Java Reflection
Explained Simply
Table of Contents

1. Introduction to Java Reflection
2. Dynamic Proxies
3. Example Uses of Java Reflection
# Introduction to Java Reflection

## 1. Introduction

- **Typical way a developer learns Java:**
  - Buys a large book on Java
  - Starts reading it
  - Stops reading about half-way through due to project deadlines
  - Starts coding (to meet deadlines) with what he has learned so far
  - Never finds the time to read the rest of the book

- **Result is widespread ignorance of many “advanced” Java features:**
  - Many such features are not complex
  - People just assume they are because they never read that part of the manual
  - Reflection is one “advanced” issue that is not complex

## What is reflection?

- When you look in a mirror:
  - You can see your reflection
  - You can act on what you see, for example, straighten your tie

- In computer programming:
  - Reflection is infrastructure enabling a program can see and manipulate itself
  - It consists of metadata plus operations to manipulate the metadata

- **Meta means self-referential**
  - So metadata is data (information) about oneself

## Widespread ignorance of Java reflection

- **Introduction to Java Reflection**

- **Java Reflection**

- **Explained Simply**

- **CiaranMcHale.com**

## Is reflection difficult?

- **When learning to program:**
  - First learn iterative programming with if-then-else, while-loop, ...
  - Later, learn recursive programming

- **Most people find recursion difficult at first**
  - Because it is an unusual way of programming
  - But it becomes much easier once you “get it”

- **Likewise, many people find reflection difficult at first**
  - It is an unusual way of programming
  - But it becomes much easier once you “get it”
  - Reflection seems natural to people who have written compilers
    (a parse tree is conceptually similar to metadata in reflection)

- **A lot of reflection-based programming uses recursion**

## Accessing metadata

- **Java stores metadata in classes**
  - Metadata for a class: `java.lang.Class`
  - Metadata for a constructor: `java.lang.reflect.Constructor`
  - Metadata for a field: `java.lang.reflect.Field`
  - Metadata for a method: `java.lang.reflect.Method`

- **Two ways to access a Class object for a class:**
  ```java
  Class c1 = Class.forName("java.util.Properties");
  Object obj = ...;
  Class c2 = obj.getClass();
  ```

- **Reflection classes are inter-dependent**
  - Examples are shown on the next slide
Examples of inter-relatedness of reflection classes

```java
class Class {
    Constructor[] getConstructors();
    Field[] getDeclaredField(String name);
    Field[] getDeclaredFields();
    Method[] getDeclaredMethods();
    ...}

class Field {
    Class getType();
    ...}

class Method {
    Class[] getParameterTypes();
    Class getReturnType();
    ...}
}
```

Introduction to Java Reflection 9

Miscellaneous Class methods

- Here are some useful methods defined in `Class`
  ```java
class Class {
    public String getName(); // fully-qualified name
    public boolean isArray();
    public boolean isInterface();
    public boolean isPrimitive();
    public Class getComponentType(); // only for arrays
    ...}
}
```

Introduction to Java Reflection 10

Invoking a default constructor

- Use `Class.newInstance()` to call the default constructor
  ```java
  abstract class Foo { ... }
  abstract class PluginManager { ... }
  abstract class Plugin {
      abstract void op1(...);
      abstract void op2(...);
      ...}
  ...
  Plugin obj = PluginManager.load("...");
  ```

Introduction to Java Reflection 11

Invoking a non-default constructor

- Slightly more complex than invoking the default constructor:
  ```java
  abstract class Plugin {
      public static Plugin load(String name) throws Exception {
          String className = props.getProperty("...");
          Class c = Class.forName(className);
          Constructor cons = c.getDeclaredConstructor();
          Plugin obj = PluginManager.load("...");
      }
  }
```

Introduction to Java Reflection 12

Metadata for primitive types and arrays

- Java associates a `Class` instance with each primitive type:
  ```java
  Class c1 = int.class;
  Class c2 = boolean.class;
  Class c3 = void.class;
```

Introduction to Java Reflection 13

Invoking a default constructor (cont’)

- This technique is used in CORBA:
  ```java
  - CORBA is an RPC (remote procedure call) standard
  - Factory operation is called `ORB.init()`
  - A system property specifies which implementation of CORBA is used
  ```

Introduction to Java Reflection 14

A plug-in architecture

- Use a properties file to store a mapping for `plugin name -> class name`
  ```java
  Many tools support plugins: Ant, Maven, Eclipse, ...
  abstract class Plugin {
      abstract void op1(...);
      abstract void op2(...);
      ...}
  ...
  Plugin obj = PluginManager.load("...");
```

Introduction to Java Reflection 15

3. Calling constructors

- Use `Class.forName()` to access the `Class` object for an array
  ```java
  Class c4 = byte.class;
  Class c5 = Class.forName("[B");
  Class c6 = Class.forName("[B");
  ```

Introduction to Java Reflection 16

Encoding scheme used by `Class.forName()`

- B -> byte; C -> char; D -> double; F -> float; I -> int; J -> long;
- Lclass-name -> class-name(); S -> short; Z -> boolean
- Use as many "["s as there are dimensions in the array

Introduction to Java Reflection 17
Chapter 1: Introduction to Java Reflection

### Passing primitive types as parameters
- If you want to pass a primitive type as a parameter:
  - Wrap the primitive value in an object wrapper
  - Then use the object wrapper as the parameter

- **Object wrappers for primitive types:**
  - boolean → java.lang.Boolean
  - byte → java.lang.Byte
  - char → java.lang.Character
  - int → java.lang.Integer
  - ...

