J-Spring

16 april 2008 Spant! - Bussum

Annotation Processing

Angelika Langer

Trainer/Consultant

http://www.AngelikaLanger.com

goal

- give an overview of annotation processing
  - what are annotations?
    - meta information
  - how are they defined?
    - language features since JDK 5.0
  - how are they processed?
    - on the source code level
    - (on the byte code level)
    - (at runtime via reflection)

speaker's qualifications

- independent trainer / consultant / author
  - teaching C++ and Java for 10+ years
  - curriculum of a dozen challenging courses
  - co-author of "Effective Java" column in JavaSpektrum
  - author of Java Generics FAQ online
  - Java champion since 2005

agenda

- **annotation language features**
- processing annotations
- case study

program annotation facility

- allows developers
  - to define custom *annotation types*
  - to *annotate* fields, methods, classes, etc. with *annotations* corresponding to these types
- allow tools to read and process the annotations
  - no direct effect on semantics of a program
  - e.g. tool can produce additional Java source files or XML documents related to the annotated program

sample usage

- annotated class

```
@Copyright("2008 Vibro Systems, Ltd.")
public class OscillationOverthruster { ... }
```

- corresponding definition of annotation type

```
public @interface Copyright { String value(); }
```

- reading an annotation via reflection

```
String copyrightHolder
= OscillationOverthruster.class.
  getAnnotation(Copyright.class).value();
```

retention

- it makes little sense to retain all annotations at run time
  - would increase run-time memory-footprint

- annotations can have different lifetime:

SOURCE:

- discarded after compilation

CLASS:

- recorded in the class file as signature attributes
- not retained until run time

RUNTIME:

- recorded in the class file *and* retained by the VM at run time
- may be read reflectively

agenda

- annotation language features
  - declaring annotation types
  - annotating program elements
  - meta annotations
- processing annotations
- case study

annotation type

- every annotation has an *annotation type*
  - takes the form of a highly restricted interface declaration
  - new "keyword" `@interface`
  - a *default value* may be specified for an annotation type member
  - permitted return types include primitive types, String, Class

```
public @interface RequestForEnhancement {
  int id();
  String synopsis();
  String engineer() default "[unassigned]";
  String date()     default "[unimplemented]";
}s
```

using annotation types

```
@RequestForEnhancement(
  id = 28,
  synopsis = "Provide time-travel functionality",
  engineer = "Mr. Peabody",
  date = "12/24/2008"
)
public static void travelThroughTime(Date destination) { ... }
```

- members with a default may be omitted

```
@RequestForEnhancement(
  id = 45,
  synopsis = "Add extension as per request #392"
)
public static void balanceFederalBudget() {
  throw new UnsupportedOperationException("Not implemented");
}
```

marker annotations

- annotation types can have no members
  - called *marker annotations*

```
public @interface Immutable { }
```

- sample usage

```
@Immutable
public class String { ... }
```

agenda

- annotation language features
  - declaring annotation types
  - **annotating program elements**
  - meta annotations
- processing annotations
- case study

annotatable program elements

- annotations may be used as modifiers in the declaration of:
  - package, class, interface, field, method, parameter, constructor, local variable, enum type, enum constant, annotation type

```
public @interface Copyright {String value();}
public @interface Default {}
```

```
@Copyright("2004 Angelika Langer")
public enum Color { RED, BLUE, GREEN, @Default NOCOLOR }
```

more annotated types

- JSR 308 (in Java 7.0) allows annotations as **type qualifiers** (on *any* use of a type)

- type parameter:

```
Map<@NonNull String,
    @NonEmpty List<@Readonly Document>> files;
```

- bounds:

```
class Folder<F extends @Existing File> { ... }
Collection<? super @Existing File> var;
```

- array:

```
Document[@Readonly][] docs1
  = new Document[@Readonly 2][12];
Document[][@Readonly] docs2
  = new Document[2][@Readonly 12];
```

disambiguation

```
Dimension getSize() @Readonly { ... }
```

- `@Readonly` annotates the type of `this`

```
@Readonly Dimension getSize() { ... }
```

