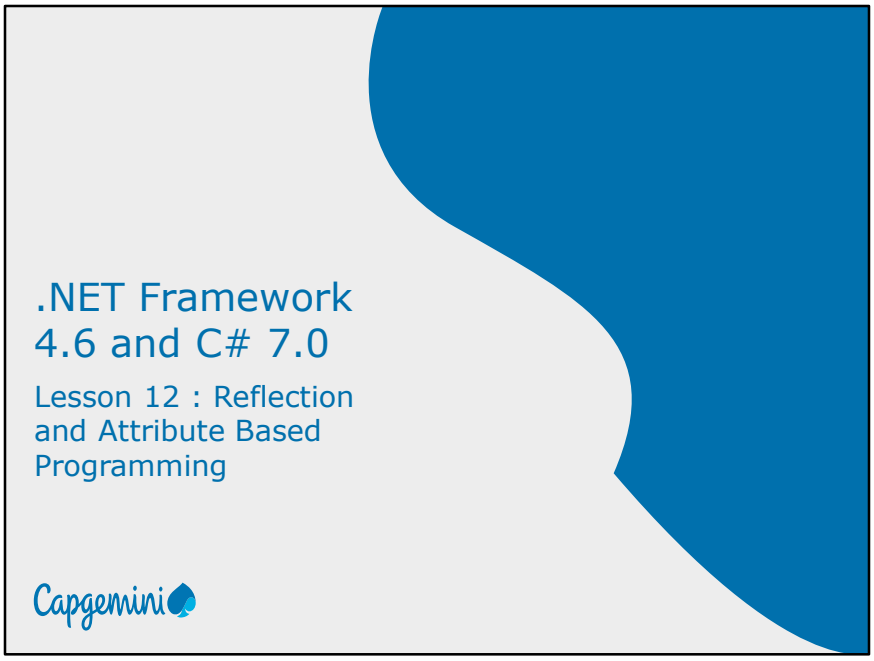

.NET Framework
4.6 and C# 7.0
Lesson 12 : Reflection
and Attribute Based
Programming
Capgemini

Explain the lesson coverage

Lesson Objectives

- In this lesson we will cover the following:
  - .NET Assemblies
  - Introduction to .NET Reflection
  - Obtaining details about types from the assembly
  - Obtaining details about methods, properties and fields
  - What are Attributes
  - Creating Custom Attributes
  - Retrieving Attribute Details with Reflection

Concept of Assembly

- An assembly can be viewed as a unit of deployment.
- An assembly is self-describing binary (DLL or EXE) containing collection of types and optional recourses.
- .NET binaries contain code constructed using Microsoft Intermediate Language (MSIL or simply IL), which is platform and CPU agnostic
- It contains "metadata" which completely describes each type

Assemblies:

Overview of .NET Assemblies:

.NET applications are constructed by piecing together any number of assemblies. Simply put, an assembly is nothing more than a versioned, self-describing binary (DLL or EXE) containing some collection of types (classes, interfaces, structures, etc.) and optional recourses (images, string tables and whatnot). One thing to be painfully aware of right now, is that the internal organization of a .NET assembly is nothing like the internal organization of a classic COM server (regardless of the shared file extensions).

The physical format of a .NET binary is actually more similar to a traditional portable executable (PE) file. The true difference is that a traditional PE file contains instructions that target a specific platform and specific CPU. In contrast, .NET binaries contain code constructed using Microsoft Intermediate Language (MSIL or simply IL), which is platform and CPU agnostic. At runtime, the internal IL is compiled on the fly (using a just in time compiler) to platform and CPU specific instructions. This of course is a powerful extension of classic COM, in that .NET assemblies are poised to be platform neutral entities which are not necessarily tied to the Windows operating system.

.NET Assemblies

- Assemblies are the building blocks of any .NET application
- All functionality of .NET application is exposed via assemblies
- Assemblies form a unit of deployment and versioning
- Assemblies contain modules which in turn contain various types (classes, structures, enumerations etc.)

