Reflection

Java™ Technology’s Secret Weapon

Steve Odendahl
Member of Technical Staff
Global eServices Engineering
Sun Microsystems, Inc.
Learn the What, How, When, Why and Where of the Java™ Reflection API
Learning Objectives

When we are done, you will be able to:

- Use the Reflection API in your own code
- Understand the engineering tradeoffs in using Reflection
- Apply some common Reflection Patterns
- Avoid inappropriate use of Reflection
Speaker’s Qualifications

Steve Odendahl has been employed as a Java™ technology applications programmer since the exciting early days of the JDK™ 1.02 release.

As a member of the Global eServices Engineering team in Enterprise Services, developing Java technology-based solutions for service and support, he takes an interest in writing effective, efficient, and maintainable Java code.
What Is Reflection?

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

- Introduction to the Reflection API
- Engineering tradeoffs
- Reflection patterns
- Real World Reflection
- When not to use Reflection
Introduction to the Reflection API
Introduction to the Reflection API

´ What can Reflection do?
´ The classes in the Reflection API
´ Sample code using the API
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—Major Players

```java
java.lang.reflect.Member
```

- `j.l.r.Field` 0..n
- `j.l.r.Method` 0..n
- `j.l.r.Constructor` 0..n

```java
java.lang.Class
```
The Reflection API—Class

```java
java.lang.reflect.Member

j.l.r.Field  j.l.r.Method  j.l.r.Constructor
0 .. n      0 .. n      0 .. n

java.lang.Class
1           1           1
```
The Reflection API—Class

Entry point to the Reflection API

<table>
<thead>
<tr>
<th>java.lang.Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ getName() : String</td>
</tr>
<tr>
<td>+ getFields() : Field[]</td>
</tr>
<tr>
<td>+ getField(String) : Field</td>
</tr>
<tr>
<td>+ getMethods() : Method[]</td>
</tr>
<tr>
<td>+ getMethod(String, Class[]) : Field</td>
</tr>
<tr>
<td>+ getConstructors() : Constructor[]</td>
</tr>
<tr>
<td>+ getConstructor(Class[]) : Constructor</td>
</tr>
</tbody>
</table>
The Reflection API—Member

java.lang.reflect.Member

j.l.r.Field  j.l.r.Method  j.l.r.Constructor

0 .. n  0 .. n  0 .. n

java.lang.Class

1  1  1
The Reflection API—Member

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

```java
java.lang.reflect.Member

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

```
java.lang.reflect.Member

j.l.r.Field  j.l.r.Method  j.l.r.Constructor
  0 .. n      0 .. n      0 .. n

java.lang.Class
  1
```

The Reflection API—Field

Get and set the value of the field

<table>
<thead>
<tr>
<th>java.lang.reflect.Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ get(Object) : Object</td>
</tr>
<tr>
<td>+ set(Object, Object)</td>
</tr>
<tr>
<td>+ getInt(Object) : int</td>
</tr>
<tr>
<td>+ setInt(Object, int)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
The Reflection API—Method

java.lang.reflect.Member

j.l.r.Field  j.l.r.Method  j.l.r.Constructor

0 .. n  0 .. n  0 .. n

1

java.lang.Class

1 1
The Reflection API—Method

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

```
java.lang.reflect.Method

+ invoke(Object, Object[]) : Object
+ getParameterTypes() : Class[]
+ getExceptionTypes() : Class[]
```
The Reflection API—Constructor

