ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

C#: reflection

Outline

- Introductory detour: quines
- Basic reflection
  - Built-in features
  - Introspection
  - Reflective method invocation
- Reflective code-generation
- What's reflection good for

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Introductory detour: quines

An introductory detour: quines

A quine is a program that outputs its own source code

- named after the philosopher Willard Van Orman Quine and his studies of self-reference
- this is an example of reflection

In pseudocode, the basic algorithm for a quine is:

Print the following sentence twice, the second time between quotes.
"Print the following sentence twice, the second time between quotes."

- Can you write a Quine in C#?

C# quine

Adapted for C# from a Java quine by Bertram Felgenhauer

```
class S{public static void Main(string[]a)
  {string s="class S{public static void
  Main(string[]a){string
  s=;System.Console.Write(s.Substring(0,52)+
  (char)34+s+
  (char)34+s.Substring(52));}}";System.Console
  .Write(s.Substring(0,52)+(char)34+s+
  (char)34+s.Substring(52));}}
```

Reflection in C#

- The CLR loader loads assemblies into application domains (boundaries around objects with the same application scope)
- Assemblies contain modules which contain types which contain members
- Reflection provides objects encapsulating assemblies, modules and types. You can use it to e.g.:
  - Access attributes (program's metadata)
  - Examine and create types inside assemblies
  - Build new types at runtime
  - Perform late binding: access methods and types created at runtime

Normal vs. reflective at a glance

Create an instance of **`MyClass`** and invoke the method **`myMethod`** on the instance

- normal C# programming

```
MyClass o = new MyClass(); o.myMethod();
```

- with reflection

```
Type t = Type.GetType("Reflection101.MyClass");
object o = Activator.CreateInstance(t);
t.InvokeMember("myMethod",
     BindingFlags.InvokeMethod, null, o, null);
// args: method name, binding bitmask,
  bindings, target, list of actual arguments
```

C#'s reflection operates at assembly level

ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Basic mechanisms for reflection

Built-in reflection

Operator `is` checks for a type

- example: overriding `Equals()`

```
public bool Equals(object obj){
        // Querying for a type at runtime
    if (!(obj is IntendedType) {
        return false;
    }
```

Operator `typeof` returns a `Type` object (see next slides)

Getting a **`Type`** object

`System.Type` is the entry point

- provides meta-info for assemblies, modules, and types
- obtainable from an object reference or a value type

```
Type t1 = myObj.GetType();
int i = 7;
Type t2 = i.GetType();
```

- obtainable from a class name

```
Type t3 = typeof(Namesp.SomeClassName);
Type t4 =
  Type.GetType("Namesp.SomeClassName");
```

Getting info from a `Type` object

- Members: `MemberInfo[] GetMembers()`
- Constructors: `ConstructorInfo[] GetConstructors()`
- Fields: `FieldInfo[] GetFields()`
- Methods: `MethodInfo[] GetMethods()`
- Properties: `PropertyInfo[] GetProperties()`
- Attributes: `IList<CustomAttributeData> GetCustomAttributeData()`
- Events: `EventsInfo[] GetEvents()`
- Base type: `Type BaseType`
- Generic arguments: `Type[] GetGenericArguments()`

There are similar variants of most of the methods above to get information about a specific item, e.g.:

```
MethodInfo GetMethod("aMethodName")
```

Introspecting non-`public` members

The getters in `Type` objects have variants that filter out members according to certain "binding flags".

- `FieldInfo[] GetFields(BindingFlags b)`
- `MethodInfo[] GetMethods(BindingFlags b)`
- `ConstructorInfo[] GetConstructors(BindingFlags b)`

Private members can be retrieved with these flags.

- For instance, to get all non-`public` methods declared in `Type t`:

```
t.GetMethods(BindingFlags.Instance
             | BindingFlags.NonPublic);
```

BindingFlags

An **enum** specifying flags that control binding and invocation done via reflection. Examples:

- **Instance** instance members will be included
- **NonPublic** non-public members will be included
- **InvokeMethod** a method will be invoked
- **SetField** a field will be set
- **Static** static members will be included
- ...

Combining flags with pipes

Multiple values can be combined with the logic pipe `|` operator ("or" for bitmasks), e.g.

> **`BindingFlags.Public | BindingFlags.SetField`**
> specifies the operation of setting the value of a field, provided it is public

The pipe operator is equivalent to a logical "and" on the properties we want to add

This has to do with the values associated to the **`enum`**, which are powers of 2 (see next page)

The bits within

Suppose:

- [0 1 0 0 0 0] encodes **A**
- [0 0 0 1 0 0] encodes **B**

Then:

- [0 1 0 0 0 0] **|** [0 0 0 1 0 0] **==** [0 1 0 1 0 0]
  encodes **A | B**

It's like adding items to a bucket: the ones just shift to the left (they represent powers of 2) and they are never stacked on top of each other. The "bucket" is the result of the or operation

Combining flags with ampersands

Suppose:

- [0 1 0 0 0 0] **|** [0 0 0 1 0 0] **==** [0 1 0 1 0 0]
  that is, **A | B** encodes **C** (the bucket)

What we actually do in the code is to check:

```
if ((theBucket & NonPublic) == NonPublic){...}
```

We check that **A** is in the bucket by computing **C & A** :

