Reflections

- Reflection in C# is used to retrieve metadata on types at runtime.
- In other words, we can use reflection to inspect metadata of the types in your program dynamically
- We can retrieve information on the loaded assemblies and the types defined in them. Reflection in C# is similar to RTTI (Runtime Type Information) of C++.
- To work with reflection in .Net, we should include the System.Reflection namespace in our program.
- In using reflection, we get objects of the type "Type" that can be used to represent assemblies, types, or modules.
- We can use reflection to create an instance of a type dynamically and even invoke methods of the type.

Reflection is useful in the following situations:

- When we have to access attributes in our program's metadata.
- For examining and instantiating types in an assembly.
- For building new types at runtime. Use classes in System.Reflection.Emit.
- For performing late binding, accessing methods on types created at run time.

Use of reflections in C#

1. Use Assembly to define and load assemblies, load modules that are listed in the assembly manifest, and locate a type from this assembly and create an instance of it.

Example :

```
using System;
using System.Reflection;
class Marvellous
{
        public int non1, no2;
        public void fun()
        {}
}

public class Infosystems
{
        public static void Main()
        {
            Type t2 = typeof(Marvellous);
            Console.WriteLine(t2.Assembly);
            //      Version=0.0.0.0,
            //      Culture=neutral,
            //      PublicKeyToken=null
```

```
                Type t1 = typeof(System.String);
                Console.WriteLine(t1.Assembly);
                //      mscorlib,
                //      Version=4.0.0.0,
                //      Culture=neutral,
                //      PublicKeyToken=b77a5c561934e089
            }
}
```

2. Use Module to discover information such as the assembly that contains the module and the classes in the module. You can also get all global methods or other specific, non global methods defined on the module.

3. Use ConstructorInfo to discover information such as the name, parameters, access modifiers (such as public or private), and implementation details (such as abstract or virtual) of a constructor. Use the GetConstructors or GetConstructor method of a Type to invoke a specific constructor.

Example :

```
using System;
using System.Reflection;

class Marvellous
{
        public Marvellous()
        {}
        public Marvellous(int i)
        {}
        public Marvellous(int i, char ch)
        {}
        public Marvellous(Marvellous obj)
        {}
        ~Marvellous()
        {}
}

public class Infosystems
{
   public static void Main()
   {
        Type t1 = typeof(Marvellous);
       Console.WriteLine("Constructors of {0} type...", t1);
        ConstructorInfo[] ci1 = t1.GetConstructors(BindingFlags.Public |
BindingFlags.Instance);

        foreach (ConstructorInfo c in ci1)
```

```
        {
            Console.WriteLine(c);
        }
        /*
                    Constructors of Marvellous type...
                    Void .ctor()
                    Void .ctor(Int32)
                    Void .ctor(Int32, Char)
                    Void .ctor(Marvellous)
        */

        Type t = typeof(System.String);

        Console.WriteLine("Constructors of {0} type...", t);
        ConstructorInfo[] ci2 = t.GetConstructors(BindingFlags.Public |
BindingFlags.Instance);

        foreach (ConstructorInfo c in ci2)
        {
            Console.WriteLine(c);
        }
        /*
                    Void .ctor(Char*)
                    Void .ctor(Char*, Int32, Int32)
                    Void .ctor(SByte*)
                    Void .ctor(SByte*, Int32, Int32)
                    Void .ctor(SByte*, Int32, Int32, System.Text.Encoding)
                    Void .ctor(Char[], Int32, Int32)
                    Void .ctor(Char[])
                    Void .ctor(Char, Int32)
        */
    }
}
```

4. Use MethodInfo to discover information such as the name, return type, parameters, access modifiers (such as public or private), and implementation details (such as abstract or virtual) of a method. Use the GetMethods or GetMethod method of a Type to invoke a specific method.

Example :

```
using System;
using System.Reflection;
class Marvellous
{
        public void fun()
        {}
```

```
        public void gun(int i, int j)
        {}
        public void sun(char ch, int i)
        {}
        public void run(double d)
        {}

}

public class ReflectionExample
{
   public static void Main()
   {
            Type t1 = typeof(Marvellous);

      Console.WriteLine("Methods of {0} type...", t1);
      MethodInfo[] ci1 = t1.GetMethods(BindingFlags.Public | BindingFlags.Instance);
      foreach (MethodInfo m in ci1)
      {
         Console.WriteLine(m);
      }
      /*
            Methods of Marvellous type...
            Void fun()
            Void gun(Int32, Int32)
            Void sun(Char, Int32)
            Void run(Double)
            System.String ToString()
            Boolean Equals(System.Object)
            Int32 GetHashCode()
            System.Type GetType()
      */
   }
}
```

5. Use FieldInfo to discover information such as the name, access modifiers (such as public or private) and implementation details (such as static) of a field, and to get or set field values.

Example :

```
using System;
using System.Reflection;

public class Marvellous
{
            public static void Main()
```

```
        {
                Type t = typeof(System.String);

                Console.WriteLine("Fields of {0} type...", t);
                FieldInfo[] ci = t.GetFields(BindingFlags.Public | BindingFlags.Static |
BindingFlags.NonPublic);
                foreach (FieldInfo f in ci)
                {
                        Console.WriteLine(f);
                }

                /*
                        Fields of System.String type...
                        System.String Empty
                        Int32 TrimHead
                        Int32 TrimTail
                        Int32 TrimBoth
                        Int32 charPtrAlignConst
                        Int32 alignConst
                */
        }
}
```

6. Use EventInfo to discover information such as the name, event-handler data type, custom attributes, declaring type, and reflected type of an event, and to add or remove event handlers.

7. Use PropertyInfo to discover information such as the name, data type, declaring type, reflected type, and read-only or writable status of a property, and to get or set property values.

8. Use ParameterInfo to discover information such as a parameter's name, data type, whether a parameter is an input or output parameter, and the position of the parameter in a method signature.

9. Use CustomAttributeData to discover information about custom attributes when you are working in the reflection-only context of an application domain. CustomAttributeData allows you to examine attributes without creating instances of them.