[C#](https://www.c-sharpcorner.com/technologies/csharp-programming)

Using Reflection with C# .NET

**Introduction**  
In this article, I’ll discuss about C# Reflection with an example. Here we’ll learn the way of getting type information using different ways and use of properties and methods of C# Reflection type class. Advanced Reflection topics like dynamically loading an assembly and late binding will also be discussed in this article. To complete the tutorial we have implemented here C# dictionary.  
  
**What is Reflection in C#?**  
Reflection provides objects (of type [Type](https://msdn.microsoft.com/en-us/library/system.type.aspx)) that describe assemblies, modules and types. We can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If we use attributes in our code, reflection enables us to access them.  
  
Here's a simple example of reflection using the static method **GetType** - inherited by all types from the **Object** base class - to obtain the type of a variable:

1. // Using GetType to obtain type information:
2. **int** i = 42;
3. System.Type type = i.GetType();
4. System.Console.WriteLine(type);

**The output is:**

System.Int32

The following example uses reflection to obtain the full name of the loaded assembly.

1. // Using Reflection to get information from an Assembly:
2. System.Reflection.Assembly info = **typeof**(System.Int32).Assembly;
3. System.Console.WriteLine(info);

**The output is:**

mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

**Real time implementation**  
  
Firstly, we will create a C# console application and add two classes named Student.cs and StudentFunction.cs. The Student.cs class contains some properties of Student and the StudentFunction.cs class contains methods returning different properties value.  
  
**Student.cs class**

1. **public** **class** Student
2. {
3. **public** **string** Name { **get**; **set**; }
4. **public** **string** University { **get**; **set**; }
5. **public** **int** Roll { **get**; **set**; }
6. }

**StudentFunction.cs class**

1. **class** StudentFunction
2. {
3. **private** Student student;
4. **public** StudentFunction()
5. {
6. student = **new** Student
7. {
8. Name = "Gopal C. Bala",
9. University = "Jahangirnagar University",
10. Roll = 1424
11. };
12. }
14. **public** **string** GetName()
15. {
16. **return** student.Name;
17. }
19. **public** **string** GetUniversity()
20. {
21. **return** student.University;
22. }
24. **public** **int** GetRoll()
25. {
26. **return** student.Roll;
27. }
28. }

Our goal is to dynamically create an instance of StudentFunction and get values from GetName(), GetUniversity() and GetRoll() method at the compile time.  
  
In Program.cs class we need to create a Dictionary which contains the method name as string. The name of the method here as string will be compared with the given string with dynamic tag formatted as [tag name]. If both matches, then we need to get the method type.  
  
The Dictionary will be like the following:

1. **private** **static** Dictionary<**string**, **string**> GetMethodsDictionary()
2. {
3. var dictionary = **new** Dictionary<**string**, **string**>
4. {
5. {"GetName", "GetName"},
6. {"GetUniversity", "GetUniversity"},
7. {"GetRoll","GetRoll"}
8. };
9. **return** dictionary;
10. }

In Main function under Program.cs class we have to call GetMethodsDictionary().

1. \_methodDictionary = **new** Dictionary<**string**, **string**>();
2. \_methodDictionary = GetMethodsDictionary();

Now we’ll write the code for creating the instance of StudentFunction at compile time.

1. var type = **typeof**(StudentFunction);
2. var studentFunctionInstance = Activator.CreateInstance(type, **new** **object**[] { });

The string will be like the following that contains the dynamic tag based on which we have to retrieve value of student properties.

