Introduction to Reflection

Mark Allen Weiss
Copyright 2000

Outline of Topics

- What is Reflection
- The Class class
- Run Time Type Identification (RTTI)
- Getting Class Information
- Accessing an arbitrary object’s fields
- Advanced features

Reflection

- Introduced in Java 1.1
- Allows you to find out information about any object, including its methods and fields, even if the type of the object is not known at compile time
- Added to the language to support Beans, Serialization, RMI, and other goodies.
- Reflection is an enabling technology.
### The Class Object

- **Class** objects represent a loaded class
- Can find out information about the class
  - its methods
  - its fields
  - its superclass
  - the interfaces it implements
  - whether it is an array

### Obtaining a Class Object

- If you know a class name, can get it:
  ```java
  Class c1 = String.class;
  Class c2 = Employee[].class;
  ```
- Can get it from any object, using `getClass`:
  ```java
  void printType(Object obj)
  {
      Class c3 = obj.getClass();
      System.out.println(c3.toString());
  }
  ```
- Can get it by loading the class using the `forName` static method:
  ```java
  Class c = Class.forName("java.util.Date");
  ```

### What’s In Class?

```java
public class Class
{
    public String getName();
    public boolean isInterface();
    public boolean isArray();
    public Class getSuperclass();
    public Class[] getInterfaces();
    public Class[] getClasses(); // inner classes
    public Object newInstance();
    public static Class forName(String name);
    public Method[] getDeclaredMethods();
    public Method[] getMethods();
}
```
Reflection Classes

- Found in java.lang.reflect
- Method: Allows you to get info about an arbitrary method, and even invoke one
- Field: Allows you to get the name and access an arbitrary field
- Constructor: Allows you to get info about an arbitrary constructor, and even invoke one
- Array: Contains static methods to create and access arbitrary arrays

Example: Array Expansion

- Want to write automatic array doubling code.
- Here is typical idea, but it does not work
  ```java
  public Object[] doubleArray(Object[] arr) {
      int newSize = arr.length * 2 + 1;
      Object[] newArray = new Object[newSize];
      for (int i = 0; i < arr.length; i++)
          newArray[i] = arr[i];
      return newArray;
  }
  ```
- But: even if arr is Foo[], actual returned object Object[] can’t be downcast to Foo[].

Solution

```java
public Object doubleArray(Object arr) {
    Class cl = arr.getClass();
    if (!cl.isArray()) return null;
    int oldSize = Array.getLength(arr);
    int newSize = oldSize * 2 + 1;
    Object newArray = Array.newInstance( 
        cl.getComponentType(), newSize);
    System.arraycopy(arr, 0, newArray, oldSize);
    return newArray;
}
```

- Notes: array can be int[]; arraycopy is faster than a loop (fewer bounds checks)
The Array Class

public class Array {

    // All of these are static
    public int getLength( Object arr );
    public Object newInstance( Class comp, int length );

    public Object get( Object arr, int index );
    public void set( Object arr, int index, Object val );

    // Various specialized versions:
    public int getInt( Object arr, int index );
    public void setInt( Object arr, int index, int val );
}

Accessing a Class’ Members

- From Class object, you can get Method objects that reflect all methods, Field objects that reflect all fields, and Constructor objects that reflect all constructors.

- Two versions (use Field as example)
  - getField gets a public field given name
  - getDeclaredField gets a field declared in this class (but not superclass); could be private
  - getFields gets an array of public fields
  - getDeclaredFields gets an array of fields declared in this class (but not superclass); could be private

Example: List Visible Class Functions

public void printClassMethods( String name )
{
    try {
        Class cl = Class.forName( name );
        Constructor c = cl.getConstructors();
        for( int i = 0; i < c.length; i++ )
            System.out.println( c.toString( ) );
        Method m [] = cl.getMethods();
        for( int i = 0; i < m.length; i++ )
            System.out.println( m.toString( ) );
    } catch( ClassNotFoundException e ) {
        System.out.println( name + " not found" );
    }
}
Using a Method Object

- From Method object
  - Can find out everything about method signature
  - Invoke a method with normal dynamic binding.
  - You can obtain a Method from a signature, or get a list of all methods.
- To specify the signature, give an array of Class objects that represent the types of the parameters.
  - Array will be zero-length if no parameters
  - Special Class objects for primitives

What’s In Method Class

- Various accessors to get info. Also invoke.
  
