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 a class or interface

```
c.isInterface ()
```

- Determine lineage
  - Superclass

```
Class c1 = c.getSuperclass ()
```

  - Superinterface

```
Class[] c2 = c.getInterfaces ()
```

Here Is How To…

- Determine implemented interfaces

```
Class[] c2 = c.getInterfaces ()
```

- 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",
                        new Class[] {…})
```

- Invoke a method

```
m.invoke (c, new Object[] {…})
```

Here Is How To…

- Determine modifiers

```
Modifiers[] mo = c.getModifiers ()
```

- Determine fields

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

- Find a specific field

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

- Modify a specific field
  - Get the value of a specific field

```
f.get (o)
```

  - Set the value of a specific field

```
f.set (o, 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]

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;
}
```