ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Java: reflection

Outline

Introductory detour: quines

Basic reflection

- Built-in features
- Introspection
- Reflective method invocation

Dynamic proxies

Reflective code-generation

What’s reflection?

A language feature that enables a program to examine itself at runtime and possibly change its behavior accordingly

- It may be cumbersome in imperative programming paradigms
  - traditional architectures distinguish between data and instructions
  - instructions are executed, while data is modified
  - this distinction is, however, purely conventional, as both are stored in memory
- The usage of metadata is the key to reflection

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
- it 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 Java?

Java quine

- From: http://www.nyx.net/~gthompso/self_java.txt
- Author: Bertram Felgenhauer

```
class S{
public static void main(String[]a){
  String s="class S{public static void
  main(String[]a){String s=;char c=34;
  System.out.println(s.substring(0,52)+c+s+c
  +s.substring(52));}}";
  char c=34;
  System.out.println(s.substring(0,52)+c+s+c
  +s.substring(52));
}}
```

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Basic mechanisms for reflection

Normal vs. reflective at a glance

Creating an instance of **MyClass** and invoking public method **myMethod** is normally straightforward:

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

Reflection makes things a bit harder:

```
Class<?> c = Class.forName("mypkg.MyClass");//1
Object o = c.newInstance();//2
//if the type is known statically we can cast
MyClass o = (MyClass)c.newInstance();//2bis
//2nd argument: formal arg. list for myMethod
Method m=c.getMethod("myMethod",(Class<?>)null);//3
//2nd argument: actual arg. list to invoke myMethod
m.invoke(o, (Object[]) null);//4
```

Small quiz: methods with parameters

Let’s assume that `myMethod` takes a `String` and an `int`:

```
MyClass o = new MyClass(); o.myMethod(“x”, 1);
```

How does the reflective code changes?

```
Class<?> c = Class.forName(“mypkg.MyClass");
Object o = c.newInstance();
//2nd argument: formal arg. list for myMethod
Method m=c.getMethod(“myMethod”, //what here?);
//2nd argument: actual arg. list to invoke myMethod
m.invoke(o, //what here?);
```

Small quiz: methods with parameters

Let’s assume that **myMethod** takes a **String** and an **int**:

```
MyClass o = new MyClass(); o.myMethod(“x”, 1);
```

How does the reflective code changes?

```
Class<?> c = Class.forName(“mypkg.MyClass");
Object o = c.newInstance();
//2nd argument: formal arg. list for myMethod
Method m=c.getMethod(“myMethod”, String.class,
  int.class);
//2nd argument: actual arg. list to invoke myMethod
m.invoke(o, new Object[]{new String(“x”),1});
```

Exceptions thrown by reflective code

```
try{
Class<?> c = Class.forName("mypkg.MyClass");//1
Object o = c.newInstance();//2
Method m=c.getMethod("myMethod",(Class<?>)null);//3
m.invoke(o, (Object[]) null);//4}
//these are only the checked exceptions thrown
catch {ClassNotFoundException e} {////thrown by 1}
catch {InstantiationException e} {//thrown by 2}
catch {IllegalAccessException e} {//thrown by 2,4}
catch {NoSuchMethodException e} {//thrown by 3}
catch {IllegalArgumentdException e} {//thrown by 4}
catch {InvocationTargetException e} {//thrown by 4}
```

Some unchecked exceptions and errors are also thrown...

Built-in reflection

Operator `instanceof`

- example: overriding `equals()`

```
public boolean equals(Object obj){
        // Querying for a type at runtime
    if (!(obj instanceof IntendedType) {
        return false;
    }
...
}
```

Getting a `Class` object

- `java.lang.Class<T>` is the entry point
  - represents the meta-info for classes
- How can I get a Class object?
  - from an object reference

```
Class<?> c1 = myObj.getClass();
```

  - from any type (including primitive types)

```
Class<?> c2 = int.class;
```

  - from a primitive type, through the wrapper

```
Class<?> c3 = Integer.TYPE;
```

  - from a (fully-qualified) class name

```
Class<?> c4 = Class.forName(“
  ch.ethz.inf.se.java.reflect.myClassName”);
```

Introspecting a class

`Class` objects provide information about:

- Modifiers: `int getModifiers()`
  - access (visibility) modifiers: `abstract`, `public`, `static`, `final`, ... encoded as an integer
  - use static method `Modifier.toString(int mod)` to get a textual representation
- Generic type parameters:

  `TypeVariable<Class<?>>[] getTypeParameters()`
- Implemented interfaces: `Class[] getInterfaces()`
- Inheritance hierarchy: `Class[] getClasses()`
- Annotations: `Annotation[] getAnnotations()`

Introspecting public class members

`Class` objects provide information about public members:

- Fields:

```
Field[] getFields()
Field getField(String fieldName)
```

- Methods:

```
Method[] getMethods()
Method getMethod(String methodName,
        Class<?>...paramTypes)
```

