PROG-3

SUBTOPIC 7

REFLECTION

Pedro J. Ponce de León

English version by Juan Antonio Pérez

Version 20111113

Contents

1. What is reflection
2. Java reflection API
3. Class objects
4. The Array class
5. Member interface
6. Method objects
7. Field objects
8. Myths of reflection
9. Capabilities not available using reflection
10. Common problems solved using reflection
    - Finding an inherited method
    - Factory with reflection
11. Bibliography

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 to see and manipulate itself at runtime
  - It consists of metadata plus operations to manipulate the metadata
- Meta means self-referential
  - So metadata is data (information) about oneself
  - Metadata is data providing information about other data (e.g., class Class in Java)

What is reflection

- Reflection can be used to...
  - construct new class instances and new arrays
  - access and modify fields of objects and classes
  - invoke methods on objects and classes
  - access and modify elements of arrays
  - etc...
- Class names do not need to be known at compile-time when using reflection. All the information needed may be provided at run time (for instance, in a string)

What is reflection

- Example
  - The string “Barco” is read from an input file, implying that an object of that class has to be created.
  - By using reflection, the program will proceed as follows:
    - ¿Is a class called “Barco” accessible?
    - If true, then it will be loaded into memory:

```
Class c = Class.forName(“Barco”)
```

    - And the constructor of the class will be invoked to create an object of class Barco:

```
Object obj = c.newInstance();
```

Java reflection API

- Java provides two different ways to obtain information about types at run-time:
  - Traditional RTTI (upcasting, downcasting)
    - When the object type is available at compile-time and run-time
  - Reflection
    - When the object type may not be available at compile-time and/or run-time.

Java reflection API

- Java reflection API
  - java.lang.Class
  - java.lang.reflect.*
- With this API our program may access classes, interfaces and objects in the JVM at runtime.
- Introduced in JDK 1.1 to support the JavaBeans specification (reusable software components that can be visually manipulated in builder tools).

Java reflection API

Core reflection classes

`java.lang.reflect`

- The reflection package
- Introduced in JDK 1.1

java.lang.reflect.Array

- Provides static methods to dynamically create and access Java arrays

`java.lang.reflect.Member`

- Interface that reflects identifying information about a single member (a field or a method) or a constructor

Java reflection API

Core reflection classes (cont'd)

java.lang.reflect.Constructor

- Provides information about, and access to, a single constructor for a class

java.lang.reflect.Field

- Provides information about, and dynamic access to, a single field of a class or an interface
- The reflected field may be a class (static) field or an instance field

java.lang.reflect.Method

- Provides information about, and access to, a single method on a class or interface.

Java reflection API

- **Commonly used classes**

`java.lang.Class`

- Represents classes and interfaces

java.lang.Package

- Provides information about a package that can be used to reflect upon a class or interface

java.lang.ClassLoader

- An abstract class
- Provides class loader (dynamic loading) services

Class objects

- Every class loaded into the JVM has an object of class Class associated with it.
  - It corresponds to a .class file
  - *The class loader* (java.lang.ClassLoader) is the responsible for finding and loaded classes into the JVM.
- **Dynamic class loading**: when instantiating an object...
  - The JVM checks whether the class has already been loaded
  - If necessary, the class is located and loaded.
  - After being loaded, the class is used for instantiating the new object.

Class objects

- Class objects represent a loaded class
- Such an object holds information about a class:
  - its methods
  - its fields
  - its superclass
  - the interfaces it implements
  - whether it's an array

Class objects

- **`Class.forName(String)`**
  - Returns a Class object corresponding to a class name (static method)

```
try
{
    // searches and loads class B
    // in case it is not already loaded
   Class c = Class.forName ("B");
    // c will point to a Class object
    // representing class B
}
catch (ClassNotFoundException e)
{ // B does not exist in the CLASSPATH
   e.printStackTrace ();
}
```

Class objects

- In the previous subtopic we studied additional ways to obtain the Class object and accessing its information:

Class literals (compile time)

All classes, interfaces, arrays, and primitive types have class literals:

```
Circulo.class
Integer.class
int.class
```

instanceOf

```
if (x instanceof Circulo)
   ((Circulo) x).setRadio(10);
```

Class objects

- **Class methods**

```
public String getName( );
```

Returns the name of the class referred to by the Class object.

```
public boolean isInterface( );
```

Returns true if the Class object refers to an interface.

```
public boolean isArray( );
```

Returns true if the Class object refers to an array type.

```
public Class getSuperclass( );
```

Returns the superclass of the current Class object.

Class objects

- **Class methods**

```
public Class[] getInterfaces( );
```

Returns array of interface classes implemented by this class.

```
public Object newInstance( );
```

Creates and returns an instance of this class.

```
public static Class forName( String name );
```

Returns a Class object corresponding to a class name (static method)

Class objects

- **Class methods**

```
public Constructor[] getConstructors( );
```

Returns an array of all public constructors in the current class.

(import java.lang.reflect.Constructor)

```
public Method[] getDeclaredMethods( );
```

Returns an array of all public and private methods declared in the current class or interface.

