynamic Source Code Generation and Compilation page at http://go.microsoft.com/fwlink/?LinkID=267870.

Defining a Type and Type Members

Defining a Type and Type Members

Defining a type with a **Main** method

```
var unit = new CodeCompileUnit();

var dynamicNamespace = new CodeNamespace("FourthCoffee.Dynamic");
unit.Namespaces.Add(dynamicNamespace);

dynamicNamespace.Imports.Add(new CodeNamespaceImport("System"));

var programType = new CodeTypeDeclaration("Program");
dynamicNamespace.Types.Add(programType);

var mainMethod = new CodeEntryPointMethod();
programType.Members.Add(mainMethod);

var expression = new CodeMethodInvokeExpression(
  new CodeTypeReferenceExpression("Console"), "WriteLine",
  new CodePrimitiveExpression("Hello Development Team..!!"));
```

Defining a type by using CodeDOM follows the same pattern as defining a type in native Visual C#. The only difference is that when using CodeDOM, you write a set of instructions that a code generator provider will interpret to generate the source code that represents your model.

The **System.CodeDOM** namespace includes the types that you can use to write these instructions. The following steps describe how to use some of the **System.CodeDOM** types to define a type that contains an entry point method named **Main**:

1. Create a **CodeCompileUnit** object to represent the assembly that will contain the type. The following code example shows how to create a **CodeCompileUnit** object.
   `var unit = new CodeCompileUnit();`
2. Create a **CodeNamespace** object to represent the namespace that the type will be scoped to and add the namespace to the **CodeCompileUnit** object. The following code example shows how to define the **FourthCoffee.Dynamic** namespace.
   ```
   var dynamicNamespace = new CodeNamespace("FourthCoffee.Dynamic");
   unit.Namespaces.Add(dynamicNamespace);
   ```
3. Import any additional namespaces that the types in the namespace will use. The following code example shows how to bring the **System** namespace into scope.
   `dynamicNamespace.Imports.Add(new CodeNamespaceImport("System"));`
4. Create a **CodeTypeDeclaration** object that represents the type you want to add to the namespace. The following code example shows how to create a type named **Program**.

```
var programType = new CodeTypeDeclaration("Program");
dynamicNamespace.Types.Add(programType);
```

5. Create a **CodeEntryPointMethod** object to represent the static main method in the **Program** type. The following code example shows how to define an entry point method named **Main** and add it to the **Program** type.

```
var mainMethod = new CodeEntryPointMethod();
programType.Members.Add(mainMethod);
```

6. Define the body of the **Main** method by using the **CodeMethodInvokeExpression**, **CodeTypeReferenceExpression**, and **CodePrimitiveExpression** classes. The parameters that you pass to the constructors of these objects enable you to define which method you want to invoke and the parameters that the method expects. The following code example shows how invoke the **Console.WriteLine** method to write the message **Hello Development Team..!!** to the console window.

```
var expression = new CodeMethodInvokeExpression(
new CodeTypeReferenceExpression("Console"),
"WriteLine",
new CodePrimitiveExpression("Hello Development Team..!!"));
mainMethod.Statements.Add(expression);
```

After you have defined your model, you can then use a code generator provider to compile and generate your code.

Additional Reading: For more information about how to define a model by using CodeDOM, refer to the Using the CodeDOM page at http://go.microsoft.com/fwlink/?LinkID=267871.

Compiling a CodeDOM Model

Compiling a CodeDOM Model

Generate source code files from your CodeDOM model

```
var provider = new CSharpCodeProvider();

var fileName = "program.cs";
var stream = new StreamWriter(fileName);
var textWriter = new IndentedTextWriter(stream);

var options = new CodeGeneratorOptions();
options.BlankLinesBetweenMembers = true;

var compileUnit = FourthCoffee.GetModel();
provider.GenerateCodeFromCompileUnit(
  compileunit,
  textWriter,
  options);

textWriter.Close();
stream.Close();
```

After you have defined the contents of your assembly by using the types in the **System.CodeDOM** namespace, you can then compile and generate an assembly. You can split the process for compiling and generating an assembly into the following parts:

1. Compiling the model and generating source code files for each type.
2. Generating an assembly that contains the necessary references and the types that are defined in the source code files.

