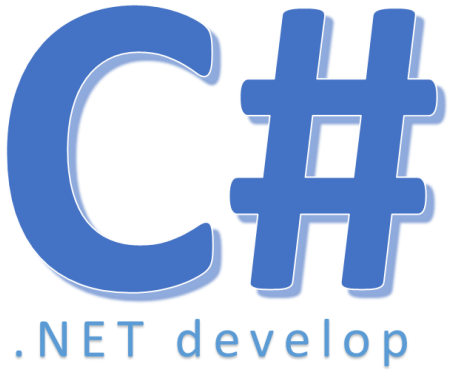
Fachhochschule Nordwestschweiz
Hochschule für Technik

C# REFLECTION



Martin Kropp, Yves Senn
University of Applied Sciences Northwestern Switzerland

Learning Targets

- You
  - know and can explain the concepts of .NET reflection and attributes
  - Can explain how reflection works for analyzing code
  - Can explain how reflection works for dynamic instantiation/plugins
  - Know the purpose of the Emit-API
  - Can apply C# reflection for developing software

Content

- What is Reflection?
- Exploring Metadata
- Building Types at Runtime
- Attributes
- Exceptions

What is Reflection?

Is the ability to inspect and change information about assemblies, types and code at runtime.

Used for

- plugin development
- performance analysis
- logging
- crash analysis
- code analysis
- mock frameworks

Use Case: Plugins

What is a plugin?

A plugin is a software component that adds a specific feature to an existing computer program. When a program supports plugins, it enables customization.

Use Case: Plugins

1. Application deployed to customers
2. Application could scan a filesystem folder for plugins
3. Plugins could be loaded automatically, and their `Init()` method called

But: You can't know all plugins at compile time

→ Impossible to add references in Visual Studio

→ Impossible to use?

Use Case: Plugins

But

- → You can’t know all plugins at compile time
- → Impossible to add references in Visual Studio
- → Impossible to use?