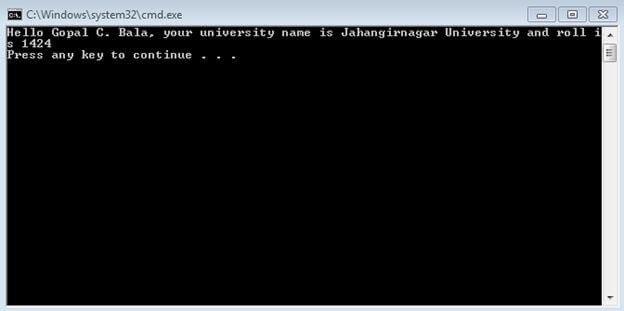
1. var testString = "Hello [GetName], your university name is [GetUniversity] and roll is [GetRoll]";

Here [GetName], [GetUniversity] and [GetRoll] are actually method names which will be compared with the methods declared in StudentFunction class.  
  
Now we will check the mentioned tag ([GetName] and others..) from the defined string using Regular Expression and invoke the value receiving from the methods of StudentFunction.

1. var match = Regex.Matches(testString, @"\[([A-Za-z0-9\-]+)]", RegexOptions.IgnoreCase);
2. **foreach** (var v **in** match)
3. {
4. var originalString = v.ToString();
5. var x = v.ToString();
6. x = x.Replace("[", "");
7. x = x.Replace("]", "");
8. x = \_methodDictionary[x];
10. var toInvoke = type.GetMethod(x);
11. var result = toInvoke.Invoke(studentFunctionInstance, **null**);
12. testString = testString.Replace(originalString, result.ToString());
13. }

So the entire Program.cs class will be like the following:

1. **class** Program
2. {
3. **private** **static** Dictionary<**string**, **string**> \_methodDictionary;
4. **static** **void** Main(**string**[] args)
5. {
6. \_methodDictionary = **new** Dictionary<**string**, **string**>();
7. \_methodDictionary = GetMethodsDictionary();
9. var type = **typeof**(StudentFunction);
10. var studentFunctionInstance = Activator.CreateInstance(type);
12. var testString = "Hello [GetName], your university name is [GetUniversity] and roll is [GetRoll]";
13. var match = Regex.Matches(testString, @"\[([A-Za-z0-9\-]+)]", RegexOptions.IgnoreCase);
14. **foreach** (var v **in** match)
15. {
16. var originalString = v.ToString();
17. var x = v.ToString();
18. x = x.Replace("[", "");
19. x = x.Replace("]", "");
20. x = \_methodDictionary[x];
22. var toInvoke = type.GetMethod(x);
23. var result = toInvoke.Invoke(studentFunctionInstance, **null**);
24. testString = testString.Replace(originalString, result.ToString());
25. }
27. Console.WriteLine(testString);
28. }
30. **private** **static** Dictionary<**string**, **string**> GetMethodsDictionary()
31. {
32. var dictionary = **new** Dictionary<**string**, **string**>
33. {
34. {"GetName", "GetName"},
35. {"GetUniversity", "GetUniversity"},
36. {"GetRoll","GetRoll"}
37. };
38. **return** dictionary;
39. }
40. }

**Output Window**  
  


Abstract

Assemblies are the core unit of deployment. At design time, we can examinethe CIL code in a set of reference assemblies with a couple of external tools such as Reflector and ildasm to peek into underlying metadata, MSIL code and the manifest.

But many times it is necessary to discover types at runtime, and this is the situation where .NET Reflection can be used. This article will also explain late binding that is very related to reflection.

What is Reflection?

Reflection typically is the process of runtime type discovery to inspect metadata, CIL code, late binding and self-generating code. At run time by using reflection, we can access the same "type" information as displayed by the ildasm utility at design time. The reflection is analogous to reverse engineering in which we can break an existing \*.exe or \*.dll assembly to explore defined significant contents information, including methods, fields, events and properties.

You can dynamically discover the set of interfaces supported by a given type using the System.Reflection namespace. This namespace contains numerous related types as follows:

|  |  |
| --- | --- |
| Types | Description |
| Assembly | This static class allows you to load, investigate and manipulate an assembly. |
| AssemblyName | Allows to exploration of abundant details behind an assembly. |
| EventInfo | Information about a given event. |
| PropertyInfo | Holds information of a specified property. |
| MethodInfo | Contains information about a specified method. |

Reflection typically is used to dump out the loaded assemblies list, their reference to inspect methods, properties etcetera. Reflection is also used in the external disassembling tools such Reflector, Fxcop and NUnit because .NET tools don't need to parse the source code similar to C++.

