

Reflection

1 MOTIVATION	2
1.1 Methodenaufruf zur Laufzeit	2
1.2 Schnittstellenerkennung	2
2 REFLECTION MIT C#	3
2.1 Grundstruktur eines Reflection-Aufrufs	3
2.1.1 Properties des Type-Objekts	4
2.2 Laden der Assembly	4
2.3 Aufruf mit Parameter	4
2.3.1 Properties des MethodInfo-Objekts	5
2.4 Bestimmter Konstruktor	5
2.5 Überladene Methoden	6
2.6 Private Methode	6
2.7 Properties	6
2.8 Statische Methode	7
2.9 ChangeType	7
2.10 Generics	7
2.10.1 List	8
2.10.2 Dictionary	8
3 AUSLESEN VON INFOS	9
3.1 Welche Methoden hat die Klasse?	9
3.2 Welche Parameter hat die Methode?	9
3.3 Welche Properties hat die Klasse?	9
3.4 Welche Konstruktoren hat die Klasse?	9
4 MÖGLICHE FEHLER	10
4.1 Falscher Assemblyname - FileNotFoundException	10
4.2 Falscher Klassenname - ArgumentNullException	10
4.3 Falscher Methodenname - NullReferenceException	10
4.4 Falsche Parameter - TargetParameterCountException	11

1 Motivation

Sinn von Reflection ist es

- Methoden von Klassen aufzurufen, wobei die genauen Klassen u. Methoden **zur Compile-Zeit nicht bekannt** sind
- von einer Assembly (bzw. DLL) herauszufinden, welche Klassen u. Methoden diese zur Verfügung stellt

1.1 Methodenaufruf zur Laufzeit

Angenommen, es gibt die Klasse **MyClass** u. darin die Methode **public int MyFunc(int x)**.
Normalerweise ruft man diese Methode so auf:

```
MyClass myClass = new MyClass();  
int result = myClass.MyFunc(666);
```

Die Idee von Reflection ist jetzt, den Namen der Klasse u. der Methode durch einen String-Parameter zu übergeben.

Also etwa so:

```
object clazz = CreateInstanceOfClass("MyClass");  
int result = CallMethod(clazz, "MyFunc", 666);
```

Die Namen werden also als String übergeben, und können somit auch aus einem TextFeld, ListBox, o.ä. eingegeben werden.

Wie diese Idee in C# umgesetzt wird, zeige ich in den folgenden Kapiteln.

In anderen Programmiersprachen funktioniert es ähnlich.

1.2 Schnittstellenerkennung

Umgekehrt ist oft gefordert, die aktuellen Signaturen der Schnittstellen zu erkennen, z.B. bei einem Web-Backend. Wenn man erkennt, welche Backend-Methoden es gibt, welche Parameter diese verlangen und welche Arten von Objekten retourniert werden, kann man daraus entsprechenden Schnittstellencode generieren.

Das wird z.B. bei Swagger/OpenAPI so gemacht.

Auch ein JSON-Serialisierer eruiert die Properties eines Objekts und erzeugt daraus einen String im JSON-Format.

2 Reflection mit C#

Als Ausgangsbasis soll folgende Klasse dienen, die in die Assembly DummyLib.dll kompiliert wird.

```
namespace DummyLib;

public class DummyClass
{
    private readonly int _factor = 10;
    public int DummyProp { get; set; }

    public DummyClass() => Console.WriteLine("    DummyClass: Default Constructor");
    public DummyClass(int factor)
    {
        _factor = factor;
        Console.WriteLine("    DummyClass: Constructor DummyClass(int factor)");
    }

    public string DummyS(string x) => $"{x}_{x}";

    public void DummyF() => Console.WriteLine("    DummyClass: DummyF()");

    public int DummyG(int val)
    {
        int result = val * _factor;
        Console.WriteLine($"    DummyClass: DummyG({val}) = {result}");
        return result;
    }

    public double DummyH(int val, double divisor)
    {
        double result = val / divisor;
        Console.WriteLine($"    DummyClass: DummyH({val},{divisor}) = {result}");
        return result;
    }

    public double DummyH(int val, double divisor, double add)
    {
        double result = val / divisor + add;
        Console.WriteLine($"    DummyClass: DummyH({val},{divisor},{add}) = {result}");
        return result;
    }

    public static void StaticDummy(int x) => Console.WriteLine($"    DummyClass: StaticDummy({x})");

    private void PrivateDummy() => Console.WriteLine($"    DummyClass: PrivateDummy()");
}
```

Konstruktoren

Methoden

überladene
Methoden

2.1 Grundstruktur eines Reflection-Aufrufs

Die einfachste Variante sieht so aus:

```
var instance = new DummyLib.DummyClass();
instance.DummyF();

Assembly assembly = Assembly.Load("DummyLib");
Type type = assembly.GetType("DummyLib.DummyClass");
object instance = Activator.CreateInstance(type);
MethodInfo methodInfo = type.GetMethod("DummyF");
methodInfo.Invoke(instance, null);
```

1-3

4-5

1

2

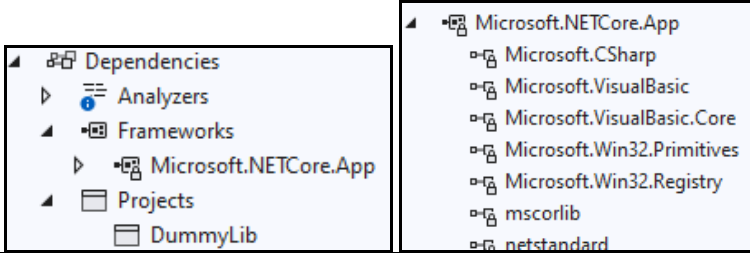
3

4

5

Da sämtliche Parameter als Strings übergeben werden, können Fehleingaben vom Compiler nicht erkannt werden. Es können daher verschiedenste Exceptions geworfen werden (siehe weiter unten).

Folgende Schritte sind wesentlich.

1	Zuerst muss man angeben, von welcher Assembly/DLL man eine Klasse laden möchte. Wichtig dabei ist, dass die Assembly vom Projekt referenziert wird:
	
2	Von der geladenen Assembly erzeugt man sich dann ein Type -Objekt. Dabei muss der fully qualified name angegeben werden. Von diesem Typ (=Klassendefinition) kann man über Properties vielerlei Informationen erhalten - am besten im Immediate Window oder Debugger ausprobieren (bzw. siehe weiter unten).
3	Mit der statischen Methode CreateInstance der Klasse Activator kann man sich eine Instanz des Typs erzeugen lassen. Dies entspricht praktisch dem Default-Konstruktor .
4	Einen Zeiger auf die gewünschte Methode bekommt man über die Methode GetMethod des Typs.
5	Der eigentliche Aufruf der Methode erfolgt über die Methode Invoke des MethodInfo-Objekts.

2.1.1 Properties des Type-Objekts

Vom **Type**-Objekt kann man sich eine Menge Informationen über die dadurch beschriebene Klasse besorgen. So z.B. ob die Klasse public definiert ist, ob sie abstract ist usw.

