Assembly and Reflection

Source	http://www.codeproject.com/csharp/IntroReflection.asp

Reflection is a notable addition to the .NET Framework. Through Reflection, a program collects and manipulates its own metadata. It is a powerful mechanism to introspect the assemblies and objects at runtime. The Reflection APIs are contained in the `System.Reflection` **namespace. Reflection allows the programmer to inspect and collect information about the type, properties, methods and events of an object and to invoke the methods of that object through the** `Invoke` **method. Reflection is a powerful tool to develop Reverse Engineering applications, class browsers and property editors.**

In this article, I will provide examples for the following uses of Reflection:

- **Collecting metadata of an assembly and discovering the types of classes in it.**
- **Dynamic invocation of methods and properties.**
- **Through late binding, the properties and methods of a dynamically instantiated object can be invoked based on type discovery.**
- **Creating types at runtime using** `Reflection.Emit`**.**
- **This is the most usable feature of reflection. The user can create new types at runtime and use them to perform required tasks.**

Reflection to find out the assemblies used by a program

```
using System;
using System.Reflection ;

namespace ReflectionDemoCSharp
{

    class ReferencedAssemblies
    {
        [STAThread]
        static void Main(string[] args)
        {
            Assembly[] appAssemblies =
              System.AppDomain.CurrentDomain.GetAssemblies ();

            foreach (Assembly assembly in appAssemblies )
            {
                Console.WriteLine (assembly.FullName );
            }
            Console.ReadLine ();
        }
    }

}
```

Output

```
mscorlib, Version=1.0.5000.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
```

```
ReflectionDemoCSharp, Version=1.0.1882.29904, Culture=neutral,
PublicKeyToken=null
```

The `System.AppDomain` **class represents an application domain.** `AppDomain` **is an isolated environment where the application executes.**

```
Assembly[] appAssemblies = System.AppDomain.CurrentDomain.GetAssemblies ();
```

The `GetAssemblies()` **method returns the assemblies loaded by** `ReflectionDemoCSharp`**. It outputs** `mscorlib.dll` **and** `ReflectionDemoCSharp`**.**

Reflecting on an assembly and Discovering the existing types

First of all, we load an assembly dynamically with the Assembly.Load() method.

```
public static Assembly.Load(AssemblyName)
```

We pass the MsCorLib*.dll*.

```
Assembly LoadedAssembly = Assembly.Load("mscorlib.dll");
```

Once the assembly is loaded, we call the GetTypes() method to get an array of Type objects.

```
System.Type[] ExistingTypes = LoadedAssembly.GetTypes ();
```

The Type returned can represent the types of classes, interfaces, or enumerators.

```
using System;
using System.Reflection ;

namespace ReflectionDemoCSharp
{
    class ReflectedTypes
    {
        [STAThread]
        static void Main(string[] args)
        {
            Assembly LoadedAssembly = Assembly.Load ("mscorlib.dll");
            System.Type[] ExistingTypes = LoadedAssembly.GetTypes ();
            foreach(Type type in ExistingTypes)
                Console.WriteLine (type.ToString ());

            Console.WriteLine (ExistingTypes.Length +
               " Types Discovered in mscorlib.dll");

            Console.ReadLine ();
        }
    }
}
```

Output (The actual output will fill several pages, so only few are shown.)

```
System.Object
System.ICloneable
System.Collections.IEnumerable
System.Collections.ICollection
System.Collections.IList
System.Array
System.Array+SorterObjectArray
System.Array+SorterGenericArray
System.Collections.IEnumerator

1480 Types Discovered in mscorlib.dll
```

Reflecting on a Single Type

In our next example, we will reflect on a single type and find out its members. The `Type.GetType(TypeName)` **method takes a string argument and returns the corresponding** `System.Type`**. We query the type using the** `Type.GetMembers()` **method to return an array of members in the type.**

```
using System;
using System.Reflection ;

namespace ReflectionDemoCSharp
{
    class ReflectedTypes
    {
        [STAThread]
        static void Main(string[] args)
        {
            Type TypeToReflect = Type.GetType("System.Int32");
            System.Reflection.MemberInfo[] Members =type.GetMembers();

            Console.WriteLine ("Members of "+TypeToReflect.ToString ());
            Console.WriteLine();
            foreach (MemberInfo member in Members )
                Console.WriteLine(member);
            Console.ReadLine ();
        }
    }
}
```

Here is the output:

```
Members of System.Int32

Int32 MaxValue
Int32 MinValue
System.String ToString(System.IFormatProvider)
System.TypeCode GetTypeCode()
System.String ToString(System.String, System.IFormatProvider)
Int32 CompareTo(System.Object)
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
```

```
System.String ToString(System.String)
Int32 Parse(System.String)
Int32 Parse(System.String, System.Globalization.NumberStyles)
Int32 Parse(System.String, System.IFormatProvider)
Int32 Parse(System.String, System.Globalization.NumberStyles,
System.IFormatProvider)
System.Type GetType()
```