### Invoking a method
- Broadly similar to invoking a non-default constructor:
  - Use `Class.getMethod(String name, Class[] parameterTypes)`
  - Then call `Method.invoke(Object target, Object[] parameters)`

### Looking up methods
- The API for looking up methods is fragmented:
  - You can lookup a public method in a class or its ancestor classes
  - Or, lookup a public or non-public method declared in the specified class

```java
class Class {
  public Method getMethod(String name, Class[] parameterTypes);
  public Method[] getMethods();
  public Method getDeclaredMethod(String name, Class[] parameterTypes);
  public Method[] getDeclaredMethods();
  ...
}
```

- A better name would have been `getPublicMethod()`

### Finding an inherited method
- This code searches up a class hierarchy for a method
  - Works for both public and non-public methods

```java
Method findMethod(Class cls, String methodName, Class[] paramTypes) {
  Method method = null;
  while (cls != null) {
    try {
      method = cls.getDeclaredMethod(methodName, paramTypes);
      break;
    } catch (NoSuchMethodException ex) {
      cls = cls.getSuperclass();
    }
  }
  return method;
}
```

### Accessing a field
- There are two ways to access a field:
  - By invoking get- and set-style methods (if the class defines them)
  - By using the code shown below

```java
Object obj = ...
Class c = obj.getClass();
Field f = c.getDeclaredField("firstName");
f.set(obj, "John");
Object value = f.get(obj);
```

- A better name would have been `getPublicField()`

### 4. Methods

### 5. Fields
Chapter 1: Introduction to Java Reflection

Finding an inherited field

- This code searches up a class hierarchy for a field
  - Works for both public and non-public fields

```java
Field findField(Class cls, String fieldName)
{
    Field field = null;
    while (cls != null) {
        try {
            field = cls.getDeclaredField(fieldName);
            break;
        } catch (NoSuchFieldException ex) {
            cls = cls.getSuperclass();
        }
    }
    return field;
}
```

6. Modifiers

- Java defines 11 modifiers:
  - abstract, final, native, private, protected, public, static, strictfp, synchronized, transient and volatile

- Some of the modifiers can be applied to a class, method or field:
  - Set of modifiers is represented as bit-fields in an integer
  - Access set of modifiers by calling `int getModifiers()`

- Useful static methods on `java.lang.reflect.Modifier`:
  - static boolean isAbstract(int modifier);
  - static boolean isFinal(int modifier);
  - static boolean isNative(int modifier);
  - static boolean isPrivate(int modifier);
  - ...

Further reading

- There are very few books that discuss Java reflection
  - An excellent one is *Java Reflection in Action* by Ira R. Forman and Nate Forman
  - It is concise and easy to understand

- Main other source of information is Javadoc documentation

Summary

- This chapter has introduced the basics of Java reflection:
  - Metadata provides information about a program
  - Methods on the metadata enable a program to examine itself and take actions

- Reflection is an unusual way to program:
  - Its "meta" nature can cause confusion at first
  - It is simple to use once you know how

- The next chapter looks at a reflection feature called *dynamic proxies*
What is a proxy?
- Dictionary definition: “a person authorized to act for another”
  - Example: if you ask a friend to vote on your behalf then you are “voting by proxy”
- In computer terms, a proxy is a delegation object (or process)
- Used in remote procedure call (RPC) mechanisms:
  - Client invokes on a (local) proxy object
  - Proxy object sends request across the network to a server and waits for a reply
- Some companies set up a HTTP proxy server:
  - Firewall prevents outgoing connections to port 80
  - So web browsers cannot connect to remote web sites directly
  - Web browsers are configured to connect via the company’s proxy server
  - Proxy server can be configured to disallow access to eBay, Amazon, ...

