Chapter 7

Reflection, Metadata and Dynamic Programming

Mrs. Swati Satpute
Fergusson College

Objective

- System.Type class, Metadata, Reflection, Assembly class, Memberinfo class,
- Standard Attributes and Custom Attributes,
- Dynamic Type, dynamic vs var, Dynamic Language Runtime (DLR)

What is Assembly?

- Assembly is the logical unit of deployment.
- Can consist of one or more compiled files.
- Assembly can be EXE or DLL

Assemblies

- Assemblies contains
  - Meta information
    - Information about the contents it is storing
    - This allows assembly to be **self descriptive**
  - Resources (Optional)
    - E.g. pictures or sound files

Structure of Assembly

• Assembly contains the executable code along with the metadata.

• This metadata helps assembly consumer to know internal details like

- Classes
- Methods
- Properties

• Metadata can be represented 2 ways

-Table of contents i.e. describing everything

- bibliography describing references to data outside the assembly.

Reflection

Meta Data

- It is the **structured description** of the code in an assembly.
- It contains
  - **Description of the assembly** (deployment unit)
    - Identity: Name, Version and culture
    - Dependencies (on other assemblies)
    - Security permission that the assembly requires to run
  - **Description of the Types**
    - classes and interfaces
  - **Custom attributes**
    - Defined by User
    - Defined by Compiler
    - Defined by Framework

Meta Data

- Meta Data is **language independent**.
- Why do we need Meta Data??
  - CLR needs this meta data **to provide compile time and runtime services**. E.g.
    - Loading of Class Files
    - Memory management
    - Debugging
    - Object Browsing
    - MSIL Translation to Native Code

Reflection API

- Reflection gives **ability to access an applications meta data.**

- Reflection is the **mechanism** of **discovering Type (class) information** solely **at runtime**.

- Type information includes information about the **type**, **properties**, **methods**, and **events** of an object.

Reflection API

- Reflection can be used to
  - get the type and its info from an existing object
  - dynamically create an instance of a type,
  - bind the type to an existing object, or
  - invoke its methods or access its fields and properties.

- If attributes are used in the code, reflection helps to access them.