```
java.lang.reflect.Member

j.l.r.Field  j.l.r.Method  j.l.r.Constructor
0 .. n      0 .. n      0 .. n

1             1          1

java.lang.Class
```

:j:l:r:Field
:j:l:r:Method
:j:l:r:Constructor

0 .. n
0 .. n
0 .. n
The Reflection API—Constructor

Create an instance—`newInstance(args)`

<table>
<thead>
<tr>
<th>java.lang.reflect.Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ <code>newInstance(Object[])</code> : Object</td>
</tr>
<tr>
<td>+ <code>getParameterTypes()</code> : Class[]</td>
</tr>
<tr>
<td>+ <code>getExceptionTypes()</code> : Class[]</td>
</tr>
</tbody>
</table>
The Reflection API—Minor Players

java.lang.reflect.InvocationHandler

1

java.lang.reflect.Proxy

java.lang.reflect.Array

java.lang.reflect.Modifier
The Reflection API—Array

java.lang.reflect.InvocationHandler

java.lang.reflect.Proxy

java.lang.reflect.Array

java.lang.reflect.Modifier
The Reflection API—Array

- Create an array
- Get/set items in the array

<table>
<thead>
<tr>
<th>java.lang.reflect.Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ newInstance(Class, int) : Object[]</td>
</tr>
<tr>
<td>+ newInstance(Class, int[]) : Object[]</td>
</tr>
<tr>
<td>+ get(Object, int) : Object</td>
</tr>
<tr>
<td>+ set(Object, int, Object)</td>
</tr>
<tr>
<td>+ getInt(Object, int) : int</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
The Reflection API—Proxy

java.lang.reflect.InvocationHandler

java.lang.reflect.Proxy

java.lang.reflect.Array

java.lang.reflect.Modifier
The Reflection API—Proxy

Create a **dynamic proxy**—an object that implements one or more interfaces

```
java.lang.reflect.Proxy

+ newProxyInstance (ClassLoader, Class[], InvocationHandler) : Object
```

```
java.lang.reflect.InvocationHandler

+ invoke (Object, Method, Object[]) : Object
```
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
}
```
How to...Get an Instance of Class

• Use the class name

• 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

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 foocls = Class.forName("Foo");
    Foo f = (Foo) foocls.newInstance();
} catch (ClassNotFoundException cnfex) {
    // handle it
} catch (InstantiationException iex) {
    // handle it
} catch (IllegalAccessException iex) {
    // handle it
}
```
Interlude—What Do I Do All About All Those !@#$%^& Exceptions?

ClassNotfoundException occurs for two reasons

- Misspelled the class name
  - In source code
  - In configuration file

Programmer error—rethrow as RuntimeException

```java
catch (ClassNotFoundException cnfex) {
    throw new RuntimeException (cnfex);
}
```
ClassNotFoundException occurs for two reasons

- Misspelled the class name
  - In source code
  - In configuration file

- System loads classes from variable locations
  - Treat like an IOException

Don’t catch the exception—force higher-level code to 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

```java
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

```java
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
```
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
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[] 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);
}
Readability

Obviously suffers in most cases

Reason enough to **avoid** use of Reflection in all but exceptional circumstances
Readability

Obviously suffers in most cases

Reason enough to **avoid** use of Reflection in all but exceptional circumstances

Alleviated by general utility methods

```java
public static Method getMethod(Object obj, String str) {
    try {
        Class cl = obj.getClass();
        return cl.getMethod(str, null);
    } catch (NoSuchMethodException nsmex) {
        throw new RuntimeException(nsmex);
    }
}
```
Readability

- Obviously suffers in most cases
- Reason enough to **avoid** use of Reflection in all but exceptional circumstances
- Can be alleviated by utility methods
- 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

- Difficult to read—difficult to extend
- A reflective implementation

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

- Concise, easy to extend
- Use utility methods to improve readability
Performance

Comparison of 2 scenarios…
  µ Reflective
  µ Cached reflective

On 3 platforms…
  µ Solaris™ 8
  µ Windows 2000
  µ Windows CE

Using different Java™ virtual machines
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);
}
```
Performance—Results

![Graph showing overhead in milliseconds for different versions of Sol 8 and Win2K, comparing Reflective and Cache performance.]

- Sol 8 1.3.1: Reflective 0.035, Cache 0.005
- Sol 8 1.4: Reflective 0.02, Cache 0.005
- Win2K 1.3.1: Reflective 0.03, Cache 0.005
- Win2K 1.4: Reflective 0.025, Cache 0.005
Performance—Results

Reflective Cache

Windows CE PersonalJava

Overhead (ms)

Reflective
Cache
Performance—Conclusion

- Cache method and invoke very competitive, especially with 1.4
- Method lookup and invoke has greater overhead, but can still be useful
Code Size

Example—AWT and JFC/Swing API development
Example—AWT and JFC/Swing API development

Implementation of EventListener interfaces

```java
public interface ActionListener {
    public void actionPerformed (ActionEvent event);
}
```
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
```
Simple GUI with 100 JButton

- Compressed jar
- Uncompressed jar
Code Size—Conclusions

- For a Java technology GUI, reflective adapters can result in
  - Smaller jar files
  - Faster startup
- Could be useful on small devices
Reflection Patterns
Reflection Patterns

\- Factory Method
\- Interpreter
\- Double Dispatch
\- Interposition
“Define an interface for creating an object, but let subclasses decide which class to instantiate” — *Design Patterns* — GOF

What if the creational method is static?
Factory Method—Example

Define an abstract base class Enum for defining type-safe enumeration classes

```java
public abstract class Enum
    implements java.io.Serializable {
    public int getIndex () ...
    public String getName () ...
}
```
Factory Method—Example

Define an abstract base class Enum for defining type-safe enumeration classes

Convenient to have a bulk-creation static method

public static Enum[] create (String[] names)
Factory Method—Example

Define an abstract base class Enum for defining type-safe enumeration classes

Convenient to have a bulk-creation static method

Subclasses will be easy to define and populate

```java
class Direction extends Enum {
    public static final Direction[] DIRS = (Direction[]) create
                                                (new String[]
                                                    {"Left", "Right", "Up", "Down"});
}
```
Factory Method—Problem

- Static methods bound at compile-time
- The create method does not know how to create instances of subclasses
Factory Method—Solution

Pass in a reference to the Class instance for the subclass

```java
public static Enum[]
    create (Class cl, String[] names)
```
(Exception handling omitted)

```java
public static Enum[] create (Class cl, String[] names) {
    Enum[] enums = Array.newInstance(cl, names.length);
    for (int i = 0; i < enums.length; i++)
    {
        enums[i] = cl.newInstance();
        enums[i].setName(names[i]);
        enums[i].setIndex(i);
    }
    return enums;
}
```
An Enum base class that makes it easy to define type-safe enumerated types

```java
public class Direction extends Enum {
    public static final Direction[] DIRS = (Direction[]) create
        (Direction.class, new String[]
        {"Left", "Right", "Up", "Down"});
}
```
Recall Readability example

<table>
<thead>
<tr>
<th>String Input</th>
<th>Method call: up()</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;up&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;down&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;left&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;right&quot;</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method call: down()</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Method call: left()</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Method call: right()</th>
</tr>
</thead>
</table>
Interpreters—Example

Use as a basis for a simple graphics language

down 6
on
down 2
right 4
up 6
off
Read and parse lines into \{ String, int[] \}

down 4 \rightarrow "down" \{ 4 \}
on \rightarrow "on" \{ \}
down 2 \rightarrow "down" \{ 2 \}
right 4 \rightarrow "right" \{ 4 \}
up 6 \rightarrow "up" \{ 6 \}
off \rightarrow "off" \{ \}
Interpreters—Example

Look up method and invoke

down 4 ➔ "down" {4} ➔ down(4);
on ➔ "on" {} ➔ on();
down 2 ➔ "down" {2} ➔ down(2);
right 4 ➔ "right" {4} ➔ right(4);
up 6 ➔ "up" {6} ➔ up(6);
off ➔ "off" {} ➔ off();
Interpreters—Example

Drawing is performed

down 4 → "down" {4} → down(4);
on → "on" {} → on();
down 2 → "down" {2} → down(2);
right 4 → "right" {4} → right(4);
up 6 → "up" {6} → up(6);
off → "off" {} → off();
String need not match method name

