Java Reflection

source:

“Using Java™ Technology Reflection to Improve Design “

by Michael T. Portwood, MS - Exuberance, LLC

Agenda

- What is reflection
- History of reflection
- How to use reflection
- Myths about reflection
- Advanced reflection issues
- Improvements to reflection
- Conclusion

What Is Reflection

- Java™ Technology provides two ways to discover information about an object at runtime
  - Traditional runtime class identification
    - The object's class is available at compile and runtime
    - Most commonly used
  - Reflection
    - The object's class may not be available at compile or runtime

What Is Reflection

- “Reflection in a programming language context refers to the ability to observe and/or manipulate the inner workings of the environment programmatically.”[1]
- “The reflection API represents, or reflects, the classes, interfaces, and objects in the current Java™ virtual machine.”[2]

1. J. R. Jackson, A. L. McClellan, Java™ 1.2 By Example, Sun Microsystems, 1999.
2. M. Campione, et al, The Java™ Tutorial Continued, Addison Wesley, 1999.

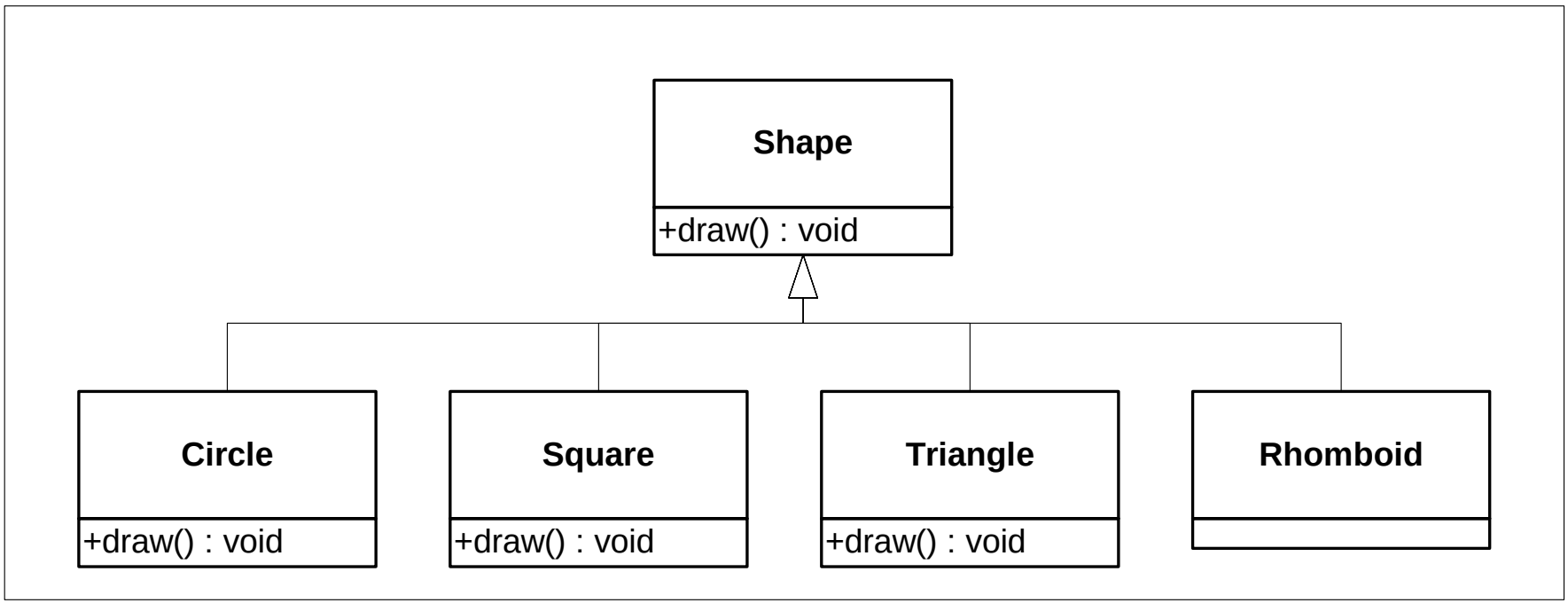
The History of Reflection

- Introduced in JDK™ 1.1 release to support the JavaBeans™ specification
- Used throughout the JDK™ software and Java runtime environment (JRE)
  - Java™ Foundation Classes API (JFC)
  - Jini™ connection technology
  - JavaMail™ API
  - JDBC™ API
- Improved in Java 1.2 SDK
- Further refined in Java 1.3 SDK

Why Runtime Class Identification

- Java™ technology takes advantage of polymorphism
  - New subclasses easily added
  - Bulk of behaviors inherited from its superclass
  - No impact on other subclasses of the superclass
  - At runtime, the JVM™ takes advantage of late dynamic binding
  - Messages are directed to the correct method

