Binary components, reflection

Software techniques

Department of
Automation and
Applied Informatics

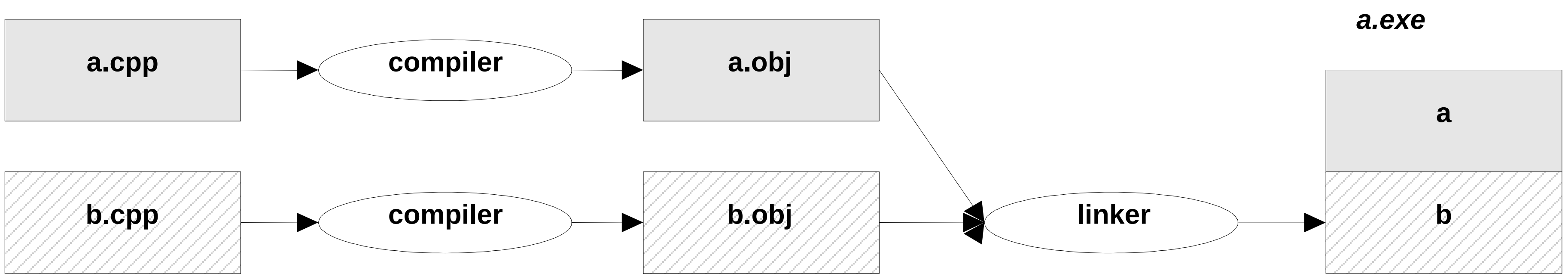
Topics

- Static and dynamic linking
  - Concepts
  - Compiling and linking
  - Static linking
  - Dynamic linking
- Binary components in C++
- Reflection in .NET environment
  - Architecture
  - Examples

Static and dynamic linking using C and C++

Process of compilation

- Particularities
  - > The compiler processes the source files one by one. The result is an .obj or an .o file for every source file
  - > The linker is responsible for address resolution

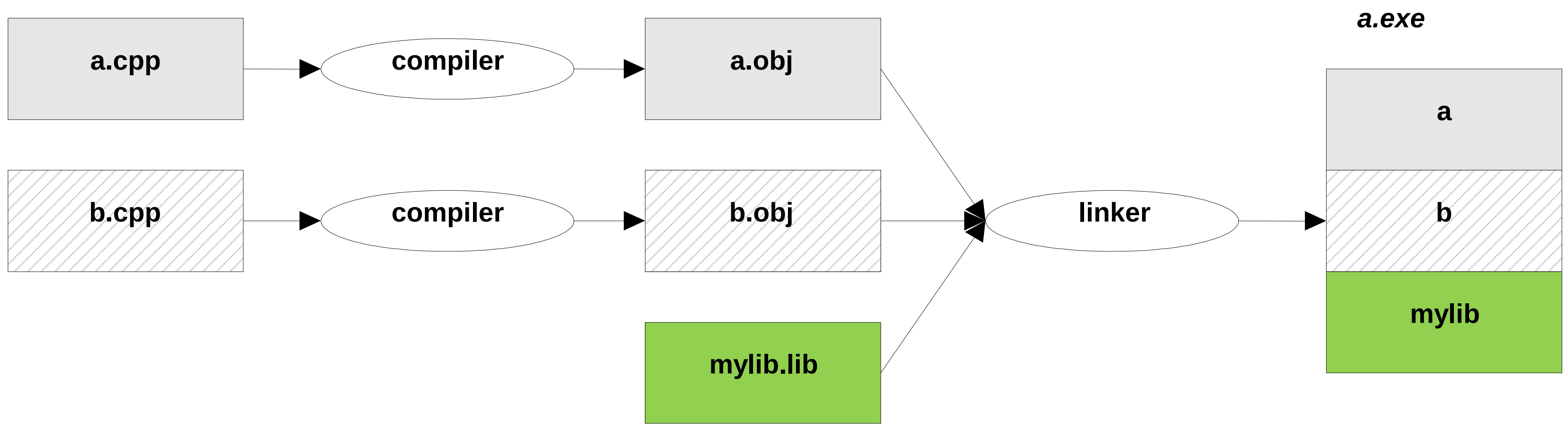


Concepts – library, binary component

- Library – C, C++, etc.
  - > A function or class library contains function and class definitions (e.g. C++ classes), constants or even resources in **compiled format**. They are called **binary components**.
- Using a library
  - > The declaration file (header file) has to be included
    - – E.g. for printf include the stdio.h header file
  - > and the library containing the definitions has to be linked to the application. There are two types of linking:
    - – A) Static
    - – B) Dynamic

Static linking

- Using static libraries



- Particularities
  > In Windows static libraries have a .lib extension. It is technically the concatenation of multiple .obj files. In Linux a statically linked library has a .a (archive) extension.

