# An Introduction to Java Reflection

## Software Design and Programming

KLM

Department of Computer Science and Information Systems
Birkbeck, University of London

`keith@dcs.bbk.ac.uk`

Spring Term 2016

# With thanks...

Developed in conjunction with Ciaran McHale, Bruce Eckel, Cay Horstmann, and Dave Matuszek

# 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 clothes
- 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

# Java reflection - why ignored? I

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 s/he has learned so far
- Never finds the time to read the rest of the book

# Java reflection - why ignored? II

Result is “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 of those *advanced* topics that is not complex

# Is reflection difficult? I

- When learning to program:
  - First learn iterative programming with if-then-else, while-loop, . . . (perhaps)
  - 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*

# Is reflection difficult? II

- Likewise, many people find reflection difficult at first
    - It is an unusual way of programming
    - But it becomes much easier once you *get it*
    - for example — 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* I

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`

# Accessing *metadata* II

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* — what?

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 I

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

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

the latter might be returned by `Method.getReturnType()`

# Metadata for primitive types and arrays II

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);
```

# Metadata for primitive types and arrays III

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

…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
  ...
}
```

# Invoking a default constructor I

- Use `Class.newInstance()` to call the default constructor
- An example may help (yes, this can be initially confusing):

```
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(...);
}
...
```

# Invoking a default constructor II

```
Foo obj = Foo.create();
obj.op1(...);
```

# Invoking a default constructor III

- This technique is used in CORBA and OSGi:
  - 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
- Same technique is used for JEE and Spring and Guice and . . . :
  - JEE is a collection of specifications
  - There are many competing implementations
  - Use a system property to specify which implementation you are using

# Example — A plug-in architecture

- Use a properties file to store a mapping for where *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();
  }
}
```

# Invoking a non-default constructor I

Slightly more complex than invoking the default constructor:

- Use `Class.getConstructor(Class[] parameterTypes)`
- Then call `Constructor.newInstance(Object[] parameters)`

# Invoking a non-default constructor II

```
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
- Reminder: 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)`

```
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

```
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[] pTypes) {
  Method method = null;
  while (cls != null) {
    try {
      method = cls.getDeclaredMethod(methodName, pTypes);
      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

```
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

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

In reality perhaps `getField` should have been called `getPublicField`

# 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 I

Both `Field` and `Method` define the following methods (which are 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” (such is life!)

# Accessing non-public fields and methods II

Example of use:

```
if (!Modifier.isPublic(field.getModifiers()) {
  field.setAccessible(true);
}

Object obj = field.get(obj);
```

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

# Further reading

- There are very few books that discuss Java reflection in detail but one, which is pretty good, is

  *Java Reflection in Action by Ira R. Forman and Nate Forman*

  which has the advantage of being concise and easy to understand,
- the chapter on reflection in Bruce Eckel's book, *Thinking in Java*, and
- the `Javadoc` documentation

# Summary

- We have 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:
  - It's "meta" nature can cause confusion at first
  - It is simple to use once you know how
- We will see reflection again when we consider *dynamic proxies*

The End