

Writing an annotation preprocessor

Davide Lissoni Mat.179878

28/10/2016

1.Introduction:

This assignment is about java annotations and java Reflection. Its goal was to create a project that take as input a simple annotated java POJO and returns, if requested through proper annotations, a new java file, containing the same POJO transformed in a java bean. The specification for the annotations will be explained in the next chapter.

Before explaining how the application work I would like to clarify the meaning of “java bean” in order to simplify the explanation of some choices:

a java bean is a java class standard. A java class for being a bean must respect the following conventions:

- It must contain a public no-argument constructor;
- All the class properties must be private and need to be accessible by get and set methods.

Furthermore the class should be serializable (this is not strictly necessary) in order to manage the bean state independently.

2.Implementation:

In order to reach the assignment requests the source code is composed by a class called Reflect.java and two annotation interfaces: MethodNote and bean.

2.1 Annotations:

The following annotations have been created in order to simplify and give some informations about input pojo and bean output proprieties to the main application.

Since we used java Reflection to discover input-class informations, both the interfaces have to be available and used at runtime.

Bean:

Bean.java (Figure 1) has been declared as class annotation, and contains a single boolean attribute: Serializable.

This class annotation is used in order to understand if the class has to be converted form java pojo to a java bean and in particular:

@bean(Serializable=true):a input class annotated in this way will be converted to a java bean and will implements java.io.Serializable.

@bean(Serializable=false): a input class annotated in this way will be converted to a java bean and will not implements java.io.Serializable.

If the class is not annotated in one of the listed above way, the input class will not be converted.

```

package reflectInspector;
import java.lang.annotation.Retention;
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface bean
{ boolean Serializable();}

```

Figure 1

MethodNote:

This is a method annotation, and contains two different String attributes : value and referTo.

The meaning of this annotation is to specify which input class methods are used as getter and which as setter, and the fields to which they relate.

For get methods the annotation must look like: @MethodNote(value:"getter" referTo:"<variable-name>")

For set methods instead: @MethodNote(value:"setter" referTo:"<variable-name>").

Note that this feature could be handled in different and clearer ways, like use two different annotation for get and set methods. I made this choice just in order to try annotations containing more than a single attribute.

2.2 Class:

Reflector:

This is the main class that, after have asked at the user path and qualified name of the input class it proceed in the following way:

1. Get information about the input-class(package name, Class name and class annotation);
2. If the input-class cotains the @bean annotation create a new file called: "<input-class-name>Gen.java", else terminate the process;

```

if (notes.length != 0) {
    for (Annotation note : notes){
        if (note.annotationType().getSimpleName().
            toString().equals("bean")){
            returnState.add(true);
            Method[] ms = note.annotationType().getMethods();
            Method m= ms[0];
            if (m.getName().equals("Serializable")){
                try {
                    if(m.invoke(note).toString().
                        equals("true")){
                        returnState.add(true);
                    }
                } catch (Exception e) {}
            }else{returnState.add(false);}
        }
    }
}

```

Figure 2: @bean annotation parsing

3. Print on the generated file the initial code lines to define a class or a serializable class, depending on the class annotation. The “initial code” include package, class name, possible implementation and a public no-arguments constructor;
4. Look for constructors with arguments in the input-class and, if presents, print it/them in the new file (Figure 3);

```

Constructor[] cons = mystery.getConstructors();
for (Constructor c : cons) {
    Class[] parTypes = c.getParameterTypes();
    if (parTypes.length != 0) {
        createConstructor(c, parTypes, writer, mystery
            .getSimpleName().toString());
    }
}

```

Figure 3: Getting constructor with arguments

5. Get public and private fields through the method `Class.getDeclaredField` and then print them as private variables in the new java file;
6. Take all the class methods and check their annotation. If a class-input method contains a `MethodNote` annotation with the attribute value equals to getter or setter, the name of this method and the field which it refers to, will be saved in the appropriated `HashMap` (Figure 4).

```

case "value":
    switch (m.invoke(note).toString()) {
        case "getter":
            if (ms[0].getName() == "referTo") {
                createGetMethod(writer, m1.getReturnType()
                    .getSimpleName(), annotatedElementName,
                    ms[0].invoke(note).toString());
                getChecker.put(ms[0].invoke(note).
                    toString(), annotatedElementName);
            } else {reprintMethod(m1, writer);}
            break;
    }