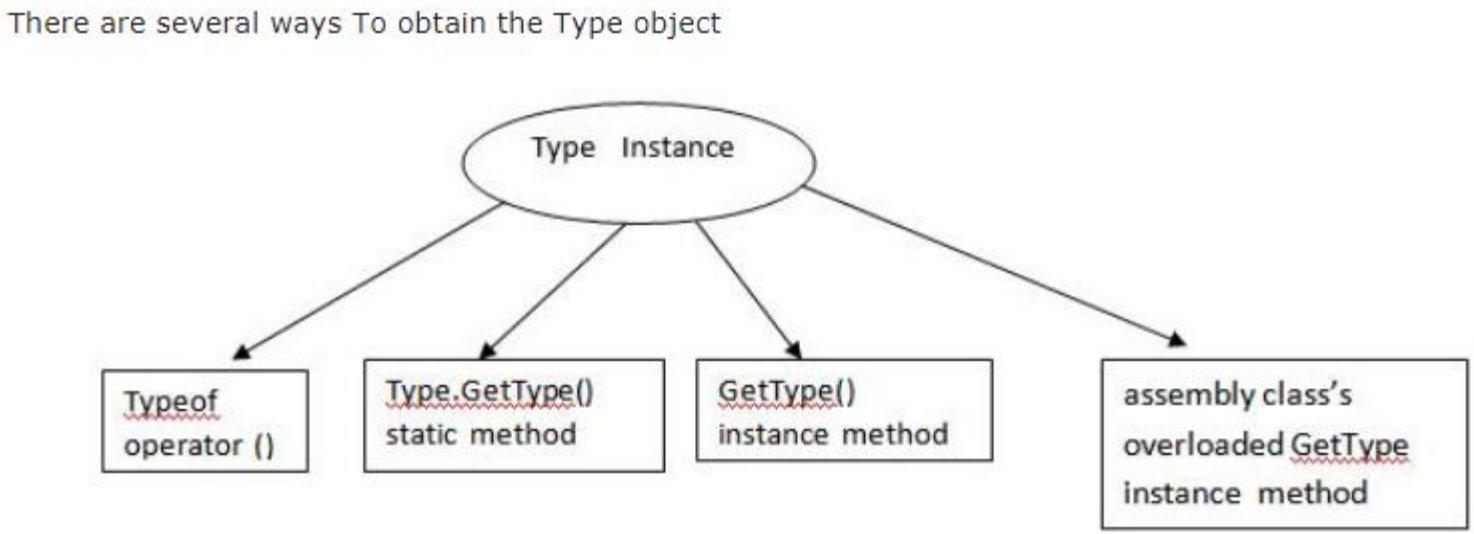
Reflection API

- **System.Reflection** namespace contains all classes and interfaces that provide
  - a **managed view of loaded types, methods, and fields**,
  - with the ability to **dynamically create and invoke types**;

- **System.Type**
  - Its an abstract class representing Type in a CTS
  - From Assembly one can query
    - Type name
    - Module details
    - Name space
    - Whether is value type or reference type

System.Type

- System.Type is the most fundamental to working with reflection functionality in .NET and it represents a type in the CTS.



- From the instance of the type, one can retrieve type information using several methods like **GetMembers()** or **GetMethods()..**

Reflection API

Example :

```
int a =10 ;
System.Type type = a.GetType();
    System.Console.WriteLine(type);
```

System.Reflection namespace can be used in the program.

```
// Using Reflection to get information from an Assembly:
System.Reflection.Assembly info = typeof (System.Int16). Assembly;
System.Console.WriteLine(info);
```

Reflection API

Assembly	Represents an assembly
EventInfo	This class holds information for a given event.
FieldInfo	This class holds information for a given field.
MemberInfo	Class is the abstract base class for classes used to obtain information about all members of a class.
MethodInfo	This class contains information for a given method.
ConstructorInfo	This class contains information for a given constructor.

Reading Type information

```
static void ReadTypeInfo()
{

    Type typeObj = typeof(Rectangle);

    Console.WriteLine("Following are the methods in Test class");
    foreach (MethodInfo mi in typeObj.GetMethods())
    {
        Console.WriteLine("Method {0}", mi.Name);
    }

    foreach (FieldInfo fi in typeObj.GetFields())
    {
        Console.WriteLine("Following are the fields in Test class");
        Console.WriteLine("Field {0}", fi.Name);
    }

    object o = Activator.CreateInstance(typeObj, new object[] {4, 6});
    MethodInfo method = typeObj.GetMethod("Display");
    Console.Write("\t");

    // Invoking Area of Rectanlge dynamically
    method.Invoke(o, null);

    Console.ReadLine();
}
```

Demo

Reflection API

- **Late binding** is a powerful technology in .NET Reflection which allows you to **create an instance** of a given type and invoke its members at runtime **without having compile-time knowledge of its existence**; this technique is also called **dynamic invocation**.

- This technique is **useful when** working with an **object which is not available at compile time**.

- In this technique, it is the **developer's responsibility** to pass the **correct signature of the methods** before invoking; otherwise, it will throw an error.