  > Basic functions (e.g. printf) are usually contained in the libc.lib file. Custom libraries can also be created.

Particularities of static linking

- Disadvantages
  - > Large application size (each application (.exe) will contain the linked library on the hard disk)
  - > Large and redundant memory consumption (if multiple running applications are using the same library it has to be loaded multiple times into the memory)
  - > Difficulties with fixing bugs: after bugfixes in the library the library has to be relinked to each application.
- Advantages
  - > The application can run as it is. No external dependencies (=> no version problems, easy to install, etc.)

Dynamic linking

- Definition: the program loads the necessary libraries only during run-time
  - The library function call references have to be resolved during runtime
  - In Windows it is a .dll (Dynamic Link Library), in Linux it is a .so (Shared Object) file.

Particularities of dynamic linking

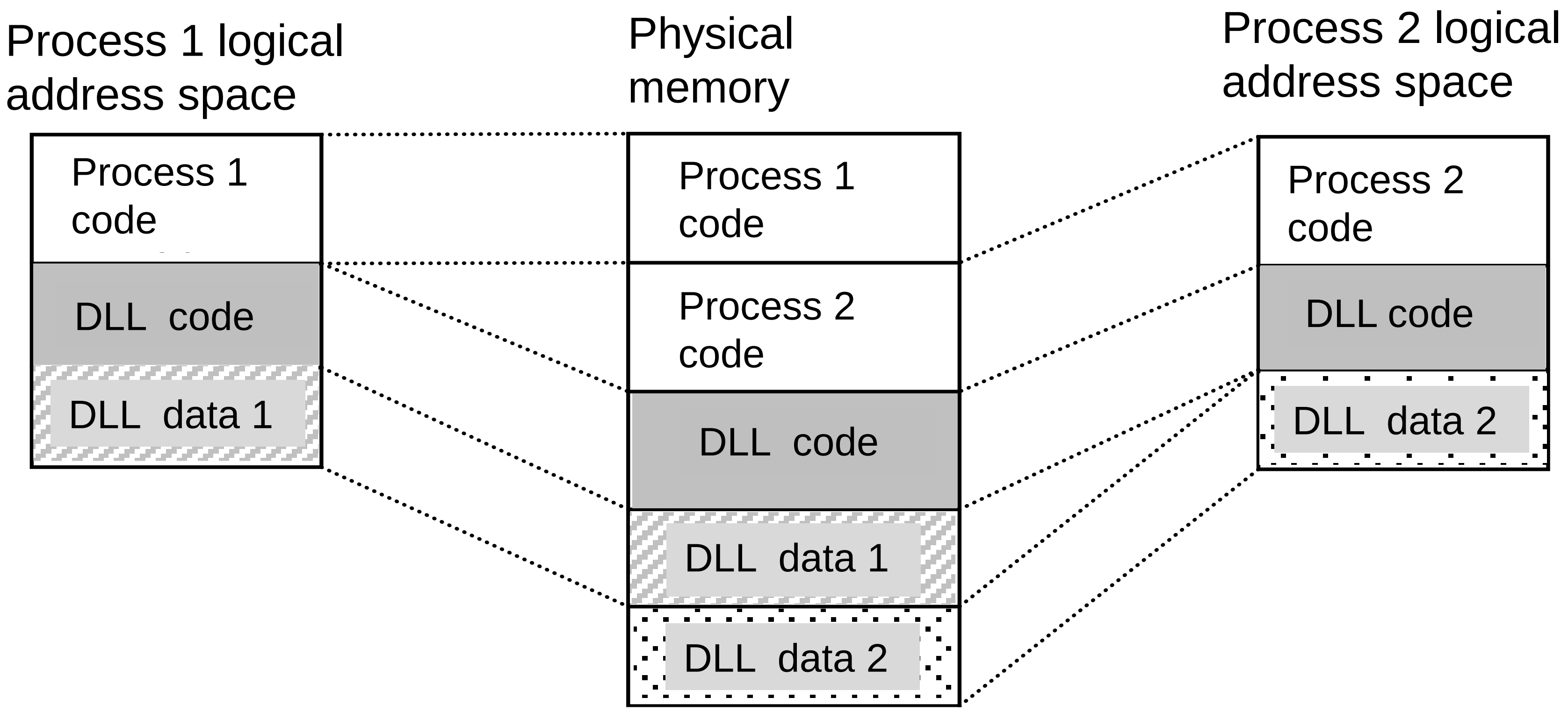
- Advantages
  - Only one instance of the library is loaded into the memory even when multiple processes use it (this is only true for the code part, the data part is loaded for each process separately).
  - They are stored only once on the hard disk if applications are using shared dll-s (e.g. from windows\system folder)
  - Small application (.exe) size
  - Bugs of the .dll can be fixed simply by replacing the dll. There is no need to relink or rebuild the application.