Benefits of Assemblies

- Assemblies provide the following benefits:
- Promote code reuse
- Establish a Type Boundary
- Are versionable and self-describing Entities
- Enable Side-By-Side Execution

Assemblies:
Benefits of Assemblies:
Assemblies Promote Code Reuse

Assemblies contain code that will be executed by the .NET runtime. As you might imagine, the types and resources contained within an assembly can be shared and reused by multiple applications. When a .NET aware language adheres to the rules of the Common Language Specification (CLS), your choice of language becomes little more than a personal preference.

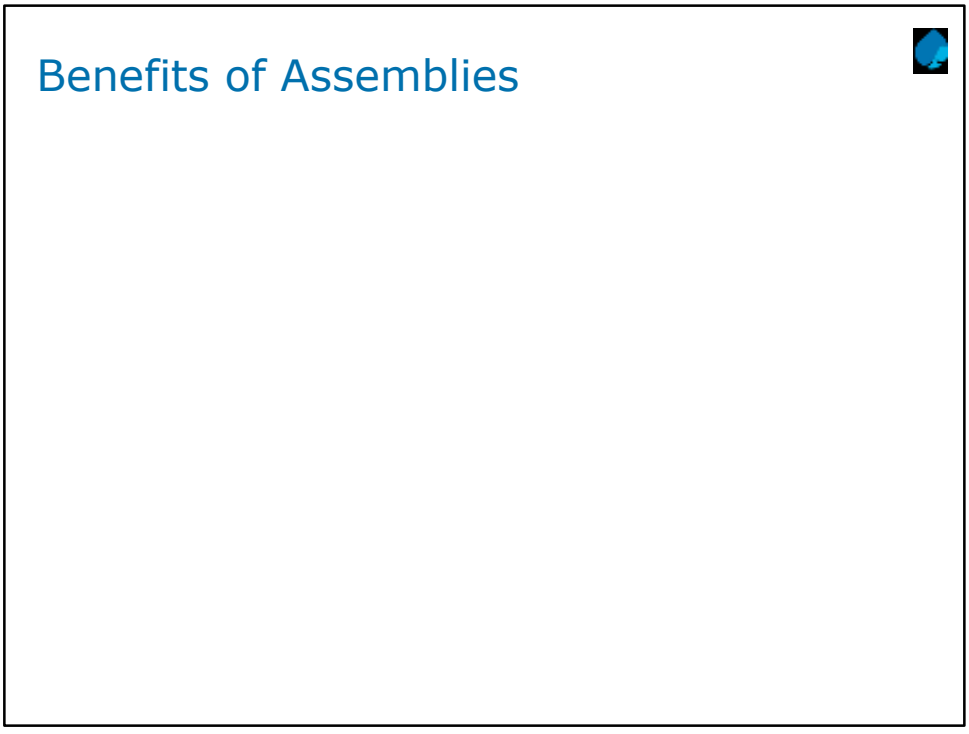
Assemblies Establish a Type Boundary

Assemblies are used to define a boundary for the types (and resources) it contains. In .NET, the identity of a given type is defined (in part) by the assembly in which it resides. Therefore, if two assemblies each define an identically named type (class, structure or whatnot) they are considered independent entities in the .NET universe.

Assemblies are Versionable and Self Describing Entities

In the world of COM, the developer is in charge of correctly versioning a binary.

For example: To ensure binary compatibility between MyComServer.dll version 1.0 and MyComServer.dll version 2.4, the programmer must use basic common sense to ensure interface definitions remained unaltered…or run the risk of breaking client code. While a healthy dose of common sense will also come in handy under the .NET universe, the problem with the COM versioning scheme is that these programmer-defined techniques are not enforced by the runtime.



Assemblies (contd.):
Benefits of Assemblies (contd.):
Assemblies Enable Side-By-Side Execution

Perhaps the biggest advantage of the .NET assembly, is the support for multiple versions of the same assembly to be loaded (and understood) by the runtime. Thus, it is possible to install and load multiple versions of the same assembly on a single machine. In this way, clients are isolated from other incompatible versions of the same assembly.

