.NET Framework 4.7 and C# 8.0

Lesson 12 : Reflection and Attribute Based Programming

Capgemini

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

.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

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.

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

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”)

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

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.

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)

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.

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.

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.

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");

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

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

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? (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.

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.

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; }
}
```

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
{
}
```

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);
```

Demo

- Creating Custom Attribute

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