Compiling a Model into a Source Code File

To compile your model and generate source code files, perform the following steps:

1. Create an instance of the code generator provider you want to use. The following code example shows how to create an instance of the **CSharpCodeProvider** class that will produce Visual C# code.

```
var provider = new CSharpCodeProvider();
```

2. Create a **StreamWriter** object that the code generator will use to write the compiled code to a file. The following code example shows how to create a **StreamWriter** object.

```
var fileName = "program.cs";
var stream = new StreamWriter(fileName);
```

3. Create an **IndentedTextWriter** object that will write the indented source code to a file. The following code example shows how to create an **IndentedTextWriter** object.

```
var textWriter = new IndentedTextWriter(stream);
```

4. Create a **CodeGeneratorOptions** object that encapsulates your code generation settings. The following code example shows how to create a **CodeGeneratorOptions** object and set the **BlankLinesBetweenMembers** property so that members are separated by a blank line.

```
var options = new CodeGeneratorOptions();
options.BlankLinesBetweenMembers = true;
```

5. Invoke the **GenerateCodeFromCompileUnit** method on the **CSharpCodeProvider** object to generate the source code. The following code example shows how to invoke the **GenerateCodeFromCompileUnit** method, passing the **CodeCompileUnit**, **IndentedTextWriter**, and **CodeGeneratorOptions** objects as parameters.

```
var compileUnit = FourthCoffee.GetModel();
provider.GenerateCodeFromCompileUnit(compileunit, textWriter, options);
```

6. Close the **IndentedTextWriter** and **StreamWriter** objects. The following code example shows how to close the **IndentedTextWriter** and **StreamWriter** objects, flushing the compiled code to the output file.

```
textWriter.Close();
stream.Close();
```

After you have executed the code to compile your model, you will have a source .cs file that contains the compiled Visual C# code.

The following code example shows the compiled Visual C# code for a model that contains the **Program** type with an entry point method called **Main.**

Compiled Visual C# Code

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.30319.17626
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------
namespace FourthCoffee.Dynamic {
using System;
public class Program {
public static void Main() {
Console.WriteLine("Hello Development Team..!!");
}
}
}
```

Compiling Source Code into an Assembly

Compiling Source Code into an Assembly

Generate an assembly from your source code files

```
var provider = new CSharpCodeProvider();

var compilerSettings = new CompilerParameters();
compilerSettings.ReferencedAssemblies.Add("System.dll");
compilerSettings.GenerateExecutable = true;
compilerSettings.OutputAssembly = "FourthCoffee.exe";

var sourceCodeFileName = "program.cs";
var compilationResults = provider.CompileAssemblyFromFile(
  compilerSettings,
  sourceCodeFileName);

var buildFailed = false;
foreach (var error in compilationResults.Errors)
{
  var errorMessage = error.ToString();
  buildFailed = true;
}
```

After you have compiled your CodeDOM model into one or more source code files, you can compile the files into an assembly.

To generate an executable assembly that contains the **FourthCoffee.Dynamic.Program** type from a source code file, perform the following steps:

1. Create an instance of the code generator provider you want to use. The following code example shows how to create an instance of the **CSharpCodeProvider** class that will produce the executable assembly.

```
var provider = new CSharpCodeProvider();
```

2. Create a **CompilerParameters** object that you will use to define the settings for the compiler, such as which assemblies to reference, and whether to generate a .dll or an .exe file. The following code example shows how to create a **CompilerParameters** object, add a reference to the System.dll file, and instruct the compiler to generate an executable file named FourthCoffee.exe.

```
var compilerSettings = new CompilerParameters();
compilerSettings.ReferencedAssemblies.Add("System.dll");
compilerSettings.GenerateExecutable = true;
compilerSettings.OutputAssembly = "FourthCoffee.exe";
```

3. Invoke the **CompileAssemblyFromFile** method on the **CSharpCodeProvider** object to generate the assembly. The following code example shows how to invoke the **CompileAssemblyFromFile** method, passing the **CompilerParameters** object and a string variable that contains the path to the source code file.

```
var sourceCodeFileName = "program.cs";
var compilationResults = provider.CompileAssemblyFromFile(
```

```
compilerSettings,
sourceCodeFileName);
```

Note: The CompileAssemblyFromFile method also accepts an array of source file names, so you can compile several source code files into a single assembly.

4. You can then use the properties that the **CompilerResults** object provides to determine whether the compilation was successful. The following code example shows how to iterate the **CompilationErrorCollection** object by using the **Errors** property.