Metadata Investigation

The following program depicts the process of reflection by creating a console based application. This program will display the details of the fields, methods, properties and interfaces for any type within the mscorlib.dll assembly. Before proceeeding, it is mandatory to import "System.Reflection".

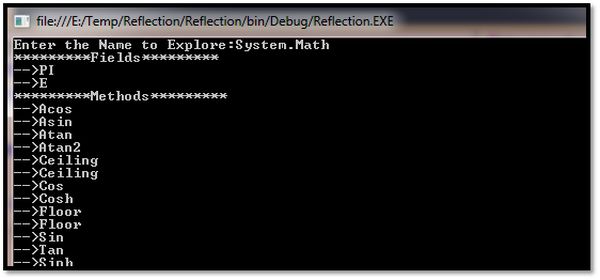
Here, we are defining a number of static methods in the program class to enumerate fields, methods and interfaces in the specified type. The static method takes a single "System.Type" parameter and returns void.

1. **static** **void** FieldInvestigation(Type t)
2. {
3. Console.WriteLine("\*\*\*\*\*\*\*\*\*Fields\*\*\*\*\*\*\*\*\*");
4. FieldInfo [] fld= t.GetFields();
5. **foreach**(FieldInfo f **in** fld)
6. {
7. Console.WriteLine("-->{0}", f.Name);
8. }
9. }
11. **static** **void** MethodInvestigation(Type t)
12. {
13. Console.WriteLine("\*\*\*\*\*\*\*\*\*Methods\*\*\*\*\*\*\*\*\*");
14. MethodInfo [] mth = t.GetMethods();
15. **foreach** (MethodInfo m **in** mth)
16. {
17. Console.WriteLine("-->{0}", m.Name);
18. }
19. }

Now we are done with the static method. It is time to call them in the main() method that first prompts the user for the fully-qualified name of the type. Once we obtain the type in string format, we pass it into the "Type.GetType()" method and send the extracted Type object into each of the static methods as:

1. **static** **void** Main(**string**[] args)
2. {
3. Console.Write("Enter the Name to Explore:");
4. **string** typName = Console.ReadLine();
6. Type t = Type.GetType(typName);
7. FieldInvestigation(t);
8. MethodInvestigation(t);
10. Console.ReadKey();
11. }

Finally run the application and examine the content of the System.Math type when the screen prompts as:



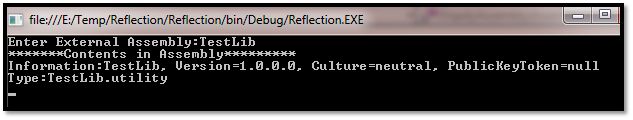
Dynamic Assembly Loading

Technically, the act of loading external assemblies on demand is known as Dynamic Loading. Using the Assembly class, we can dynamically load both private and shared assemblies from the local location to a remote location as well as, explore its properties.

To illustrate dynamic loading, we are creating a console based application that loads an external TestLib.dll assembly. During the execution, the application asks the user to specify the dynamic loading assembly name and that reference is passed to the helper method that is responsible for loading the assembly as:

1. **using** System;
2. **using** System.Reflection;
4. **namespace** Reflection
5. {
6. **class** Program
7. {
8. **static** **void** Main(**string**[] args)
9. {
10. Console.Write("Enter External Assembly:");
11. **string** typName = Console.ReadLine();
12. **try**
13. {
14. Assembly asm = Assembly.Load(typName);
15. DispalyAssembly(asm);
16. }
17. **catch**
18. {
19. Console.WriteLine("Can't Load Assembly");
20. }
21. Console.ReadKey();
22. }
24. **static** **void** DispalyAssembly(Assembly a)
25. {
26. Console.WriteLine("\*\*\*\*\*\*\*Contents in Assembly\*\*\*\*\*\*\*\*\*");
27. Console.WriteLine("Information:{0}",a.FullName);
28. Type[] asm = a.GetTypes();
29. **foreach** (Type tp **in** asm)
30. {
31. Console.WriteLine("Type:{0}", tp);
32. }
33. }
34. }
35. }