IsAbstract	false	bool	IsGenericType	false	bool	IsNotPublic	false	bool
IsAnsiClass	true	bool	IsGenericTypeDefinition	false	bool	IsPointer	false	bool
IsArray	false	bool	IsImport	false	bool	IsPrimitive	false	bool
IsAutoClass	false	bool	IsInterface	false	bool	IsPublic	true	bool
IsAutoLayout	true	bool	IsLayoutSequential	false	bool	IsSealed	false	bool
IsByRef	false	bool	IsMarshalByRef	false	bool	IsSecurityCritical	true	bool
IsCOMObject	false	bool	IsNested	false	bool	IsSecuritySafeCritical	false	bool
IsClass	true	bool	IsNestedAssembly	false	bool	IsSecurityTransparent	false	bool
IsConstructedGenericType	false	bool	IsNestedFamANDAssem	false	bool	IsSerializable	false	bool
IsContextful	false	bool	IsNestedFamORAssem	false	bool	IsSpecialName	false	bool
IsEnum	false	bool	IsNestedFamily	false	bool	IsUnicodeClass	false	bool
IsExplicitLayout	false	bool	IsNestedPrivate	false	bool	IsValueType	false	bool
IsGenericParameter	false	bool	IsNestedPublic	false	bool	IsVisible	true	bool

2.2 Laden der Assembly

Beim Laden der Assembly hat man grundsätzlich 4 Möglichkeiten:

- **Assembly.LoadFile()**: dabei muss den vollständigen Namen (also inkl. Pfad u. Dateierdung) angeben, also z.B. **Assembly.LoadFile("D:\Temp\System.Xml.dll")**
- **Assembly.Load()**: damit kann man eine Assembly laden, die bei den Dependencies angegeben ist. Bsp: **Assembly.Load("System.Xml")**
- **Assembly.GetExecutingAssembly()**: Liefert die Assembly des aktuellen Programms (das ist ja ein exe, und keine DLL).
- **typeof(...).Assembly**: kann man verwenden, wenn man die Assembly eines bekannten Typs erhalten möchte. Bsp: **typeof(DummyLib.DummyClass).Assembly**

2.3 Aufruf mit Parameter

Angenommen, es soll folgende Methode aufgerufen werden:

```
public int DummyG(int val)
{
    int result = val * _factor;
    Console.WriteLine($"    DummyClass: DummyG({val}) = {result}");
    return result;
}
```

Die Parameterübergabe an die Methode erfolgt durch ein object-Array:

```
Assembly assembly = Assembly.Load("DummyLib");
Type type = assembly.GetType("DummyLib.DummyClass");
object instance = Activator.CreateInstance(type);
MethodInfo methodInfo = type.GetMethod("DummyG");
object result = methodInfo.Invoke(instance, new object[] { 666 });
```

Der Rest ist wie gehabt. Das Ergebnis des Aufrufs ist vom Typ object. Dieses kann dann auf den richtigen Typ gecastet werden.

Mit „method chaining“:

```
var type = Assembly.Load("DummyLib").GetType("DummyLib.DummyClass");
object instance = Activator.CreateInstance(type);
object result = type?.GetMethod("DummyG")?.Invoke(instance, new object[] { 666 });
```

Vergleich ohne Reflection:

```
var instance = new DummyLib.DummyClass();
object result = instance.DummyG(666);
```

2.3.1 Properties des MethodInfo-Objekts

Auch über die Methode selbst kann man einige Infos erfragen:

IsAbstract	false	IsStatic	false
IsAssembly	false	IsVirtual	false
IsConstructor	false	MemberType	Method
IsFamily	false	MetadataToken	100663308
IsFamilyAndAssembly	false	MethodHandle	{System.RuntimeMethodHandle}
IsFamilyOrAssembly	false	MethodImplementationFl...	IL
IsFinal	false	Module	{DummyLib.dll}
IsGenericMethod	false	Name	"gDummy"
IsGenericMethodDefinition	false	ReflectedType	{Name = "DummyClass" FullName = "DummyLib.DummyClass"}
IsHideBySig	true	ReturnParameter	{Int32 }
IsPrivate	false	ReturnType	{Name = "Int32" FullName = "System.Int32"}
IsPublic	true	ReturnTypeCustomAttribu...	{Int32 }