Particularities of dynamic linking

- Disadvantages
  - >All the needed DLLs have to be present in the system otherwise the application cannot be executed.
  - >Using shared DLLs (e.g. from windows\system folder) may cause versioning problems (DLL hell)
- DLLs for an application are searched for in
  - >the directory that contains the .exe
  - >the current working directory
  - ><windows>\<system32> folder
  - ><windows> folder
  - >folders listed in the PATH environment variable

DLLs share code in memory among applications

- The code area of a DLL is loaded only once into the memory, but the data area of the DLL is loaded separately for each process that uses the DLL at the moment. This is the only way to protect the processes from each other.



Creating a DLL

- Steps
  - Functions that are supposed to be called outside of the library has to be **exported**
  - Depends on the development tool. E.g. in Visual C++ the function declaration has to be decorated with:

    __declspec(dllexport)

Creating a DLL– exporting a function

- Complex.h

```
struct complex
{
   double re, im;
};
__declspec(dllexport) struct complex add(struct complex a, struct complex b);
```

- Complex.c

```
__declspec(dllexport) struct complex add(struct complex a, struct complex b)
{
   struct complex c;
   c.re=a.re+b.re;
   c.im=a.im+b.im;
   return c;
}
```

Creating a DLL – exporting a function

- C++ - name mangling
  - > This was C until this point. In C++ function names are overloaded: a DLL may export multiple functions with the same name. That's why C++ uses name mangling and somehow encodes the parameter list to the ultimate function name.
  - > Example: the add function after mangling

    ?add@@YA?AUcomplex@@U1@0@Z

  It cannot be called from C in this form. Name mangling can be turned off in C++ if it is a problem. (extern "C").

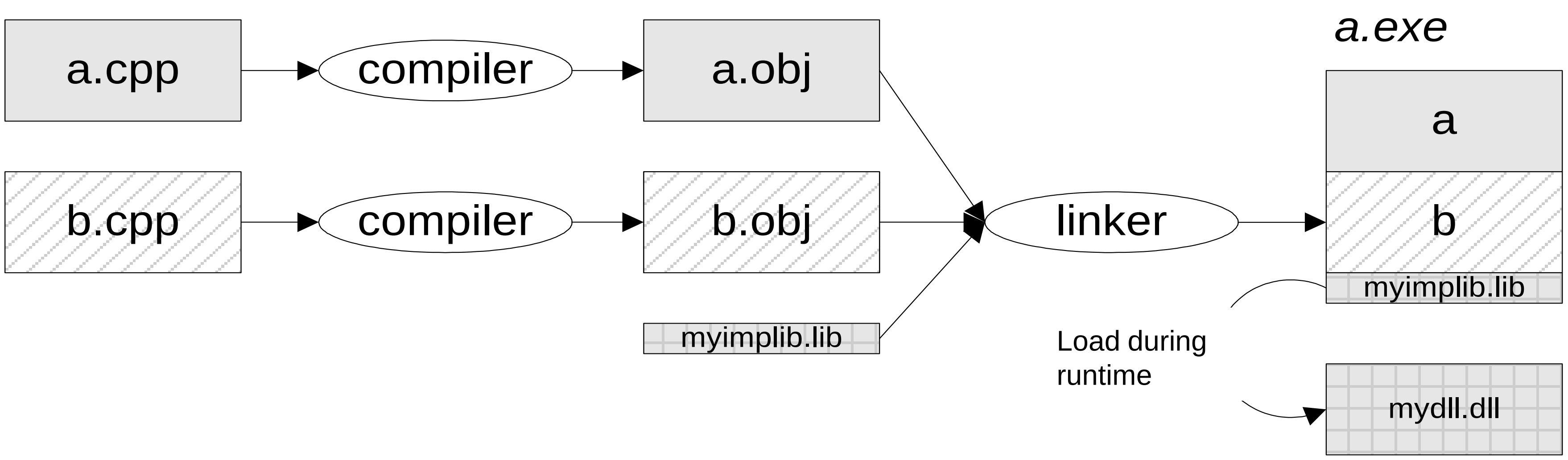
```
struct complex
{
  double re, im;
};

extern "C" __declspec(dllexport) struct complex add(struct complex a,
  struct complex b);
```