After successfully compiling the application, the important point to remember is that you need to copy the dynamically loading TestLib.dll binary into the solution Bin\Debug folder to run this program properly. Here, the output is as follows:



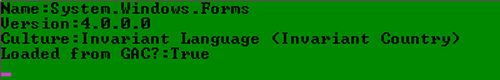
If you wish to make this program more flexible then you can update your code to load the external assembly using the LoadFrom() method. In such a situation you don't need to place the external assembly in the Bin\Debug folder.

1. **static** **void** Main(**string**[] args)
2. {
3. **try**
4. {
5. Assembly asm = Assembly.LoadFrom(@"E:\TestLib.dll");
6. DispalyAssembly(asm);
7. }
8. **catch**
9. {
10. Console.WriteLine("Can't Load Assembly");
11. }
12. }

We can also implement reflection on shared assemblies. We must pass sets of information that identify an assembly, such as assembly name, version, publicKeyToken as in the following:

1. **class** Program
2. {
3. **static** **void** Main(**string**[] args)
4. {
5. **try**
6. {
7. **string** info = @"System.Windows.Forms," + "Version=4.0.0.0," +
8. "PublicKeyToken=B77A5C561934E089," +
9. @"Culture=""";
10. Assembly asm = Assembly.Load(info);
11. DispalyAssembly(asm);
12. }
13. **catch**
14. {
15. Console.WriteLine("Can't Load Assembly");
16. }
17. Console.ReadKey();
18. }
19. **static** **void** DispalyAssembly(Assembly a)
20. {
22. Console.WriteLine("Name:{0}", a.GetName().Name);
23. Console.WriteLine("Version:{0}", a.GetName().Version);
24. Console.WriteLine("Culture:{0}", a.GetName().CultureInfo.DisplayName);
25. Console.WriteLine("Loaded from GAC?:{0}", a.GlobalAssemblyCache);
26. }
27. }

The output of that program will be as follows:



Late Binding

The .NET framework can create an instance of a given type using early binding or late binding. In early binding, we typically set the external assembly reference in the project and allocate the type using the new operator. Early binding allows us to determine errors at compile time rather than at runtime.

Whereas in late binding, you can create an instance of a given type and invoke its methods at runtime without having knowledge at compile time. There is no provision to set an external assembly reference in this construct.

We can create a late binding instance of an external assembly using the CreateInstance() method of the System.Activator static class. Here, we are dynamically instantiating a utility class of the TestLib.dll; the code is refreshingly simple, as in the following:

1. **using** System;
2. **using** System.Reflection;
4. **namespace** Reflection
5. {
6. **class** Program
7. {
8. **static** **void** Main(**string**[] args)
9. {
10. **try**
11. {
12. Console.WriteLine("\*\*\*\*\*\*\*Assembly Late Binding\*\*\*\*\*\*\*\*\*");
14. Type t = Type.GetType("TestLib.utility,TestLib");
15. **object** obj = Activator.CreateInstance(t);
16. Console.WriteLine("Create a {0} using late binding", obj);
18. MethodInfo mth = t.GetMethod("Test");
19. mth.Invoke(obj, **null**);
20. Console.WriteLine("Method Invoked");
21. }
22. **catch**
23. {
24. Console.WriteLine("Can't Create Assembly Instance");
25. }
26. Console.ReadKey();
27. }
28. }
29. }

Here you don't need to put the external assembly binary copy in the solution Bin/Debug folder like earlier. Once the code is complete and you compile this application, you obtain this output as:

