Reflection

.NET

Reflection assists languages to operate in networks by providing libraries for serialization, bundling, and varying data formats. Reflection is a language more suited to network-oriented code.

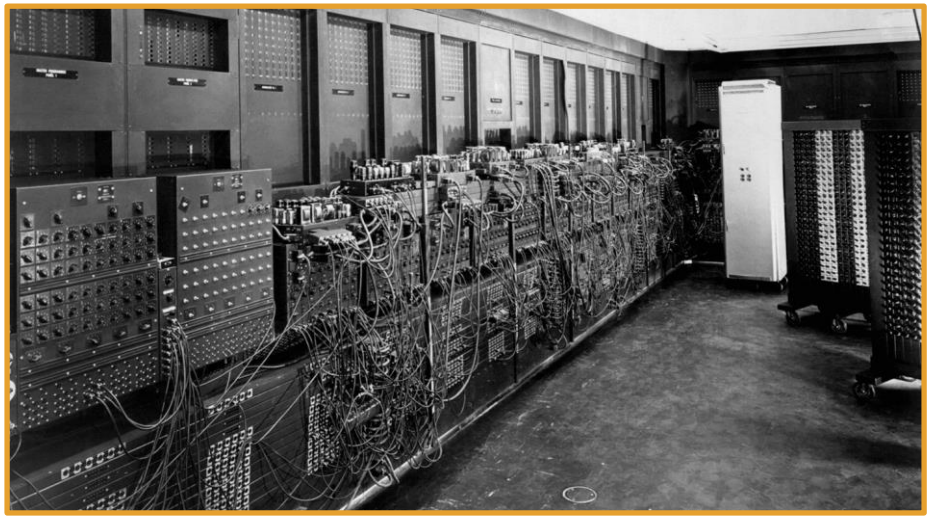
HTTPS://EN.WIKIPEDIA.ORG/WIKI/REFLECTIVE_PROGRAMMING

A History of Reflection.

The earliest computers were inherently reflective. Their architectures were programmed by defining instructions as data and using self-modifying code.

As programming moved to higher-level, compiled languages (C,C++) this reflective ability largely disappeared until new programming languages with built-in reflection appeared.

In 1982, Brian Smith proposed computational reflection in procedural programming languages and the notion of the meta-circular interpreter as a component of 3-Lisp.



What is Reflection?

https://en.wikipedia.org/wiki/Reflective_programming

Reflection can be used for observing and modifying program execution during runtime based on the state of different program elements. A reflection-oriented program component can monitor the execution of code and then modify itself according to predefined goals.

In OOP languages, reflection:

- allows inspection of classes, interfaces, fields and methods at runtime without knowing the names of the interfaces, fields, methods at compile time.
- allows instantiation of new objects and invocation of methods.
- is used as part of testing for mocking objects and bringing otherwise unreachable, and thereby untestable, code into scope.

In some languages, reflection can be used to bypass member accessibility rules. It is also possible to find non-public class methods and manually invoke them.

Reflection in .NET

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection

Reflection provides type Type objects that describe assemblies, modules, and types.

You 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.

```
// Using GetType() to obtain
type information:
int myInt = 42;
Type type = myInt.GetType();
Console.WriteLine(type);

/* prints
System.Int32
 */
```

When to use Reflection

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection#reflection-overview
https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection

---

Reflection is useful in the following situations:

- When you need to access attributes in your program's metadata.
  - Retrieving Information Stored in Attributes.
- For examining and instantiating Types in an assembly.
- Use System.Reflection.Emit for building new types at run time.
- For performing late binding, accessing methods on types created at run time.
  - See Dynamically Loading and Using Types.

```
// Without reflection
Foo foo = new Foo();
foo.PrintHello();

// With reflection
Object foo =
Activator.CreateInstance("complete.classpath.and.Foo");
MethodInfo method =
foo.GetType().GetMethod("PrintHello");
method.Invoke(foo, null);
```

Type Class: A Reflections datatype

https://docs.microsoft.com/en-us/dotnet/api/system.type?view=net-6.0

Type Class – Represents all class ***types*** (interface, array, value, enum, generic, ***type*** parameters).

Type is the root of System.Reflection and is the primary way to access metadata. Use the members and extension methods of ***Type*** to get information about any ***type***, about the members of the ***type*** (class constructors, methods, fields, properties, events), and the module and assembly of the class.

Type access requires no special permissions. A derived class can access protected members of the calling code's base classes. Access is allowed to assembly members of the calling code's assembly.

In the code at right, .GetType() and .Name are chained to get the official name of the type.

```
object[] values = { "word", true, 120, 136.34, 'a' };
foreach (var value in values)
   Console.WriteLine("{0} - type {1}", value,
                     value.GetType().Name);

// The example displays the following output:
//       word - type String
//       True - type Boolean
//       120 - type Int32
//       136.34 - type Double
//       a - type Char
```

MemberInfo

https://learn.microsoft.com/en-us/dotnet/api/system.reflection.memberinfo?view=net-6.0

MemberInfo contains information about the attributes of a class member and provides access to member metadata. Use Type.GetMembers() to get an array of the members of a ***type***.