- Object creation / method invocation using reflection has **performance cost.**

Lab Assignment

Late Binding and Dynamic method invocation

```
using System.Reflection;
try
        {
           String strTypeName = "CarLibrary.Car";
           Assembly a = Assembly.LoadFrom(@"E:\SAmple Projects\CarLibrary.dll");
           Object myCar = a.CreateInstance(strTypeName);

           MethodInfo mi = a.GetType("CarLibrary.Car").GetMethod("StartCar");
           if (mi != null)
              mi.Invoke(myCar,null);

           mi = a.GetType("CarLibrary.Car").GetMethod("StopCar");
           if (mi != null)
              mi.Invoke(myCar, new object[] { "SUV" });
        }
        catch (Exception ex)
        {
           Console.WriteLine("Exception Details: {0}", ex.Message);
        }
```

Demo

Attributes

Attributes

- **Attributes are objects** that provide a **powerful method of associating metadata, or declarative information, with code** (Assembly, Class, Method, Delegate, Enum, Event, Field, Interface, Property and Struct)

Properties

- Attributes **add metadata to your program**.
  - *Metadata* is information about the types defined in a program.
  - Custom attributes can be used to specify any additional information that is required
- One can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can **accept arguments** in the **same way as methods and properties.**
- program can examine its own metadata or the metadata in other programs by using reflection

Using Attributes

- A declarative tag is depicted by square [ ] brackets placed above the element.

- Attributes are **used for adding metadata**, such as compiler instruction and other information such as comments, description, methods and classes to a program.

- Two types of attributes: *the pre-defined* attributes and *custom built attributes*.

Predefined Attributes

Attributes	Description
[Serialization]	By marking this attributes, a class is able to persist its current state into stream.
[NonSerialization]	It specify that a given class or filed should not persisted during the serialization process.
[Obsolete]	It is used to mark a member or type as deprecated. If they are attempted to be used somewhere else then compiler issues a warning message.
[DllImport]	This allows .NET code to make call an unmanaged C or C++ library.
[WebMethod]	This is used to build XML web services and the marked method is being invoked by HTTP request.
[CLSCompliant]	Enforce the annotated items to conform to the semantics of CLS.

Serializable

- This attribute informs **compiler that this class is serializable.**

```
C#

[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

DLLImport

- Indicates that the attributed method is exposed by an unmanaged dynamic-link library (DLL) as a static entry point.

C#

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

Conditional

- This 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**.

- it displays the values of the variables while debugging a code.

```
[Conditional("DEBUG")]
public static void Message(string msg)
{
    Console.WriteLine(msg);
}
```

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.**
- [**Obsolete** (message, iserror)]
  - The parameter *message*, is a string describing the reason why the item is obsolete and what to use instead.
  - *iserror*, is a Boolean value.
    - If value is **true**, the compiler should **treat** the use of the item as an error.

Pre-defined Attributes

```
static class IntrinsicAttribute
{
    [Obsolete("This method is obsolete...", false
    public static void DoSomeWork()
    {
        Console.WriteLine("Obsolete Method");
    }

    [Conditional("DEBUG")]
    public static void Message(string msg)
    {
        Console.WriteLine(msg);
    }

    public static void function1()
    {
        IntrinsicAttribute.Message("In Function 1
```

Attribute targets

- The *target* of an attribute is the entity to which the attribute applies e.g. class , method etc

- By default, an attribute applies to the element that it precedes.

- To explicitly identify an attribute target,

```
C#

[target : attribute-list]
```

The list of possible target values

Target value	Applies to
`assembly`	Entire assembly
`module`	Current assembly module
`field`	Field in a class or a struct
`event`	Event
`method`	Method or `get` and `set` property accessors
`param`	Method parameters or `set` property accessor parameters
`property`	Property
`return`	Return value of a method, property indexer, or `get` property accessor
`type`	Struct, class, interface, enum, or delegate

You would specify the `field` target value to apply an attribute to the backing field created for an auto-implemented property.

Attributes on assemblies and modules

C#