- `System.Reflection.MemberInfo[] Members =type.GetMembers()` **- returns all the members of the** `Type` **being queried.**
- `System.Reflection.MethodInfo[] Methods =Type.GetMethods()` **- returns only the methods in the** `Type` **being queried.**
- `System.Reflection.FieldInfo[] Fields =Type.GetFields()` **- returns only the fields in the** `Type` **being queried.**
- `System.Reflection.PropertyInfo[] Properties = type.GetProperties ()` **- returns the properties in the** `Type` **being queried.**
- `System.Reflection.EventInfo[] Events = type.GetEvents()` **- returns the events in the** `Type` **being queried.**
- `System.Reflection.ConstructorInfo[] Constructors = type.GetConstructors ()` **- returns the constructors in the** `Type` **being queried.**
- `System.Type[] Interfaces = type.GetInterfaces()` **- returns the interfaces in the** `Type` **being queried.**

Dynamic Invocation with Type.InvokeMember()

The next example demonstrates how to dynamically invoke a method using the method `type.InvokeMember()`**. The "Equals" method of** `System.String`**, which compares two strings for equality, is invoked using the** `InvokeMember()` **method. The program passes two string arguments for comparison. The** `type.InvokeMember()` **allows us to execute methods by name.**

Parameters of `InvokeMember()` **method are:**

1. **The first parameter to** `InvokeMember()` **is the name of the member we want to invoke. It is passed as a string.**
2. **The second parameter is a member of the** `BindingFlags` **enumeration.** `BindingFlags` **enumeration specifies flags that control binding and the way in which to look for members and types.**
3. **The third parameter is a Binder object that defines a set of properties and enables binding. Or it can be null, in which case the default Binder will be used. The Binder parameter gives the user explicit control over how the reflection selects an overloaded method and converts arguments.**
4. **The fourth parameter is the object on which to invoke the specified member.**
5. **The fifth parameter is an array of arguments to pass to the member to invoke.**

```
using System;
using System.Reflection ;

namespace ReflectionDemoCSharp
{
    class ReflectedTypes
```

```
    {
        [STAThread]
        static void Main(string[] args)
        {
            Type TypeToReflect = Type.GetType("System.String");
            object result = null;

            object[] arguments = {"abc","xyz"};
            result = TypeToReflect.InvokeMember ("Equals",
              BindingFlags.InvokeMethod, null, result, arguments);
            Console.WriteLine (result.ToString ());
            Console.ReadLine ();
        }
    }
}
```

The output in this case is `false`.

Reflection.Emit - Creating Types Dynamically at Runtime and Invoking their Methods

`Reflection.Emit` **supports dynamic creation of new types at runtime. You can create an assembly dynamically, and then you can define the modules, types and methods you want included in it. The assembly can run dynamically or can be saved to disk. The methods defined in the new assembly can be invoked using the** `Type.InvokeMember()` **method.** `Reflection.Emit` **also allows the compiler to emit its metadata and Microsoft Intermediate Language (MSIL) during runtime.**

Let us create a class `DoMath` **and define a method** `DoSum` **in the assembly named Math. The first thing to do is to create an object of type A**`ssemblyName` **and give it a name.**

```
AssemblyName assemblyName = new AssemblyName();

assemblyName.Name = "Math";
```

Next, we use the `AssemblyBuilder` **class to define a dynamic assembly, in the Current Domain of the application. We have to pass two parameters** `AssemblyName` **and an enumeration value of** `AssemblyBuilderAccess` **(Run, RunAndSave or Save). The value of** `AssemblyBuilderAccess` **determines whether the assembly can be run only or it can be saved to the disk.**

```
AssemblyBuilder CreatedAssembly =
AppDomain.CurrentDomain.DefineDynamicAssembly(assemblyName,
AssemblyBuilderAccess.RunAndSave );
```

Now, in our dynamically created assembly, we create an assembly Module. For this, use the created `AsemblyBuilder` **object and call its** `DefineDynamicModule()` **method, which in turn returns a** `ModuleBuilder` **object. We pass the name of the Module and the filename in which it will be saved.**

```
ModuleBuilder AssemblyModule =
         CreatedAssembly.DefineDynamicModule("MathModule","Math.dll");
```

Our next step is to create a public class in this `AssemblyModule`**. Let's define a public class named "**`DoMath`**".**