```

Figure 4: case for get methods

This is made in order to know for which variable are provided get and set methods in the input class, in order to auto-generate the missing methods;

7. Print all the input-class methods (with arguments attached) in the new file;
8. Generate and print in the new file the missing get and set methods (Figure 5);

```

static void createGetMethod(PrintWriter writer, String type,
    String name, String variable) {
    writer.println("private " + type + " " + name + "(){");
    writer.println("return " + variable + ";");
    writer.println("}\n");
}

```

Figure 5: generate a missing get method for a variable

9. Print the end lines of the class and then terminate the process.

Note that since is not possible to get class and methods bodies of the input class, the new bean generated will contains just declarations and return statements of the methods written in a well

formed java bean document.

3.Deployment:

Since java reflection works on compiled java files, but the program does not provide compilation at runtime, the first step in order to compile:

- The pojo that will be converted;
- Bean.java(the class annotation);
- MethodNode.java(the method annotation).

Note that the annotated pojo must be compiled together the annotation files or the compilation it would be caused unknown annotation errors.

Furthermore if the input-class has been compiled together the whole application there is no the need to recompile it separately (the classes should be located in the same package).

To compile the classes digit:

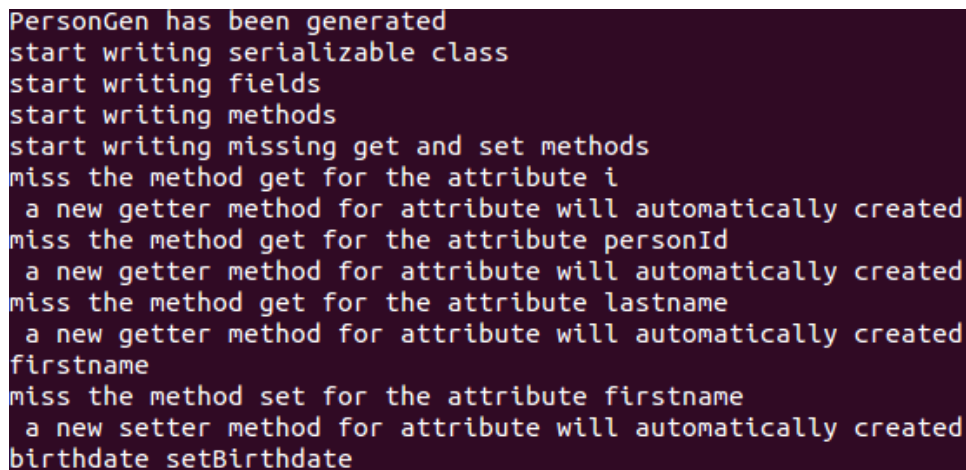
```
javac bean.java MethodNote.java <input-class>.java
```

Now the input-class is ready and the last thing to do is to start the application and then follow the instructions given at run time.

To execute the program use the command:

```
java -jar Inspector.jar
```

With the appropriate settings the console output should be look like (Figure 6):



```
PersonGen has been generated
start writing serializable class
start writing fields
start writing methods
start writing missing get and set methods
miss the method get for the attribute i
  a new getter method for attribute will automatically created
miss the method get for the attribute personId
  a new getter method for attribute will automatically created
miss the method get for the attribute lastname
  a new getter method for attribute will automatically created
firstname
miss the method set for the attribute firstname
  a new setter method for attribute will automatically created
birthdate setBirthdate
.
```

Figure 6

And a new file called <input-class>Gen will be created and located in the same path of the Inspector.jar.

Note that for testing the project, Inspector.jar already contain a compiled pojo called “Person” located in the reflectInspector.test package.