Some important ***MemberInfo*** Properties and extension methods are:

Method/Property	Usage
.MemberType	Contains a MemberTypes object (method, constructor, event,etc).
.Name	Contains the members official name.
.GetType()	Gets the Type of the current instance.

```
foreach ( Type t in b.GetExportedTypes() )
{
    Display(0, "");
    Display(indent, "Type: {0}", t);

    // For each type, show its members & their custom attributes.

    indent += 1;
    foreach (MemberInfo mi in t.GetMembers() )
    {
        Display(indent, "Member: {0}", mi.Name);
        DisplayAttributes(indent, mi);

        // If the member is a method, display information about its parameters.

        if (mi.MemberType==MemberTypes.Method)
        {
            foreach ( ParameterInfo pi in ((MethodInfo) mi).GetParameters() )
            {
                Display(indent+1, "Parameter: Type={0}, Name={1}", pi.ParameterType, pi.Name);
            }
        }

        // If the member is a property, display information about the property's accessor method
        if (mi.MemberType==MemberTypes.Property)
        {
            foreach ( MethodInfo am in ((PropertyInfo) mi).GetAccessors() )
            {
                Display(indent+1, "Accessor method: {0}", am);
            }
        }
    }
    indent -= 1;
}
```

MethodInfo

https://learn.microsoft.com/en-us/dotnet/api/system.reflection.methodinfo?view=net-6.0

The ***MethodInfo*** Class contains the attributes of a method and provides access to method metadata. Use Type.GetMethods() to get references to the methods in that ***Type*** object.

Some important ***MethodInfo*** Properties and extension methods are:

Method/Property	Usage
.Name	Contains the Method Name.
.ReturnParameter	A ParameterInfo object representing the return ***type*** of the method.
.GetMethodBody()	Returns a MethodBody object
.GetParameters()	Returns the methods parameters.
.Invoke()	Invokes the method and returns an object representing the return.

```
using System;
using System.Reflection;

class Example
{
    static void Main()
    {
        Type t = typeof(String);

        MethodInfo substr = t.GetMethod("Substring",
            new Type[] { typeof(int), typeof(int) });

        Object result =
            substr.Invoke("Hello, World!", new Object[] { 7, 5 });
        Console.WriteLine("{0} returned \"{1}\".", substr, result);
    }
}
```

FieldInfo

https://learn.microsoft.com/en-us/dotnet/api/system.reflection.fieldinfo?view=net-6.0

Fields are variables defined in a class. ***FieldInfo*** provides access to the metadata for a classes ***fields.***Use Type.GetFields() to get references to the fields of a ***type*** object. The class is not loaded into memory until .invoke() or get() is called on the object.

Some important ***FieldInfo*** Properties and extension methods are:

Method/Property	Usage
.Attributes	Get the attributes associated with the field.
.FieldType	Gets the type of this field object.
.Name	Gets the official name of the field.
.GetValue(Obj)	Gets the value of the field.
.SetValue(Obj, Obj)	Sets the value of the field.

```
public class FieldInfoClass
{
    public int myField1 = 0;
    protected string myField2 = null;
    public static void Main()
    {
        FieldInfo[] myFieldInfo;
        Type myType = typeof(FieldInfoClass);
        // Get the type and fields of FieldInfoClass.
        myFieldInfo = myType.GetFields(BindingFlags.NonPublic | BindingFlags.Instance
            | BindingFlags.Public);
        Console.WriteLine("\nThe fields of " +
            "FieldInfoClass are \n");
        // Display the field information of FieldInfoClass.
        for(int i = 0; i < myFieldInfo.Length; i++)
        {
            Console.WriteLine("\nName            : {0}", myFieldInfo[i].Name);
            Console.WriteLine("Declaring Type  : {0}", myFieldInfo[i].DeclaringType);
            Console.WriteLine("IsPublic        : {0}", myFieldInfo[i].IsPublic);
            Console.WriteLine("MemberType      : {0}", myFieldInfo[i].MemberType);
            Console.WriteLine("FieldType       : {0}", myFieldInfo[i].FieldType);
            Console.WriteLine("IsFamily        : {0}", myFieldInfo[i].IsFamily);
        }
    }
}
```

PropertyInfo

https://learn.microsoft.com/en-us/dotnet/api/system.reflection.propertyinfo?view=net-6.0

The ***PropertyInfo*** Class provides access to property metadata. A property's value is typically accessible through get and set accessors. Properties may be read-only, so a set may not be supported. Use Type.GetProperties() to get references to the ***properties*** in that ***Type*** object.

Some important ***PropertyInfo*** Properties and extension methods are:

Method/Property	Usage
.Name	Gets the official name of the field.
.GetMethod()	Gets the get accessor for this property.
.PropertyType	Gets the type of this property.
.GetValue(Obj)	Gets the value of the property.
.SetValue(Obj, Obj)	Sets the value of the property.

```
foreach (MemberInfo mi in t.GetMembers() )
{
    Display(indent, "Member: {0}", mi.Name);
    DisplayAttributes(indent, mi);

    // If the member is a method, display information about its parameters.

    if (mi.MemberType==MemberTypes.Method)
    {
        foreach ( ParameterInfo pi in ((MethodInfo) mi).GetParameters() )
        {
            Display(indent+1, "Parameter: Type={0}, Name={1}", pi.ParameterType, pi.Name);
        }
    }

    // If the member is a property, display information about the property's accessor methods.
    if (mi.MemberType==MemberTypes.Property)
    {
        foreach ( MethodInfo am in ((PropertyInfo) mi).GetAccessors() )
        {
            Display(indent+1, "Accessor method: {0}", am);
        }
    }
}
```