Using a DLL

- An application can use the classes or functions of the library
- Function call references have to be resolved during runtime. There are two ways:
  - Implicit linking
  - Explicit linking

Using a DLL with implicit linking

- A small import library (stub) is needed. This is statically linked to the application. It is responsible for loading the DLL into the memory when the application is started and it is also used to forward the calls to the DLL.



- Advantage: easy to use.

Using a DLL with explicit linking

- The DLL-s are loaded into the memory from code and the address of the function to call is also retrieved manually.
- Example: adding two complex numbers

```
#include "complex.h"
typedef struct complex(*ADD_TYPE)(struct complex, struct complex);

int main()
{
   struct complex a,b,c;
   HMODULE hDLL = LoadLibrary("complex.dll"); // load the DLL
   padd=(ADD_TYPE)GetProcAddress(hDLL,"add"); // Get the address of function add
   …
  c=(*padd)(a,b);
  …
  FreeLibrary(hDLL);
}
```

- Flexible: the DLL can be loaded and unloaded at any time

Binary components: C++ vs .NET

Binary C++ components

- Let us assume we have a component (eg. Exe or DLL), which uses a DLL, eg. Calls a method of a class defined in the DLL.
- If the structure of a class in the DLL changes (eg. We add a private member variable), then we must also recompile the component that uses the DLL! This is true even if the interface (public members) of the class stays the same!
- This makes it harder to develop versioned multicomponent applications!
- The problem has a C++ solution: using interfaces (abstract classes that only have pure virtual methods).
  - > This is somewhat tedious, making C++ less than ideal for developing applications with multiple components.
- Modern frameworks (.NET, Java) make this much easier.

Binary components in .NET

- In .NET DLLs and EXE files are regarded as binary components (in essence assemblies)
- Visual Studio projects usually create one component each.
- If the internals of a class changes in a DLL (but its interface stays the same), then the components that use the DLL need not be recompiled – contrary to C++. This makes the development of multicomponent .NET applications simpler.
- This works because components are not dependent on the internal details of classes defined in other components.