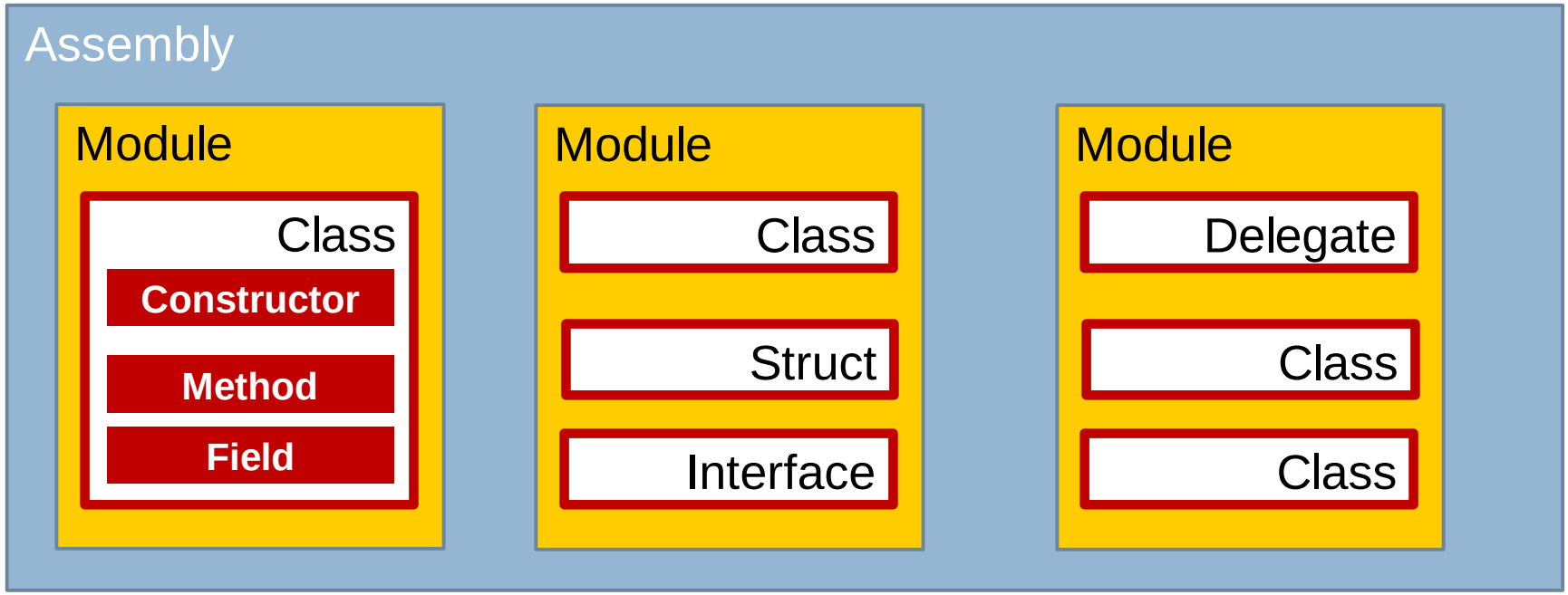
However

- → If we could inspect the plugin code and check if the plugin supports the expected API
- → That means, use **reflection**

The big picture

- Assemblies contain modules
- Modules contain types
- Types contain code

And all information about it



A Simple Example

- Loading an assembly

```
var a = Assembly.LoadFrom("MyPlugin.dll");
```

or

```
var b = Assembly.GetExecutingAssembly();
```

or

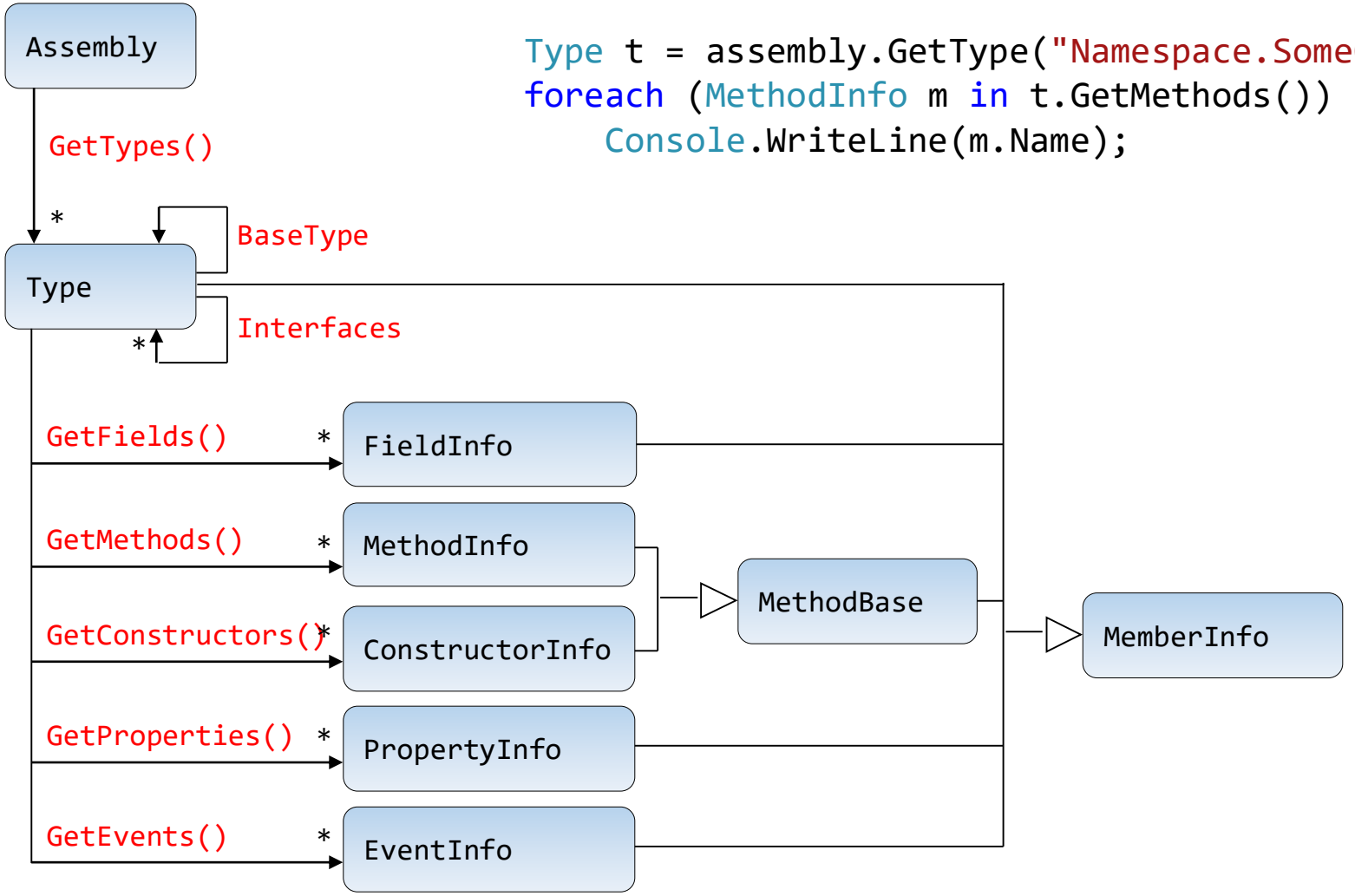
```
var c = Assembly.GetAssembly(typeof(string));
```