We use a `TypeBuilder` **class to dynamically define a class. For this, we call the** `AssemblyModule.DefineType()` **method. The** `DefineType()` **returns a** `TypeBuilder` **object.**

```
TypeBuilder MathType = AssemblyModule.DefineType("DoMath",
       TypeAttributes.Public | TypeAttributes.Class);
```

In the "DoSum" class, we create a method "Sum" which adds two integers and returns the result. The `MethodBuilder` **class is used to define the method, its parameter types and the return type. We call the** `TypeBuilder` **object's** `DefineMethod()` **method and pass the name of the method, its attributes, return type, and an array of types of the parameters.**

```
System.Type [] ParamTypes = new Type[] { typeof(int),typeof(int) };

MethodBuilder SumMethod = MathType.DefineMethod("Sum",
             MethodAttributes.Public, typeof(int), ParamTypes);
```

Next, we define the two parameters of the method "Sum" using the ParameterBuilder class. We call the `DefineParameter()` **method of the** `MethodBuilder` **object, passing the position, attribute of the parameter, and an optional name for the parameter.**

```
ParameterBuilder Param1 =
    SumMethod.DefineParameter(1,ParameterAttributes.In ,"num1");

ParameterBuilder Param2 =
    SumMethod.DefineParameter(2,ParameterAttributes.In ,"num2");
```

We then use the `MethodBuilder` object created earlier to get an `ILGenerator` object.

```
ILGenerator ilGenerator = SumMethod.GetILGenerator();
```

It is the `ILGenerator` **object that emits the** `opcode` **or Microsoft Intermediate Language (MSIL) instruction. These** `opcodes` **are the same** `opcodes` **generated by a C# compiler. The** `OpCodes` **class contains fields that represent MSIL instructions. We use these fields to emit the actual** `opcode`**. So we emit the** `opcode` **of the two arguments of the "Sum" method. The opcodes are pushed into the stack. Then we specify the operation – in our case, add two numbers. Now the stack will contain the sum of the two arguments. The** `OpCodes.Ret` **will return the value in the stack.**

```
ilGenerator.Emit(OpCodes.Ldarg_1);
ilGenerator.Emit (OpCodes.Ldarg_2);

ilGenerator.Emit (OpCodes.Add );
ilGenerator.Emit(OpCodes.Ret);
```

Now we create the class and return the assembly.

```
MathType.CreateType();
```

```
return CreatedAssembly;
```

Here is the example code:

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace ConsoleApplication1
{
    public class ReflectionEmitDemo
    {
        public Assembly CreateAssembly()
        {
            AssemblyName assemblyName = new AssemblyName();
            assemblyName.Name = "Math";

            AssemblyBuilder CreatedAssembly =
                 AppDomain.CurrentDomain.DefineDynamicAssembly(assemblyName,
                 AssemblyBuilderAccess.RunAndSave );

            ModuleBuilder AssemblyModule =
               CreatedAssembly.DefineDynamicModule("MathModule","Math.dll");

            TypeBuilder MathType =
                   AssemblyModule.DefineType("DoMath", TypeAttributes.Public
|
                   TypeAttributes.Class);

            System.Type [] ParamTypes = new Type[] { typeof(int),typeof(int)
};

            MethodBuilder SumMethod = MathType.DefineMethod("Sum",
                   MethodAttributes.Public, typeof(int), ParamTypes);

            ParameterBuilder Param1 =
                   SumMethod.DefineParameter(1,ParameterAttributes.In,
"num1");

            ParameterBuilder Param2 =
                   SumMethod.DefineParameter(2,ParameterAttributes.In,
"num2");

            ILGenerator ilGenerator = SumMethod.GetILGenerator();
            ilGenerator.Emit(OpCodes.Ldarg_1);
            ilGenerator.Emit (OpCodes.Ldarg_2);
            ilGenerator.Emit (OpCodes.Add );
            ilGenerator.Emit(OpCodes.Ret);

            MathType.CreateType();

            return CreatedAssembly;
        }
    }
}
```

Our next aim is to invoke the "Sum" method of the dynamically created Type. Let us create an object of the `ReflectionEmitDemo` **class and call its** `CreateAssembly` **method to return the dynamically created assembly. Then we reflect on the** `EmitAssembly` **to find out the "DoMath" type.**

```
ReflectionEmitDemo EmitDemo = new ReflectionEmitDemo();

Assembly EmitAssembly = EmitDemo.CreateAssembly();
System.Type MathType = EmitAssembly.GetType("DoMath");
```

Next, we prepare the parameters for the "Sum" method, and create an instance of the "DoMath" type on which we do the Invoke. We call the `Type.InvokeMember()` **method to invoke the "Sum" method. In the example below, we've passed the parameters 5 and 9.**

```
object[] Parameters = new object [2];

Parameters[0] =  (object) (5);

