![](data:None;base64,)

* [Home](http://hannesdorfmann.com/)
* [Categories](http://hannesdorfmann.com/categories)
* [About](http://hannesdorfmann.com/about)
* [Tags](http://hannesdorfmann.com/tags)
* [Feed](http://hannesdorfmann.com/feed.xml)

In this blog entry I would like to explain how to write an annotation processor. So here is my tutorial. First, I am going to explain to you what annotation processing is, what you can do with that powerful tool and finally what you cannot do with it. In a second step we will implement a simple annotation processor step by step.

# **ANNOTATION PROCESSING 101**

by Hannes Dorfmann —

# **The Basics**

To clarify a very important thing from the very beginning: we are not talking about evaluating annotations by using reflections at runtime (run time = the time when the application runs). Annotation processing takes place at compile time (compile time = the time when the java compiler compiles your java source code).

Annotation processing is a tool build in javac for scanning and processing annotations **at compile time**. You can register your own annotation processor for certain annotations. At this point I assume that you already know what an annotation is and how to declare an annotation type. If you are not familar with annotations you can find more information in the [official java documentation](http://docs.oracle.com/javase/tutorial/java/annotations/index.html). Annotation processing is already available since Java 5 but a useable API is available since Java 6 (released in December 2006). It took some time until the java world realized the power of annotation processing. So it has become popular in the last few years.

An annotation processor for a certain annotation takes java code (or compiled byte code) as input and generate files (usually .java files) as output. What does that exactly means? You can generate java code! The generated java code is in a generated **.java** file. So you **can not**manipulate an existing java class for instance adding a method. The generated java file will be compiled by javac as any other hand written java source file.

# **AbstractProcessor**

Let’s have a look at the Processor API. Every Processor extends from **AbstractProcessor** as follows:

**package com.example;**

**public class MyProcessor extends AbstractProcessor {**

**@Override**

**public synchronized void init(ProcessingEnvironment env){ }**

**@Override**

**public boolean process(Set<? extends TypeElement> annoations, RoundEnvironment env) { }**

**@Override**

**public Set<String> getSupportedAnnotationTypes() { }**

**@Override**

**public SourceVersion getSupportedSourceVersion() { }**

**}**

* **init(ProcessingEnvironment env)**: Every annotation processor class **must have an empty constructor**. However, there is a special init() method which is invoked by the annotation processing tool with the **ProcessingEnviroment** as parameter. The ProcessingEnviroment provides some useful util classes **Elements**, **Types** and **Filer**. We will use them later.
* **process(Set<? extends TypeElement> annotations, RoundEnvironment env)**: This is kind of **main()** method of each processor. Here you write your code for scanning, evaluating and processing annotations and generating java files. With **RoundEnviroment**passed as parameter you can query for elements annotated with a certain annotation as we will see later.
* **getSupportedAnnotationTypes()**: Here you have to specify for which annotations this annotation processor should be registered for. Note that the return type is a set of strings containing full qualified names for your annotation types you want to process with this annotation processor. In other words, you define here for which annotations you register your annotation processor.
* **getSupportedSourceVersion()**: Used to specify which java version you use. Usually you will return **SourceVersion.latestSupported()**. However, you could also return **SourceVersion.RELEASE\_6** if you have good reasons for stick with Java 6. I recommend to use **SourceVersion.latestSupported();**

With Java 7 you could also use annotations instead of overriding **getSupportedAnnotationTypes()** and **getSupportedSourceVersion()** like that:

**@SupportedSourceVersion(SourceVersion.latestSupported())**

**@SupportedAnnotationTypes({**

**// Set of full qullified annotation type names**

**})**

**public class MyProcessor extends AbstractProcessor {**

**@Override**

**public synchronized void init(ProcessingEnvironment env){ }**

**@Override**

**public boolean process(Set<? extends TypeElement> annoations, RoundEnvironment env) { }**

**}**

For compatibility reasons, especially for android, I recommend to override **getSupportedAnnotationTypes()** and **getSupportedSourceVersion()** instead of using **@SupportedAnnotationTypes** and **@SupportedSourceVersion**

The next thing you have to know is that the annotation processor runs in it’s own jvm. Yes you read correctly. javac starts a complete java virtual machine for running annotation processors. So what that means for you? You can use anything you would use in any other java application. Use guava! If you want to you can use dependency injection tools like dagger or any other library you want to. But don’t forget. Even if it’s just a small processor you should take care about efficient algorithms and design patterns like you would do for any other java application.