- `@Readonly` annotates the return type

```
@Override
@NonNull Dimension getSize() { ... }
```

- `@NonNull` annotates the return type
- `@Override` annotates the method declaration

- `@Target` meta-annotation indicates the intent:

```
@Target(ElementType.TYPE)
public @interface ReadOnly
{}
@Target(ElementType.TYPE)
public @interface NonNull
{}
@Target(ElementType.METHOD)
public @interface Override
{}
```

agenda

- annotation language features
  - declaring annotation types
  - annotating program elements
  - **meta annotations**
- processing annotations
- case study

meta annotations

`@Target(ElementType[])`

- indicates the program elements to which an annotation type can be applied
- values: TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE, PACKAGE
- default: applicable to *all* program elements

`@Documented`

- indicates that annotations are documented in javadoc

`@Retention(RetentionPolicy)`

- indicates how long annotations are to be retained
- values: SOURCE, CLASS, RUNTIME
- default: CLASS

sample usage

- self-referential meta-annotation

```
@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Retention { RetentionPolicy value(); }
```

agenda

- annotation language features
- **processing annotations**
- case study

annotation processing

- can happen on 3 levels
  - introspectors
    - process *runtime-visible* annotations of their own program elements
    - use reflection and need annotations with RUNTIME retention
  - byte code analyzers
    - process annotations in `.class` files
    - e.g. stub generators
  - source code analyzers
    - process annotations in Java source code
    - e.g. compilers, documentation generators, class browsers

agenda

- annotation language features
- processing annotations
  - **reflection**
  - pluggable annotation processing in 6.0
- case study

introspection

- using reflection
  - to inspect its own program elements
  - search for annotated elements
  - retrieve annotations and their content
- reflection API has been extended
  - to support introspective annotation processing

extensions to the reflection API

- additional methods in `Package`, `Class`, `Field`, `Constructor`, `Method`

```
<A extends Annotation>
A getAnnotation(Class<A> annotationClass)
```

  - returns the specified annotation if present on this element

```
Annotations[] getAnnotations()
Annotations[] getDeclaredAnnotations()
```

  - returns all annotations that are (directly) present on this element

```
boolean isAnnotationPresent
    (Class<? extends Annotation> annotationClass)
```

  - returns true if an annotation for the specified type is present on this element

reading annotations

```
@RequestForEnhancement(
  id = 28,
  synopsis = "Provide time-travel functionality",
  engineer = "Mr. Peabody",
  date = "24/12/2008"
)
public static void travelThroughTime(Date destination) { ... }
```

- accessed reflectively:

```
Method m = TimeTravel.class.getMethod
           ("travelThroughTime", new Class[] {Date.class});
RequestForEnhancement rfe
  = m.getAnnotation(RequestForEnhancement.class);
int    id       = rfe.id();
String synopsis = rfe.synopsis();
String engineer = rfe.engineer();
String date     = rfe.date();
```

agenda

- annotation language features
- processing annotations
  - reflection
  - **pluggable annotation processing in 6.0**
- case studies

annotation processing in Java 6.0

- annotation processing integrated into `javac` compiler
  - since Java 6.0; known as *pluggable annotation processing*
  - compiler automatically searches for annotation processors
  - unless disabled with `-proc:none` option
  - processors can be specified explicitly with `-processor` option
  - details at `java.sun.com/javase/6/docs/technotes/tools/windows/javac.html#processing`

- example:

```
javac -processor MyAnnotationProcessor MyAnnotatedClass.java
```

annotation processor

- implement a processor class
  - must implement `Processor` interface
  - typically derived from `AbstractProcessor`
  - new package `javax.annotation.processing`