(import java.lang.reflect.Method)

```
public Method[] getMethods( );
```

Returns an array of all public methods in the current class, as well as those in all superclasses and superinterfaces.

Class objects

- **Class methods**

```
public Method getMethod( String methodName,
                         Class[] paramTypes );
```

Returns a Method object that reflects the method identified by name and parameter types in the current class and all superclasses. Method must be public.

```
public Method getDeclaredMethod( String
methodName,
Class[] paramTypes );
```

Returns a Method object that reflects the method identified by name and parameter types in the current class. Method may be private.

The Array class

- **The Array class**

```
public class Array {
    // all static methods:
    public int getLength( Object arr );
    public Object newInstance( Class elements, int
    length );
    public Object get( Object arr, int index );
    public void set( Object arr, int index, Object val );
    // Various specialized versions, such as...
    public int getInt( Object arr, int index );
    public void setInt( Object arr, int index, int val );
}
```

The Array class

- **Array samples**

```
Canine[] kennel = new Canine[10];
.
int n = Array.getLength( kennel );

// set the contents of an array element
Array.set( kennel, (n-1), new Canine( "Spaniel" ) );

// get an object from the array, determine its class,
// and display its value:
Object obj = Array.get( kennel, (n-1) );
Class c1 = obj.getClass( );
System.out.println( c1.getName( )
    + "-->"
    + obj.toString( ) );
```

The Array class

- **Two ways to declare an array**

```
// first:
Canine kennel = new Canine[10];


// second:
Class c1 = Class.forName( "Canine" );
Canine kennel = (Canine[]) Array.newInstance( c1,
10 );
```

The Array class

- **Example: expanding an array**

Problem statement: write a function that receives an arbitrary array, allocates storage for twice the size of the array, copies the data to the new array, and returns the new array

The Array class

- **Example: expanding an array**

Why won't this code work?

```
public static Object[] doubleArrayBad( 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;
}
```

The Array class

- **Example: expanding an array**

Answer: This method always returns an array of Object, rather than the type of the array being copied.

```
public static Object[] doubleArrayBad( 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;
}
```

The Array class

- **Example: expanding an array**

Use reflection to get the array type:

```
public Object[] doubleArray( Object[] arr )
{
    Class c1 = arr.getClass( );
    if( !c1.isArray( ) ) return null;

    int oldSize = Array.getLength( arr );
    int newSize = oldSize * 2 + 1;

    Object[] newArray = (Object[]) Array.newInstance(
        c1.getComponentType( ), newSize );

    for( int i = 0; i < arr.length; i++ )
        newArray[ i ] = arr[ i ];

    return newArray;
}
```

Member interface

Implemented by Constructor, Method, and Field

Class getDeclaringClass( )

returns the Class object representing the class or interface that declares the member or constructor represented by this Member.

int getModifiers( )

returns the Java language modifiers for the member or constructor represented by this Member, as an integer.

String getName( )

returns the simple name of the underlying member or constructor represented by this Member.

Method objects

- Using a Method object, you can...
    - get its name and parameter list and
    - invoke the method

- Obtain a Method from a signature, or get a list of all methods.

- To specify the signature, create an array of Class objects representing the method’s parameter types.
    - Array will be zero-length if no parameters

Method objects

```
public class Method implements Member
{
    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.

Method objects

- Retrieve the name of a method:

```
(Method meth;)
String name = meth.getName( );
```

- Retrieve an array of parameter types:

```
Class parms[] = meth.getParameterTypes( );
```

- Retrieve a method's return type:

```
Class retType = meth.getReturnType( );
```

Method objects

- **`Method.invoke()`**

```
public Object invoke(Object obj, Object[] args)
```

- If the parameters or return types are primitives, they are wrapped using one of the eight wrapper classes.
  - example: Integer.class, Integer.TYPE
- The first parameter to invoke is the controlling object
  - (use null for static methods)
- The second parameter is the parameter list
  - array of objects
- Disadvantages to using invoke( ):
  - executes more slowly than static invocation
  - you have to handle all checked exceptions
  - you lose lots of compile-time checks

Method objects

- **`Method.invoke() and exceptions`**
- If invoked method throws an exception, invoke( ) will throw an InvocationTargetException
  - get the actual exception by calling 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

Method objects

- **Steps to invoke a method**

- Get a Class object, c.
- Get a Method object m, from c:
  - Form an array of parameter types that match the method you want to invoke
  - Call getDeclaredMethod( ), passing it the name of the method and the array of parameter types. Returns m.
- Form an array of Object that contains the arguments to pass (second argument to m.invoke).
  - new String[ ] { "Breathing", "Fire" }
- Pass the controlling object (or null if calling a static method) as the first parameter.
- Call m.invoke( ), and catch
  InvocationTargetException

Method objects

Example: invoking main( )

Calling: main(String[] args)

Simplified, with no error checking:

```
Class cl = Class.forName( className );
Class[] paramTypes = new Class[] { String[].class };
Method m = cl.getDeclaredMethod( "main", paramTypes );
Object[] args = new Object[]
       { new String[] { "Breathing", "Fire" } }
m.invoke( null, args );
```

Method objects

Example: invoking a constructor

Call getConstructor( ), then call newInstance( ),
catch InstantiationException

```
Class c1 = Class.forName("Circulo");
Class[] paramTypes = new Class[] {Coordenada.class,
                     Float.class };
Constructor m = c1.getConstructor( paramTypes );

Object[] arguments = new Object[] {
   new Coordenada(10,20),
   new Float(20) };

Figura2D c = (Figura2D) m.newInstance(arguments);
```

Field objects

Getting Field objects from a Class

```
public Field getField( String name )
       throws NoSuchFieldException, SecurityException
```

Returns a public Field object.