Furthermore, it is possible to control which version of a (shared) assembly is to be loaded using application configuration files. These files are little more than a simple text file describing (via XML syntax) the version, and possibly location, of the assembly to be used by the calling application.

Private Assemblies

- Private assemblies are a collection of types that are only used by the application with which it has been deployed.
- Private assemblies are required to be located within the main directory of the owing application.

Assemblies:

Private Assemblies:

Private assemblies are a collection of types that are only used by the application with which it has been deployed.

For example: CarLibrary.dll is a private assembly used by the CSharpCarClient and VBCarClient applications. When you create a private assembly, the assumption is that the collection of types will only be used by the 'owning' application, and not shared with other applications on the system.

Private assemblies are required to be located within the main directory of the owing application (termed the 'application directory') or a subdirectory thereof. Uninstalling (or replicating) an application which makes exclusive use of private assemblies is a no-brainer. Delete (or copy) the application folder. Unlike classic COM, you do not need to worry about numerous orphaned registry settings. More importantly, you do not need to worry that the removal of private assemblies will break any other applications on the machine!

Identity of a Private Assembly

- The 'identity' of a private assembly consists of friendly string name and numerical version
- Both are recorded in the Assembly Manifest
- The friendly name is created based on the name of the binary module which contains the Assembly's Manifest

Assemblies:

The Identity of a Private Assembly:

The 'identity' of a private assembly consists of a friendly string name and numerical version, both of which are recorded in the assembly manifest. The friendly name is created based on the name of the binary module which contains the assembly's manifest.

For example: If you examine the manifest of the CarLibrary.dll assembly, you will find the following:

```
.assembly CarLibrary as "CarLibrary"
{
…
  .ver 1:0:332:34940
}
```

However, given the nature of a private assembly, it should make sense that the .NET runtime does not bother to apply any version policies when loading the assembly. The assumption is that private assemblies do not need to have any elaborate version checking, given that the client application is the only entity that 'knows' of its existence.

The only mandatory requirement of a shared assembly is that its friendly string

name is unique among all assemblies in the application. Unlike COM, private assemblies do not need a universally unique ID (AppID) as only one entity is aware of its existence. As an interesting corollary, you should understand that it will be (very) possible for a single machine to have multiple copies of the same private assembly in various application directories.

Shared Assemblies

- Shared assemblies can be used by several clients on a single machine
- Shared assemblies are installed into a machine wide "Global Assembly Cache" (GAC)
- A shared assembly must be assigned a "shared name" (also known as a "strong name")

Assemblies:

Understanding Shared Assemblies:

Like a private assembly, a "shared" assembly is a collection of types and (optional) resources contained within some number of modules. The most obvious difference between shared and private assemblies is the fact that shared assemblies can be used by several clients on a single machine. Clearly, if you wish to create a machine wide class library, a shared assembly is the way to go.

A shared assembly is typically not deployed within the same directory as the application making use of it. Rather, shared assemblies are installed into a machine wide "Global Assembly Cache", which lends itself to yet another colorful acronym in the programming universe: the GAC. The GAC itself is located under the <drive>: \ Windows \ Assembly subdirectory.

This is yet another major difference between the COM and .NET architectures. In COM, shared applications can reside anywhere on a given machine, provided they are property registered. Under .NET, shared assemblies are typically placed into a centralized well-known location (the GAC).

10.2: Strong Names

Understanding Strong Names

- For creating a Shared assembly, create a unique "strong name" for the assembly.
- A strong name contains the following information:
  - Friendly string name and optional culture information (just like a private assembly)
  - Version identifier
  - Public key value
  - Embedded digital signature

Strong Names:

Unlike private assemblies, a shared assembly requires additional version information beyond the friendly text string + numerical version combination.

As you may have guessed, the .NET runtime does enforce version checking for a shared assembly before it is loaded on behalf of the calling application. In addition, a shared assembly must be assigned a "shared name" (also known as a "strong name").