Example UML


Shape
+draw() : void
Circle
+draw() : void
Square
+draw() : void
Triangle
+draw() : void
Rhomboid

Runtime Class Identification Example Code

- Class loading occurs at first instantiation
- When the object is retrieved from the list, it is cast to the superclass, ***Shape***
- The object remembers its class and responds with the correct ***draw*** method

```
List s = new ArrayList ();
s.add (new Circle ());
s.add (new Square ());
s.add (new Triangle ());
for (Iterator e = s.iterator ();
                  e.hasNext ();)
    ((Shape) e.next ()).draw ();
```

How the Class Object Works

- Every class loaded into the JVM™ has a Class object
  - Corresponds to a .class file
  - The ClassLoader is responsible for finding and loading the class into the JVM™
- At object instantiation…
  - The JVM™ checks to see if the class is already loaded into the virtual machine
  - Locates and loads the class if necessary
  - Once loaded, the JVM™ uses the loaded class to instantiate an instance

Proof of Dynamic Loading

```
public static void main (String[] args)
{
   System.out.println("inside main");
   new A();
   System.out.println("After creating A");
   try
   {
      Class.forName("B");
   }
   catch (ClassNotFoundException e)
   {
      e.printStackTrace();
   }
   System.out.println("After forName (\"B\")");
   new C();
   System.out.println("After creating C");
}
```

Late Dynamic Binding

- The JRE does not require that all classes are loaded prior to execution
  - Different from most other environments
- Class loading occurs when the class is first referenced
- Late Dynamic Binding is…
  - Important for polymorphism
    - Message propagation is dictated at runtime
    - Messages are directed to the correct method
  - Essential for reflection to be possible

Class Literals

- A class literal is an expression consisting of
  - the name of a class, interface, array, or primitive type
  - followed by a '.'
  - And the token class (e.g. `class`, TYPE)
- Using class literals is the second way to reference an object's class
  - Added in the JDK™ 1.1 release
- Primitive types have corresponding wrapper classes
- Examples:

```
Integer.TYPE   →  int
Integer.class  →  class java.lang.Integer
int.class      →  int
```

The *instanceof* Keyword

- The ***instanceof*** keyword is the third way to reference an object's class
- Used with both classes and interfaces
- Returns true if the object is a species of a specified class
  - Subclasses will also answer true
  - Code becomes structurally bound to the class hierarchy
- Several limitations on the referenced class
  - Must be a named class or interface
  - The class constant cannot be the ***Class*** class
- Example:

```
if (x instanceof Circle)
  ((Circle) x).setRadius(10);
```

The Reflection API

- The reflection API is the fourth way to reference an object's class
- Reflection allows programs to interrogate and manipulate objects at runtime
- The reflected class may be…
  - Unknown at compile time
  - Dynamically loaded at runtime

Core Reflection Classes

- **`java.lang.reflect`**
  - The reflection package
  - Introduced in JDK 1.1 release
- **`java.lang.reflect.AccessibleObject`**
  - The superclass for ***`Field`***, ***`Method`***, and ***`Constructor`*** classes
  - Suppresses the default Java language access control checks
  - Introduced in JDK 1.2 release

Core Reflection Classes (Cont.)

- **`java.lang.reflect.Array`**
  - Provides static methods to dynamically create and access Java arrays
- **`java.lang.reflect.Constructor`**
  - Provides information about, and access to, a single constructor for a class

Core Reflection Classes (Cont.)

- **`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

Core Reflection Classes (Cont.)

- **`java.lang.reflect.Member`**
  - Interface that reflects identifying information about a single member (a field or a method) or a constructor
- **`java.lang.reflect.Method`**
  - Provides information about, and access to, a single method on a class or interface
- **`java.lang.reflect.Modifier`**
  - Provides static methods and constants to decode class and member access modifiers

Core Reflection Classes (Cont.)

- JDK 1.3 release additions
  - **`java.lang.reflect.Proxy`**
    - Provides static methods for creating dynamic proxy classes and instances
    - The superclass of all dynamic proxy classes created by those methods
  - **`java.lang.reflect.InvocationHandler`**
    - Interface
    - Interface implemented by the invocation handler of a proxy instance

Commonly Used Classes

- **`java.lang.Class`**
  - Represents classes and interfaces within a running Java™ technology-based program
- **`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 services

Using Reflection

- Reflection allows programs to interrogate an object at runtime without knowing the object's class
- How can this be…
  - Connecting to a JavaBean™ technology-based component
  - Object is not local
    - RMI or serialized object
  - Object dynamically injected

What Can I Do With Reflection

- Literally everything that you can do if you know the object's class
  - Load a class
  - Determine if it is a class or interface
  - Determine its superclass and implemented interfaces
  - Instantiate a new instance of a class
  - Determine class and instance methods
  - Invoke class and instance methods
  - Determine and possibly manipulate fields
  - Determine the modifiers for fields, methods, classes, and interfaces
  - etc.