Parameters[1] =  (object) (9);

object EmitObj = Activator.CreateInstance(MathType,false);

object Result = MathType.InvokeMember("Sum",
     BindingFlags.InvokeMethod ,null,EmitObj,Parameters);

Console.WriteLine ("Sum of {0}+{1} is {2}",
   Parameters[0],Parameters[1],Result.ToString ());
```

The output is:

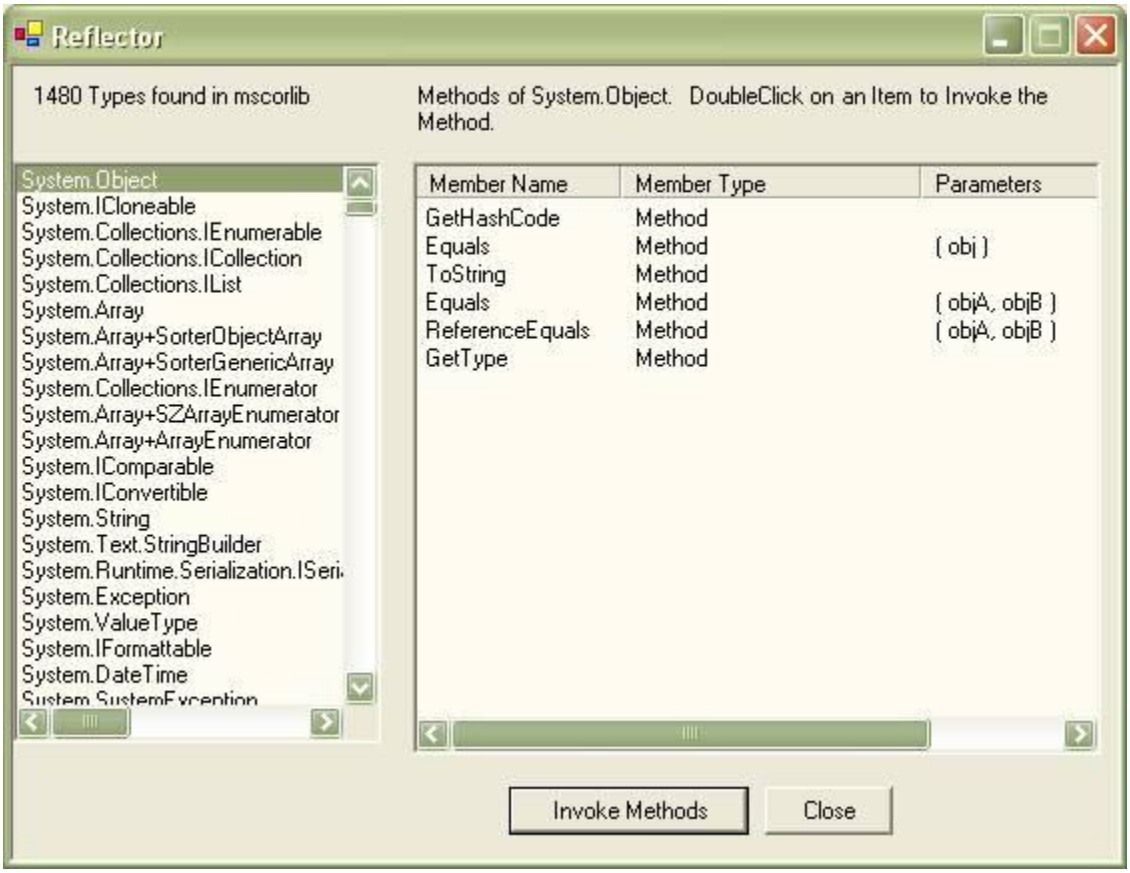
```
Sum of 5+9 is 14
```

Here is the example code:

```
using System;
using System.Reflection;

namespace ConsoleApplication1
{
    public class EmitDemoTest
    {
        static void Main()
        {
            ReflectionEmitDemo EmitDemo = new ReflectionEmitDemo();
            Assembly EmitAssembly = EmitDemo.CreateAssembly();

            System.Type MathType = EmitAssembly.GetType("DoMath");
            object[] Parameters = new object [2];
            Parameters[0] = (object) (5);
            Parameters[1] = (object) (9);
            object EmitObj = Activator.CreateInstance (MathType,false);

            object Result = MathType.InvokeMember("Sum",
                BindingFlags.InvokeMethod ,null,EmitObj,Parameters);
```

```
            Console.WriteLine("Sum of {0}+{1} is {2}",
                 Parameters[0],Parameters[1],Result.ToString());
            Console.ReadLine();
        }
    }
}
```

The attached ZIP file contains a Windows application which lets you examine the assemblies and its types.



~~~ End of Article ~~~