Understanding Strong Names

- To provide a strong name for an assembly, generate the public / private key data using sn.exe utility.
  - This will create a strong name key file that contains data for two distinct but mathematically related keys, the "public" key and the "private" key.
- Inform the C# compiler about the location of the *.snk file.
  - It will record the full public key value in the Assembly Manifest using the publickey token at the time of compilation.

Understanding Strong Names

- The compiler will also generate a unique hash code based on the contents of the entire assembly.
- This hash code is combined with the private key data within the *.snk file to yield the digital signature embedded within the assembly's CLR header data.
- The CLR Header is a block of data that all .NET files must support in order to be hosted by the CLR.

Assembly Metadata

- Assembly MetaData:
  - Every Assembly is self describing i.e. it consists of meta data which describes itself.
  - Metadata is defined as "Data about data".
  - Metadata is generated by compiler and stored in the EXE or DLL.
  - Metadata also contains data about System Level attributes.

Common Language Runtime (CLR):

To enable the runtime to provide services to managed code, language compilers must emit metadata. Metadata is binary information describing your program that is stored either in a common language runtime portable executable (PE file) or in memory. When you compile your code into a PE file, metadata is inserted into one portion of the file, while your code is converted to Microsoft intermediate language (MSIL) and inserted into another portion of the file. Every type and member defined and referenced in a module or assembly is described within metadata. When code is executed, the runtime loads metadata into memory and references it to discover information about your code's classes, members, inheritance, and so on.

Metadata describes every type and member defined in your code in a multi-language fashion. Metadata stores the following information:

Description of the assembly:

Identity (name, version, culture, public key)

The types that are exported

Other assemblies that this assembly depends on

Security permissions needed to run

Description of types:

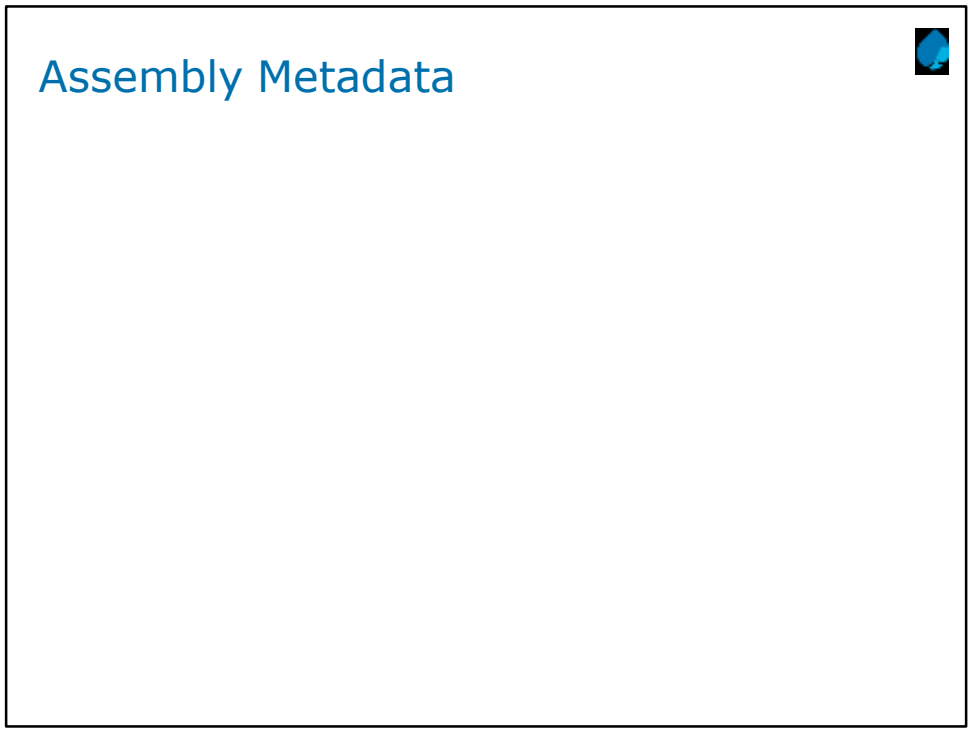
Name, visibility, base class, and interfaces implemented
Members (methods, fields, properties, events, nested types)