~~Never~~ use `Assembly.LoadFile`

- List the types in an assembly

```
Type[] types = a.GetTypes();
foreach (var t in types)
    Console.WriteLine(t.FullName);
```

Reflection API (partial)




```
Type t = assembly.GetType("Namespace.SomeClass");
foreach (MethodInfo m in t.GetMethods())
    Console.WriteLine(m.Name);
```

System.Type

Represents type declarations: class types, interface types, array and value types, enumeration types, type parameters, and more.

- Describes
  - Category: Primitive, Enum, Struct or Class
  - Methods and Constructors
  - Parameter- and Return-Types
  - Fields and Properties, Arguments and Attributes
  - Events, Delegates and Namespaces

- Returned by `someObject.GetType()` or `typeof(string)` or from `assembly.GetTypes();`

System.Type API

Value, Interface or Class?
→ `IsValueType, IsInterface, IsClass`

Public, Private or Sealed ?
→ `IsNotPublic, IsSealed`

Abstract or Implementation?
→ `IsAbstract`

Worksheet – Part 1

Type checking (Recap)

```
try
{
    var castObject = (Customer)o;
    …
}
catch(InvalidCastException) { }
```

```
if (o.GetType() == typeof(Customer)) { … }
```

```
if (o is Customer ) { … }
```

```
var castObject = o as Customer;

if (castObject != null) { … }
```

Dynamic member invocation

- Create a new instance of a type:

```
Assembly a = Assembly.LoadFrom("MyPlugin.dll");
object o = a.CreateInstance("MyPlugin.HelloWorld");
```

- Invoke `.ToString()`, without parameters:

```
Type hwClass = a.GetType("MyPlugin.HelloWorld");
MethodInfo mi = hwClass.GetMethod("ToString");
object retVal = mi.Invoke(o, null);
```

By default, GetMethod returns only public methods

Dynamic member invocation

Invoking a private Member:

```
public class SaySomething
{
    private string SayIt()
    {
        return "Private method call!";
    }
}
```

```
var saySomething = new SaySomething();
MethodInfo mi = saySomething.GetType().GetMethod("SayIt",
                                    BindingFlags.Instance |
                                    BindingFlags.NonPublic);
object text = mi.Invoke(saySomething, null);
Console.WriteLine(text);
```

Worksheet – Part 2

Attributes

Attributes

```
[Serializable]
class C {...} // marks the class as serializable
```

- Associate meta data or declarative information with code
- Can be attached to types, members, assemblies, etc.
- Can be queried at run time
- Often used by CLR services (serialization, remoting, COM interop)
- Similar to Java annotations

Common attributes

- Assembly Attributes

```
[assembly: AssemblyTitle("ConsoleApp")]
[assembly: AssemblyDescription("Route Planner Application")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("FHNW")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("M.Kropp,S.Felix,M.Schnyder,Y.Senn")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

used in `AssemblyInfo.cs` by VS

- Conditional

```
[Conditional("TRACE_ON")] //calls ignored if not #define TRACE_ON
```

Many more: http://msdn.microsoft.com/en-us/library/system.attribute.aspx#inheritanceContinued

Implement Attributes

- Are implemented as classes that are derived from `System.Attribute`:

```
// will force compiler to produce a message
public class ObsoleteAttribute : Attribute
{
    public string Message { get {...} }
    public bool IsError { get {...} set {...} }
    public ObsoleteAttribute() {...}
    public ObsoleteAttribute(string msg) {...}
    public ObsoleteAttribute(string msg, bool error) {...}
}
```

- Append the suffix `Attribute` for the class name, but use it without: `[Obsolete]`

Attributes with parameters

- Positional parameter

```
[Obsolete("Message Use class C1 instead", true)]
public class C {.. }
```

- Name parameter

```
[Obsolete(IsError=true, Message="Use class C1 instead")]
public class C {.. }
```

- Mixed


```
[Obsolete("Use class C1 instead", IsError=true)]
public class C {.. }
```

Name parameters

- Positional parameter: parameter of the constructor
- Name parameter: a public property
- Any public property can be passed as a name parameter:

```
[Obsolete("Use class C1 instead", IsError=true)]
public class C {.. }


public class ObsoleteAttribute : Attribute
{
     public bool IsError { get {...} set {...} }
     public ObsoleteAttribute(string msg) {...}
}
```

Note: There is no constructor that takes a boolean parameter

Querying of attributes

- Declaration

```
class CommentAttribute : Attribute {
     string text, author;
     public string Text { get {return text;} }
     public string Author { get {return author;} set {author = value;} }
     public CommentAttribute (string text) { this.text = text; author =“SF"; }
}
```

- Usage

```
[Comment("This is a demo class for Attributes", Author="XX")]
class C { ... }
```

- Querying the attribute at runtime

```
Type t = typeof(C);
object[] a = t.GetCustomAttributes(typeof(Comment), true);
Comment ca = (Comment)a[0];
Console.WriteLine(ca.Text + ", " + ca.Author);
```

AttributeUsage attribute

- Describes how user-defined attributes are to be used

```
public class AttributeUsageAttribute : Attribute {
  public AttributeUsageAttribute (AttributeTargets validOn) {...}
  public bool AllowMultiple { get; set; }    // default: false
  public bool Inherited { get; set; }        // default: true
  public AttributeTargets ValidOn { get; set; } // default: All
  public virtual Object TypeId { get;}
}
```

validOn	to which program elements is the attribute applicable?
AllowMultiple	can it be applied to the same program element multiple times?
Inherited	is it inherited by subclasses?
TypeID	when implemented, gets unique identifier for derived attribute classes

- Usage

```
[AttributeUsage(AttributeTargets.Class |
                AttributeTargets.Interface, AllowMultiple=false)]
public class MyAttribute : Attribute {...}
```

Exceptions

Exceptions


```
try
{
    //...
}
catch (FileNotFoundException e)
{
    Console.WriteLine($"{e.FileName} not found");
}
catch (IOException)
{
    throw;
}
catch
{
    Console.WriteLine("Unknown error");
}
finally
{
    someAPI.Close();
}
```

- `catch` clauses are checked in sequential order
- `finally` clause is always executed
- Exception parameter name can be omitted
- Exceptions derive from `System.Exception`
- Exceptions should be re-thrown by `throw;`

Exceptions

- Searching for a `catch` clause
  - Caller chain is traversed backwards until a method with a matching catch clause is found.
  - Program aborts if none is found
- Different than Java
  - Exceptions don't have to be dealt with (no "checked exceptions")
  - No `throws` keyword in method signature

System.Exception

Properties

`e.Message`	error message as a string, set by `new Exception(msg);`
`e.StackTrace`	trace of the call stack as a string
`e.Source`	the assembly that threw the exception
`e.TargetSite`	the method that threw the exception
...	

Methods

`e.ToString()`	returns the name of the exception and the stack trace
...	

More about Exceptions

- Null-coalescing operator can throw exceptions:

```
return someVariable ?? throw new NullReferenceException();
```

- Conditional operator can throw exceptions:

```
var x = (j > 5) ? throw new ArgumentException() : j;
```

- Exceptions can be filtered:

```
try
{
    //...
}
catch (COMException cex) when (cex.HResult == 42)
{
    Console.WriteLine("Error 42 occurred");
}
```

Generating Code Dynamically

- The Emit-API

Demo Factorial