```
public Field[] getFields()
       throws SecurityException
```

Returns an array containing public fields of current class, interface, superclasses, and superinterfaces.

```
public Field[] getDeclaredFields()
       throws SecurityException
```

Returns an array containing all fields of current class and interfaces.

Field objects

- Things you can do with a Field object:
  - Get the field's name - String getName( )
  - Get the field's type – getType( )
  - Get or set a field's value – get( ), set( )
  - Check for equality – equals( )
  - Get its declaring class – Class getDeclaringClass( )
  - Get its modifiers - getModifiers( )

Field objects

Important Field methods:

- Implements Member interface: getName( ), getModifiers( ), and getDeclaringClass( )
- Class getType( )
  - returns a Class object that identifies the declared type for the field represented by this Field object.
- Object get(Object obj)
  - Returns the value of the field represented by this Field, on the specified object.
- void set(Object obj, Object value)
  - sets the field represented by this Field object on the specified object argument to the specified new value.

(When referencing a static field, the obj argument is null)

Field objects

Important Field methods:

- Specific "get" methods:
  - boolean getBoolean(Object obj) gets the value of a static or instance boolean field.
  - Also: getByte, getChar, getDouble, getFloat, getInt, getLong, and getShort
- Specific "set" methods:
  - void setBoolean(Object obj, boolean z) sets the value of a field as a boolean on the specified object.
  - Also: setByte, setChar, setDouble, setFloat, setInt, setLong, and setShort

Field objects

Get and Set for Field:

- For instance:

```
Object d = new Hero( );
Field f = d.getClass( ).getField( "strength" );
System.out.println( f.get( d ) );
```

- Possible exceptions:
  - NoSuchFieldException, IllegalAccessException

Four Myths of Reflection

- “Reflection is only useful for JavaBeans technology-based components”
- “Reflection is too complex for use in general purpose applications”
- “Reflection reduces performance of applications”
- “Reflection cannot be used with the 100% *Pure Java certification standard*”

“Reflection Is Only Useful for JavaBeans Technology-based Components”

- False
- Reflection is a common technique used in other pure object oriented languages like Smalltalk and Eiffel
- Benefits
  - Reflection helps keep software robust
  - Can help applications become more
    - Flexible
    - Extensible
    - Pluggable

“Reflection Is Too Complex for Use in General Applications”

- False
- For most purposes, use of reflection requires mastery of only several method invocations
- The skills required are easily mastered
- Reflection can significantly…
  - Reduce the footprint of an application
  - Improve reusability

“Reflection Reduces the Performance of Applications”

- False
- Reflection can actually increase the performance of code
- Benefits
  - Can reduce and remove expensive conditional code
  - Can simplify source code and design
  - Can greatly expand the capabilities of the application

“Reflection Cannot Be Used With the 100% Pure Java Certification Standard”

- False
- There are some restrictions
  - “The program must limit invocations to classes that are part of the program or part of the JRE” (Sun Microsystems, 100% Pure Java Certification Guide, version 3.1, May 2000).

Capabilities Not Available Using Reflection

- What are a class’ subclasses?
  - Not possible due to dynamic class loading in the JVM
- What method is currently executing?
  - Not the purpose of reflection
  - Other APIs provide this capability

Common problems solved using reflection

- Finding an inherited method
- Factory pattern with reflection: misuse of switch/case statements

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;
}
```

Example: findMethod(Figura2D.class, “equals”, new Class[] {})

Factory method without reflection

```
public static Shape getFactoryShape (String s)
{
   Shape temp = null;
   if (s.equals (“Circle”))
      temp = new Circle ();
   else
      if (s.equals (“Square”))
         temp = new Square ();
      else
         if (s.equals (“Triangle”)
            temp = new Triangle ();
         else
           // …
           // continues for each kind of shape
   return temp;
}
```

Factory method with reflection

```
public static Shape getFactoryShape (String s)
{
   Shape temp = null;
   try
   {
      temp = (Shape) Class.forName (s).newInstance ();
   }
   catch (Exception e)
   {
   }
   return temp;
}
```

Bibliography

- **Introduction to Object Oriented Programming, 3rd Ed,** Timothy A. Budd

  Ch. 25 : Reflection and Introspection

- **Java reflection in action**
  - Ira R. Forman and Nate Forman, 2004

  http://www.manning.com/forman/

- **Piensa en Java, 4th ed.**
  - Bruce Eckl

  Ch. 14 : Información de tipos