- specify supported annotation + options
  - by means of annotations:
    `@SupportedAnnotationTypes`
    `@SupportedOptions`
    `@SupportedSourceVersion`

annotation processor - example

```
@SupportedAnnotationTypes({"Property"})
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class PropertyAnnotationProcessor extends AbstractProcessor {
  public boolean process(Set<? extends TypeElement> annotations,
                         RoundEnvironment env) {
    … process the source file elements using the mirror API …
  }
}
```

rounds

- annotation processing happens in a sequence of *rounds*

- 1st round:
  - compiler parses source files on the command line
    - to determine what annotations are present
  - compiler queries the processors
    - to determine what annotations they process
  - when a match is found, the processor is invoked

claim

- a processor may "claim" annotations
  - no further attempt to find any processors for those annotations
  - once all annotations have been claimed, compiler stops looking for additional processors

- claim is specified as return value of `process()` method
  - `true`: annotations are claimed;
    no subsequent processors are asked to process them
  - `false`: annotations are unclaimed;
  - subsequent processors are asked to process them

subsequent rounds

- if processors generate new source files, another round of annotation processing starts
  - newly generated source files are parsed and annotations are processed as before
  - processors invoked on previous rounds are also invoked on all subsequent rounds
- this continues until no new source files are generated

last round

- after a round where no new source files are generated:
  - annotation processors are invoked one last time
    - to give them a chance to complete work they still need to do
  - compiler compiles original and all generated source files

- compilation and/or processing is controlled by `-proc` option
  - `-proc:only`: only annotation processing, no subsequent compilation
  - `-proc:none`: compilation takes place without annotation processing

environment

- processor environment provides
  - `Filer` for creation of new source, class, or auxiliary files
  - `Messager` to report errors, warnings, and other notices
- inherited as protected field from `AbstractProcessor`

```
protected ProcessingEnvironment processingEnv
```

  - implicitly initialized on construction of the processor

processor arguments

- process() method takes 2 arguments:

```
Set<? extends TypeElement> annotations
```

  - the annotation types requested to be processed
  - subset of the supported annotations

```
RoundEnvironment roundenv
```

  - environment for information about the current and prior round
  - supplies elements annotated with a given annotation or all root elements in the source

annotation processor - example

```
public boolean process(Set<? extends TypeElement> annotations,
                       RoundEnvironment roundEnv) {
    for (Element t : roundEnv.getRootElements()) {
      if (t.getModifiers().contains(Modifier.PUBLIC)) {
         for (ExecutableElement m :
               ElementFilter.methodsIn(t.getEnclosedElements())) {
            Property p = m.getAnnotation(Property.class);
            if (p != null) { ... process property ... }
…
```

elements and types

- *elements* and *types* from `javax.lang.model.*` packages
  - represent *declarations* and *types* in the Java source code
- *element* is a static language construct
  - like the declaration of `java.util.Set`
- a family of *types* is associated with an element
  - like the raw type `java.util.Set`, and the parameterized types `java.util.Set<String>` and `java.util.Set<T>`

filers

```
private void writeGeneratedFile(String beanClassName) {
  FileObject sourceFile
        = processingEnv.getFiler().createSourceFile(beanClassName);
  PrintWriter out = new PrintWriter(sourceFile.openWriter());
  out.print("public class ");
  …
  out.close();
}
```

- `Filers` are obtained from the *processing* environment
  - not from the *round* environment

agenda

- annotation language features
- processing annotations
- **case study**

@Comparator annotation

- define a `@Comparator` annotation
  - that can be used to annotate methods that perform a comparison
- build an annotation processor that generates a `Comparator` class
  - for each annotated method

intended use of annotation

file: `data\Name.java`

```
public class Name {
  private final String first;
  private final String last;
  public Name(String f, String l) {
    first = f;
    last = l;
  }
  @Comparator("NameByFirstNameComparator")
  public int compareToByFirstName(Name other) {
    if (this == other) return 0;
    int result;
    if ((result = this.first.compareTo(other.first)) != 0)
      return result;
    return this.last.compareTo(other.last);
  }
}
```

class to be generated

file: data\NameByFirstNameComparator.java

```
public class NameByFirstNameComparator
  implements java.util.Comparator<Name> {

  public int compare(Name o1, Name o2) {
    return o1.compareToByFirstName(o2);
  }
  public boolean equals(Object other) {
    return this.getClass() == other.getClass();
  }
}
```

define the @Comparator annotation

file: `processor/Comparator.java`

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Comparator {
    String value();
}
```

- applicable to methods only
- present in source code only
- value is the name of the `Comparator` class to be generated

annotation processor

file: `processor/ComparatorAnnotationProcessor.java`

