**Reflection**

Inspecting assembly metadata at runtime.

It is used to find all the types in an assembly and/or all the methods in assembly.

Use:

When you drag and drop a button on a winforms application or in asp.net application. The properties window uses all the properties available for the class. So, reflection is extensively used by Ide or UI UX developers.

Late binding can be achieved by reflection. We can create instance of a type without having any information of that type at compile time. So, reflection enables you to use code that is not available at compile time.

Late binding : creating the instance of class at runtime rather than compile time.

Consider an example where we have two alternate implementations of a interface. You want to give user chance to pick one or the other using a config file.. With reflection you can simple read the name of the class whose implementation you want to use from the config file, and instantiate an instance of the class. This is another example of late binding.

First we declare a class, and for that class we declare Type T type for that class.

Three ways to declare the Type T:

* Type T = Type.GetType("Reflection\_EG.Employee");
* Type T=typeof(Employee);
* Employee employee = new Employee();

Type T=employee.GetType();

Example:

internal class Program

{

static void Main(string[] args)

{

// Type T = Type.GetType("Reflection\_EG.Employee");

// Type T=typeof(Employee);

Employee employee = new Employee();

Type T=employee.GetType();

Console.WriteLine($"Full Name={T.FullName}");

Console.WriteLine($"Just Name: {T.Name}");

Console.WriteLine($"Just the Namespace:{T.Namespace}");

Console.WriteLine();

Console.WriteLine("Property Information");

PropertyInfo[] property=T.GetProperties();

foreach (PropertyInfo pi in property)

{

Console.WriteLine($"Type:{pi.PropertyType.Name} Name:{pi.Name}");

}

Console.WriteLine();

Console.WriteLine("Method Information");

MethodInfo[] methodInfo=T.GetMethods();

foreach (MethodInfo mi in methodInfo)

{

Console.WriteLine($"Method return Type:{mi.ReturnType.Name} Method Name: {mi.Name}");

}

Console.WriteLine();

Console.WriteLine("Constructor Information");

ConstructorInfo[] constructors=T.GetConstructors();

foreach(ConstructorInfo ci in constructors)

{

Console.WriteLine(ci.Name+" "+ci.GetType().Name+" "+ci.ToString());

}

}

}

public class Employee

{

public int Id { get; set; }

public string Name { get; set; }

public Employee(int id, string name)

{

Id = id;

Name = name;

}

public Employee()

{

Id = -1;

Name = null;

}

}

Output :

Full Name=Reflection\_EG.Employee

Just Name: Employee

Just the Namespace:Reflection\_EG

Property Information

Type:Int32 Name:Id

Type:String Name:Name

Method Information

Method return Type:Int32 Method Name: get\_Id

Method return Type:Void Method Name: set\_Id

Method return Type:String Method Name: get\_Name

Method return Type:Void Method Name: set\_Name

Method return Type:Boolean Method Name: Equals

Method return Type:Int32 Method Name: GetHashCode

Method return Type:Type Method Name: GetType

Method return Type:String Method Name: ToString

Constructor Information

.ctor RuntimeConstructorInfo Void .ctor(Int32, System.String)

.ctor RuntimeConstructorInfo Void .ctor()

Note : By specifying BindingFlags.NonPublic flag in the reflection calls, we indicate that we're interested in accessing non-public members of the class. This allows us to bypass the private access modifier and access private members at runtime.

Example:

FieldInfo[] fields=T.GetFields(BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.Public);

foreach (FieldInfo field in fields)

{

Console.WriteLine($"Field Name: {field.Name}");

Console.WriteLine($"Field Type: {field.FieldType}");

Console.WriteLine($"Is Public: {field.IsPublic}");

Console.WriteLine($"Is Private: {field.IsPrivate}");

Console.WriteLine($"Is Static: {field.IsStatic}");

Console.WriteLine();

}

Output :

FieldInfo

Field Name: <Id>k\_\_BackingField

Field Type: System.Int32

Is Public: False

Is Private: True

Is Static: False

Field Name: <Name>k\_\_BackingField // property field

Field Type: System.String

Is Public: False

Is Private: True

Is Static: False

Field Name: <Age>k\_\_BackingField

Field Type: System.String

Is Public: False

Is Private: True

Is Static: False

Field Name: val // non-property field

Field Type: System.String

Is Public: True

Is Private: False

Is Static: False

**Late Binding and Early Binding**

When we have information of class at the time of compile time, we use early binding.

When we don’t have information of class at the time of compile time, we use late binding.

Disadvantages:

1. Slower than early binding.
2. If we misspell the name of methods then it will compile, but gives null reference exception at run time while in early binding it will identify those type of errors at compile time itself
3. Complex than early binding.

Assembly assembly = Assembly.GetExecutingAssembly(); // load the execution assembly.

Type employeeType = assembly.GetType("Late\_binding\_EG.Employee");

object employeeInstance = Activator.CreateInstance(employeeType);

MethodInfo getFullName = employeeType.GetMethod("GetFullName");

string[] param = new string[2];

param[0] = "Nikhita";

param[1] = "Palla";

string fullName=(string)getFullName.Invoke(employeeInstance, param);

Console.WriteLine(fullName);

**Get and set values of parameters using reflection**

// Display initial values

Console.WriteLine("Initial values:");

Console.WriteLine($"Id: {employee.Id}, Name: {employee.Name}");

// Get the type of Employee

Type employeeType = typeof(Employee);

// Change properties using reflection

PropertyInfo idProperty = employeeType.GetProperty("Id");

idProperty.SetValue(employee, 100); // Set new value for Id

PropertyInfo nameProperty = employeeType.GetProperty("Name");

nameProperty.SetValue(employee, "Jane"); // Set new value for Name

// Display updated values

Console.WriteLine("\nUpdated values:");

Console.WriteLine($"Id: {employee.Id}, Name: {employee.Name}");

Console.ReadLine();