```
using System;
using System.Reflection;
[assembly: AssemblyTitleAttribute("Production assembly 4")]
[module: CLSCompliant(true)]
```

Common uses for attributes

- Marking methods using the **WebMethod** attribute in Web services to indicate that the method should be callable over the SOAP protocol.
- MarshalAsAttribute: Describing how to marshal method parameters when interoperating with native code.
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the DllImportAttribute class.
- Describing your assembly in terms of **title, version, description, or trademark.**
- Describing which members of a class to **serialize** for persistence.
- Describing how to map between class members and XML nodes for **XML serialization.**
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

AttributeUsage

- Describes how a custom attribute class can be used.
- It specifies the types of items to which the attribute can be applied.
- [**AttributeUsage** (validon, AllowMultiple = allowmultiple, Inherited = inherited)]
  - **validon** specifies the language elements on which the attribute can be placed.
    - default value is ***AttributeTargets.All***
  - ***allowmultiple*** (optional) provides a Boolean value. If this is true, the **attribute is multiuse** i.e. multi instance.
  - **inherited** (optional) provides value for the *Inherited* property of this attribute, a Boolean value. default is false.

```
[AttributeUsage(
   AttributeTargets.Class |
   AttributeTargets.Constructor |
   AttributeTargets.Field |
   AttributeTargets.Method |
   AttributeTargets.Property,
   AllowMultiple = true)]
```

Custom Attributes

- Custom attributes that can be **used to store declarative information** and can be **retrieved at run-time.**
- 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

Use Case for custom attribute

- Construct a custom attribute named *DeBugInfo*, which stores the information obtained by debugging any program.
- It stores the following information:
  - The code number for the bug (private)
  - Name of the developer who identified the bug (private)
  - Date of last review of the code (private)
  - A string message for storing the developer's remarks (public)

Constructing the Custom Attribute

```
//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...
    public string Developer...
    public string LastReview...
    public string Message
    {
        get
        {
            return message;
        }
        set
        {
            message = value;
        }
    }
}
```

Using Custom Attribute

```
[DeBugInfo(45, "Mayur Kulkarni", "12/8/2012", Message = "Return type mismatch")]
[DeBugInfo(49, "Sarah khan", "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, "Kiran Deshapnade", "19/10/2012", Message = "Return type mismatch")]
    public double GetArea()
    {
        return length * width;
    }

    [DeBugInfo(56, "Kiran Deshapnade", "19/10/2012")]
    public void Display()
    {
        Console.WriteLine("Length: {0}", length);
        Console.WriteLine("Width: {0}", width);
        Console.WriteLine("Area: {0}", GetArea());
    }
```

Class Level

Method Level

Accessing through Reflection

```
static void ReadCustomAttribute()
{
    Type type = typeof(Rectangle);

    //iterating through the attribtues of the Rectangle class
    foreach (Object attributes in type.GetCustomAttributes(false))
    {
        DeBugInfo dbi = (DeBugInfo)attributes;

        if (null != dbi)
        {
            Console.WriteLine("Bug no: {0}", dbi.BugNo);
            Console.WriteLine("Developer: {0}", dbi.Developer);
            Console.WriteLine("Last Reviewed: {0}", dbi.LastReview);
            Console.WriteLine("Remarks: {0}", dbi.Message);
        }
    }

    //iterating through the method attribtues
    foreach (MethodInfo m in type.GetMethods())
    {
        foreach (Attribute a in m.GetCustomAttributes(true))
        {
            DeBugInfo dbi = a as DeBugInfo;

            if (null != dbi)
            {
                Console.WriteLine("Bug no: {0}, for Method: {1}", dbi.BugNo, m.Name);
                Console.WriteLine("Developer: {0}", dbi.Developer);
                Console.WriteLine("Last Reviewed: {0}", dbi.LastReview);
                Console.WriteLine("Remarks: {0}", dbi.Message);
            }
        }
    }
```

Demo

Dynamic Type

- Introduced in C# 4.0
- A new type which **avoids compile time – type checking.**
- A dynamic type **resolves type at run time.**
- At compile time, an element that is typed as dynamic is assumed to **support any operation.**
- The dynamic types do not have intellisense support in visual studio.

Dynamic Type