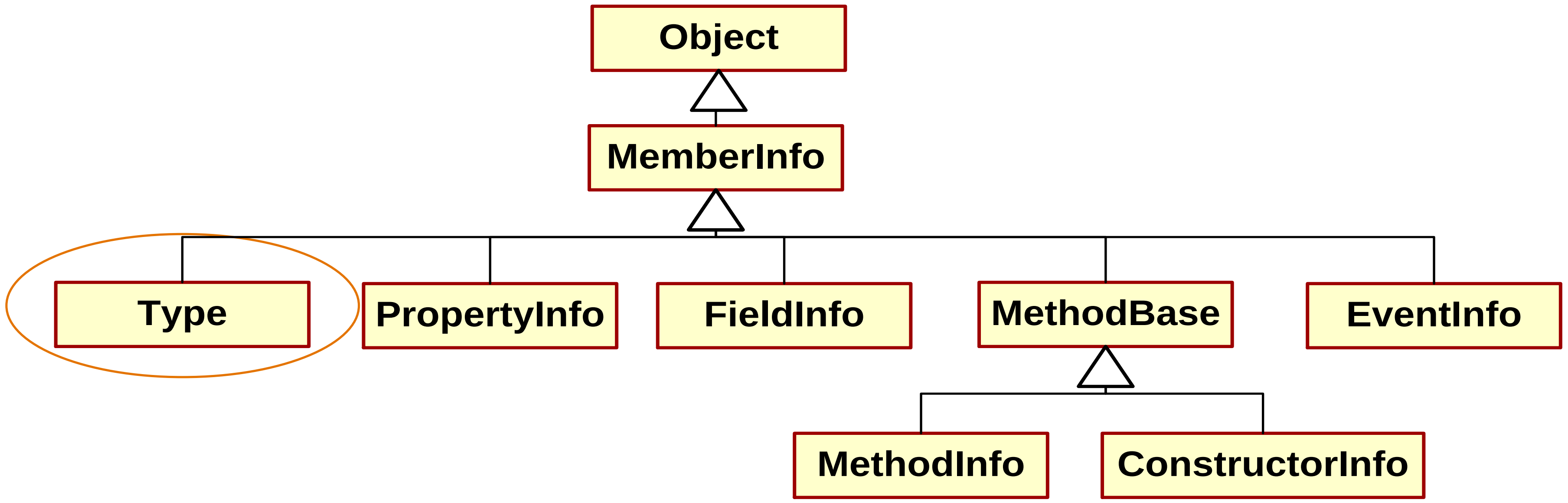
REFLECTION

Reflection

- General goal
  - > Provide meta-data on types and assemblies
- Concrete uses
  - > The types defined in an assembly can be queried at run-time
  - > The components of types (classes, interfaces, etc.) can be queried at run-time: e.g.: member variables, methods, properties, events, etc.
    - – The values of the variables can be changed
    - – The methods can be called directly (bypassing calling conventions like protection rules)
  - > We can instantiate objects without knowing their class at compile time (we provide the class' name as a string at run-time).
  - > The list of attributes for every language element (class, interface, etc.) can be queried. We can add our own custom attributes.

Reflection

- *Type* class – represents a type

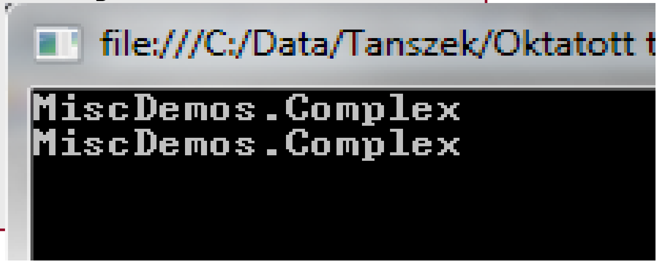


- Demos: Reflection\MiscDemos

Reflection

- Example on how to query the type of an object and of a class

```
Complex c1 = new Complex(10, 10);
Type t1 = c1.GetType(); // GetType is defined in the Object base class
Console.WriteLine(t1.FullName);
// typeof is a C# operator
Type t2 = typeof(Complex);
Console.WriteLine(t2.FullName);
```



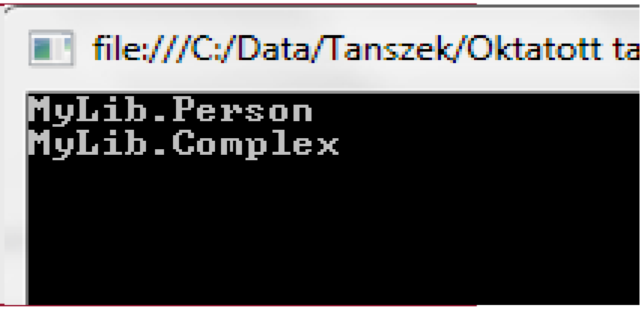
file:///C:/Data/Tanszek/Oktatott t

```
MiscDemos.Complex
MiscDemos.Complex
```

  > Type.FullName returns the type name with the namespace

- Example on how to list all the types defined in MyLib.dll

```
Assembly assembly;
assembly = Assembly.LoadFrom("MyLib.dll");
Type[] types = assembly.GetTypes();
foreach (Type type in types)
   Console.WriteLine(type.FullName);
```



file:///C:/Data/Tanszek/Oktatott ta

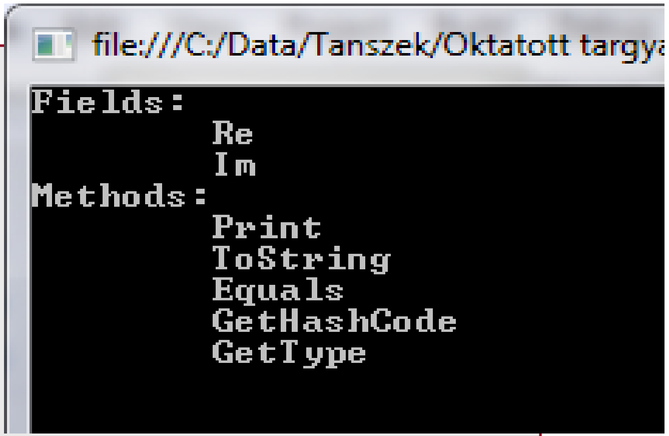
```
MyLib.Person
MyLib.Complex
```

  > The Assembly type represents an assembly loaded into the memory(.dll or.exe)

