The Art of Metaprogramming in Java

Falguni Vyas
Dec 08, 2012
SIMPLY EXPLAINED: METADATA

WHAT'S "METADATA"?

A WORD WITH 8 LETTERS
What is Metadata?

Data that describes other data

Defined as data providing information about one or more aspects of the data, such as:

- Means of creation of the data
- Purpose of the data
- Time and date of creation
- Creator or author of data
- Location on a computer network where the data was created
- Standards used

For example,

- A digital image
- A text document
- SVN Checkout Metadata
How is Metadata expressed?

Implicit
  • Adhering to a certain convention

Explicit
  • External to the code
    • DSL, XML, RDF
  • Internal to the code
    • As comments on code, Javadoc
    • As the code itself like HTML Meta tags, Java Annotations
How is Metadata being used?

Schema
  • Data dictionary in RDBMS
  • Check constraints (JSR 303)

Semantics
  • WSDL : Web Services Description Language
  • RDF : Resource Description Framework

Data Management
  • .svn

Configuration
Metaprogramming
Metaprogramming
From Wikipedia, the free encyclopedia

Metaprogramming is the writing of computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime. In some cases, this allows programmers to minimize the number of lines of code to express a solution (hence reducing development time), or it gives programs greater flexibility to efficiently handle new situations without recompilation.
What is Metaprogramming?

Write code that writes code

Writing programs that write or manipulate other programs or themselves based on some metadata

Supported in many languages and across several platforms

An underused feature

Back bone of many of the most successful frameworks
How does Metaprogramming work?

Expose the internals of the run-time engine to the programming code through APIs.

Dynamic execution of expressions that contain programming commands

Program transformation system
Metaprogramming: Benefits

Keeps code small, simple, DRYer, lighter, more intuitive and more scalable

Rapid prototyping

Minimize the LOC to express a solution

Makes good programmers really productive
Metaprogramming : Techniques?

No well defined best practices

Many techniques focused on individual aspects of metaprogramming are often Language specific
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 (2011) - invokedynamic
- JDK 1.8 (2012) - lambda expressions
Advanced Java Metaprogramming
Advanced Java Metaprogramming

Annotation Processing

Class Loading / instrumentation

Dynamic Proxies
JSR 175 Metadata facility for Java

JSR 175 was released as a part of J2SE 5.0 (Tiger)

java.lang.annotation package

Annotation does not effect the program semantics

Benefits:

- Allows us to extend the language with new metadata.
- Helps us “color” the code
- Great expressive power
- Can be retained until runtime
Built-In Annotations

Annotations applied to java code

- @Deprecated
- @Override
- @SupressWarning

Annotations applied to other annotations

- @Documented
- @Inherited
- @Target
- @Retention
## Common Annotations

**JSR 250 (Common Annotations for the Java Platform)**

<table>
<thead>
<tr>
<th>Annotation name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generated</td>
<td>Marks sources that have been generated</td>
</tr>
<tr>
<td>Resource</td>
<td>Declares a reference to a resource, e.g. a database</td>
</tr>
<tr>
<td>Resources</td>
<td>Container for multiple Resource annotations</td>
</tr>
<tr>
<td>PostConstruct</td>
<td>Is used on methods that need to get executed after dependency injection is done to perform any initialization.</td>
</tr>
<tr>
<td>PreDestroy</td>
<td>Is used on methods that are called before the instance is removed from the container</td>
</tr>
<tr>
<td>RunAs</td>
<td>Defines the role of the application during execution in a Java EE container</td>
</tr>
<tr>
<td>RolesAllowed</td>
<td>Specifies the security roles permitted to access method(s) in an application.</td>
</tr>
<tr>
<td>PermitAll</td>
<td>Specifies that all security roles are permitted to access the annotated method, or all methods in the annotated class.</td>
</tr>
<tr>
<td>DenyAll</td>
<td>Specifies that no security roles are allowed to invoke the specified method(s).</td>
</tr>
<tr>
<td>DeclareRoles</td>
<td>Used to specify the security roles by the application.</td>
</tr>
</tbody>
</table>
@Target

@Target ElementType

<table>
<thead>
<tr>
<th>Annotation Type</th>
<th>Annotation type declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>Constructor declaration</td>
</tr>
<tr>
<td>Field</td>
<td>Field declaration (includes enum constants)</td>
</tr>
<tr>
<td>Local Variable</td>
<td>Local variable declaration</td>
</tr>
<tr>
<td>Method</td>
<td>Method declaration</td>
</tr>
<tr>
<td>Package</td>
<td>Package declaration</td>
</tr>
<tr>
<td>Parameter</td>
<td>Parameter declaration</td>
</tr>
<tr>
<td>Type</td>
<td>Class, interface (including annotation type), or enum declaration</td>
</tr>
</tbody>
</table>

@Target({ElementType.METHOD, ElementType.LOCAL_VARIABLE})
@Retention

Says whether VM retains the annotation for reflective access at runtime

Retention Policy

- Source: discarded at compile time
- Class: Kept in bytecode but not loaded into VM this is default
- Runtime: available for reflective access at runtime
Custom Annotation : Create

Let's create a custom annotation and use it.
Custom Annotation: Annotate

```java
package com.foo.custom;

public class Annotated {
    @Copyright(info = "@2012")
    public void foo1(String myParam)
    {
        System.out.println("Copyright " + myParam);
    }
    @Copyright(info = "@2013")
    public void foo2(String myParam)
    {
        System.out.println("Copyright " + myParam);
    }
}
```
Custom Annotation : Annotation Parser

```java
package com.foo.custom;
import java.lang.reflect.Method;

public class TestAnnotationParser {
    public void parse(Class clazz) {
        try {
            Method[] methods = clazz.getMethods();
            for (Method method : methods) {
                if (method.isAnnotationPresent(Copyright.class)) {
                    Copyright test = method.getAnnotation(Copyright.class);
                    String info = test.info();
                    if (@2012.equals(info)) {
                        method.invoke(
                            Annotated.class.newInstance(), info);
                    }
                    if (@2013.equals(info)) {
                        method.invoke(
                            Annotated.class.newInstance(), info);
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```
Custom Annotation: In Action