```
C#

class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

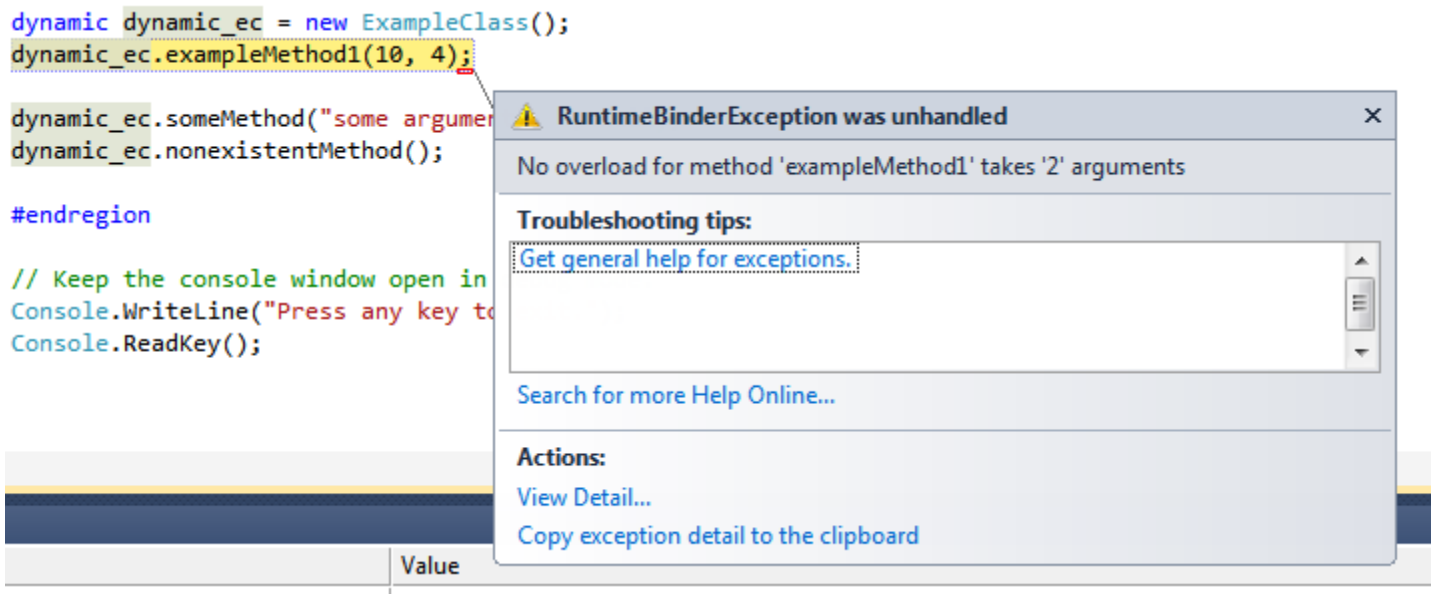
```
static void Main()
{

    #region DynamicType demo
    ExampleClass ec = new ExampleClass();

    dynamic dynamic_ec = new ExampleClass();
    dynamic_ec.exampleMethod1(10, 4);

    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();

    #endregion
```

Compilation of this code gives 0 Errors.

Once a variable is declared as having type **dynamic**, **operations** on these value **are not done nor verified at compile time**, but instead happen **entirely at runtime**. This is known also as ***duck typing.***