2.4 Bestimmter Konstruktor

Will man das Objekt nicht über den Default-Konstruktor erzeugen, sondern mit einem speziellen Konstruktor, muss das bei **CreateInstance** entsprechend angegeben werden.

```
public DummyClass(int factor)
{
    this.factor = factor;
    Console.WriteLine($"    DummyClass: Constructor DummyClass(int factor)");
}
```

```
Assembly assembly = Assembly.Load("DummyLib");
Type type = assembly.GetType("DummyLib.DummyClass");
object instance = Activator.CreateInstance(type, new object[] { 5 });
MethodInfo methodInfo = type.GetMethod("DummyG");
object result = methodInfo.Invoke(instance, new object[] { 666 });
```

Mit „method chaining“ könnte man den Code wieder kompakter schreiben:

```
var type = Assembly.Load("DummyLib").GetType("DummyLib.DummyClass");
object instance = Activator.CreateInstance(type, 5);
object result = type
    ?.GetMethod("DummyG")
    ?.Invoke(instance, new object[] { 666 });
```

2.5 Überladene Methoden

Aufpassen muss man, wenn eine Methode der Klasse überladen ist, wenn es also denselben Methodennamen mehrfach gibt, wobei sich die Methoden nur über die Signatur unterscheiden, also überladen sind:

```
public double DummyH(int val, double divisor)
{
    double result = val / divisor;
    Console.WriteLine($"    DummyClass: DummyH({val},{divisor}) = {result}");
    return result;
}
public double DummyH(int val, double divisor, double add)
{
    double result = val / divisor + add;
    Console.WriteLine($"    DummyClass: DummyH({val},{divisor},{add}) = {result}");
    return result;
}
```

Der Aufruf mit Reflection sieht dann so aus:

```
var type = Assembly.Load("DummyLib").GetType("DummyLib.DummyClass");
object instance = Activator.CreateInstance(type);
var methodInfo = type.GetMethod("DummyH",
    new[] { typeof(int), typeof(double) });
object result = methodInfo?.Invoke(instance,
    new object[] { 666, 2 });
Console.WriteLine($"Result: {result}");

var methodInfo3p = type.GetMethod("DummyH",
    new[] { typeof(int), typeof(double), typeof(double) });
object resultB = methodInfo3p?.Invoke(instance,
    new object[] { 666, 2, 123 });
Console.WriteLine($"Result: {resultB}");
```

Die Parametertypen der gewünschten Methode werden also über ein Type-Array festgelegt. Die Typen können dabei entweder durch **typeof(int)** aber auch durch **int x=3; x.GetType()** angegeben werden.


2.6 Private Methode

Normalerweise findet **GetMethod** nur Methoden, die public sind. Mit zusätzlichen Flags kann man aber auch private Methoden aufrufen:

```
var type = Assembly.Load("DummyLib").GetType("DummyLib.DummyClass");
var methodInfo = type.GetMethod("PrivateDummy", BindingFlags.NonPublic | BindingFlags.Instance);
methodInfo.Invoke(Activator.CreateInstance(type), null);
```

2.7 Properties

Für Properties funktioniert es sehr ähnlich wie für Funktionen. Man besorgt sich über den Namen der Property ein **PropertyInfo**-Objekt. Dieses stellt dann zusätzliche Infos über die Property zur Verfügung:

CanRead	true
CanWrite	true
CustomAttributes	Count = 0
DeclaringType	{Name = "DummyClass" FullName = "DummyLib.DummyClass"}
GetMethod	{Int32 get_DummyProp()}
IsSpecialName	false
MemberType	Property
MetadataToken	385875970
Module	{DummyLib.dll}
Name	"DummyProp" 
PropertyType	{Name = "Int32" FullName = "System.Int32"}
ReflectedType	{Name = "DummyClass" FullName = "DummyLib.DummyClass"}
SetMethod	{Void set_DummyProp(Int32)}

Außerdem kann man über die Methoden **GetValue/SetValue** den Wert lesen bzw. schreiben.

