**Lecture 12 – reflection**

Reflection is the ability of a managed code to read its own metadata for the purpose of finding assemblies, modules and type information at runtime. In other words, reflection provides objects that encapsulate assemblies, modules and types. A program reflects on itself by extracting metadata from its assembly and using that metadata either to inform the user or to modify its own behavior. Reflection is similar to C++ RTTI (Runtime Type Information), but much broader in scope and capability. By using Reflection in C#, one is able to find out details of an object, method, and create objects and invoke methods at runtime. The System.Reflectionnamespace contains classes and interfaces that provide a managed view of loaded types, methods, and fields, with the ability to dynamically create and invoke types. When writing a C# code that uses reflection, the coder can use the typeof operator to get the object's type or use the GetType() method to get the type of the current instance. Here are sample codes that demonstrate the use of reflection:

**Example 1**

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

using System;

using System.Reflection;

public class MyClass

{

public virtual int AddNumb(int numb1,int numb2)

{

int result = numb1 + numb2;

return result;

}

}

class MyMainClass

{

public static int Main()

{

Console.WriteLine ("\nReflection.MethodInfo");

*// Create MyClass object*

MyClass myClassObj = new MyClass();

*// Get the Type information.*

Type myTypeObj = myClassObj.GetType();

*// Get Method Information.*

MethodInfo myMethodInfo = myTypeObj.GetMethod("AddNumb");

object[] mParam = new object[] {5, 10};

*// Get and display the Invoke method.*

Console.Write("\nFirst method - " + myTypeObj.FullName + " returns " +

myMethodInfo.Invoke(myClassObj, mParam) + "\n");

return 0;

}

}

Firstly, the code snippet below will get the type information:

Hide   Copy Code

Type myTypeObj = Type.GetType("MyClass");

And myTypeObj will now have the required information about MyClass. Therefore we can now check if the class is an abstract class or a regular class by using either of these statements:

Hide   Copy Code

myTypeObj.IsAbstract

or:

Hide   Copy Code

myTypeObj.IsClass

The code snippet below will get the method's information. And the method that we are interested in this case isAddNumb:

Hide   Copy Code

Methodinfo myMethodInfo = myTypeObj.GetMethod("AddNumb");

The following code snippet will invoke the AddNumb method:

Hide   Copy Code

myMethodInfo.Invoke(myClassObj, mParam);

*//Example2: In this example, we will use the typeof keyword to obtain the*

*// System.Type object for a type.*

Public class MyClass2

{

int answer;

public MyClass2()

{

answer = 0;

}

public int AddNumb(intnumb1, intnumb2)

{

answer = numb1 + numb2;

return answer;

}

}

The code snippet below gets the System.Type object for the MyClass2 type.

Hide   Copy Code

Type type1 = typeof(MyClass2);

So we will now be able to create an instance of the type1 object by passing the type1 object to theActivator.CreateInstance(type1) method.

Hide   Copy Code

object obj = Activator.CreateInstance(type1);

Then we can now invoke the AddNumb method of the MyClass2 class by first creating an array of objects for the arguments that we would be passing to the AddNumb(int, int) method.

Hide   Copy Code

object[] mParam =newobject[] {5, 10};

Finally, we would invoke the AddNumb(int, int) method by passing the method name AddNumb toSystem.Type.InvokeMember() with the appropriate arguments.

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

int res = (int)type1.InvokeMember("AddNumb", BindingFlags.InvokeMethod,null,

obj, mParam);

*//Here is the complete code:*

using System;

using System.Reflection;

namespace Reflection

{

class Class1

{

public int AddNumb(int numb1, int numb2)

{

int ans = numb1 + numb2;

return ans;

}

[STAThread]

static void Main(string[] args)

{

Type type1 = typeof(Class1);

*//Create an instance of the type*

object obj = Activator.CreateInstance(type1);

object[] mParam = new object[] {5, 10};

*//invoke AddMethod, passing in two parameters*

int res = (int)type1.InvokeMember("AddNumb", BindingFlags.InvokeMethod,

null, obj, mParam);

Console.Write("Result: {0} \n", res);

}

}

}

**Attributes**

An **attribute** is a declarative tag that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc. in your program. You can add declarative information to a program by using an attribute. A declarative tag is depicted by square ([ ]) brackets placed above the element it is used for.

Attributes are used for adding metadata, such as compiler instruction and other information such as comments, description, methods and classes to a program. The .Net Framework provides two types of attributes: *the pre-defined* attributes and *custom built*attributes.

## **Specifying an Attribute**

Syntax for specifying an attribute is as follows:

[attribute(positional\_parameters, name\_parameter = value, ...)]

element

Name of the attribute and its values are specified within the square brackets, before the element to which the attribute is applied. Positional parameters specify the essential information and the name parameters specify the optional information.

## **Predefined Attributes**

The .Net Framework provides three pre-defined attributes:

* AttributeUsage
* Conditional
* Obsolete

## **AttributeUsage**

The pre-defined attribute **AttributeUsage** describes how a custom attribute class can be used. It specifies the types of items to which the attribute can be applied.

Syntax for specifying this attribute is as follows:

[AttributeUsage(

validon,

AllowMultiple=allowmultiple,

Inherited=inherited

)]

Where,

* The parameter validon specifies the language elements on which the attribute can be placed. It is a combination of the value of an enumerator *AttributeTargets*. The default value is *AttributeTargets.All*.
* The parameter *allowmultiple* (optional) provides value for the *AllowMultiple*property of this attribute, a Boolean value. If this is true, the attribute is multiuse. The default is false (single-use).
* The parameter inherited (optional) provides value for the *Inherited* property of this attribute, a Boolean value. If it is true, the attribute is inherited by derived classes. The default value is false (not inherited).