```java
package com.foo.custom;
public class Demo {
    public static void main(String[] args) throws Exception {
        TestAnnotationParser parser = new TestAnnotationParser();
        parser.parse(Annotated.class);
    }
}
```

```
<terminated> Demo
Copyright @2012
Copyright @2013
```
Annotation Processing

APT : Annotation Processing Tool

Compile time

• Annotation Processing at build time JSR 269 : Pluggable Annotation Processing API

Load-time

• Java Profiling with the java.lang.instrument Package

Runtime

• Java Reflection API
package com.foo.custom;

import java.util.Set;

@SupportedAnnotationTypes({"*"})
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class TestAnnotationProcessor extends AbstractProcessor {

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        if(!roundEnv.processingOver()){
            for (Element t : roundEnv.getRootElements()){
                for (Element element : t.getEnclosedElements()) {
                    if (element.getKind() == ElementKind.METHOD) {
                        if(element.getAnnotation(Copyright.class) != null){
                            processingEnv.getMessager().printMessage(Diagnostic.Kind.NOTE,
                            element.getSimpleName() + " has Annotation " +
                            element.getAnnotation(Copyright.class));
                        }else if (element.getAnnotation(AnotherCustomAnnotation.class) != null){
                            processingEnv.getMessager().printMessage(Diagnostic.Kind.NOTE,
                            element.getSimpleName() + " has Annotation " +
                            element.getAnnotation(AnotherCustomAnnotation.class));
                        }else{
                            processingEnv.getMessager().printMessage(Diagnostic.Kind.WARNING,
                            element.getSimpleName() + " method is not annotated!");
                        }
                    }
                }
            }
            return false;
        }
    }
}
package com.foo.custom;

public class Annotated {
    @Copyright(info = "@2012")
    public void foo1(String myParam)
    {
        System.out.println("Copyright " + myParam);
    }
    @AnotherCustomAnnotation(info = "@2013")
    public void foo2(String myParam)
    {
        System.out.println("Copyright " + myParam);
    }
    public void foo3(String myParam)
    {
        //empty!
    }
}
JSR 269 Exploiting: In Action

```
d:\MetaProgramming>javac -processor com.foo.custom.TestAnnotationProcessor Annotated.java
Note: foo1 has Annotation @com.foo.custom.Copyright<info=©2012>
Note: foo2 has Annotation @com.foo.custom.AnotherCustomAnnotation<info=©2013>
warning: foo3 method is not annotated!
```
Instrumentation

Modification of java byte code for the purpose of gathering data

AOP frameworks like AspectJ use the instrumentation framework to implement load time weaving of their advices to the point-cuts
Reflection

Definition

- Ability to introspect metalevel information about the program structure itself at runtime.
- Mechanisms to change the program interpretation or meaning at the runtime (intercession)

Usually this metalevel information is modeled using the general abstraction mechanisms available in the language.
Reflection : Dynamic Proxy Classes

Dynamic mechanism to create an implementation to some particular interface at runtime

Basic Idea:

• Associate an invocation handler to the set of interfaces
• Every method call to the interface is dispatched to the invocation handler
Metaprogramming in the wild
Metaprogramming in the wild
SPRING: Annotations

Annotation based transactions

Annotation based container configuration
Groovy : MetaClass

```groovy
Integer.metaClass.isEven = { -> // only (->) sign indicates that isEven() method is no argument method
delegate%2 == 0
}
```
Metaprogramming with Groovy

```java
import java.util.logging.Level
import java.util.logging.Logger

class MyLogger {
    static String fyi(Logger self, String msg) {
        return self.log(new CustomLevel('FYI', level.INFO.intValue()), msg)
    }
    static String ouch(Logger self, String msg) {
        return self.log(new CustomLevel('OUCH', level.SEVERE.intValue()), msg)
    }

    static void main (def args) {
        Logger log = Logger.getLogger(this.className)
        use(MyLogger) {
            log.fyi 'now you know it...'
            log.ouch 'that hurts!'
        }
    }
}

<terminated> MyLogger [Groovy Script] C:\Program Files (x86)\Java\re6\bin\javaw.exe (02-Dec-2012 10:22:53 PM)
FYI: now you know it...
OUCH: that hurts!
```
Rails : DRY Metaprogramming

```ruby
class Post < ActiveRecord::Base
  validate_inclusion_of :status, :in => ['draft', 'published', 'spam']

  def self.all_draft
    find(:all, :conditions => { :status => 'draft' })
  end

  def self.all_published
    find(:all, :conditions => { :status => 'published' })
  end

  def self.all_spam
    find(:all, :conditions => { :status => 'spam' })
  end

  def draft?
    self.status == 'draft'
  end

  def published?
    self.status == 'published'
  end

  def spam?
    self.status == 'spam'
  end
end
```
Rails : DRY Metaprogramming

```ruby
class Post < ActiveRecord::Base

STATUSES = ['draft', 'published', 'spam']
validate_inclusion_of :status, :in => STATUSES

class << self
  STATUSES.each do |status_name|
    define_method "all_#{status_name}" do
      find(:all, :conditions => { :status => status_name })
    end
  end
end

STATUSES.each do |status_name|
  define_method "#{status_name}" do
    self.status == status_name
  end
end
end
```
Thank You.