Techniques of Java Programming: Reflection

Manuel Oriol

May 15, 2006

1 Introduction

In Object-Oriented Programming, reflection is the mechanism for discovering data about the program as it runs and use that data to adapt the behavior accordingly. Introspection allows to discover data. Full reflection is allowing richer features: like reflexive calls and potentially dynamic code modification.

In Java, the possibilites for reflection are limited to introspection and reflexive invocation.

2 Introspection and Reflexive Invocation

Introspection is mostly achieved in Java by using the classes `Class` and `Field` while reflexive invocation is usually achieved using `Method` and `Field`. In the following subsections we show the possibilities offered by each of the reflexive classes.

2.1 Class `Object`

`Object` is only used to get the `Class` instance of the object through the method `getClass()` (see Table 1).

Table 1: Method from Object participating to reflection

java.lang.Object	
Class<? extends Object> getClass()	Returns the class of the Object.

2.2 Class `Class<?>`

The class `Class<?>` is used to get fields and methods of a class. It can also be used as a way to get the information to make invocations, assign fields... Note that to obtain the `Class` object of a class C it is needed to write `C.class`. Note that arrays and primitive types each have their own `Class` objects. It is also interesting to see that interfaces as well as local classes have their own instances of the class `Class`, tests being available to check that out.

As another interesting point, the class `Class` is serializable, which can allow code mobility and code storing.

Table 2: Method on classes

java.lang.Class<T>	
T cast(Object obj)	Casts an object.
static Class<?> forName(String className)	Returns the class corresponding to the name, with initialization.
static Class<?> forName(String name, boolean initialize, ClassLoader loader)	Loads a class with a given name in a given class loader and may initialize it.
Class[] getClasses()	Get all classes/interfacces implemented.
ClassLoader getClassLoader()	Return the loader of this class.
Constructor<T> getConstructor(Class... parameterTypes)	Returns the constructor adapted to arguments types.
Constructor[] getConstructors()	Returns an array of constructors.
Field getField(String name)	Returns the field with name name.
Field[] getFields()	Returns an array of available fields.
Method getMethod(String name, Class... parameterTypes)	Returns a Method with given name and types.
Method[] getMethods()	Returns all methods
int getModifiers()	Returns the modifier encoded in an int.
String getName()	Returns the class name.
boolean isInstance(Object obj)	Checks if the object is an instance.

2.3 Class `Field`

As shown in Table 3, `Field`s can be interrogated and assigned. As an example:

```
o.f=3;
```

is equivalent to:

```
(o.getClass().getField("f")).setInt(o,3);
```

2.4 Class `Constructor`

The constructor class allows for initiating instances. It is, for example, equivalent to write:

```
o=new O(3);
```

and:

```
(O.class).getConstructor(int.class).newInstance(3);
```

2.5 Class `Method`

The `Method` class provides facilities to interogate and invoke code in a reflexive manner. As an example, it is equivalent to write:

Table 3: Method from Field

java.lang.reflect.Field	
Object get(Object obj)	Gets this field value in obj.
Note: An equivalent method exists also for each primitive type.	
Class<?> getDeclaringClass()	Returns declaring class.
int getModifiers()	Returns the modifiers encoded in an int.
String getName()	Returns the name of the field.
Class<?> getType()	Returns the type of the field.
void set(Object obj, Object value)	Sets the field's value.
Note: An equivalent method exists also for each primitive type.	

Table 4: Method from Constructor

java.lang.reflect.Constructor<T>	
Class<?> getDeclaringClass()	Returns declaring class.
int getModifiers()	Returns the modifiers encoded in an int.
String getName()	Returns the name of the field.
TypeVariable<Method> getTypeParameters()	Returns the types of the paramters.
boolean isVarArgs()	Checks if the constructor has a variabl number of arguments.
T newInstance(Object... initargs)	Creates a new instance

Table 5: Method from Method

java.lang.reflect.Method	
Class<?> getDeclaringClass()	Returns declaring class.
int getModifiers()	Returns the modifiers encoded in an int.
String getName()	Returns the name of the field.
TypeVariable<Method> getTypeParameters()	Returns the types of the paramters.
boolean isVarArgs()	Checks if the constructor has a variabl number of arguments.
Object invoke(Object obj, Object... args)	Invokes the method on obj with args.

```
t.test();
```

and:

```
t.getClass().getMethod("test").invoke(t);
```

3 Proxy Classes and Instances

It is possible to set up `Proxy` classes as well as proxy instances. A dynamic proxy class is a class that is created dynamically to encapsulate calls made on a set of interfaces as sown in Table 6. The way it works is that it redirects all calls (except for public, non-final methods of `Object`) on the class to a handler.

Table 6: Proxy class and InvocationHandler interface

`java.lang.reflect.`*InvocationHandler*	
Object invoke(Object proxy, Method method, Object[] args)	Mehtod called in place of the regular method.
java.lang.reflect.Proxy	
protected Proxy(InvocationHandler h)	Constructor.
static InvocationHandler getInvocationHandler(Object proxy)	Gets the handler associated with proxy
static Class<?> getProxyClass(ClassLoader loader, Class<?>... interfaces)	Creates a Proxy class in a given loader for implementing the specified interfaces.
static boolean isProxyClass(Class<?> cl)	Returns true if the class is a proxy class.
static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)	Creates a proxy instance

As shown in the Java APIs[1], it is easy to create a Proxy instance:

```
Foo f = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader(),
                                     new Class[] { Foo.class },
                                     handler);
```

Accordingly to what intuition suggests, `proxy instanceof Foo` returns `true` and `(Foo) proxy` does not trigger any `ClassCastException`

4 Performance Considerations

A simple benchmark program gives very interesting results:

```
import java.lang.reflect.*;
public class Test{
   public static final int MAX=1000000;
   int c=0;
   public void test(){
      c++;
   }
   public static void main(String[] args) throws Exception{
      Test t=new Test();
```

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Proxy.html

```
        long start,startReflex;
        long end,endReflex;

        t.test();
        start=System.nanoTime();
        for(int i=0;i<MAX;i++)
           t.test();
        end=System.nanoTime();

        t=new Test();
        Method m=t.getClass().getMethod("test");
        m.invoke(t);
        startReflex=System.nanoTime();
        for(int i=0;i<MAX;i++)
           m.invoke(t);
        endReflex=System.nanoTime();

        System.out.println("Mean time (ns) for regular call: "
           +(end-start)/MAX);
        System.out.println("Mean time (ns) for reflexive call:"
           +(endReflex-startReflex)/MAX);
        System.out.println("-------\nSlowdown factor:"
         +(endReflex-startReflex)/(end-start)+"x");
}
}
```

Gives the following results:

```
Mean time (ns) for regular call: 7
Mean time (ns) for reflexive call: 262
-------
Slowdown factor: 33x
```

It then appears that reflexivity is to be used in Java only when really needed... This point of view is generally shared by industrials that need reasonable performance.