For example,

[AttributeUsage(AttributeTargets.Class |

AttributeTargets.Constructor |

AttributeTargets.Feild |

AttributeTargets.Method |

AttributeTargets.Property,

AllowMultiple = true)]

## **Conditional**

This predefined attribute marks a conditional method whose execution depends on a specified preprocessing identifier.

It causes conditional compilation of method calls, depending on the specified value such as**Debug** or **Trace**. For example, it displays the values of the variables while debugging a code.

Syntax for specifying this attribute is as follows:

[Conditional(

conditionalSymbol

)]

For example,

[Conditional("DEBUG")]

The following example demonstrates the attribute:

#define DEBUG

using System;

using System.Diagnostics;

public class Myclass

{

[Conditional("DEBUG")]

public static void Message(string msg)

{

Console.WriteLine(msg);

}

}

class Test

{

static void function1()

{

Myclass.Message("In Function 1.");

function2();

}

static void function2()

{

Myclass.Message("In Function 2.");

}

public static void Main()

{

Myclass.Message("In Main function.");

function1();

Console.ReadKey();

}

}

When the above code is compiled and executed, it produces the following result:

In Main function

In Function 1

In Function 2

## **Obsolete**

This predefined attribute marks a program entity that should not be used. It enables you to inform the compiler to discard a particular target element. For example, when a new method is being used in a class and if you still want to retain the old method in the class, you may mark it as obsolete by displaying a message the new method should be used, instead of the old method.

Syntax for specifying this attribute is as follows:

[Obsolete(

message

)]

[Obsolete(

message,

iserror

)]

Where,

* The parameter *message*, is a string describing the reason why the item is obsolete and what to use instead.
* The parameter *iserror*, is a Boolean value. If its value is true, the compiler should treat the use of the item as an error. Default value is false (compiler generates a warning).

The following program demonstrates this:

using System;

public class MyClass

{

[Obsolete("Don't use OldMethod, use NewMethod instead", true)]

static void OldMethod()

{

Console.WriteLine("It is the old method");

}

static void NewMethod()

{

Console.WriteLine("It is the new method");

}

public static void Main()

{

OldMethod();

}

}

When you try to compile the program, the compiler gives an error message stating:

Don't use OldMethod, use NewMethod instead

## **Creating Custom Attributes**

The .Net Framework allows creation of custom attributes that can be used to store declarative information and can be retrieved at run-time. This information can be related to any target element depending upon the design criteria and application need.

Creating and using custom attributes involve four steps:

* Declaring a custom attribute
* Constructing the custom attribute
* Apply the custom attribute on a target program element
* Accessing Attributes Through Reflection

The Last step involves writing a simple program to read through the metadata to find various notations. Metadata is data about data or information used for describing other data. This program should use reflections for accessing attributes at runtime. This we will discuss in the next chapter.

## **Declaring a Custom Attribute**

A new custom attribute should is derived from the **System.Attribute** class. For example,

//a custom attribute BugFix to be assigned to a class and its members

[AttributeUsage(AttributeTargets.Class |

AttributeTargets.Constructor |

AttributeTargets.Field |

AttributeTargets.Method |

AttributeTargets.Property,

AllowMultiple = true)]

public class DeBugInfo : System.Attribute

In the preceding code, we have declared a custom attribute named *DeBugInfo*.

## **Constructing the Custom Attribute**

Let us construct a custom attribute named *DeBugInfo*, which stores the information obtained by debugging any program. Let it store the following information:

* The code number for the bug
* Name of the developer who identified the bug
* Date of last review of the code
* A string message for storing the developer's remarks

The *DeBugInfo* class has three private properties for storing the first three information and a public property for storing the message. Hence the bug number, developer's name, and date of review are the positional parameters of the DeBugInfo class and the message is an optional or named parameter.

Each attribute must have at least one constructor. The positional parameters should be passed through the constructor. The following code shows the *DeBugInfo* class:

//a custom attribute BugFix to be assigned to a class and its members

[AttributeUsage(AttributeTargets.Class |

AttributeTargets.Constructor |

AttributeTargets.Field |

AttributeTargets.Method |

AttributeTargets.Property,

AllowMultiple = true)]

public class DeBugInfo : System.Attribute

{

private int bugNo;

private string developer;

private string lastReview;

public string message;

public DeBugInfo(int bg, string dev, string d)

{

this.bugNo = bg;

this.developer = dev;

this.lastReview = d;

}

public int BugNo

{

get

{

return bugNo;

}

}

public string Developer

{

get

{

return developer;

}

}

public string LastReview

{

get

{

return lastReview;

}

}

public string Message

{

get

{

return message;

}

set

{

message = value;

}

}

}

## **Applying the Custom Attribute**

The attribute is applied by placing it immediately before its target:

[DeBugInfo(45, "Zara Ali", "12/8/2012", Message = "Return type mismatch")]

[DeBugInfo(49, "Nuha Ali", "10/10/2012", Message = "Unused variable")]

class Rectangle

{

//member variables

protected double length;

protected double width;

public Rectangle(double l, double w)

{

length = l;

width = w;

}

[DeBugInfo(55, "Zara Ali", "19/10/2012", Message = "Return type mismatch")]

public double GetArea()

{

return length \* width;

}

[DeBugInfo(56, "Zara Ali", "19/10/2012")]

public void Display()

{

Console.WriteLine("Length: {0}", length);

Console.WriteLine("Width: {0}", width);

Console.WriteLine("Area: {0}", GetArea());

}

}

**Question:**

1. **What mechanism does reflection provide to programmer?**
2. **What classes are used to get the information about assembly?**
3. **How are attributes help to create extensible applications?**