```
@SupportedAnnotationTypes({"processor.Comparator"})
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class ComparatorAnnotationProcessor
  extends AbstractProcessor {
    public boolean process(
        Set<? extends TypeElement> annotations,
        RoundEnvironment roundEnv) {
      … see next slide …
    }
}
```

- supports no options
- processes only the `@Comparator` annotation

processing @Comparator

```
public void process() {
  for (Element t : roundEnv.getRootElements()) {
    if (t.getModifiers().contains(Modifier.PUBLIC)) {
      for (ExecutableElement m :
         ElementFilter.methodsIn(t.getEnclosedElements()) ) {
         Comparator a = m.getAnnotation(Comparator.class);
         if (a != null) {
           … see next slide …
         }
    } }
} } }
```

- process all type declarations in the source file
- ignore non-public ones
- process all methods of the type
- ignore methods without a `@Comparator` annotation

checking the annotated method

```
TypeMirror returnType = m.getReturnType();
if (!(returnType instanceof PrimitiveType) ||
    ((PrimitiveType)returnType).getKind()!=TypeKind.INT
   ) {
   processingEnv.getMessager().printMessage(Diagnostic.Kind.ERROR,
   "@Comparator can only be applied to methods that return int");
   continue;
}
```

… see next slide …

- check whether return type is `int`
- print error message

preparing code generation

```
String comparatorClassName     = a.value();
String comparetoMethodName     = m.getSimpleName();
String theProcessedClassesName = t.getQualifiedName();}

writeComparatorFile(theProcessedClassesName,
                    comparatorClassName,
                    comparetoMethodName);
```

- retrieve the name of the `Comparator` class to be generated
  - from the `@Comparator` annotation
- retrieve the compare method's name
  - from the annotated method
- retrieve the enclosing class's name
  - from the processed type declaration

generating the source file

```
private void writeComparatorFile(
            String fullClassName,
            String comparatorClassName,
            String compareToMethodName) throws IOException {
  int i = fullClassName.lastIndexOf(".");
  String packageName = fullClassName.substring(0, i);

  FileObject sourceFile = processingEnv.getFiler().
    createSourceFile(packageName+"."+comparatorClassName);
  PrintWriter out = new PrintWriter(sourceFile.openWriter());

  if (i > 0) { out.println("package "+packageName); }
   … see next slide …
}
```

- get output destination from environment
- create a source file and provide the class name
  - package directory and .java suffix are determined automatically

invoke compiler

- invoke the `javac` compiler for annotation processing
  - it generates a class for each annotated method
  - in the package of the method's enclosing class

```
>javac -processor
processor.ComparatorAnnotationProcessor
data\Name.java
```

wrap-up

- annotations permit associating information with program elements
  - consist of member-value-pairs and an annotation type
  - annotation types are a restricted variant of interfaces

- annotations have different lifetime
  - SOURCE, CLASS, RUNTIME
  - runtime annotations can be read via reflection
  - source code annotation processing supported by `javac` compiler

wrap-up

- 6.0 pluggable annotation processing support
  - an easy way of processing annotations and generating side files
    - not an exhaustive exploration of the possibilities
    - case study intends to provide an idea of what can be done with annotated source files

author

Angelika Langer

Training & Mentoring

Object-Oriented Software Development in C++ & Java

Email: **contact@AngelikaLanger.com**

http: **www.AngelikaLanger.com**

annotation processing

Q & A