- Constructors:

```
Constructor<?>[] getConstructors()
Constructor<?> getConstructor(String
constructorName, Class<?>...paramTypes)
```

Introspecting all class members

- Fields:

```
Field[] getDeclaredFields()
Field getDeclaredField(String fieldName)
```

- Methods:

```
Method[] getDeclaredMethods()
Method getDeclaredMethod(String methodName, Class<?>...paramTypes)
```

- Constructors:

```
Constructor<?>[] getDeclaredConstructors()
Constructor<?> getDeclaredConstructor(Class<?>...paramTypes)
```

To make a non-visible field accessible via reflection, invoke:

```
f.setAccessible(true)//what's the type of f?
```

Reflection and security

- Method `setAccessible(boolean flag)` in classes `Field` and `Method` toggles runtime access checking
- The security manager of the JVM can disable `setAccessible` altogether
- The default security manager allows `setAccessible` on members of classes loaded by the same class loader as the caller

Reflection and exceptions

Besides the already mentioned checked exceptions, reflection may trigger the following un-checked exceptions and errors:

- `SecurityException`
- `NullPointerException`
- `ExceptionInInitializerError`
- `LinkageError`

While we don’t have to handle these exceptions and errors, we do have to handle the checked ones, bloating the code even more

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Dynamic proxies

Dynamic proxies

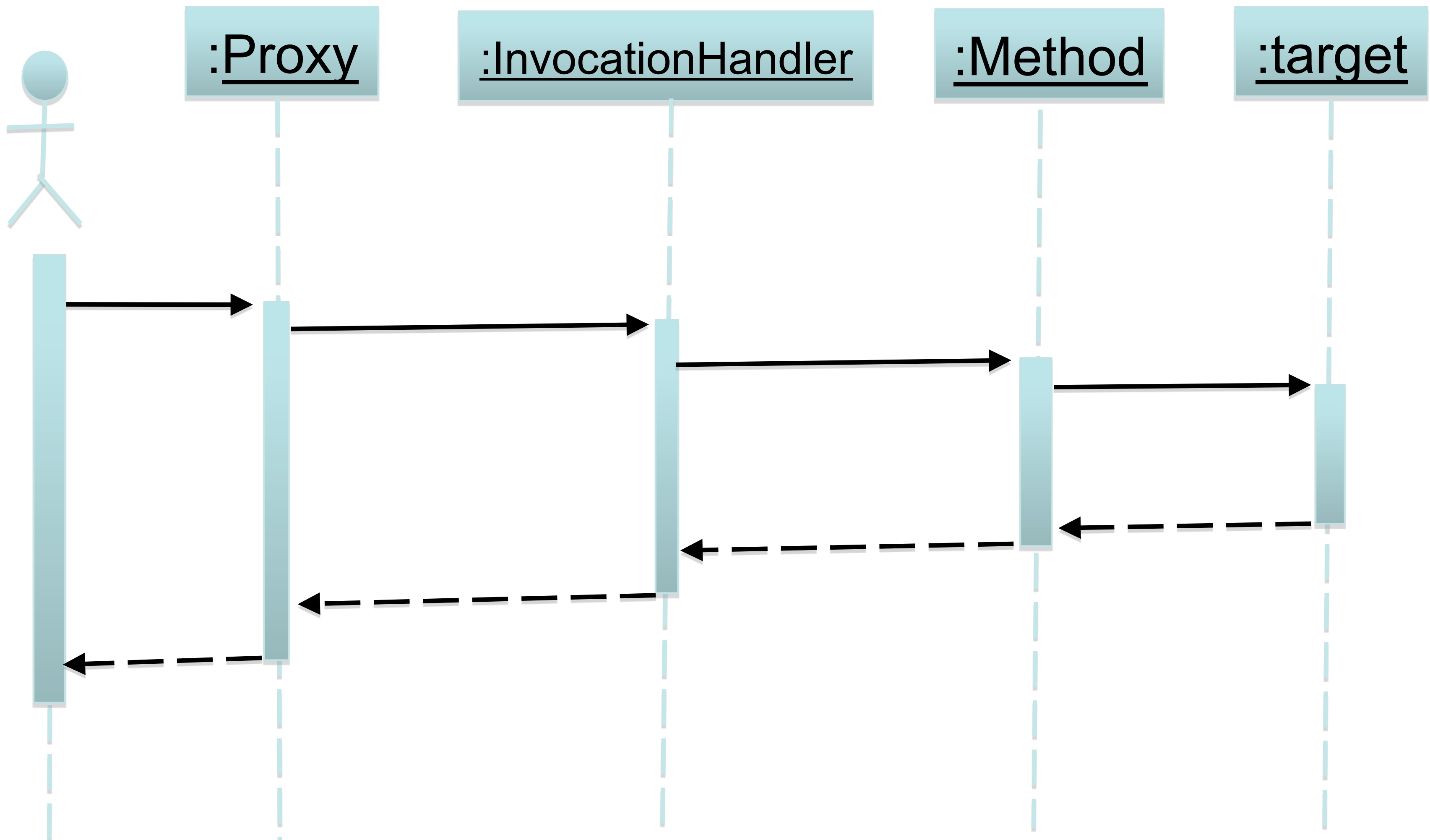
The idea comes from the Proxy design pattern (GoF):

