Languages in Depth Series: Java Programming

Prof. Dr. Bertrand Meyer

Java Reflection

Marco Piccioni

Der Plan

- Basic reflection
  - Built-in facilities
  - Introspection
  - Invoking methods reflectively
- Dynamic proxies
- Reflective code generation
- Assessing reflection

What’s reflection?

- A language feature that enables a program to examine itself at runtime and possibly change its behavior accordingly
- Feasible because a program represents itself using metadata
- `Class`, `Method`, `Field`, are sample metadata classes

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;
  }
//Do something to enforce equals() contract
  ...}
```

Getting a Class object

- `java.lang.Class` is the entry point
    - Represents the meta-info on a certain class

```
//If you have an object reference
Class<?> cl1 = myObj.getClass();
//If you have a primitive type like int
Class<?> cl2 = int.class;//or using wrapper
Class<?> cl3 = Integer.TYPE;
//If you have the fully qualified class name
Class<?> cl4 =
  Class.forName(“ch.ethz.inf.se.java.reflect.myClassName”);
```

Introspecting a class

Using a `Class` object we can get info about the following

- Modifiers
  - access modifiers, `abstract`, `static`, `final`
  - `int getModifiers()`
- Generic type parameters

`TypeVariable<Class<?>>[] getTypeParameters()`

- Implemented interfaces
  - `Class[] getInterfaces()`
- Inheritance path
  - `Class[] getClasses()`
- Annotations
  - `Annotation[] getAnnotations()`

More introspection: class members

Using a `Class` object we can also get info about the following

- Fields, each represented by a `Field` object
  - `Field[] getFields()`
- Methods, each represented by a `Method` object
  - `Method[] getMethods()`
- Constructors, each represented by a `Constructor` object
  - `Constructor[] getConstructors()`

- Private members are not returned

Even more introspection on members

- To get info on all the members declared in the current class, including private ones

- `getDeclaredFields()`
- `getDeclaredMethods()`
- `getDeclaredConstructors()`

- Accessibility can be set programmatically (more on this later)

Creating an object reflectively

```
...
//Reading the class name in a config file
ResourceBundle rb =
  ResourceBundle.getBundle("ch.ethz.inf.se.java.reflect.myPropFile");
String s = rb.getString("myClassName");
Class<?> c1 = Class.forName(s);
//We need to know the class name here
TestClass tc1=(TestClass)(c1.newInstance());
...
```

Invoking a method reflectively

```
...
//We know just the method name to invoke, and
  the arg types; c1 is as before
Class<?>[] argArray={(Class<?>)
  Class.forName("java.lang.String"),
  Integer.TYPE};
Method m=c1.getMethod("setInfo",argArray);
Object o=c1.newInstance();
//These are the values we want to pass
Object[] stringArray={”aVal”, new Integer(4)};
m.invoke(o,stringArray);
...
```

Some reflection exceptions

- ClassNotFoundException
- InstantiationException
- SecurityException
- NoSuchMethodException
- IllegalAccessException
- InvocationTargetException

Quiz: Does this compile and execute?

```
package ch.ethz.inf.se.java.reflect;
public class AClass {
  private String s;
  public AClass(String s) { this.s=s;}
}
  public static void main(String[] args) {
  Class<?> c=Class.forName("ch.ethz.AClass");
     c.newInstance();
  }
}
```

Quiz solution

The code does NOT compile

- `ClassNotFoundException`, `InstantiationException` and `IllegalAccessException` have to be handled

- Even if we deal with the exceptions at compile time by using `try-catch` or `throws`, at execution time the application throws an `InstantiationException`, as there is no no-args constructor

Dynamic Proxies

- Dynamically created classes that can implement interfaces

- Objects from dynamic proxy classes are typically used to intercept calls to some other classes that implement the same interfaces

- A standard Java solution to some problems AOP (Aspect Oriented Programming) tries to solve

- The cross-cutting concerns are centralized

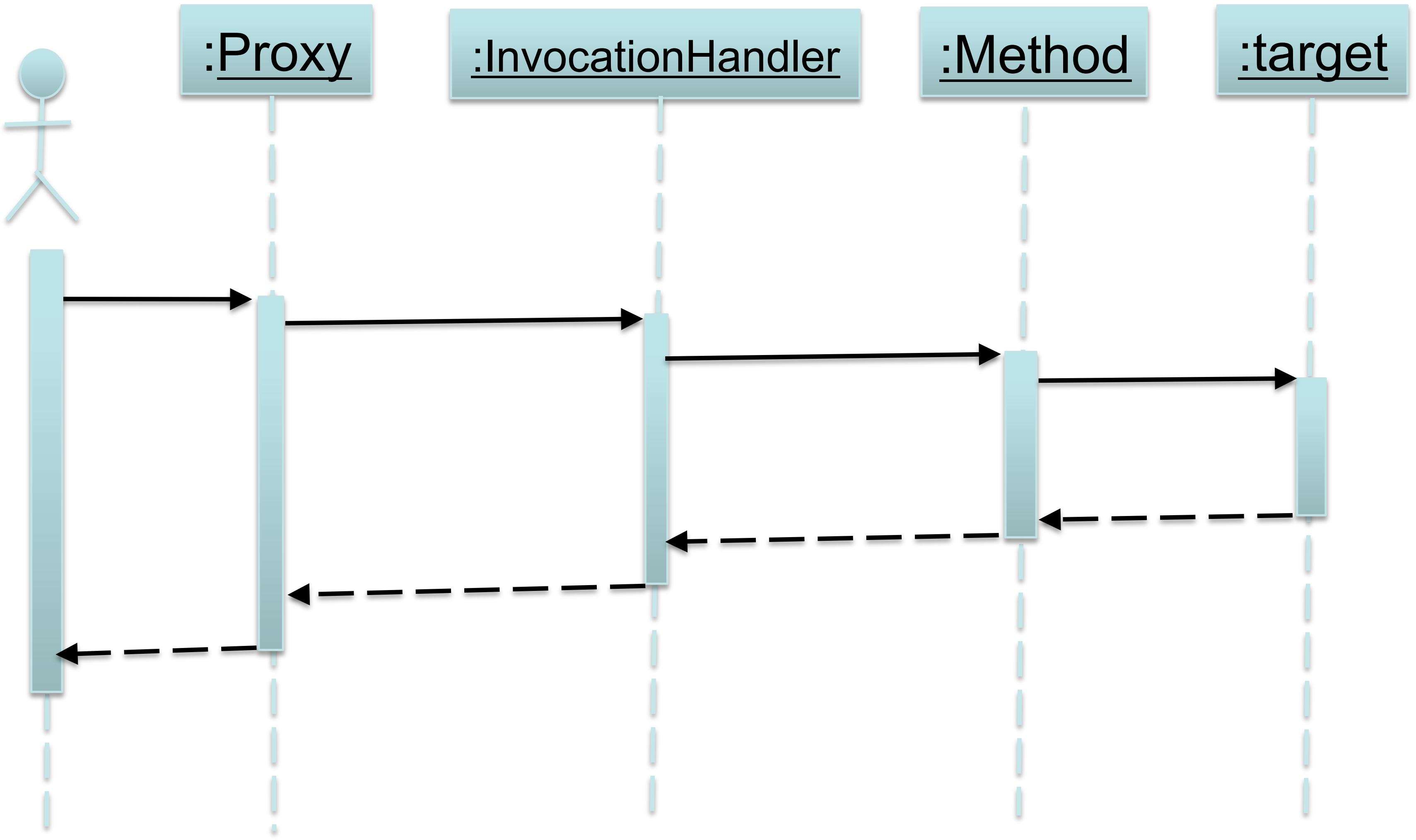
Java support: java.lang.reflect.Proxy

- Each proxy class constructed by this factory method extends `Proxy`, implements the proxied interfaces and wraps an `InvocationHandler`
  - `Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)`

- `InvocationHandler` is an interface to handle methods that are declared in proxied interfaces

- Invocation handlers are objects that handle a method call for a proxy instance, and are responsible for holding references to the target object

Proxy sequence diagram



Creating a Proxy

