Reflection

Java™ Technology’s Secret Weapon

Steve Odendahl
Member of Technical Staff
Global eServices Engineering
Sun Microsystems, Inc.

Goal

Learn the What, How, When, Why and Where of the Java™ Reflection API
What Is Reflection?

Reflection is Java™ technology's crowbar—a blunt instrument for dirty jobs that can't be done any other way.

What Can Reflection Do?

- Provide runtime information on the fields and methods of a class
What Can Reflection Do?

- Provide runtime information on the fields and methods of a class
- Instantiate objects and arrays given the class name

What Can Reflection Do?

- Provide runtime information on the fields and methods of a class
- Instantiate objects and arrays given the class name
- Invoke static and instance methods given the method name
What Can Reflection Do?

- Provide runtime information on the fields and methods of a class
- Instantiate objects and arrays given the class name
- Invoke static and instance methods given the method name
- Create a class at runtime that implements one or more interfaces
The Reflection API—Class

- Entry point to the Reflection API

```
java.lang.Class

+ getName() : String
+ getFields() : Field[]
+ getField(String) : Field
+ getMethods() : Method[]
+getMethod(String, Class[]) : Field
+ getConstructors() : Constructor[]
+ getConstructor(Class[]) : Constructor
```

The Reflection API—Member

- Name (of the field or method)
- Modifiers (public, static, transient...)

```
java.lang.reflect.Member

+ getName() : String
+ getDeclaringClass() : Class
+ getModifiers() : int
```
The Reflection API — Field

- Get and set the value of the field

```java
java.lang.reflect.Field
+ get(Object) : Object
+ set(Object, Object)
+ getInt(Object) : int
+ setInt(Object, int)
...```

The Reflection API — Method

- Call the method — `invoke(target, args)`
- Static method — `invoke(null, args)`
- No arguments — `invoke(target, null)`

```java
java.lang.reflect.Method
+ invoke(Object, Object[]) : Object
+ getParameterTypes() : Class[]
+ getExceptionTypes() : Class[]
```
The Reflection API—Constructor

- Create an instance—`newInstance(args)`

```java
java.lang.reflect.Constructor
+ newInstance(Object[]) : Object
+ getParameterTypes() : Class[]
+ getExceptionTypes() : Class[]
```

It All Begins With Class

- An instance of `java.lang.Class` for every reference type (classes, interfaces, and arrays)
- An instance of `java.lang.Class` for every primitive type
- Most reflective techniques start with retrieval of a Class instance
How to...Get an Instance of Class

- Use the class name

```java
try {
    Class strcl = Class.forName("java.lang.String");
    Class strarray = Class.forName("Ljava.lang.String;" );
} catch (ClassNotFoundException cnfex) {
    // handle it
}
```

- Use a class literal

```java
Class strcl = String.class;
Class intcl = int.class;
Class strarray = String[].class;
```
How to...Get an Instance of Class

- Use the class name
- Use a class literal
- Retrieve it from an instance

```java
Class strcl = "hello".getClass();
Class strarray =
    new String[] {}.getClass();
```

How to...Get an Instance of Class

- Use the class name
- Use a class literal
- Retrieve it from an instance
- Use the constant defined in the wrapper class

```java
Class intcl = Integer.TYPE;
```
How to...Create an Object

** Use the Class object

```java
try {
    Class foocl = Class.forName("Foo");
    Foo f = (Foo) foocl.newInstance();
} catch (ClassNotFoundException cnfex) {
    // handle it
} catch (InstantiationException iex) {
    // handle it
} catch (IllegalAccessException iex) {
    // handle it
}
```

How to...Create an Object

** Use the Class object

** Use a Constructor

```java
try {
    Constructor c =
        foocl.getConstructor(null);
    Foo f = (Foo) c.newInstance(null);
} // catch blocks omitted
```
How to...Create an Array

• Use the Array.newInstance method

```
try {
    Foo[] foos = (Foo[])
    Array.newInstance (foocl, 100);
} // catch blocks omitted
```

How to...Get a Method

• Use the method name and parameters to retrieve it from the Class instance

```
Class[] paramTypes = { String.class };
Class cl = java.io.PrintWriter.class;
try {
    Method m = cl.getMethod
        ("println", paramTypes);
} // catch blocks omitted
```
**How to...Get a Field**

- Use the field name

```java
Class cl = System.class;
try {
    Field f = cl.getField("out");
} // catch blocks omitted
```

---

**How to...Get the Value of a Field**

- Use the Field instance

```java
try {
    java.io.PrintWriter out =
        (java.io.PrintWriter) f.get(null);
} // catch blocks omitted
```
How to...Invoke an Instance Method

** Use the Method object