```
var buildFailed = false;
foreach (var error in compilationResults.Errors)
{
var errorMessage = error.ToString();
buildFailed = true;
}
```

You have now generated an assembly that displays the message **Hello Development Team..!!** when it is executed.

Additional Reading: For more information about how to compile a CodeDOM model, refer to the Generating and Compiling Source Code from a CodeDOM Graph page at http://go.microsoft.com/fwlink/?LinkID=267872.

Lesson 4: Versioning, Signing, and Deploying Assemblies

Lesson 4: Versioning, Signing, and Deploying Assemblies

- What Is an Assembly?
- What Is the GAC?
- Signing Assemblies
- Versioning Assemblies
- Installing an Assembly into the GAC
- Demonstration: Signing and Installing an Assembly into the GAC
- Demonstration: Specifying the Data to Include in the Grades Report Lab

Lesson Overview

When you finish developing an application, you should sign and version the assembly before you distribute it to users. You must also consider how and where the assembly is going to be installed.

In this lesson, you will learn how to version assemblies and install an assembly into the GAC.

OBJECTIVES

After completing this lesson, you will be able to:

- Describe the key features of assemblies.
- Describe the GAC.
- Explain how to sign an assembly.
- Describe the key features of assembly versioning.
- Install an assembly in the GAC.

What Is an Assembly?

What Is an Assembly?

- An assembly is a collection of types and resources
- An assembly is a versioned deployable unit
- An assembly can contain:
  - IL code
  - Resources
  - Type metadata
  - Manifest

An assembly is a collection of types and resources that form a unit of functionality. An assembly might consist of a single portable executable (PE) file, such as an executable (.exe) program or dynamic link library (.dll) file, or it might consist of multiple PE files and external resource files, such as bitmaps or data files. An assembly is the building block of a .NET Framework application because an application consists of one or more assemblies.

Contents of Assemblies

An assembly consists of the following components:

- *Intermediate language (IL) code*. The compiler translates the source code into IL, which is the set of instructions that the just-in-time (JIT) compiler then translates to CPU-specific code before the application runs.
- *Resources*. These include images and assembly metadata. Assembly metadata exists in the form of the assembly manifest.
- *Type metadata*. Type metadata provides information about available classes, interfaces, methods, and properties, similar to the way that a type library provides information about COM components.
- *The assembly manifest*. This contains assembly metadata and provides information about the assembly such as the title, the description, and version information. The manifest also contains information about links to the other files in the assembly. The information in the manifest is used at run time to resolve references and validate loaded assemblies. The assembly manifest can be stored in a separate file but is often part of one of the PE files.

Boundaries of Assemblies

By arranging your code into assemblies, you create a set of boundaries that you can use to isolate configuration to a particular assembly. The following list describes some of the boundaries that assemblies provide:

- *Security boundary*. You set security permissions at an assembly level. You can use these permissions to request specific access for an application, for example, file I/O permissions if the application must write to a disk. When the assembly is loaded at run time, the permissions that are requested are entered into the security policy and used to determine whether permissions can be granted.
- *Type boundary*. An assembly provides a boundary for data types, because each type has the assembly name as part of its identity. As a result, two types can have the same name in different assemblies without any conflict.
- *Reference scope boundary*. An assembly provides a reference scope boundary by using the assembly manifest to resolve type and resource requests. This metadata specifies which types and resources are exposed outside the assembly.

Benefits of Assemblies

Assemblies provide you with the following benefits:

- *Single units of deployment*. The client application loads assemblies when it needs them, which enables a minimal download strategy where appropriate.
- *Versioning*. An assembly is the smallest unit in a .NET Framework application that you can version. The assembly manifest describes the version information and any version dependencies that are specified for any dependent assemblies. You can only version assemblies that have strong names.

Additional Reading: For more information about assemblies, see the Assemblies in the Common Language Runtime page at http://go.microsoft.com/fwlink/?LinkID=267873.

What Is the GAC?

What Is the GAC?

- The GAC provide a robust solution to share assemblies between multiple application on the same machine
- Find the contents of the GAC at C:\Windows\assembly
- Benefits:
  - Side-by-side deployment
  - Improved loading time
  - Reduced memory consumption
  - Improved search time
  - Improved maintainability

When you create an assembly, by default you create a private assembly that a single application can use. If you need to create an assembly that multiple applications can share, you should give the assembly a strong name and install the assembly into the GAC.

A strong name is a unique name for an assembly that consists of the assembly's name, version number, culture information, and a digital signature that contains a public and private key.

The GAC stores the assemblies that you want to share between multiple applications. When you add an assembly to the GAC, the GAC performs integrity checks on all of the files that form the assembly. These checks ensure that nothing has tampered with an assembly. For example, the GAC checks for changes to a file that the manifest does not reflect.

You can examine the GAC by using File Explorer. Browse to C:\Windows\assembly to see the list of assemblies in the GAC. The information in the list of installed assemblies includes the following:

- The global assembly name
- The version number of the assembly
- The culture of the assembly, if applicable
- The public key token of the assembly in the strong name
- The type of assembly

Benefits of Using the GAC

Although you can install strong-named assemblies in directories on the computer, the GAC offers several benefits, including the following:

- *Side-by-side deployment and execution*. Different versions of an assembly in the GAC do not affect each other. Therefore, applications that reference different versions of an assembly do not fail if a later incompatible version of the assembly is installed into the cache.
- *Improved loading time*. When you install a strong-named assembly in the GAC, it undergoes strong-name validation, which ensures that the digital signature is valid. The verification process occurs at installation time, so strong-named assemblies in the GAC load faster at run time than assemblies that are not installed in the GAC.
- *Reduced memory consumption*. If multiple applications reference an assembly, the operating system loads only one instance of the assembly, which can reduce the total memory that is used on the computer.
- *Improved search time*. The CLR can locate a strong-named assembly in the GAC faster than it can locate a strong-named assembly that is in a directory. This is because the runtime checks the GAC for a referenced assembly before it checks other locations.
- *Improved maintainability*. With a single file that multiple applications share, you can easily make fixes that affect all of the applications.

Signing Assemblies

Signing Assemblies

Sign an assembly:

- Create a key file
- Associate the key file with an assembly