Allows for object level access control by acting as a pass through entity or a placeholder object

Dynamically created classes that implement some interfaces

- Typical usage of dynamic proxy objects: intercept calls to objects of different classes implementing the same interfaces
- Standard Java approach to Aspect Oriented Programming (AOP): cross-cutting concerns are centralized

Proxy sequence diagram



java.lang.reflect.Proxy

Java's dynamic proxy factory:

- The factory produces objects of classes extending class **`Proxy`**
- They also implement the proxied interfaces and associate an **`InvocationHandler`** object

```
Object newProxyInstance(ClassLoader loader,
  Class<?>[] interfaces, InvocationHandler h)
```

- **`InvocationHandler`** is an interface to wrap objects providing methods that can handle method calls to proxy instances
- The handler object holds a reference to the target object

Example: a proxy for shapes

```
public interface IDrawable {
  public void draw();
}

public class Shape implements IDrawable {
  public void draw(){
      //draw a shape
  }
...
}
```

A factory for shapes

The clients gets an **IDrawable** object:

```
public class DrawablesFactory{

public static IDrawable getDrawable(){
   Shape s = new Shape();
   return Proxy.newProxyInstance(
       this.getClass().getClassLoader(),
       new Class[]{IDrawable.class},
       new CustomInvocationHandler(s));
}
}
```

Sample invocation handler

```
class CustomInvocationHandler
            implements InvocationHandler{
  private proxied;
  public CustomInvocationHandler(Shape s){
      proxied = s; }

  public Object invoke(Object proxy, Method m,
  Object[] args) throws Throwable{
      // Pre-processing here
      Object result = m.invoke(proxied, args);
      // Post-processing here
      return result;
  }
}
```

Proxy usage: example

```
/* If the client does not know which
  specific type comes from the factory */
IDrawable s =
  DrawablesFactory.getDrawable();
/* If the client wants to use other
  features of Shape as well*/
Shape s = (Shape)
  DrawablesFactory.getDrawable();


s.draw();
```

Dynamic Proxies hints and tips

- You can only proxy for an interface, not for a class
- Use handlers to process requests
- `instanceof` can be used on proxy objects
- Casting works with proxy objects

What is a Class Loader

- For every class in the system, the JVM maintains a copy of the class code in the form of an instance of **`java.lang.Class`**
  - the **`class`** attribute of any **`Object`** returns it
- Every class is loaded in the JVM by an instance of **`java.lang.ClassLoader`**
  - reflection is really built-in the JVM
- Within the JVM, a class is uniquely defined by:
  - its fully-qualified name (i.e., including the package name)
  - and the instance of the class loader that loaded it
- User-defined class loaders may make different usages of the same class incompatible (if loaded by unrelated class loaders)

Possible usages of class loaders

- Load resources bundled in JARs
- Load, unload, update modules at runtime
- Use different versions of a library at the same time
- Isolate different applications running within the same VM (static variables could be a problem otherwise)
- Exercise control over where the code comes from (e.g. a network)

ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Reflective code-generation

Reflective code generation

- Basic Java reflection is limited
- Dynamic proxies are more powerful, but their level of granularity is the method
- We may need to change the behavior of a method at runtime
- Code generation is a solution
- Class-to-class transformation is an example of code generation

Class-to-class transformation

- Input: a class
- Output: another class, obtained by transforming the input
- Use reflection to examine the input class (no parser needed)
- Load generated classes dynamically at runtime

Generating static HelloWorld (1/2)

```
class HelloGenerator {
   public static void main(String[] args)
        throws Exception {
        // Step 1: generate class text on file
        PrintWriter pw = new PrintWriter( new
                           FileOutputStream("Hello.java"));
        pw.println("... class text here ...");
        // Step 2: compile .java file into bytecode
        Process p = Runtime.getRuntime().exec(
              new String[]{"javac","Hello.java"});
        p.waitFor();
        // continues on next slide
```

Generating static HelloWorld (2/2)

```
    // continues from previous slide
    // If compilation went fine...
    if(p.exitValue() == 0){
    // now the runtime knows about the Hello class
    // Step 3: use dynamically generated class
    Class<?> helloObj = Class.forName("Hello");
    Method m = helloObj.getMethod("main", String[].class);
    // null target because 'main' is static
    m.invoke(null, new Object[]{new String[]{}});
    }
    else{     /* handle I/O errors */ }


    }
}
```