```java
Object[] params = { "Hello, world!" };
try {
    m.invoke (out, params);
} // catch blocks omitted
```

Putting It All Together...

** A familiar program

```java
public static void main (String[] args)
    throws Exception {
    Field f = System.class.getField ("out");
    PrintStream out = (PrintStream) f.get (null);
    Class[] paramTypes = { String.class };
    Method m = PrintStream.class.getMethod
        ("println", paramTypes);
    String[] params = (String[]) Array.newInstance
        (String.class, 1);
    Array.set (params, 0, "Hello, world!");
    m.invoke (out, params);
}
```
Engineering Tradeoffs

- Readability
- Performance
- Code size

Readability

- Obviously suffers in most cases

**Direct Code**

```java
public static void main (String[] args) {
    System.out.println ("Hello, world!");
}
```

**Reflective Code**

```java
public static void main (String[] args) throws Exception {
    Field f = System.class.getField ("out");
    PrintStream out = (PrintStream) f.get (null);
    Class[] paraTypes = { String.class };
    Method m = PrintStream.class.getMethod
        ("println", paraTypes);
    String[] paraArgs = { String.valueOf (new String[0]));
    m.invoke (out, paraArgs);
}
```
Readability

- Obviously suffers in most cases

- In certain circumstances, code employing Reflection can be more readable than non-reflective code

Readability—Example

- Take various actions based on a String

String Input:
- "up"
- "down"
- "left"
- "right"

- Method call: up()
- Method call: down()
- Method call: left()
- Method call: right()
Readability—Example

- Take various actions based on a String
- A straight-forward implementation

```java
public void takeAction (String str) {
    if (str.equals("up")) {
        up ();
    } else if (str.equals("down")) {
        down ();
    } else if (str.equals("left")) {
        left ();
    } else if (str.equals("right")) {
        right ();
    }
}
```

Readability—Example

- A reflective implementation

```java
public void takeAction (String str) {
    Util.invoke (str, this, null);
}
```

- Concise, easy to extend
Performance

- Direct method invocation
  
  ```java
  for (int i = 0; i < ITERATIONS; i++) {
    incrementable.increment();
  }
  ```

Performance

- Direct method invocation
- Method lookup and invocation
  
  ```java
  for (int i = 0; i < ITERATIONS; i++) {
    Method m = cl.getMethod("increment", null);
    m.invoke(incrementable, null);
  }
  ```
Performance

- Direct method invocation
- Method lookup and invocation
- Cache method lookup and invoke

```java
Method m = cl.getMethod("increment", null);
for (int i = 0; i < ITERATIONS; i++) {
    m.invoke(incrementable, null);
}
```

Code Size

- Example—AWT and JFC/Swing API development
- Implementation of EventListener interfaces

```java
public interface ActionListener {
    public void actionPerformed (ActionEvent event);
}
```
Code Size

- Example—AWT and JFC/Swing API development
- Implementation of EventListener interfaces
- Traditionally done through inner classes

```java
button.addActionListener
   (new ActionListener () {
       public void actionPerformed
           (ActionEvent event)
           { doAction (); }
   });
```

Code Size

- Example—AWT and JFC/Swing API development
- Implementation of EventListener interfaces
- Traditionally done through inner classes
- Results in many small classes
Code Size—Solution

** Use a reflective adapter with a cached Method

```java
public class ReflectiveAdapter
    implements ActionListener {
    private Object target;
    private Method method; // cached Method

    public ReflectiveAdapter
        (Object target, String action) {
        this.target = target;
        method = getMethod (target, action);
    }
    ...}
```

Code Size—Solution

** Implement the actionPerformed method

```java
public class ReflectiveAdapter
    implements ActionListener {
    ...
    public void actionPerformed (ActionEvent evt) {
        try {
            method.invoke (target, null);
        } catch (Exception ex) {
            throw new RuntimeException (ex);
        }
    }
}
```

** Result—one class, many instances
Reflection Patterns

- Factory Method
- Interpreter
- Double Dispatch
- Interposition

When Not to Use Reflection
Reflection Considered Harmful?

- Don’t use reflection when...
  - Direct invocation does the job
    - More simply
    - More clearly
Reflection Considered Harmful?

- Don’t use reflection when...
  - Direct invocation does the job
  - Type safety is more important than flexibility
    - Errors in spelling class or method names caught at runtime, not compile-time
Summary

- Apply the Reflection API to problems direct code can’t solve
- Know the tradeoffs of using Reflection
- Employ the common patterns of Reflection use