Here Is How To…

- Load a class
  `Class c = Class.forName("Classname")`
- Determine if an interface, an array class, or primitive type
  `c.isInterface() / c.isArray() / c.isPrimitive()`
- Determine lineage
  - Super-class
    `Class c1 = c.getSuperclass()`
  - Implemented interface(s)
    `Class[] c2 = c.getInterfaces()`

Here Is How To…

- Determine constructors

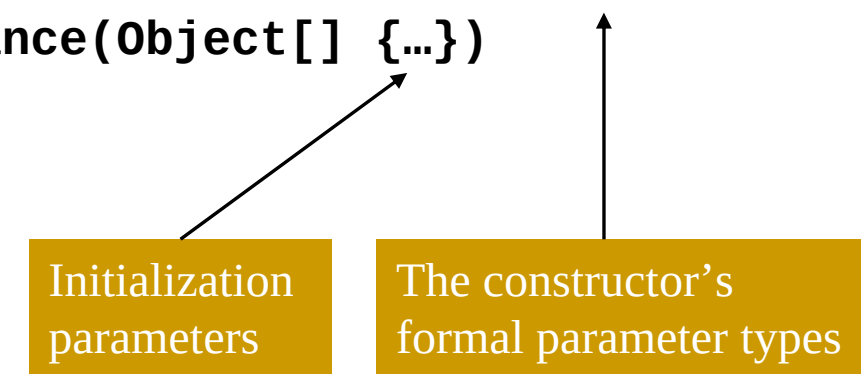
```
Constructor[] c0 = c.getDeclaredConstructors()
```

- Instantiate an instance
  - Default constructor

```
Object o1 = c.newInstance()
```

  - Non-default constructor

```
Constructor c1 = c.getConstructor(Class[]{…})
Object i = c1.newInstance(Object[] {…})
```



Here Is How To…

- Determine methods

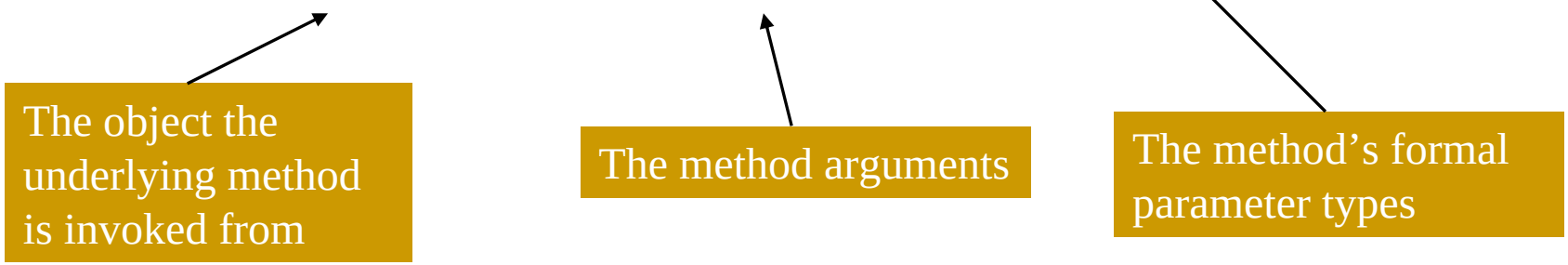
```
Methods[] m1 = c.getDeclaredMethods()
```

- Find a specific method

```
Method m = c.getMethod("methodName",
                       Class[] {…})
```

- Invoke a method

```
m.invoke (c, Object[] {…})
```



Here Is How To…

- Determine modifiers

  ```
  int mo = c.getModifiers ()
  ```

  The modifier encodings are defined in *The Java Virtual Machine Specification*

- Determine fields

  ```
  Field[] f = c.getDeclaredFields ()
  ```

- Find a specific field

  ```
  Field f = c.getField("fieldName")
  ```

- Modify a specific field
  - Get the value of a specific field on a specified object

    ```
    Object val_obj = f.get(obj)
    ```

  - Set the value of a specific fieldon a specified object

    ```
    f.set(obj, value)
    ```

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"[3]

3. Sun Microsystems, 100% Pure Java™ Certification Guide, version 3.1, May 2000.

Advanced Reflection Issues

- Why use reflection
- Using reflection with object-oriented design patterns
- Common problems solved using reflection
  - Misuse of switch/case statements
  - User interface listeners