Dynamic Type

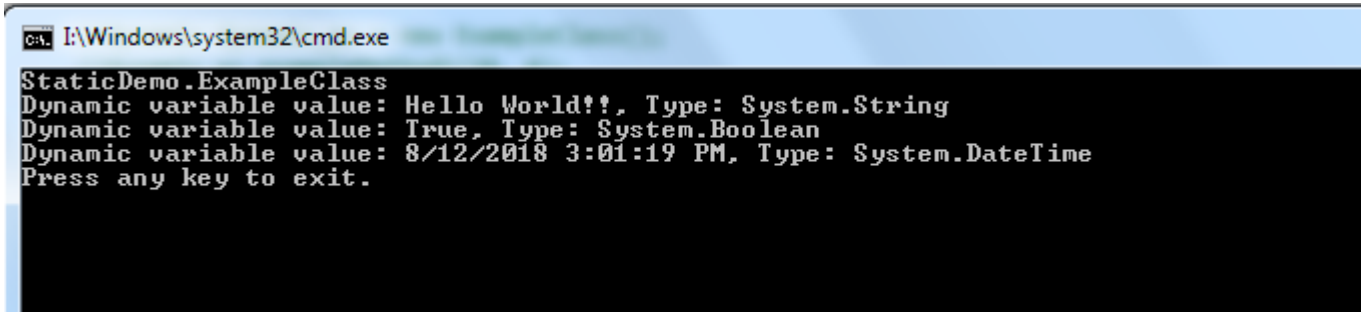
- During runtime, if **methods not resolved** then **exception will be thrown**



Dynamic Type – GetType

```
#region Dynamic Type GEtType
dynamic dynamicVariable = new ExampleClass();
Console.WriteLine(dynamicVariable.GetType().ToString());

dynamicVariable = "Hello World!!";
Console.WriteLine("Dynamic variable value: {0}, Type: {1}", dynamicVariable, dynamicVariable.GetType().ToString());

dynamicVariable = true;
Console.WriteLine("Dynamic variable value: {0}, Type: {1}", dynamicVariable, dynamicVariable.GetType().ToString());

dynamicVariable = DateTime.Now;
Console.WriteLine("Dynamic variable value: {0}, Type: {1}", dynamicVariable, dynamicVariable.GetType().ToString());
```

Demo



Implicit Typed Local Variable - var

- var **references a type** in an implicit way.
- var can only **be defined** in a method as a local variable.
- The **compiler** will **infer its type** based on the value to the right of the **"=" operator.**
- Var can be used in the following different contexts:
  - Local variable in a function
  - For loop
  - Foreach loop
  - Using statement
  - As an anonymous type
  - In a LINQ query expression

Implicit Typed Local Variable - var

```
static void Main(string[] args)
{
    var i = 10;
    Console.WriteLine("Type of i is {0}",i.GetType().ToString());

    var str = "Hello World!!";
    Console.WriteLine("Type of str is {0}", str.GetType().ToString());

    var d = 100.50d;
    Console.WriteLine("Type of d is {0}", d.GetType().ToString());

    var b = true;
    Console.WriteLine("Type of b is {0}", b.GetType().ToString());
}
```

```
Type of i is System.Int32
Type of str is System.String
Type of d is System.Double
Type of b is System.Boolean
```

Implicit Typed Local Variable - var

Points to Remember :

1. ***var*** can only be **declared and initialized in a single statement**. Following
   var i; i = 10; // is not valid

2. ***var*** **cannot be** used as **a field type** at the class level.

3. ***var*** **cannot be used** in an expression like
   var i += 10;

4. **Multiple *vars* cannot be declared** and **initialized** in a single statement.
   For example, var i=10, j=20; is invalid.

Dynamic vs var type

- var is **statistically typed** and compiler will perform strong type checking.
- dynamic is **dynamically typed** and compiler will ignore type checking.

```
// Can a dynamic change type?
dynamic test = 1;
test = "i'm a string now";  // compiles and runs just fine
var test2 = 2;
test2 = "i'm a string now"; // will give compile error
```

Dynamic vs var type

```
// Can a dynamic change type?
dynamic test = 1;
Console.WriteLine("Dynamic as " + test.GetType() + ": " + test);
test = "i'm a string now";  // compiles and run just fine
Console.WriteLine("Dynamic as " + test.GetType() + ": " + test);
var test2 = 2;
  test2 = "i'm a string now"; // will give compile error
Console.WriteLine("Var as " + test2.GetType() + ": " + test2);
```