Steps required to create a dynamic proxy
- Step 1:
  - Write a class that implements InvocationHandler
  - Your implementation of invoke() should:
    - Use Method.invoke() to delegate to the target object
    - Provide some “added value” logic
- Step 2:
  - Call Proxy.newInstance(), with the following parameters:
    - targetObj: target object
    - targetObj.getClass().getClassLoader()
    - targetObj.getClass().getInterfaces()
    - InvocationHandler object “wrapper” around the target object
- Step 3:
  - Typecast the result of Proxy.newInstance() to an interface implemented by the target object

Sample code
```java
public class Handler implements InvocationHandler {
    private Object target;
    public Handler(Object target) {
        this.target = target;
    }
    public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
        Object result = null;
        try {
            ... // added-value code
            result = m.invoke(target, args);
        } catch(InvocationTargetException ex) {
            ... // added-value code
            throw ex.getCause();
        }
        return result;
    }
}
```

Java Reflection
Explained Simply

License
Copyright © 2008 Ciaran McHale.
Permission is hereby granted, free of charge, to any person obtaining a copy of this
training course and associated documentation files (the “Training Course”), to deal in
the Training Course without restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Training
Course, and to permit persons to whom the Training Course is furnished to do so,
subject to the following conditions:
The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Training Course.
THE TRAINING COURSE IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY
KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE TRAINING COURSE
OR THE USE OR OTHER DEALINGS IN THE TRAINING COURSE.
## Example uses for dynamic proxies

- **Added-value code might:**
  - Enforce security checks
  - Begin and commit or rollback a transaction
  - Use reflection & recursion to print details of all parameters
  (for debugging)

- **In a testing system, a proxy might “pretend” to be target object**
  - Returns “test” values instead of delegating to a real object
  - EasyMock ([www.easymock.org](http://www.easymock.org)) makes it easy to write tests in this way
Example Uses of Java Reflection

Java Reflection

Explained Simply

1. Basic uses of Java reflection

Example Ant build file

```xml
<project name="example build file" ...>
    <property name="src.dir" value="..."/>
    <property name="build.dir" value="..."/>
    <property name="lib.dir" value="..."/>
    <property name="target" value="do-everything"/>
    <mkdir dir="..."/>
    <mkdir dir="..."/>
    <javac srcdir="..." destdir="..." excludes="..."/>
    <jar jarfile="..." basedir="..." excludes="..."/>
    <foo .../>
</target>
<taskdef name="foo" classname="com.example.tools.Foo"/>
</project>
```

Auto-completion in a text editor

- Some Java editors and IDEs provide auto-completion
  - Example: you type "someObj." and a pop-up menu lists fields and methods for the object’s type
- The pop-up menu is populated by using Java reflection

JUnit

- JUnit 3 uses reflection to find methods whose names start with "test"
- The algorithm was changed in JUnit 4
  - Test methods are identified by an annotation
  - Reflection is used to find methods with the appropriate annotation

Below is an extract from a Spring configuration file:

```xml
<beans ...>
    <bean id="employee1" class="com.example.xyz.Employee">
        <property name="firstName" value="John"/>
        <property name="lastName" value="Smith"/>
        <property name="manager" ref="manager"/>
    </bean>
    <bean id="manager" class="com.example.xyz.Employee">
        <property name="firstName" value="John"/>
        <property name="lastName" value="Smith"/>
        <property name="manager" ref="manager"/>
    </bean>
    ...
</beans>
```

Spring
### 2. Code generation and bytecode manipulation

**Code generators (cont')**
- Compile-time code generation in a project:
  - Use technique described on previous slide to generate code
  - Then run Java compiler to compile generated code
  - Use Ant to automate the code generation and compilation
- Runtime code generation:
  - Use techniques described on previous slide to generate code
  - Then invoke a Java compiler from inside your application:
    - Can use (non-standard) API to Sun Java compiler
    - Provided in tools.jar, which is shipped with the Sun JDK
    - Or can use Janino (an open-source, embeddable, Java compiler)
      - Hosted at www.janino.net
    - Finally, use Class.forName() to load the compiled code

### Uses for Java bytecode manipulation

- Compilers:
  - Write a compiler for a scripting language and generate Java bytecode
    - Result: out-of-the-box integration between Java and the language
  - Groovy (groovy.codehaus.org) uses this technique
- Optimization:
  - Read a .class file, optimize bytecode and rewrite the .class file
- Code analysis:
  - Read a .class file, analyze bytecode and generate a report
- Code obfuscation:
  - Mangle names of methods and fields in .class files
- Aspect-oriented programming (AOP):
  - Modify bytecode to insert “interception” code
  - Generate proxies for classes or interfaces
  - Spring uses this technique

### Tools for bytecode manipulation

- Example open-source projects for bytecode manipulation:
  - ASM (http://asm.objectweb.org)
  - BCEL (http://jakarta.apache.org/bcel/)
  - SERP (serp.sourceforge.net)
- CGLIB (Code Generation LIBrary):
  - Built on top of BCEL
  - Provides a higher-level API for generating dynamic proxies
  - Used by other tools, such as Spring and Hibernate

### 3. Summary
Chapter 3: Example Uses of Java Reflection

Summary

- A lot of tools use Java reflection:
  - Plugins to extend functionality of an application (Ant)
  - Auto-completion in Java editors and IDEs
  - Use naming conventions of methods to infer semantics (JUnit test methods)
  - Tie components together (Spring)
  - Compile-time code generation
  - Runtime code generation
    - Generate proxies
    - Generate servlets from a markup language (JSP)
  - Evaluate Java expressions entered interactively by a user