Java Reflection

Java Reflection

Explained Simply

CiaranMcHale.com — *Complexity explained simply*

Copyright License

About the Author

Ciaran McHale has a Ph.D. in computer science from Trinity College Dublin. He has been working for IONA Technologies (www.iona.com) since 1995, where he is a principal consultant. His primary talent is the ability to digest complex ideas and re-explain them in simpler ways. He applies this talent to subjects that stir his passion, such as multi-threading, distributed middleware, code generation, configuration-file parsers, and writing training courses. You can find details of some of his work at his personal web site: www.CiaranMcHale.com. You can email him at Ciaran@CiaranMcHale.com.

Acknowledgements

Ciaran McHale’s employer, IONA Technologies (www.iona.com) generously gave permission for this training material to be released under the stated open-source license.

Table of Contents

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

Introduction to Java Reflection

Java Reflection

Explained Simply

CiaranMcHale.com
Complexity explained simply

License

1. Introduction

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

- 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

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

2. Metadata

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:

```
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

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

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

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


Metadata for primitive types and arrays

- Java associates a `Class` instance with each primitive type:

```
Class c1 = int.class;
Class c2 = boolean.class;
Class c3 = void.class;
```

Might be returned by `Method.getReturnType()` (→ `void.class`)

- Use `Class.forName()` to access the `Class` object for an array

```
Class c4 = byte.class;                // byte
Class c5 = Class.forName("[B");       // byte[]
Class c6 = Class.forName("[[B");      // byte[][]
Class c7 = Class.forName("[Ljava.util.Properties");
```

- 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

Miscellaneous Class methods

- Here are some useful methods defined in `Class`

```
class Class {
  public String getName(); // fully-qualified name
  public boolean isArray();
  public boolean isInterface();
  public boolean isPrimitive();
  public Class getComponentType(); // only for arrays
  ...
}
```


3. Calling constructors

Invoking a default constructor

- Use `Class.newInstance()` to call the default constructor
  Example:

```
abstract class Foo {
    public static Foo create() throws Exception {
        String className = System.getProperty(
                "foo.implementation.class",
                "com.example.myproject.FooImpl");
        Class c = Class.forName(className);
        return (Foo)c.newInstance();
    }
    abstract void op1(...);
    abstract void op2(...);
}
...
Foo obj = Foo.create();
obj.op1(...);
```

Name of property → "foo.implementation.class"

Default value → "com.example.myproject.FooImpl"

Invoking a default constructor (cont')

- This technique is used in CORBA:
  - CORBA is an RPC (remote procedure call) standard
  - There are many competing implementations of CORBA
  - Factory operation is called `ORB.init()`
  - A system property specifies which implementation of CORBA is used
- A CORBA application can be written in a portable way
  - Specify the implementation you want to use via a system property (pass -D<name>=<value> command-line option to the Java interpreter)
- Same technique is used for J2EE:
  - J2EE is a collection of specifications
  - There are many competing implementations
  - Use a system property to specify which implementation you are using

A plug-in architecture

- Use a properties file to store a mapping for *plugin name → class name*
  - Many tools support plugins: Ant, Maven, Eclipse, …

```
abstract class Plugin {
    abstract void op1(...);
    abstract void op1(...);
}
abstract class PluginManager {
    public static Plugin load(String name)
                                throws Exception {
        String className = props.getProperty(name);
        Class c = Class.forName(className);
        return (Plugin)c.newInstance();
    }
}
...
Plugin obj = PluginManager.load("...");
```


Invoking a non-default constructor

- Slightly more complex than invoking the default constructor:
  - Use `Class.getConstructor(Class[] parameterTypes)`
  - Then call `Constructor.newInstance(Object[] parameters)`

```
abstract class PluginManager {
    public static Plugin load(String name)
                                throws Exception {
        String className = props.getProperty(name);
        Class c = Class.forName(className);
        Constructor cons = c.getConstructor(
           new Class[]{String.class, String.class});
        return (Plugin)cons.newInstance(
                          new Object[]{"x", "y"});
    }
}
...
Plugin obj = PluginManager.load("...");
```

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`
  - `...`

4. Methods

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)`

```
Object obj = ...
Class c = obj.getClass();
Method m = c.getMethod("doWork",
              new Class[]{String.class, String.class});
Object result= m.invoke(obj, new Object[]{"x","y"});
```


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

A better name would have been `getPublicMethod()`

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

Finding an inherited method

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

```
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;
}
```


5. Fields

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

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


Looking up fields

- The API for looking up fields is fragmented:
  - You can lookup a *public* field in a class or its ancestor classes
  - Or, lookup a public or non-public field *declared* in the specified class

A better name would have been `getPublicField()`

```
class Class {
    public Field   getField(String name);
    public Field[] getFields();
    public Field   getDeclaredField(String name);
    public Field[] getDeclaredFields();
    ...
}
```

Finding an inherited field

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

```
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 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);
...
```


Accessing non-public fields and methods

- Both `Field` and `Method` define the following methods (inherited from `java.lang.reflect.AccessibleObject`):

```
boolean isAccessible();
void setAccessible(boolean flag);
static void setAccessible(AccessibleObject[] array,
                          boolean flag);
```

- Better terminology might have been "SuppressSecurityChecks" instead of "Accessible"
- Example of use:

```
if (!Modifier.isPublic(field.getModifiers()) {
    field.setAccessible(true);
}
Object obj = field.get(obj);
```

Hibernate uses this technique so it can serialize non-public fields of an object to a database

7. Further reading and summary

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*

Dynamic Proxies

Java Reflection

Explained Simply

CiaranMcHale.com
Complexity explained simply

License

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, …

Dynamic proxies in Java

- Java 1.3 introduced dynamic proxies
  - The API is defined in the `java.lang.reflect` package

```
class Proxy {
    public static Object newProxyInstance(
                         ClassLoader loader,
                         Class[] interfaces,
                         InvocationHandler h) throws ...
    ...
}

interface InvocationHandler {
    Object invoke(Object proxy,
                  Method m,
                  Object[] args) throws Throwable;
}
```

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.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

How does this work?

- The `Proxy.newProxyInstance()` method:
  - Uses runtime code generation techniques
  - Generates a “hidden” class with a name of the form `$Proxy<int>` (Use of “`$`” prevents namespace pollution)
  - Generated class:
    - Implements the specified interfaces
    - Each method puts parameters into `Object[]` and calls `InvocationHandler.invoke()`
- Can use a dynamic proxy *only if* a class implements 1+ interfaces
  - Use of interfaces is a good programming practice
  - So this requirement is not a problem in practice

Sample code

```
public class Handler implements InvocationHandler {
    private Object target;

    private 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;
    }
    ... // continued on the next slide
```

The `proxy` parameter is usually ignored

Sample code (cont')

```
    ... // continued from the previous slide

    public static Object createProxy(Object target)
    {
        return Proxy.newProxyInstance(
                    target.getClass().getClassLoader(),
                    target.getClass().getInterfaces(),
                    new Handler(target));
    }
}
```

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) makes it easy to write tests in this way

Example Uses of Java Reflection

Java Reflection

Explained Simply

CiaranMcHale.com
Complexity explained simply

License

1. Basic uses of Java reflection

Ant

- Ant reads build (compilation) instructions from an XML file
- Ant is hard-coded to know how to process top-level elements
  - `property`, `target`, `taskdef` and so on
- Each Ant task (used inside `target` elements) is a plug-in
  - See example Ant build file on the next slide for examples of tasks
- Many task plug-ins are bundled with the Ant distribution (`jar`, `javac`, `mkdir`, …)
  - A properties file provides a mapping for *task-name* → *class-that-implements-task*
- Users can use `taskdef` to tell Ant about user-written tasks
  - See example on the next slide

Example Ant build file

```
<?xml version="1.0"?>
<project name="example build file" ...>
    <property name="src.dir" value="..."/>
    <property name="build.dir" value="..."/>
    <property name="lib.dir" value="..."/>

    <target name="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 (Annotations were introduced in Java 1.5)
  - Reflection is used to find methods with the appropriate annotation

Spring

- Below is an extract from a Spring configuration file:

```
<?xml version="1.0"?>
<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>
```

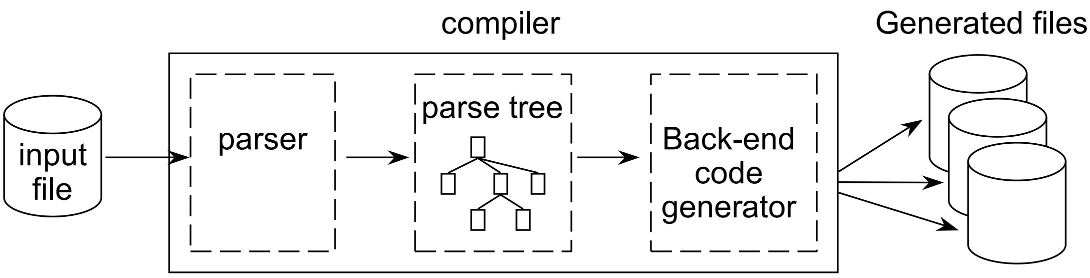
Spring (cont')

- Spring uses reflection to create an object for each `bean`
  - The object's type is specified by the `class` attribute
- By default, the object is created with its default constructor
  - You can use `constructor-arg` elements (nested inside `bean`) to use a non-default constructor
- After an object is constructed, each `property` is examined
  - Spring uses reflection to invoke `obj.setXxx(value)`
    - Where `Xxx` is the capitalized name of property `xxx`
  - Spring uses reflection to determine the type of the parameter passed to `obj.setXxx()`
  - Spring can support primitive types and common `Collection` types
  - The `ref` attribute refers to another `bean` identified by its `id`

2. Code generation and bytecode manipulation

Code generators

- Most compilers have the following architecture



- Java's reflection metadata is conceptually similar to a parse tree
- You can build a Java code generation tool as follows:
  - Do *not* write a Java parser. Instead run the Java compiler
  - Treat generated .class files as your parse tree
  - Use reflection to navigate over this "parse tree"

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 runtime code generation

- Runtime code generation is used…
- By JSP (Java Server Pages)
  - To generate servlets from .jsp files
- By IDEs and debuggers
  - To evaluate Java expressions entered by user

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

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