  ```java
  public class Method {
    public Class getReturnType();
    public Class[] getParameterTypes();
    public String getName();
    public int getModifiers();
    public Class[] getExceptionTypes();
    public Object invoke(Object obj, Object[] args);
  }
  ```

  - The modifiers are stored as a bit pattern; class Modifier has methods to interpret the bits.

Some Details

- Parameters and return types are Objects. If the actual types are primitives, they will be wrapped using one of the eight wrapper classes.
- The first parameter to invoke is the controlling object (good idea to use null for static methods, but not required). The second parameter is the parameter list.
- When you use invoke beware:
  - It is much much slower than static invocation
  - You have to handle all the exceptions
  - You lose lots of compile-time checks
Exceptions

● If invoked method throws an exception, invoke will throw an InvocationTargetException
● Can get original via getException
● Lots of other exceptions to worry about before you call invoke:
  – Did class load? ClassNotFoundException
  – Was method found? NoSuchMethodException
  – Can you access method? IllegalAccessException

Representing the Primitive Types

● Special Class objects for the primitives:
  – Integer.TYPE is the Class object for int
  – There is a type for each of the eight primitives
  – Void.TYPE is the Class object for void
● Not the same as
  – Integer.class which is the Class object for Integer wrapper
● Also Class types for arrays
  – for example, class type for int[] [] is Integer.TYPE[][].class

Steps To Invoke A Method

● Get a Class object for the class that contains the method
● Get a Method object, m. Will need name of method, and an array of Class objects.
● Form an array of Object that contains the parameters to pass (second argument to m.invoke). Pass the controlling object or null (if static method) as the first parameter.
● Catch InvocationTargetException
Example: Run any main

```java
// Assumes import statements present
// Run the main for any class className
// This is the main logic; exception handling is on next slide
public static void invokeMain( String className, Object[] params )
{
    try {
        Class cl = Class.forName( className );
        Class[] mainParamTypes = new Class[] { String.class };
        Method mainMethod = cl.getMethod( "main", mainParamTypes );
        System.out.println( "Oops... main is not static!" );
        else if( Modifier.isStatic( mainMethod.getModifiers() ) )
            System.out.println( "Oops... main doesn't return void!" );
            mainMethod.invoke( null, params );
    }
```

Example: Run any main (exceptions)

```java
catch( ClassNotFoundException e ) {
    System.out.println( "Cannot find " + className );
}
catch( NoSuchMethodException e ) {
    System.out.println( "Cannot find main in " + className );
}
catch( IllegalArgumentException e ) {
    System.out.println( "Cannot invoke main in " + className );
}
catch( InvocationTargetException e ) {
    System.out.println( "main threw an exception" );
    e.printStackTrace();
}
```

The Field Class

- Can get list of all fields from a Class object.
- Once you have a Field class representation of an object, you can get or set its value.
- For instance (assume Date has month field, as a string):
  ```java
  Object d = new Date( "July 1, 1993" );
  Field f = d.getClass().getField( "month" );
  System.out.println( f.get( d ) );
  ```
- Security check is performed: if field is inaccessible, an IllegalArgumentException is thrown. And fields should be private!!
get and set For Field

- get and set return value in an Object.
- Primitives are wrapped.
- Special versions for convenience (e.g. `getInt`, `getDouble`, `setInt`, etc.)

Java 1.2: Accessible Objects

- Can request that Field, Method, and Constructor objects be “accessible.”
- Request granted if no security manager, or if the existing security manager allows it.
- Can invoke method or access field, even if inaccessible via privacy rules.
- Blatant security hole, means now you need to know what a security manager is. Stay tuned....