Reflection

- Example on how to query the member variables and methods of a class

```
Type type = typeof (Complex);
Console.WriteLine("Fields:");
foreach (FieldInfo fi in type.GetFields())
    Console.WriteLine("\t" + fi.Name);
Console.WriteLine("Methods:");
foreach (MethodInfo mi in type.GetMethods())
    Console.WriteLine("\t" + mi.Name);
```



```
Fields:
        Re
        Im
Methods:
        Print
        ToString
        Equals
        GetHashCode
        GetType
```

  - Print is defined in the Complex class, all other methods are inherited from the Object base class

- FieldInfo
  - Name
  - DeclaringType
  - IsPublic
  - MemberType

  …

Reflection

- Example on instantiating an object, accessing its members

```
// Create a Complex type
Type complexType = Type.GetType("MiscDemos.Complex");
ConstructorInfo ci = complexType.GetConstructor(new Type[0]);
Object instance = ci.Invoke(null);

// Get the „Re” field
FieldInfo fi = complexType.GetField("Re");
Console.WriteLine(fi.GetValue(instance));
// Set the „Re” field
fi.SetValue(instance, 10);
// Call print
MethodInfo mi = complexType.GetMethod("Print");
mi.Invoke(instance, null);
```

- Particularities
  - Very **flexible**, as strings are used to identify the members of a class
  - **Very slow!**

Reflection– example for attributes

- **Reflection example for attributes**
  - Example for: creating a custom attribute, using an attribute, querying attributes

The problem:

- Create a class that can save any kind of object into a data stream (e.g. into a file).
- Provide a way to configure which objects (by restricting to certain classes) and what variables of the selected objects are saved. Make it possible for the user of the class to specify the name under which a variable is saved.

Basic idea for the solution:

- Define a **StorableClass** custom attribute: only classes with this attribute will be saved.
- Define a **Storable** custom attribute: only member variables with this attribute will be saved. The attribute will have a name parameter that defines the storage label for the member variable.

Reflection – example for using attributes

- Demo
- This is how saving works

```
Student student = new Student("James Bond", "007007");
SaveUtils.Save(student); // Save the student object
```

- This is how we control which variables are saved

```
[StorableClass]
class Student
{
    [Storable("Name")]
    private string name;
    [Storable("Neptun", SaveType= true)]
    private string neptun;
     …
}
```

Reflection – example for using attributes

- Defining the **StorableClass** and **Storable** attributes

```
[AttributeUsage(AttributeTargets.Class)]
class StorableClassAttribute: System.Attribute {
    public StorableClassAttribute() { }
}
```

```
[AttributeUsage(AttributeTargets.Field)]
public class StorableAttribute: System.Attribute
{
    string name;

    public StorableAttribute(string name)
    {
        this.name = name;
     }

     public string Name { get { return name; } }
}
```

- Derive from the System.Attribute base class
- AttributeUsage determines where it can be used

Reflection – using an attribute: saving

```
class SaveUtils
{
    public static void Save(object o) {
        Type type = o.GetType();

        // If no StorableClassAttribute is assigned to the class dont serialize it
        object[] attributes =
            type.GetCustomAttributes(typeof(StorableAttribute) , false);
        if (attributes.Length == 0)
            return;

       FieldInfo[] fieldInfos = type.GetFields(BindingFlags.Public | BindingFlags.NonPublic
                         | BindingFlags.Instance);
        foreach (FieldInfo fi in fieldInfos)
        {
            object[] fieldAttributes = fi.GetCustomAttributes(
                                Type.GetType("Storage.StorableAttribute"), false);
            if (fieldAttributes.Length == 0)
                continue;
            StorableAttribute attr = (StorableAttribute)fieldAttributes[0];

            // In this demo we dont really save it, just print it onto the screen
            Console.WriteLine("Name: {0}", attr.Name);
            Console.WriteLine("Value: {0}", fi.GetValue(o).ToString());
        }
    }
```