```
...
public static Object createProxy(Object
  objToProxy)
{
  return Proxy.newProxyInstance
  (obj.getClass().getClassLoader(),
  obj.getClass().getInterfaces(),
  new CustomInvocationHandler(objToProxy));
}
...
```

Creating an InvocationHandler

```
class CustomInvocationHandler implements
  InvocationHandler{
...
  public Object invoke(Object proxy, Method
  method, Object[] args) throws Throwable
  {
  //Some pre-processing here
     Object result=method.invoke(target, args);
  //Some post-processing here
     return result;
  }
...
}
```

Dynamic Proxies hints and tips

- You can only proxy for an interface (not for a class)
- Use handlers to process requests
- instanceof is true with a proxy object
- Also casting works fine with a proxy object

Reflective code generation

- Basic Java reflection is limited
- Proxies are ok, but can only act before or after a method invocation
- Sometimes the capacity of completely change a method behavior at runtime is needed
- Code generation is a solution
- Class-to-class transformation is a useful example of code generation

Class-to-class-transformations

- Take a class as input and generate another class
- Use introspection to examine the input class
- A Java parser is therefore not needed
- Generated classes can be dynamically loaded into a running program

Generating HelloWorld

```
class HelloGenerator {
  public static void main(String[] args) throws
  Exception{
      PrintWriter pw = new PrintWriter(new
                FileOutputStream(“Hello.java”));
      pw.println(“class text here”);
      pw.flush();
      Process pro= Runtime.getRuntime().exec
      (“javac Hello.java”)
      pro.waitFor();
      ...
```

Generating HelloWorld (continued)

```
...
  if(pro.exitValue() == 0) {
     Class<?> helloObj = Class.forName(“Hello”);
     Class<?>[] parList = {String[].class};
     Method m = helloObj.getMethod(“main”,parList);
     m.invoke(null, new Object[]{new String[]{}});
  }
  else
  {
     //handle I/O issues
  }
...
}
```

Reflection pros and cons

- Increased flexibility
  - Provide good solutions for specific problems
  - Good for infrastructure code

- The price to pay
  - Performance penalty (slighter with latest JVM)
  - Security restrictions
  - Encapsulation violation
  - More code, and more difficult to understand

Applications for which reflection is good

- Class browsers
- Object inspectors
- Code analysis tools
- J2EE servers
- Dynamic code generation for cross-cutting concerns

Is reflection really inefficient?

- Construction overhead
  - One-time cost, practically never matters
- Execution overhead
  - A reflexive call is typically slower w.r.t. a normal call
  - Matters only with heavy use and when the program does little more, which seldom happens
- Hint: choose reflection when and where is the right design choice
- Hint: sometimes you just need it to create objects of unknown classes. To access them directly, use their interface

Reflection and security

- Method `setAccessible(boolean flag)` in classes `Field` and `Method` can suppress or enable runtime access checking
- Setting accessibility can be disabled in the security manager
- The default security manager permits to set accessibility on members of classes loaded by the same class loader as the caller

References

[1] http://java.sun.com/docs/books/tutorial/reflect/

[2] http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html

[3] Ira R. Forman, Nate Forman Java Reflection in Action, Manning 2005

[4] Joshua Bloch, Effective Java, Addison Wesley 2001