Attributes:

Additional descriptive elements that modify types and members

Assembly Metadata

- Some items of Metadata defined in .NET framework are:
  - Description of Deployment unit (Assembly)
  - Name, Version Culture (language used)
  - Public key for verification
  - Types exported by Assembly
  - Dependencies (other assemblies which this assembly depends upon).
  - Security permission needed to run
  - Base Classes and interfaces used by the assembly
  - Custom attributes (Optional)



Common Language Runtime (CLR):
Benefits of Metadata:
Metadata is the key to a simpler programming model, eliminating the need for Interface Definition Language (IDL) files, header files, or any external method of component reference. Metadata allows .NET languages to describe themselves automatically in a multilanguage manner, unseen by both the developer and the user. Additionally, metadata is extensible through the use of attributes. Metadata provides the following major benefits:

Self-describing files

Common language runtime modules and assemblies are self-describing. A module's metadata contains everything needed to interact with another module. Metadata automatically provides the functionality of IDL in COM, allowing you to use one file for both definition and implementation. Runtime modules and assemblies do not even require registration with the operating system. As a result, the descriptions used by the runtime always reflect the actual code in your compiled file, which increases application reliability.

Language interoperability and easier component-based design

Metadata provides all the information required about compiled code for you to inherit a class from a PE file written in a different language. You can create an instance of any class written in any managed language (any language that targets the Common Language Runtime) without worrying about explicit marshaling or using custom interoperability code.

Attributes

The .NET Framework allows you to declare specific kinds of metadata, called attributes, in your compiled file. Attributes can be found throughout the .NET Framework and are used to control in more detail how your program behaves at run time. Additionally, you can emit your own custom metadata into .NET Framework files through user-defined custom attributes.

Features

- Multiple Language Integration and support:
  - CLR is designed to support Multiple Languages.
  - CLR allows complete integration amongst these languages.
  - CLR enforces a Common Type System (CTS). This makes .NET languages work together transparently.

Common Language Runtime (CLR):

The Common Language Runtime makes it easy to design components and applications whose objects interact across languages. Objects written in different languages can communicate with each other, and their behaviors can be tightly integrated.

For example: You can define a class and then use a different language to derive a class from your original class, or call a method on the original class.

You can also pass an instance of a class to a method on a class written in a different language. This cross-language integration is possible because language compilers and tools that target the runtime use a common type system, that is, a subset of language features that is supported by a broad set of compliant tools.

CLS-compliant components and tools are guaranteed to interoperate with other CLS-compliant components and tools, defined by the runtime, and they follow the runtime's rules for defining new types, as well as for creating, using, persisting, and binding to types.

Features

- Multiple Language Integration and support (contd.):
  - Previously one language could instantiate components written in another by using COM. However, sub classing such a component was difficult and required wrappers
  - In .NET framework, using one language to subclass a class written in another language is very straight forward.
  - A class in VB can inherit from a base class written in C# or COBOL.
  - In fact, the VB Program does not even need to know the language used for base class.

Common Language Runtime (CLR):