Eine Codesequenz zum Schreiben einer bestimmten Property könnte z.B. so aussehen:

```
private static void CallProperty(string propertyName, int value)
{
    LogMethodName();
    var obj = new DummyClass();
    var propertyInfo = obj.GetType().GetProperty(propertyName);
    if (propertyInfo == null) return;
    if (propertyInfo.CanWrite) propertyInfo.SetValue(obj, value, null);
    var actualValue = propertyInfo.GetValue(obj, null);
    Console.WriteLine($"Calling property {propertyName}: value now is {actualValue} ...");
}
```

2.8 Statische Methode

Etwas anders muss man vorgehen, wenn man eine statische Methode aufrufen will, also eine Methode, bei der **keine Objektinstanz** benötigt wird:

```
public static void StaticDummy(int x) => Console.WriteLine($"    DummyClass: StaticDummy({x})");
```

Der Aufruf erfolgt direkt über das Type-Objekt. Das ist nicht verwunderlich, würde diese statische Methode ohne Reflection ja ebenfalls mit dem Klassennamen aufgerufen (DummyClass.StaticDummy(666)):

```
var type = Assembly.Load("DummyLib").GetType("DummyLib.DummyClass");
var method = type
    .GetMethod("StaticDummy")?
    .Invoke(type, new object[] { 666 });
```

Die Parameterübergabe erfolgt aber wie bekannt über ein object-Array.

Durch Method chaining kann man den Ausdruck wieder LINQ-mäßig schreiben:

```
Assembly.Load("DummyLib")
    .GetType("DummyLib.DummyClass")
    .GetMethod("StaticDummy")?
    .Invoke(type, new object[] { 666 });
```

Eine andere Möglichkeit für den Aufruf einer statischen Methode wäre folgender Code:

```
var result = Assembly.Load("DummyLib")
    .GetType("DummyLib.DummyClass")
    .InvokeMember("StaticDummy",
        BindingFlags.Default | BindingFlags.InvokeMethod,
        null,
        null,
        new object[] { 666 });
```

2.9 ChangeType

Oft muss man bei Reflection einen Typ in einen anderen umwandeln, z.B. einen String in einen Integer. Dazu kann man die Methode **Convert.ChangeType** benutzen.

Beispiele:

```
object iVal = Convert.ChangeType("123", typeof(int));
object sVal = Convert.ChangeType(666, typeof(string));
DateTime dateVal = (DateTime)Convert.ChangeType("22.3.2019", typeof(DateTime));
```

2.10 Generics

Für generische Typen muss man über **MakeGenericType** den entsprechenden Typ explizit erzeugen, der die gewünschten generischen Parameter enthält.

2.10.1 List

Für eine List<> würde das so aussehen – **List`1** heißt dabei generischer Typ mit einem Parameter

```
var ass = Assembly.Load("mscorlib");
var typeList = ass.GetType("System.Collections.Generic.List`1");
typeList = typeList.MakeGenericType(typeof(string));
Console.WriteLine($"typeList = {typeList.Name}");
```

2.10.2 Dictionary

Bei einem Dictionary<,> verhält es sich ähnlich, aufgrund der 2 Parameter braucht man entsprechend ein

Dictionary`2:

```
var typeDict = ass.GetType("System.Collections.Generic.Dictionary`2"); ;
typeDict = typeDict.MakeGenericType(new Type[] { typeof(int), typeof(string) });
Console.WriteLine($"typeDict = {typeDict.Name}");
```

Man hätte auch folgende Möglichkeit, wobei aber eben der Typ explizit angegeben wird und nicht als String:

```
var typeDict = typeof(Dictionary<,>);
typeDict = typeDict.MakeGenericType(new Type[] { typeof(int), typeof(string) });
Console.WriteLine($"typeDict = {typeDict.Name}");
```


