Metaprogramming & Reflection in Java

Pascal Costanza

Advanced Java Metaprogramming

- Annotation processing.
- Class loading / instrumentation.
- Dynamic proxies.

Annotations in Java

- public @interface RequestForEnhancement {
    int id();
    String synopsis();
    String engineer() default “[unassigned]”;
    String date()   default “[unimplemented]”;
  }

- @RequestForEnhancement{
    id   = 2868724,
    synopsis = “Enable time-travel”,
    engineer = “Mr. Peabody”,
    date   = “4/1/3007”
  }
  public static void travelThroughTime(Date destination) { ... }

Testing framework example

- @Retention(RetentionPolicy.RUNTIME) // CLASS, RUNTIME or SOURCE
  @Target(ElementType.METHOD) // PACKAGE, TYPE, METHOD, FIELD, ...
  public @interface Test { }

- public class Foo {
    @Test public static void m1() { }
    ...
  }

- for (Method m: Class.forName(...).getMethods()) {
    if (m.isAnnotationPresent(Test.class))
      try { m.invoke(null); }
      catch (Throwable exc) { System.out.println("Test failed: " + exc); }
  }

Annotations

- Annotations can be used at compile time, load time or runtime:
  + Annotation processing at compile time with javac.
  + Reflective access at load time.
  + Reflective access at runtime.

- http://java.sun.com/javase/6/docs/
  + Java Programming Language
    + Enhancements in JDK 5
      + Annotations

- Package java.lang.annotation

Annotation Processing (compile time)

- Define an annotation processor
  + by implementing javax.annotation,processing.Processor
  + or by subclassing javax.annotation.processing,AbstractProcessor
  + important methods: init, process, getSupportedAnnotationTypes
  + targets are represented as Java objects (ASTs)
  + note: targets cannot be modified, but only new files can be generated!
  + new source code is subject to further processing

- Annotation processors can be provided in jar files as “service providers”, or by passing them by way of the -processor option to javac.

- http://java.sun.com/javase/6/docs/
  + JDK Tool and Utility Documentation
    + javac

Instrumentation (load-time)

- Define a class file transformer
  + by implementing java.lang.instrument.ClassFileTransformer
  + important methods
    - transform for modifying bytecodes
    - premain for registering the transformer (addTransformer)
  + classes are represented as byte arrays!
  + classes may be redefined or retransformed,
    but this may not add/remove fields or methods
  + transformation is strictly ordered

- Main documentation: package description for java.lang.instrument

- Better representation by way of gnu.bytecode
  + gnu.bytecode.readClassType
  + gnu.bytecode.ClassType.writeToArray

Annotations at runtime

- ...accessed by way of java.lang.Class and java.lang.reflect.*
  (as in the “test framework” example)

Dynamic Proxy Classes: Interception of Message Sending

- Define a dynamic proxy
  + by implementing java.lang.reflect.InvocationHandler
  + important method: invoke

- http://java.sun.com/javase/6/docs/
  + Reflection
  + Dynamic Proxy Classes

Example: Observer Pattern

- @Retention(RetentionPolicy.RUNTIME)
  @Target(ElementType.METHOD)
  public @interface Setter { }

- public interface Observer<T> {
    public void update(T object);
  }

Example: Observer Pattern

- public class ObserverPattern implements InvocationHandler {

```
    protected Object delegate;

    public Object attach(Object delegate) {
      this.delegate = delegate;
      return Proxy.newProxyInstance(
                delegate.getClass().getClassLoader(),
                delegate.getClass().getInterfaces(),
                this
            );
    }

    protected Vector<Observer> observers = new Vector<Observer>();

...
```

Example: Observer Pattern

- ...

```
protected boolean isSetter(Object target, Method method) {
   return method.getAnnotation(Setter.class) != null;
}
```

```
public Object invoke(Object proxy, Method method, Object[] args)
                                                        throws Throwable {
   Object result = method.invoke(delegate, args);
   if (isSetter(delegate, method)) {
     for (Observer observer: observers) {
        observer.update(delegate);
     }
   }
}
```

Example: Observer Pattern

- public interface SimpleModelInterface {
    public @Setter void setData(String s);
    public String getData();
  }

- public class SimpleModel implement SimpleModelInterface {
    private String s = null;

    public @Setter void setData(String s) { this.s = s; }

    public String getData() { return s; }
}

Example: Observer Pattern

- ObserverPattern op = new ObserverPattern();

```
SimpleModelInterface model =
   (SimpleModelInterface)op.attach(new SimpleModel());

op.addObserver(new Observer<SimpleModelInterface> () {
   public void update(SimpleModelInterface obj) {
      System.out.println("changed: " + model.getData());
   }
});


...    model.setData("some string")   ...
```

invokedynamic

- New bytecode for custom method invocation semantics.
- Scheduled for inclusion in Java 7.

History

- JDK 1.0 (1996) - no metaprogramming features
- JDK 1.1 (1998) - java.lang.reflect (but only introspection)
- JDK 1.3 (2001) - dynamic proxies classes
- JDK 1.4 (2003) - generics
- JDK 1.5 (2004) - annotations + annotation processing + instrumentation
- JDK 1.6 (2006) - annotation processing part of javac
- JDK 1.7 (2010) - invokedynamic

Summary

- More and more metaprogramming and reflective features get adopted.
- Better learn them to stay ahead.