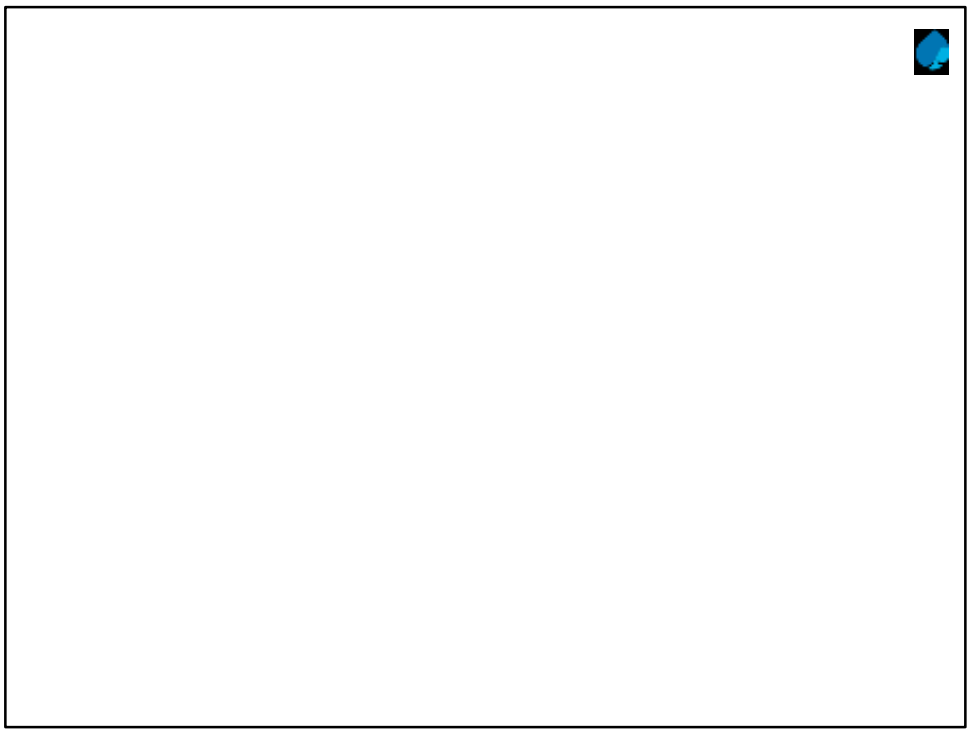
The Common Type System (CTS) defines how types are declared, used, and managed in the runtime. CTS is also an important part of the runtime's support for cross-language integration.

The common type system performs the following functions:

Establishes a framework that enables cross-language integration, type safety, and high performance code execution.

Provides an object-oriented model that supports the complete implementation of many programming languages.

Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.



Compilation and Execution in .NET:

Security engine: This provides evidence-based security using information about the origin of the code as well as the user. The security engine offers the safe execution of the semi-trusted code, protects from malicious software and several kinds of attacks, and allows controlled, identity-based access to resources.

Debug engine: This provides support for such tools as debuggers and tracers. Several classes in the Microsoft .NET Framework class library rely on the services provided by the debug engine. For example, we can use the Debug and Trace classes found in the System.Diagnostics namespace.

Type checker: This guarantees type safety guarding us from unsafe casts and uninitialized variables. Type checking is performed at the JIT-compilation step in the process called “verification”. At that point, it examines code and attempts to determine whether or not the code is type safe.

Exception manager: This provides structured exception handling. Exception handling is the essential part of the Common Language Runtime and is supported through the huge hierarchy of classes based on the Exception class and its derived classes.

Thread management: This supports multithreading. It includes starting and ending threads, suspending and scheduling threads, and thread pooling, as well as

providing thread manipulation support at runtime. This is achieved via the Thread class implemented in the System.Threading namespace.

COM Marshaler: This provides marshaling to and from COM. It includes the data type conversions between COM types and .NET types and back.

.NET Framework class library support: This provides integration between the runtime and code via the Common Type System.

Demo

- Demo on Shared Assemblies

Overview

- Reflection is ability to find information about types contained in an assembly at run time.
- .NET provides a whole new set of APIs to introspect assemblies and objects.
- All the APIs related to reflection are located under System.Reflection namespace.
- .NET reflection is a powerful mechanism which allows you to inspect type information and invoke methods on those types at runtime.

Reflection is ability to find information about types contained in an assembly at run time. Prior to .NET languages like C++ provided such ability in a limited sense. .NET provides a whole new set of APIs to introspect assemblies and objects. .NET reflection is a powerful mechanism which not only allows you to inspect type information but also allows you to invoke methods on those types at runtime. Certain reflection APIs also allow creating of assembly in memory dynamically and use it in your code.

Obtaining details about types

- Before obtaining any information about types contained in an assembly we must first load the assembly
  - Assembly myassembly = Assembly.LoadFrom("employee.dll");
- The next step is to obtain a list of various types contained in the assembly.
  - Types mytypes[] = myassembly.GetTypes();
  - Type mytype=myassembly.GetType("Company.Employee");

This statement loads an assembly called employee.dll. You can substitute your own path here. Assembly class has a static method called LoadFrom that loads the specified assembly in memory. The method returns an instance of assembly class itself.