3 Auslesen von Infos

3.1 Welche Methoden hat die Klasse?

Die Methode **GetMethods()** der Klasse **Type** liefert ein Array von **MethodInfo**-Objekten:

```
type.GetMethods()
{System.Reflection.MethodInfo[9]}
[0]: {Void fDummy()}
[1]: {Int32 gDummy(Int32)}
[2]: {Double hDummy(Int32, Double)}
[3]: {Double hDummy(Int32, Double, Double)}
[4]: {Void staticDummy(Int32)}
[5]: {System.String ToString()}
[6]: {Boolean Equals(System.Object)}
[7]: {Int32 GetHashCode()}
[8]: {System.Type GetType()}
```

Weitere Methoden der Klasse Type am besten im Immediate Window ausprobieren.

3.2 Welche Parameter hat die Methode?

Die Methode **GetParameters()** der Klasse **MethodInfo** liefert ein Array von **ParameterInfo**-Objekten:

```
method.GetParameters()
{System.Reflection.ParameterInfo[2]}
[0]: {Int32 val}
[1]: {Double divisor}
```

Auch hier hilft wieder das Immediate Window.

3.3 Welche Properties hat die Klasse?

Die Methode **GetProperties()** der Klasse **Type** liefert ein Array von **PropertyInfo**-Objekten.

```
type.GetProperties()
{System.Reflection.PropertyInfo[1]}
[0]: {Int32 DummyProp}
```

3.4 Welche Konstruktoren hat die Klasse?

Die Methode **GetConstructors()** der Klasse **Type** liefert ein Array von **ConstructorInfo**-Objekten. Diese Klasse ConstructorInfo sind praktisch identisch mit der Klasse MethodInfo.

```
type.GetConstructors()
{System.Reflection.ConstructorInfo[2]}
[0]: {Void .ctor()}
[1]: {Void .ctor(Int32)}
```

Wichtig: der Name der Konstruktoren ist immer **.ctor**!

4 Mögliche Fehler

Da die wesentlichen Parameter entweder als String oder als object-Array übergeben werden, können sich leicht Fehler einschleichen. Dementsprechend können verschiedene Exceptions geworfen werden. Untenstehende Programmausschnitte sollen das veranschaulichen.

4.1 Falscher Assemblyname - FileNotFoundException

```
private static void CallWrongLib()
{
    try
    {
        Assembly assembly = Assembly.Load("DummyLibWrongName");
    }
    catch (FileNotFoundException exc)
    {
        DisplayException(exc);
    }
}
```

Denselben Fehler bekommt man, wenn die Assembly nicht bei den References angegeben wurde.

4.2 Falscher Klassenname - ArgumentException

```
private static void CallWrongClass()
{
    Assembly assembly = Assembly.Load("DummyLib");
    try
    {
        Type type = assembly.GetType("DummyLib.DummyClassWrongName");
        object obj = Activator.CreateInstance(type);
    }
    catch (ArgumentException exc)
    {
        DisplayException(exc);
    }
}
```

4.3 Falscher Methodenname - NullReferenceException

```
private static void CallNonExistingMethod()
{
    Assembly assembly = Assembly.Load("DummyLib");
    Type type = assembly.GetType("DummyLib.DummyClass");
    object obj = Activator.CreateInstance(type);
    try
    {
        MethodInfo method = type.GetMethod("DummyFWrongName");
        method.Invoke(obj, null);
    }
    catch (NullReferenceException exc)
    {
        DisplayException(exc);
    }
}
```

4.4 Falsche Parameter - TargetParameterCountException

```
private static void CallWithWrongParameters()
{
    Assembly assembly = Assembly.Load("DummyLib");
    Type type = assembly.GetType("DummyLib.DummyClass");
    object obj = Activator.CreateInstance(type);
    MethodInfo method = type.GetMethod("DummyG");
    try
    {
        object result = method.Invoke(obj, new object[] { 666, 123 });
    }
    catch (TargetParameterCountException exc)
    {
        DisplayException(exc);
    }
}
```