[0 1 0 1 0 0] **&** [0 1 0 0 0 0] **==** [0 1 0 0 0 0] **== A**

Object creation with reflection

```
// s contains the name of the created class
// it may come from any runtime input
string s = GetClassNameFromInput();
// set the class name here
Type t = Type.GetType(s);
// create an Object
Object o = System.Activator.CreateInstance(t);

// cast if we know the class name statically
MyCls o = (MyCls)
  (System.Activator.CreateInstance(t));
```

Method invocation with reflection

Let's use reflection to invoke `private` method "SetInfo" on object `o` of class `c`; the method has signature `(string, int)`

```
// get a Type object from o
  Type t = o.GetType();
// set the binding flags for a private method
  BindingFlags bf = BindingFlags.InvokeMethod
                          | BindingFlags.NonPublic;
// set the actual value of the arguments
  Object[] a = {"aVal", 4};
// invoke the method on object o (of class c)
  t.InvokeMember("SetInfo", bf, null, o, a);
```

`Type.InvokeMember` method

```
Object InvokeMember(string name, BindingFlags
  bitmask, Binder binder, Object target, Object[]
  args);
```

`name`: method name

`bitmask`: bitmask to specify how to conduct the search

`binder`: enables binding (e.g. selection of an overloaded method)

`target`: object on which to invoke the selected member

`args`: array containing the args to pass to the member to invoke

```
t.InvokeMember("SetInfo", bf, null, o, a);
```

passing `null` for the binder argument selects the `DefaultBinder` property, which applies to most common cases. If you need different behavior, inherit from `Binder` and pass an instance of that instead of `null`

Reflection and exceptions

Reflection may trigger run-time exceptions such as:

- From `System`
  - `TypeLoadException`
  - `UnauthorizedAccessException`
  - `MissingMemberException`
    - `MissingFieldException`
    - `MissingMethodException`

- From `System.Reflection`
  - `TargetException`
  - `TargetInvocationException`
  - `TargetParameterCountException`

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Reflective code-generation

Reflective code generation

- We may need to change the behavior of a method at runtime

- Code generation is a solution
  - You can use the services collected in the `System.Reflection.Emit` namespace to edit directly CIL code
  - Class-to-class transformation is another example of code generation (similarly to what we have seen in Java)

Steps to realize a dynamic HelloWorld

1. Create a dynamic assembly and a module
2. Create a new type for the Main method
3. Create the Main method
4. Generate the Intermediate Language (IL) for Main
5. Create a Type object for the result of the previous process
6. Invoke the Main
7. Set the entry point for the application
8. Save the executable file

The sample code in the following pages is by Joel Pobar (slightly adapted):
http://blogs.msdn.com/b/joelpob/archive/2004/01/21/61411.aspx

Code using `System.Reflection.Emit` (1/3)

```
using System; using System.Reflection;
using System.Reflection.Emit; using System.Threading;
namespace EmitHelloWorld

class MainClass {
   public static void main(String[] args) {
        //step 1: create a dynamic assembly and a module
        AssemblyName assemblyName = new AssemblyName();
        assemblyName.Name = "HelloWorld";
        AssemblyBuilder assemblyBuilder =
Thread.GetDomain().DefineDynamicAssembly(assemblyName,
AssemblyBuilderAccess.RunAndSave);
        ModuleBuilder module =
        assemblyBuilder.DefineDynamicModule("HelloWorld.exe");
```

Code using `System.Reflection.Emit` (2/3)

```
// step 2: create a new type for the Main method
TypeBuilder typeBuilder = module.DefineType("HelloWorldType",
TypeAttributes.Public | TypeAttributes.Class)

// step 3: create the Main method
MethodBuilder methodbuilder = typeBuilder.DefineMethod("Main",
   MethodAttributes.HideBySig MethodAttributes.Static |
   MethodAttributes.Public, typeof(void), new Type[]
   {typeof(string[])});

// step 4: generate the IL for Main
ILGenerator ilGenerator = methodbuilder.GetILGenerator();
ilGenerator.EmitWriteLine("hello, world");
ilGenerator.Emit(OpCodes.Ret);
```

Code using `System.Reflection.Emit` (3/3)

```
// step 5: create a Type object for the result of the previous
   process
Type helloType = typeBuilder.CreateType();

// step 6: invoke the Main method
helloType.GetMethod("Main").Invoke(null, new string[] {null});

// step 7: set the entry point for the application
assemblyBuilder.SetEntryPoint
   (methodbuilder, PEFileKinds.ConsoleApplication);

// step 8: Save the executable file
assemblyBuilder.Save("HelloWorld.exe");
}
}
}
```

What's reflection good for

- Class browsers
- Object inspectors
- Code analysis tools
- Testing applications
- Enterprise server code

The reflection trade-off

- Powerful reflection mechanisms increases flexibility
  - Powerful solutions to specific problems
  - Very useful for infrastructure code
- Flexibility comes at a price
  - Performance penalty
  - Security restrictions (reflective code may not run in certain restricted environments)
  - Encapsulation violation
  - More code, more difficult to understand

Performance overhead with reflection

- Class construction overhead
  - One-time cost, usually negligible
- Execution overhead
  - A reflexive call is typically slower than a normal call
  - Can be significant if the application does heavy usage of reflection
- Bottom line: choose reflection when and where is the right design choice