There are two methods to get type information . The method GetTypes returns an array of System.Type objects. The method GetType returns a type object having details of specified object. Note that in our example Company is the namespace. In case your assembly do not contain any namespace you will simply write the type name

Obtaining type details

- The Type class has following properties that gives details about the type under consideration :
  - Name : Gives name of the type
  - FullName : Give fully qualified name of the type
  - Namespace : Gives namespace name
  - IsClass
  - IsInterface
  - IsAbstract
  - IsSealed
  - IsPublic

Obtaining details about methods, properties and fields

- Each type may have fields (member variables), properties and methods.
- The details about each of these types are obtained by following methods of the Type object.
  - GetMembers() : Gives array of MemberInfo objects
  - GetFields() : Gives array of FieldInfo objects
  - GetProperties() : Gives array of PropertyInfo objects
  - GetMethods() : Gives array of MethodInfo objects

Note that you can also get information about specific method, property or field using GetMethod("mymethod"), GetProperty("myprop") or GetField("myfield") methods.

```
MethodInfo[] mymethods= mytype.GetMethods();
MethodInfo mymethod = mytype.GetMethod("GetSalary");
```

Properties and methods of MethodInfo Object

- Name
- IsPrivate
- IsPublic
- IsStatic
- IsConstructor
- ReturnType
- GetParameters()
- Invoke()

Properties and methods of PropertyInfo Object

- Name
- CanRead
- CanWrite
- PropertyType
- GetValue()
- SetValue()

Properties and methods of FieldInfo Object

- Name
- FieldType
- IsPublic
- IsPrivate
- IsStatic
- GetValue()
- SetValue()

Demo

- Demo on Using Type Class to look into an Assembly

Add the notes here.

What are Attributes?

- Attributes concept in .NET is a way to mark or store meta data about the code in assembly.
- Often it is an instruction meant for the runtime.
- The Runtime can change its behavior or course of action based on the attribute present.
- In .NET framework there are many built in attributes
- For e.g. : Serializable, NonSerlalized, XmlIgnore, WebMethod to name few

What are Attributes

Attributes provide a powerful method of associating declarative information with C# code (types, methods, properties, and so forth). Once associated with a program entity, the attribute can be queried at run time and used in any number of ways.

The common language runtime allows you to add keyword-like descriptive declarations, called attributes, to annotate programming elements such as types, fields, methods, and properties. Attributes are saved with the metadata of a Microsoft .NET Framework file and can be used to describe your code to the runtime or to affect application behavior at run time. While the .NET Framework supplies many useful attributes, you can also design and deploy your own.

Essentially attributes are a means of decorating your code with various properties at compile time. This can be as marking a class as Serializable with the SerializableAttribute. In this case when you try serializing an object, the runtime checks to see if the object has the Serializable attribute applied to it.

In this case the attribute is nothing more than a flag to let the runtime know that the classes author gave the OK for serialization (since this could expose some sensitive data to someone who should see it).

What are Attributes? (contd…)

- In .NET an Attribute is a sub class of System.Attribute.
- Conventionally Attribute class names would end with "Attribute" word.
- Attributes can be applied to various code parts in .NET and are called Targets for an attribute
- For instance class, fields, methods, constructors can be targets for an attribute.
- We can apply multiple attributes to a target.

Applying Attributes

Use the following process to apply an attribute to an element of your code.

Define a new attribute or use an existing attribute by importing its namespace from the .NET Framework.

Initialize the attribute directly preceding the element that you want described, calling the attribute's constructor with the desired flag or information.