Example Of Accessing Private Data

```java
import java.lang.reflect.*;

class Hidden
{
    private static int SECRET = 3737;
}

class Spy
{
    public static int getHiddenSecret()
    {
        try
        {
            Field f = Hidden.class.getDeclaredField("SECRET");
            f.setAccessible(true); // Make private field accessible
            int x = f.getInt(null);
            return x;
        }
        catch (NoSuchFieldException e) { }
        catch (IllegalAccessException e) { }
        catch (java.security.AccessControlException e) { System.out.println("Security manager objects to this!"); return -1; }
    }
}
```
Added In Java 1.3
- Dynamic Proxy Classes
- Automates the creation of proxies
- We will discuss a use of the proxy pattern in more detail later in the course when we discuss Java 1.2 garbage collection

The Problem
- Suppose you have an interface and an implementation

```java
public interface Foo
{
    void meth1();
    int meth2();
    ...
}
class FooImpl implements Foo
{
    ...
}
- You want to have a new class that does everything each Foo method in FooImpl does, with a little before or after the call

You Need a Proxy Class
- Easy to write: Proxy class stores a reference to the Foo. For instance to print Hello,

```java
class FooProxy implements Foo
{
    public FooProxy( Foo d )
    { delegate = d; }

    public void meth1()
    { System.out.println( "Hello" ); delegate.meth1(); }
    public int meth2()
    { System.out.println( "Hello" ); return delegate.meth2(); }
    ...
    Foo delegate;
}
Proxy Pattern

● With the proxy pattern, FooImpl and FooProxy are not usually constructed directly by the user. Instead, they are handed out by a FooFactory class and only Foo is visible:

```java
public class FooFactory {
    public static Foo allocateFoo() {
        return new FooProxy(new FooImpl());
    }
    private FooFactory() {
    } // No FooFactory objects
}
```

● With this pattern, user is oblivious to the fact that they have a proxy!
● Easy to change implementation of the concrete Foo instances

Dynamic Proxies

● Proxies useful to
  – do security checks prior to each call
  – do logging that calls are being made and completed
  – do lazy loading or copying
  – represent remote objects

● If interfaces are large, the code to write new proxies is cumbersome and repeated.
● Reflection can do this for you automatically.
● Downside is that reflection might be too slow; depends on what the proxy is doing.

Code Is Straightforward

● Uses two classes:
  – InvocationHandler interface; must implement its invoke method to do delegation
  – Proxy; usually call its newProxyInstance method with parameters that explain the class loader to use, interface being implemented, and a ref to an invocation handler object.
  – Proxy pattern is important; you should understand the pattern; automatic generation is not so important now
Generation of Foo Proxy Class

public class FooProxy {
    public static Foo allocateFoo() {
        return (Foo) Proxy.newProxyInstance( Foo.class.getClassLoader(),
                                                new Class[] { Foo.class, new FooHandler(new FooImpl) } );
    }
    private FooFactory() {} // No FooFactory objects

    class FooHandler implements InvocationHandler {
        public FooHandler( Object d ) {
            delegate = d;
        }
        public Object invoke( Object proxy, Method meth, Object[] args )
            throws Throwable {
            System.out.println( "Hello" );
            return meth.invoke( delegate, args );
        }
        private Object delegate;
    }
}

Dynamic Proxy Details

● Can have several interfaces implemented; order of interfaces matters if interfaces declare common methods
● Generated Proxy classes
  - public, final, not abstract
  - extend java.lang.reflect.Proxy
  - implement the specified interfaces
  - constructor populates Proxy base class public reference to invocation handler by calling super
● newProxyInstance actually calls getProxyClass to get a Class object, and then newInstance on the Class object.

What The New Proxy Class Is

public final class GeneratedProxy extends Proxy implements Foo {
    public GeneratedProxy( InvocationHandler h ) {
        super( h );
    }
    public int meth2() {
        Object ret = null;
        try {
            Method m = myClass.getMethod( "meth2", new Class[] {} );
            ret = handler.invoke( this, m, new Object[] {} );
            if( m instanceof RuntimeException ) throw (RuntimeException) m;
            if( m instanceof Error ) throw (Error) m;
            return ((Integer) ret).intValue();
        }
        finally {
            handler = super.handler;
        }
    }
    private InvocationHandler handler;
    private static final Class myClass = Foo.class;
}
Summary

- Reflection lets you do some cool stuff and is relatively easy to use.
- Allows RTTI, which is occasionally useful to you, and crucial for other Java stuff.