The Java Specialists' Newsletter
Issue 167 2008-12-05 Category: Language Java version: Java 6

1 Subscribe Free RSS Feed Print

Annotation Processing Tool

by Dr. Heinz M. Kabutz

Abstract:
In this newsletter we answer the question: "How do we force all subclasses to contain a public no-args constructor?" The Annotation Processing Tool allows us to check conditions like this at compile time, rather than only at runtime.

Welcome to the 167th issue of **The Java(tm) Specialists' Newsletter**. A special welcome to my subscribers from Ethiopia and Mongolia, increasing our **countries to 120**! We are still missing Albania and a few countries in Africa and Asia. I am also not sure about some disputed territories in South America. Greenland, Antarctica and the Vatican are also still missing. So, if you have friends who are Java programmers in these regions, please ask them to subscribe and send me a quick note :-)

Upcoming Java Master Courses:
Duesseldorf, Germany, Aug 22
Chania, Crete, Sep 6
Cape Town, South Africa, Sep 12

In-house courses if these dates or locations do not suit you. Note that the course in Crete may also be attended remotely via webinar.

Annotation Processing Tool

Here is a pop quiz for you, thanks to Christoph Engelhardt: "How do we force all subclasses to contain a public no-args constructor?"

To do this at runtime would be trivial. You could simply check inside the superclass' constructor whether the subclass returned by `this.getClass()` contains a no-args constructor like so:

```
import java.lang.reflect.Constructor;

public abstract class ABC {
  public ABC() {
    checkNoArgsConstructor();
  }
  private void checkNoArgsConstructor() {
    Class clazz = getClass();
    try {
      Constructor noargs = clazz.getConstructor();
    } catch (NoSuchMethodException e) {
      throw new AssertionError("Class " + clazz.getName() +
          " needs a no-args constructor");
    }
  }
}
```

That was so easy it would hardly qualify as a quiz. Let's up the ante a bit by requiring that we want to discover subclasses of ABC without a no-args constructor at *compile time*.

It is possible to do this since Java 5. However, in Java 5, the mechanism to do this was separate to the `javac` compiler and the classes we needed were in the `com.sun.*` package structure. In Java 6, the mechanism is incorporated into javac and we have a set of classes in the `javax.*` package that we can use to examine the annotations.

The first step is to define an annotation `NoArgsConstructor`:

```
package eu.javaspecialists.tools.apt;

import java.lang.annotation.*;

@Inherited
@Documented
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface NoArgsConstructor {
}
```

Newsletter Links
Book Review
Concurrency
Exceptions
GUI
Inspirational
Language
Performance
Software Engineering
Tips and Tricks

The annotation is marked with meta-annotations. **@Inherited** is necessary so that the subclasses inherit the annotation. We would like that annotation to appear in the JavaDocs, so we mark it as **@Documented**. We use the **@Retention** meta-annotation to specify that the annotation will only be available to APT and not at runtime with reflection. Lastly, we use **@Target** to define that the annotation is only allowed for classes.

Next we need to write a processor for this annotation. APT uses a variation of the Visitor design pattern. Unfortunately I found the documentation to be a little bit hard to understand. Hopefully this sample code will make it easier for you to write your own annotation processors.

```
package eu.javaspecialists.tools.apt;

import javax.annotation.processing.*;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.*;
import javax.lang.model.type.*;
import javax.lang.model.util.SimpleTypeVisitor6;
import javax.tools.Diagnostic;
import java.util.Set;

@SupportedAnnotationTypes(
    "eu.javaspecialists.tools.apt.NoArgsConstructor")
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class NoArgsConstructorProcessor extends AbstractProcessor {
  public boolean process(Set<? extends TypeElement> annotations,
                         RoundEnvironment env) {
    for (TypeElement type : annotations) {
      processNoArgsConstructorClasses(env, type);
    }
    return true;
  }

  private void processNoArgsConstructorClasses(
      RoundEnvironment env, TypeElement type) {
    for (Element element : env.getElementsAnnotatedWith(type)) {
      processClass(element);
    }
  }

  private void processClass(Element element) {
    if (!doesClassContainNoArgsConstructor(element)) {
      processingEnv.getMessager().printMessage(
          Diagnostic.Kind.ERROR,
          "Class " + element + " needs a No-Args Constructor");
    }
  }

  private boolean doesClassContainNoArgsConstructor(Element el) {
    for (Element subelement : el.getEnclosedElements()) {
      if (subelement.getKind() == ElementKind.CONSTRUCTOR &&
          subelement.getModifiers().contains(Modifier.PUBLIC)) {
        TypeMirror mirror = subelement.asType();
        if (mirror.accept(noArgsVisitor, null)) return true;
      }
    }
    return false;
  }

  private static final TypeVisitor<Boolean, Void> noArgsVisitor =
      new SimpleTypeVisitor6<Boolean, Void>() {
        public Boolean visitExecutable(ExecutableType t, Void v) {
          return t.getParameterTypes().isEmpty();
        }
      };
}
```

The process() method is called by the compiler when it processes the annotation. Since we have specified that we are only interested in the NoArgsConstructor annotation, we will only be given classes that use this annotation or whose ancestor uses it.

In the `doesClassContainNoArgsConstructor()` method, we iterate through all of the class elements and pick out the public constructors. A class that does not have any, will automatically fail the test. Note that when we do not specify any constructor, Java automatically adds a default no-args constructor.

Once we have found our constructor element, we visit it with to determine whether the argument list is empty. Using the Visitor pattern in this case helps us to avoid a downcast.

The code looks very easy now that it has been completed, but it took me a while to get it to this state.

Once you have written the annotation processor, you need to package it into a jar file. In addition, you should include a file `META-INF/services/javax.annotation.processing.Processor` with the text `eu.javaspecialists.tools.apt.NoArgsConstructorProcessor` as a single line.

Once we have all these files packaged in a jar file, let's call it **apttools.jar**, we can add a processing parameter to our call to javac:

```
javac -classpath .;apttools.jar -processorpath apttools.jar *.java
```

As a test case, we define several classes:

```
import eu.javaspecialists.tools.apt.NoArgsConstructor;

@NoArgsConstructor
public abstract class NoArgsSuperClass {
  public NoArgsSuperClass() {
  }
}

// Passes
```

```
public class PublicNoArgsConstructor extends NoArgsSuperClass {
  public PublicNoArgsConstructor() {
  }
}

// Passes
public class DefaultConstructor extends NoArgsSuperClass {
}

// Passes
public class SeveralConstructors extends NoArgsSuperClass {
  public SeveralConstructors(String as) {
  }

  public SeveralConstructors(int ai) {
  }

  public SeveralConstructors() {
  }
}

// Fails
public class NonPublicConstructor extends NoArgsSuperClass {
  NonPublicConstructor() {
  }
}

// Fails
public class WrongConstructor extends NoArgsSuperClass {
  public WrongConstructor(String aString) {
  }
}
```

When we compile these classes with the NoArgsConstructorProcessor, we see the following compiler errors:

```
    error: Class NonPublicConstructor needs a No-Args Constructor
    error: Class WrongConstructor needs a No-Args Constructor
    2 errors
```

There are lots of possibilities with the new Java 6 Annotation Processing Tool. In this newsletter, we showed how we can use annotations to restrict the Java language in ways that were not originally in the design.

Kind regards

Heinz

Language Articles Related Java Course Discuss at The Java Specialist Club