Why Use Reflection

- Reflection solves problems within object-oriented design:
  - Flexibility
  - Extensibility
  - Pluggability
- Reflection solves problems caused by…
  - The static nature of the class hierarchy
  - The complexities of strong typing

Use Reflection With Design Patterns

- Design patterns can benefit from reflection
- Reflection can …
  - Further decouple objects
  - Simplify and reduce maintenance

Design Patterns and Reflection

- Many of the object- oriented design patterns can benefit from reflection
- Reflection extends the decoupling of objects that design patterns offer
- Can significantly simplify design patterns

- Factory
- Factory Method
- State
- Command
- Observer
- Others

Factory 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 With Reflection

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

Design Pattern Implications

- Product classes can be added, changed, or deleted without affecting the factory
  - Faster development (one factory fits all)
  - Reduced maintenance
  - Less code to develop, test, and debug

Design Strategies for Using Reflection

- Challenge switch/case and cascading if statements
  - Rationale
    - The switch statement should scream “redesign me” to the developer
    - In most cases, switch statements perform pseudo subclass operations
  - Steps
    - Redesign using an appropriate class decomposition
    - Eliminate the switch/case statement
    - Consider a design pattern approach
  - Benefits
    - High level of object decoupling
    - Reduced level of maintenance

Challenge UI Listeners

- Can a generalized listener function for several components or does each component need a unique listener?
  - Consider using the Command design pattern
  - Steps
    - Use the ***`setActionCommand`*** method to set the method to reflect upon for each component
    - Instantiate only one instant of the listener
  - Benefits
    - Smaller program memory footprint
    - Faster performance due to less class loading
    - Behavior placed in the appropriate place

Listener Without Reflection

```
addBT.addActionListener (new ActionListener ()
   {
      public void actionPerformed (ActionEvent e)
      {
         Outer.this.setTransactionState (EFrameState.add);
         Outer.this.setPromptMode ();
         Outer.this.clearForm ();
         Outer.this.enableForm (true);
         Outer.this.queryBT.setEnabled (false);
         Outer.this.deleteBT.setEnabled (false);
         Outer.this.addBT.setEnabled (true);
         Outer.this.addBT.setSelected (true);
         Outer.this.beforeTransaction ();
         … // other code excluded for clarity
      }
   });
```

Listener With Reflection

```
protected ActionListener actionAdapter = new ActionListener ()
   {
      final static Class[] emptyClass   = new Class[] {};
      final static Object[] emptyObject = new Object[] {};

      public void actionPerformed (ActionEvent e)
      {
         try
         {
            Outer.this.getClass ().getMethod (e.getActionCommand (),
                  emptyClass).invoke (Outher.this, emptyObject);
// alternatively
//          Outer.class.getMethod (e.getActionCommand (),
//                emptyClass).invoke (Outer.this, emptyObject);
         }
         catch (Exception ee)
         {
         }
      }
   };
```

Improvements to Reflection in JDK™ 1.2 Release

- Numerous small changes throughout the API
- Most changes “under the hood”
- Two classes added
  - ***AccessibleObject*** class
    - Allows trusted applications to work with private, protected, and default visibility members
  - ***ReflectPermission*** class
    - Complements the ***AccessibleObject*** class
    - Governs the access to objects and their components via reflection

Improvements to Reflection in JDK™ 1.3 Release

- Two significant additions to the API
  - ***Proxy*** Class
    - Implements a specified list of interfaces
    - Delegates invocation of the methods defined by those interfaces to a separate ***InvocationHandler*** object
  - ***InvocationHandler*** Interface
    - Defines a single invoke method that is called whenever a method is invoked on a dynamically created ***Proxy*** object

Capabilities Not Available Using Reflection

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

Review

- The JRE allows 4 ways to reference a class
  - The class' class definition
  - Class literals
  - The ***`instanceof`*** keyword
  - Reflection
- Reflection is the only pure runtime way
  - Provides full access to the object's capabilities
  - Provides runtime capabilities not otherwise available
  - Improves the quality of an application

Review

- Solves several design issues
  - Simplifies the static complexity of methods by providing elimination of…
    - Nested if/else constructs
    - The switch/case construct
  - Improves user interface code by…
    - Removing redundant inner classes
    - Reducing application footprint
    - Placing behaviors where they belong
  - Extends the power of classic object-oriented design patterns

Benefits of Reflection

- Reflection provides…
  - High level of object decoupling
  - Reduced level of maintenance
  - Programs become…
    - Flexible
    - Extensible
    - Pluggable
  - Software becomes “soft”