Map String => Method

<table>
<thead>
<tr>
<th>String</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;on&quot;</td>
<td>public void penOn ()</td>
</tr>
<tr>
<td>&quot;off&quot;</td>
<td>public void penOff ()</td>
</tr>
<tr>
<td>&quot;up&quot;</td>
<td>public void moveUp (int amount)</td>
</tr>
<tr>
<td>&quot;down&quot;</td>
<td>public void moveDown (int amount)</td>
</tr>
<tr>
<td>&quot;left&quot;</td>
<td>public void moveLeft (int amount)</td>
</tr>
<tr>
<td>&quot;right&quot;</td>
<td>public void moveRight (int amount)</td>
</tr>
</tbody>
</table>
Interpreters—Conclusion

See *Languages for the Java™ VM*

- 160 languages for the JVM™
- Many use Reflection for implementation
Suppose we want a logging interface that will support various types of input.

```java
public interface Logger {
    public void log (Object obj);
}
```
Double Dispatch

A possible implementation—

```java
public class LoggerImpl implements Logger {
    public void log (Object obj) ...
    private void log (String str) ...
    private void log (Date date) ...
    private void log (Exception ex) ...
}
```
Double Dispatch—Problem

Leads to chains of if ... else if

```java
public void log (Object obj) {
    if (obj instanceof String) {
        log ((String) obj);
    } else if (next instanceof Date) {
        log ((Date) obj);
    } else if (next instanceof Exception) {
        log ((Exception) obj);
    }
}
```
Define a loggable interface

```java
public interface Loggable {
    public void logThis (Logger logger);
}
```
Double Dispatch—Classic Solution

- Define a loggable interface

- Add type-specific methods to the Logger interface

```java
public interface Logger {
    public void log (Object obj);
    public void log (String str);
    public void log (Date date);
    public void log (Exception ex);
}
```
Double Dispatch—Classic Solution

- Define a loggable interface
- Add type-specific methods to the Logger interface
- Implementations of Loggable invoke the appropriate method

```java
public class LoggableDate implements Loggable {
    private Date date;
    public void logThis (Logger logger) {
        logger.log (date);
    }
}
```
Double Dispatch—Classic Solution

- Define a loggable interface
- Add type-specific methods to the Logger interface
- Implementations of Loggable invoke the appropriate method
- Implementation of Logger.log

```java
public void log (Object obj) {
    ((Loggable) obj).logThis (this);
}
```
Double Dispatch—Classic Problems

- Must define a Loggable adapter or subclass for every type
  - Some types may be final—java.lang.String
- Must add a method to the Logger interface for each new type, as well as an implementation method in LoggerImpl
Double Dispatch—Reflection

Select method based on argument class

```java
private Method getLogMethod (Object obj) {
  try {
    Class cl = obj.getClass ();
    return getClass ().getDeclaredMethod
      ("log", new Class[] { cl });
  } // catch blocks omitted
}
```
Double Dispatch—Reflection

Logger implementation

```java
public void log (Object obj) {
    try {
        getLogMethod (obj).invoke (this, new Object[]{obj});
    } // catch blocks omitted
}

private void log (String string) ...
private void log (Date date) ...
private void log (Exception exception) ...
```
private Method getLogMethod (Object obj) {
    Class cl = obj.getClass ();
    return getClass ().getDeclaredMethod("log", new Class[]{cl});
}

′ Fails if log is called with an Exception subclass
′ Need more sophisticated utility method
  μ Search all methods
  μ isAssignableFrom
  μ Search methods from superclasses
Double Dispatch—Results

- Easy to extend for new types
- No need to subclass or wrap the various loggable types
- Loss of type safety
- “Thinking in Patterns” —Chapter 12—Bruce Eckel
  - [http://www.mindview.net](http://www.mindview.net)
- “Reflect on the Visitor design pattern” —Jeremy Blosser
Interposition

A means for adding to or amending behavior across a group of methods
Interposition

**Service**
- `getEmployee()`
- `getManager()`
- `addEmployee()`
- `removeEmployee()`
- `addManager()`
- `removeManager()`