```
sn –k FourthCoffeeKeyFile.snk
```

Delay the signing of an assembly:

1. Open the properties for the project

```
[assembly: AssemblyKeyFileAttribute("FourthCoffeeKeyFile.snk")]
```

2. Click the **Signing** tab
3. Select the **Sign the assembly** check box
4. Specify a key file
5. Select the **Delay sign only** check box

When you *sign* an assembly, you give the assembly a *strong name*. A strong name provides an assembly with a globally unique name that applications use when they reference your assembly. This ensures that no one else can compile an assembly with the same name as yours and impersonate your assembly. This helps to avoid malicious code overwriting one of your assemblies and then being run from an application that expects to be using your authentic code.

A strong name requires two cryptographic keys, a public key and a private key, known as a *key pair*. The compiler uses the key pair at build time to create the strong name. The strong name consists of the simple text name of the assembly, the version number, optional culture information, the public key, and a digital signature.

Creating Key Pairs

You must have a key pair to sign an assembly with a strong name. To create a key pair, use the Strong Name tool (Sn.exe) that the .NET Framework provides. To create a key pair file, perform the following steps:

1. Open the Visual Studio 2012 command prompt.
2. In the **Command Prompt** window, use the Sn.exe tool with the **K** switch to create a new key file. The following code example shows how to create a new key file with the name **FourthCoffeeKeyFile**.

```
sn -k FourthCoffeeKeyFile.snk
```

After you have created a key file, you can then sign your assembly.

Signing an Assembly

When you have created the key pair, you can then associate the key file with your assembly. You can achieve this using the **Signing** tab in the project properties pane. When you specify a key file in the properties pane, Visual Studio adds the **AssemblyKeyFileAttribute** attribute to the **AssemblyInfo** class in your application.

The following code example shows how to associate the FourthCoffeeKeyFile.snk file with your assembly by using the **AssemblyKeyFileAttribute** attribute.

The **AssemblyKeyFileAttribute** attribute

```
[assembly: AssemblyKeyFileAttribute("FourthCoffeeKeyFile.snk")]
```

Delay-Signing an Assembly

When you sign an assembly, you might not have access to a private key. For example, for security reasons, some organizations restrict access to their private key to just a few individuals. The public key will generally be available because as its name implies, it is publicly accessible. In this situation, you can use delayed signing at build time. You provide the public key and reserve space in the PE file for the strong-name signature. However, you defer the addition of the private key until a later stage, typically just before the assembly ships.

You can enable delay-signing on the **Signing** tab of the project properties window as follows:

1. In Solution Explorer, right-click the project, and then click **Properties**.
2. In the properties window of the project, click the **Signing** tab.
3. Select the **Sign the assembly** check box.
4. Specify a key file.
5. Select the **Delay sign only** check box.

You cannot run or debug a delay-signed project. You can, however, use the Sn.exe tool with the **-Vr** option to skip verification, which means that the identity of the assemblies will not be checked. However, you should only use this option at development time because it creates a security vulnerability.
The following code example turns off verification for an assembly called FourthCoffee.Core.dll.
Disabling Verification