The following code example shows how to declare System.ObsoleteAttribute, which marks code as obsolete. The string "Will be removed in next version" is passed to the attribute. This attribute causes a compiler warning that displays the passed string when code that the attribute describes is called.

```
using System;
public class MainApp
{
   public static void Main()
   {
      //This generates a compile-time warning.
      int MyInt = Add(2,2);
```

```
    }
    [Obsolete("Will be removed in next version")]
    public static int Add( int a, int b)
    {
       return (a+b);
    }
}
```

Creating Custom Attributes

- We can define our own attributes by sub classing System.Attribute.
- While creating our own attribute we also decide about attribute usages
  - for example whether it can be applied multiple times, what are targets? etc.
- Same as built in framework attributes these attributes can be further be applied to the program elements as per valid targets allowed.

The .NET Framework also allows you to define your own attributes. Clearly, these attributes will not have any effect on the compilation process, because the compiler has no intrinsic awareness of them. However, these attributes will be emitted as metadata in the compiled assembly when they are applied to program elements.

By itself, this metadata might be useful for documentation purposes, but what makes attributes really powerful is that by using reflection, your code can read this metadata and use it to make decisions at runtime. This means that the custom attributes that you define can have a direct effect on how your code runs. For example, custom attributes can be used to enable declarative code access security checks for custom permission classes, associate information with program elements that can then be used by testing tools, or be used when developing extensible frameworks that allow the loading of plugins or modules.

Creating Custom Attributes

- Consider this Custom Attribute

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class AuthorAttribute : Attribute
{
        private string name;
        public AuthorAttribute(string name)
        {
                this.name = name;
        }

        public string AuthorName { get; set; }
}
```

Declaring an Attribute Class

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute : System.Attribute
{
  pivate string Url;
  private string topic;

  public string Topic              // Topic is a named parameter
  {
    get
    {
      return topic;
    }
    set
    {

      topic = value;
    }
  }

  public HelpAttribute(string url)  // url is a positional parameter
  {
    this.Url = url;
  }
}
```

Creating Custom Attributes

- This Attribute as per usage setting can be applied only to a class or a structure.
- It has 1 parameter for Author Name
- This attribute can be applied to the class in the following way

```
[Author("Steven Spielberg")]
public class Order
{
}
```

Using an Attribute Class
Here's a simple example of using the attribute declared in the previous section:

```
[HelpAttribute("http://localhost/MyClassInfo")]
public class MyClass
{
}
```

In this example, the HelpAttribute attribute is associated with MyClass.
Note By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET Framework. However, you do not need to specify the attribute suffix when using attributes in code. For example, you can specify HelpAttribute as follows:

```
[Help("http://localhost/MyClassInfo")]
public class MyClass
{
}
```

Retrieving Attribute Details

- The information regarding the attributes that are applied can be obtained using Reflection API in following way.

```
AuthorAttribute atr = (AuthorAttribute )
            Attribute.GetCustomAttribute(
                  typeof(Order), typeof (AuthorAttribute ));
if(atr == null)
  Console.WriteLine("The attribute was not applied.");
else
  Console.WriteLine("The Name Paramter Value is: {0}." , atr.Name);
```

Accessing Attributes Through Reflection

Once attributes have been associated with program elements, reflection can be used to query their existence and values. The main reflection methods to query attributes are contained in the System.Reflection.MemberInfo class (GetCustomAttributes family of methods). The following example demonstrates the basic way of using reflection to get access to attributes:

```
publlic class MainClass
{
  public static void Main()
  {
    System.Reflection.MemberInfo info = typeof(MyClass);
    object[] attributes = info.GetCustomAttributes(true);
    for (int i = 0; i < attributes.Length; i ++)
    {
      System.Console.WriteLine(attributes[i]);
    }
  }
}
```

Demo

- Creating Custom Attribute

Add the notes here.

Summary

- In this module we studied:
- What is Reflection
- The use of Type class
- Obtaining details about types from assembly
- Obtaining details about methods, properties and Fields.
- What are attributes
- Applying Attributes and Attribute Usages
- How we can create our own attributes
- Retrieving Attribute details

Review Questions

- All the APIs related to reflection are located under _____ Namespace
- Which class is used to Load the Assembly?
- What are Attributes?
- What is Attribute Usage?
- Name few built in attributes in Framework