**Take action before invocation**

**Take action after invocation**

**Caller**

**Caller**
Interposition—Implementation

Use a dynamic proxy

- Substitutes for the “real” object
- Intercepts the method call
  - Perform action (before)
  - Invoke method on real object
  - Perform action (after)
Exception handling Interposition

```java
public class ExceptionHandler implements InvocationHandler {
    private Object source;

    public ExceptionHandler (Object source) {
        this.source = source;
    }

    public void invoke (Object proxy, Method method, Object[] args)
        throws Throwable ...
```
public void invoke (Object proxy, Method method, Object[] args) throws Throwable {
    try {
        method.invoke (source, args);
    } catch (InvocationTargetException itex) {
        log (itex.getCause ());
        throw itex.getCause ();
    } catch (IllegalAccessException iaex) {
        throw new RuntimeException (iaex);
    }
}

Interposition—Implementation
Interposition—Sample Usage

RMI version of Service

```java
public interface Service extends Remote ...
public class ServiceImpl implements Service ...

// create a dynamic proxy
Service proxy = (Service) Proxy.newInstance
    (impl.getClass ().getClassLoader (),
     new Class[] { Service.class },
     new ErrorHandler (impl));
Naming.rebind ("//habanero/Service", proxy);
```
Any exception from any method is logged
Interposition—Another Example

Provides a read-only view of an object

```java
public void invoke (Object proxy, Method method, Object[] args) throws Throwable {
    try {
        if (method.getName ().startsWith ("set")) {
            throw new SecurityException ("not allowed to set");
        }
        method.invoke (source, args);
    } // catch blocks omitted
}
```
Interposition—Conclusion

Applications

- Logging
- Persistence
- Error handling
Interposition—Links

“Explore the Dynamic Proxy API” —Jeremy Blosser

“Using java.lang.reflect. Proxy to Interpose on JavaTM Class Methods” —Tom Harpin
http://developer.java.sun.com/developer/technicalArticles/JavaLP/Interposing/

Aspect Oriented Programming
http://www.aspectj.org
Real World Reflection
Real World Reflection

- Jakarta Tomcat Servlet engine
- jEdit text editor
- Java™ core libraries
Real World Reflection—Tomcat

- Reference implementation of the Java™ Servlet API and JavaServer Pages™ technology

- http://jakarta.apache.org/tomcat

- Reflection used to configure server from XML files
Tomcat—Example

The **server.xml** file contains basic server configuration information:

```
<Server port="8005" shutdown="SHUTDOWN">
  <Service name="Tomcat-Standalone">
    ...
```

![Diagram](server.xml -> SAX parser)
An internal map associates tag name to class name

```xml
<Server port="8005" shutdown="SHUTDOWN">
  <Service name="Tomcat-Standalone">
    ...
  </Service>
</Server>
```

<table>
<thead>
<tr>
<th>Tag Name</th>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Server&quot;</td>
<td>&quot;org.apache.catalina.core.StandardServer&quot;</td>
</tr>
<tr>
<td>&quot;Service&quot;</td>
<td>&quot;org.apache.catalina.core.StandardService&quot;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Tomcat—Example

- A tag name is encountered during parsing
- The associated class name is retrieved
- Class instance retrieved by Class.forName
- Instance created by Class.newInstance

- server.xml -> SAX parser -> StandardServer
  create
Tomcat—Example

- The attributes of the tag are parsed
- Associated **set** method found via Reflection
  - Look first for set method with a String parameter
  - Look for set method with int/boolean parameter
- Method is invoked on the newly created object
  `<Server port="8005" shutdown="SHUTDOWN">`
  - `standardServer.setPort(8005);`
  - `standardServer.setShutdown("Shutdown");`
Object configured from tag attributes

```xml
<Server port="8005" shutdown="SHUTDOWN">
```

```
configure

server.xml  -->  SAX parser  -->  StandardServer
```
Tomcat—Example