```
sn –Vr FourthCoffee.Core.dll
```

You can then submit the assembly to the signing authority of your organization for the actual strong-name signing. Use the **–R** option with the Sn.exe tool to resign a delay-signed assembly.
The following code example signs an assembly called FourthCoffee.Core.dll with a strong name by using the sgKey.snk key pair.
Signing an Assembly

```
sn -R FourthCoffee.Core.dll sgKey.snk
```

Additional Reading: For more information about delay signing, refer to the Delay Signing an Assembly page at http://go.microsoft.com/fwlink/?LinkID=267874.

Versioning Assemblies

Versioning Assemblies

A version number of an assembly is a four-part string:

```
<major version>.<minor version>.<build number>.<revision>
```

Applications reference particular versions of assemblies

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="...">
     <dependentAssembly>
       <assemblyIdentity name="FourthCoffee.Core"
          publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
       <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>
     </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

All assemblies are given a version number by Visual Studio, which is typically 1.0.0.0. It is the responsibility of the developer to increment the assembly's version number as the assembly evolves.
It is important to version assemblies so that you can keep track of which version of your application users are using. Without a version number, debugging and reproducing production issues can be difficult.

Assembly Version Number

The version number of an assembly is a four-part string with the following format: <*major version*>.<*minor version*>.<*build number*>.<*revision*>. For example, version 1.2.3.0 indicates 1 as the major version, 2 as the minor version, 3 as the build number, and 0 as the revision number.
The assembly manifest stores the version number along with other identity information such as the assembly name and public key. The CLR uses the version number, in conjunction with the available configuration information, to load the proper version of a referenced assembly.
By default, applications only run with the version of an assembly with which they were built. To change an application to use a different version of an assembly, you can create a version policy in one of the configuration files: the application configuration file, the publisher policy file, or the computer's administrator configuration file.

Redirecting Binding Requests

When you want to update a strong-named component without redeploying the client application that uses it, you can use a publisher policy file to redirect a binding request to a newer instance of the component.
When a client application makes a binding request, the runtime performs the following tasks:

- It checks the original assembly reference for the version to be bound.
- It checks the configuration files for version policy instructions.

The following code example shows a publisher policy file. To redirect one version to another, use the **<bindingRedirect>** element. The **oldVersion** attribute can specify either a single version or a range of versions. For

example, **<bindingRedirect oldVersion="1.1.0.0-1.2.0.0" newVersion="2.0.0.0"/>** specifies that the runtime should use version 2.0.0.0 instead of the assembly versions between 1.1.0.0 and 1.2.0.0.

Assembly Binding Redirect

```
<configuration>
<runtime>
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
<dependentAssembly>
<assemblyIdentity name="FourthCoffee.Core"
publicKeyToken="32ab4ba45e0a69a1"
culture="en-us" />
<bindingRedirect oldVersion="1.0.0.0"
newVersion="2.0.0.0"/>
</dependentAssembly>
</assemblyBinding>
</runtime>
</configuration>
```

Additional Reading: For more information about versioning, refer to the Assembly Versioning page at http://go.microsoft.com/fwlink/?LinkID=267875.

Installing an Assembly into the GAC

Installing an Assembly into the GAC

Install an assembly in the GAC by using:

- Global Assembly Cache tool
- Microsoft Windows Installer

Examples:

- Install an assembly by using Gacutil.exe:

```
gacutil -i "<pathToAssembly>"
```

- View an assembly by using Gacutil.exe:

```
gacutil -l "<assemblyName>"
```

The GAC is a folder on file system where you can install your assemblies. You can install your assemblies into the GAC in a variety of ways, which include the following:

- *Global Assembly Cache tool (Gacutil.exe)*. You can use Gacutil.exe to add strong-named assemblies to the GAC and to view the contents of the GAC. Gacutil.exe is for development purposes. You should not use the tool to install production assemblies into the GAC.
- *Microsoft Windows Installer 2.0*. This is the recommended and most common way to add assemblies to the GAC. The installer provides benefits such as reference counting of assemblies in the GAC.

Installing an Assembly into the GAC by Using Gacutil.exe

To install an assembly into the GAC by using the Gacutil.exe command-line tool, perform the following steps:

1. Open the Visual Studio 2012 command prompt as an administrator.
2. In the **Command Prompt** window, type the following command:
   `gacutil -i "<pathToAssembly>"`

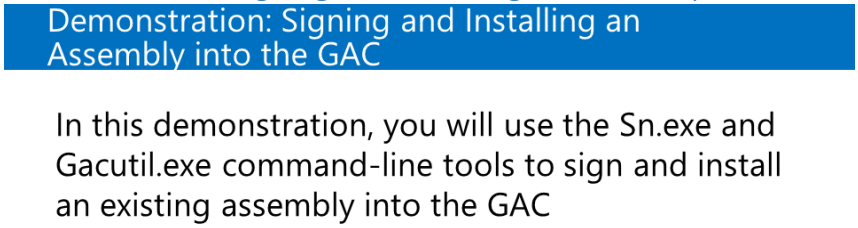
Viewing an Assembly in the GAC by Using Gacutil.exe

To view an assembly that is installed into the GAC by using the Gacutil.exe command-line tool, perform the following steps:

1. Open the Visual Studio 2012 command prompt as an administrator.
2. In the **Command Prompt** window, type the following command:
   `gacutil -l "<assemblyName>"`

Additional Reading: For more information about the GAC, refer to the Global Assembly Cache page at http://go.microsoft.com/fwlink/?LinkID=267876.

Demonstration: Signing and Installing an Assembly into the GAC

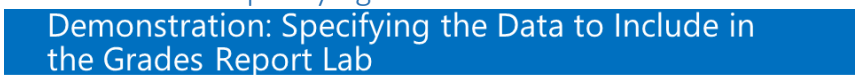


In this demonstration, you will use the Sn.exe and Gacutil.exe command-line tools to sign and install an existing assembly into the GAC.

Demonstration Steps

You will find the steps in the **Demonstration: Signing and Installing an Assembly into the GAC** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD12_DEMO.md.

Demonstration: Specifying the Data to Include in the Grades Report Lab



In this demonstration, you will see the tasks that you will perform in the lab for this module.

Demonstration Steps

You will find the steps in the **Demonstration: Specifying the Data to Include in the Grades Report Lab** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD12_DEMO.md.

Lab Scenario

Lab Scenario

You decide to update the Grades application to use custom attributes to define the fields and properties that should be included in a grade report and to format them appropriately. This will enable further reusability of the Microsoft Word reporting functionality.

You will host this code in the GAC to ensure that it is available to other applications that require its services.

Lab: Specifying the Data to Include in the Grades Report

Lab: Specifying the Data to Include in the Grades Report

- Exercise 1: Creating and Applying the IncludeInReport attribute
- Exercise 2: Updating the Report
- Exercise 3: Storing the Grades.Utilities Assembly Centrally (If Time Permits)

Estimated Time: 75 minutes

Scenario

You decide to update the Grades application to use custom attributes to define the fields and properties that should be included in a grade report and to format them appropriately. This will enable further reusability of the Microsoft Word reporting functionality.

You will host this code in the GAC to ensure that it is available to other applications that require its services.

Objectives

After completing this lab, you will be able to:

- Define custom attributes.
- Use reflection to examine metadata at run time.
- Sign an assembly and deploy it to the GAC.

Lab Setup

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD12_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD12_LAK.md.

Module Review and Takeaways

Module Review and Takeaways

- Review Questions

In this module, you learned how to consume existing assemblies by using reflection and how to add additional metadata to types and type members by using attributes. You also learned how to generate code at run time by using CodeDOM and how you can ensure that your assemblies are versioned and available to other applications by using the GAC.

Review Questions

1. You are developing an application that enables users to browse the object model of a compiled type. At no point will the application attempt to execute any code; it will merely serve as a viewer. You notice the code that loads the assembly uses the **Assembly.LoadFrom** static method. This is the most suitable method taking into account the requirements of the application.

( )True
( )False

2. You are developing a custom attribute. You want to derive your custom attribute class from the abstract base class that underpins all attributes. Which class should you use?

( )Attribute
( )ContextAttribute
( )ExtensionAttribute
( )DataAttribute
( )AddInAttribute

3. You are reviewing some code that uses CodeDOM to generate managed Visual C# at run time. What does the following line of code do?

```
var method = new CodeEntryPointMethod();
```

( )Defines an instance method with a random name.
( )Defines an instance method named EntryPoint.
( )Defines a static method named EntryPoint.
( )Defines an instance method named Main.
( )Defines a static method named Main.

4. The **FourthCoffee.Core.dll** assembly has 2.1.0.24 as its version number. The number 24 in the version number refers to the build number.

( )True
( )False