Process repeated for child tags

```xml
<Server port="8005" shutdown="SHUTDOWN">
  <Service name="Tomcat-Standalone">
  
    StandardServer
    port:8005
    shutdown:"SHUTDOWN"
  
    create & configure
    StandardService
    name:"Tomcat-Standalone"

  </Service>
</Server>
```
An internal map also associates the tag name with a method name from the parent:

```xml
<Server port="8005" shutdown="SHUTDOWN" debug="0">
    <Service name="Tomcat-Standalone">
        ...
    </Service>
</Server>
```

<table>
<thead>
<tr>
<th>Tag Name</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Service&quot;</td>
<td>&quot;addService&quot;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Tomcat—Example

The associated method is retrieved from the parent class (StandardServer)

The method is invoked on the parent object with the child object as the argument

```xml
<Server port="8005" shutdown="SHUTDOWN" debug="0">
    <Service name="Tomcat-Standalone">
        ...
        standardServer.addService (standardService);
    </Service>
</Server>
```
Tomcat—Example

The child object is added to the object graph

```
server.xml
```

```
SAX parser
```

```
invoke
```

```
StandardServer
port:8005
shutdown:"SHUTDOWN"
```

```
StandardService
name:"Tomcat-Standalone"
```
Tomcat—Conclusion

- A complex data structure configured from an XML file
- Performance not critical—used at startup
- Flexibility in configuration is critical
  - Plug in different service types by tag name
  - Build structure via various methods
Real World Reflection—jEdit

- Text editor based entirely on Java™ technology
- http://www.jedit.org
- Uses Reflection to discover library capabilities at runtime

Display Directory Tree → Does java.io.File provide info on filesystem roots?
java.io.File.listRoots method was added in 1.2

// try using Java 2 method first
try{
    method = File.class.getMethod
        ("listRoots",new Class[0]);
} catch(Exception e) {
    fsView = FileSystemView.getFileSystemView();
}
Reflection used to select workaround for old versions

- Display Directory Tree
- Does java.io.File provide info on filesystem roots?
  - Yes (1.2)
    - Use it.
  - No (1.1)
    - Provide a workaround.
Object Serialization
- Reassemble Object from Stream
- Disassemble Object into Stream

Diagram:
- File, Socket, ...
- File, Socket, ...
- InputStream to Object
- Object to OutputStream
- InputStream
- OutputStream
Object Serialization

Reflection used to determine fields
- Even private ones!

```
Field[] getClass().getDeclaredFields()
```
Reflection used to get the value of fields for writing to the stream

```java
for (int i = 0; i < fields.length; i++)
    fields[i].get(emp).out.write(val)
```

Employee

<table>
<thead>
<tr>
<th>name</th>
<th>id</th>
<th>salary</th>
</tr>
</thead>
</table>

"Steve"

File
Reflection used to set the value of fields from data read from the stream

```
for (int i = 0; i < fields.length; i++)
```

```
in.read()
```

```
"Steve"
```

```
fields[i].set(emp, val)
```

```
Employee
```

```
name
```

```
id
```

```
salary
```
Object Serialization—Conclusion

- Some other applications
  - Object-Relational Mapping
  - Debugging

- Code Generation vs. Reflection
When Not to Use Reflection
Reflection Considered Harmful?

Or… “When You Have a Crowbar, Everything Looks Like A Wall”

Don’t use reflection when…
Reflection Considered Harmful?

Or… “When You Have a Crowbar, Everything Looks Like A Wall”

Don’t use reflection when…

- Direct invocation does the job
  - More simply
  - More clearly
Reflection Considered Harmful?

Or… “When You Have a Crowbar, Everything Looks Like A Wall”

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
Reflection Considered Harmful?

Or... “When You Have a Crowbar, Everything Looks Like A Wall”

Don’t use reflection when...

- Direct invocation does the job
- Type safety is more important than flexibility
- Performance is critical
- Inside loops
Conclusion
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
- Use utility methods to hide some of the complexity of the Reflection API
And...

Don’t fear Reflection—fear indiscriminate use of Reflection.