# **Register Your Processor**

You may ask yourself “*How do I register MyProcessor to javac?”*. You have to provide a **.jar**file. Like any other .jar file you pack your (compiled) annotation processor in that file. Furthermore you also have to pack a special file called **javax.annotation.processing.Processor** located in **META-INF/services** in your .jar file. So the content of your .jar file looks like this:

**MyProcessor.jar**

**- com**

**- example**

**- MyProcessor.class**

**- META-INF**

**- services**

**- javax.annotation.processing.Processor**

The content of the file **javax.annotation.processing.Processor** (packed in MyProcessor.jar) is a list with full qualified class names to the processors with new line as delimiter:

**com.example.MyProcessor**

**com.foo.OtherProcessor**

**net.blabla.SpecialProcessor**

With **MyProcessor.jar** in your buildpath javac automatically detects and reads the **javax.annotation.processing.Processor** file and registers **MyProcessor** as annotation processor.

# **Example: Factory Pattern**

It’s time to for a concrete example. We will use maven as our build system and dependency management tool. If you are not familiar with maven, don’t worry maven is not necessary. The whole code can be found [on github](https://github.com/sockeqwe/annotationprocessing101).

First of all I have to say, that it’s not so easy to find a simple problem for a tutorial that we can solve with an annotation processor. Here we gonna implement a very simple factory pattern (not abstract factory pattern). It should give you just a brief introduction on the annotation processing API. So the problem statement may be a little bit dump and not a real world one. Once again, you will learn about annotation processing and not about design patterns.

So here is the problem: We want to implement a pizza store. The pizza store offers to it’s customers 2 Pizzas (“Margherita” and “Calzone”) and Tiramisu for dessert.

Have a look at this code snippets, which should not need any further explanation:

**public interface Meal {**

**public float getPrice();**

**}**

**public class MargheritaPizza implements Meal {**

**@Override public float getPrice() {**

**return 6.0f;**

**}**

**}**

**public class CalzonePizza implements Meal {**

**@Override public float getPrice() {**

**return 8.5f;**

**}**

**}**

**public class Tiramisu implements Meal {**

**@Override public float getPrice() {**

**return 4.5f;**

**}**

**}**

To order in our **PizzaStore** the customer has to enter the name of the meal:

**public class PizzaStore {**

**public Meal order(String mealName) {**

**if (mealName == null) {**

**throw new IllegalArgumentException("Name of the meal is null!");**

**}**

**if ("Margherita".equals(mealName)) {**

**return new MargheritaPizza();**

**}**

**if ("Calzone".equals(mealName)) {**

**return new CalzonePizza();**

**}**

**if ("Tiramisu".equals(mealName)) {**

**return new Tiramisu();**

**}**

**throw new IllegalArgumentException("Unknown meal '" + mealName + "'");**

**}**

**public static void main(String[] args) throws IOException {**

**PizzaStore pizzaStore = new PizzaStore();**

**Meal meal = pizzaStore.order(readConsole());**

**System.out.println("Bill: $" + meal.getPrice());**

**}**

**}**

As you see, we have a lot of *if statements* in the **order()** method and whenever we add a new type of pizza we have to add a new if statement. But wait, with annotation processing and the factory pattern we can let an annotation processor generate this *if statements*. So what we want to have is something like that:

**public class PizzaStore {**

**private MealFactory factory = new MealFactory();**

**public Meal order(String mealName) {**

**return factory.create(mealName);**

**}**

**public static void main(String[] args) throws IOException {**

**PizzaStore pizzaStore = new PizzaStore();**

**Meal meal = pizzaStore.order(readConsole());**

**System.out.println("Bill: $" + meal.getPrice());**

**}**

**}**

The **MealFactory** should look as follows:

**public class MealFactory {**

**public Meal create(String id) {**

**if (id == null) {**

**throw new IllegalArgumentException("id is null!");**

**}**

**if ("Calzone".equals(id)) {**

**return new CalzonePizza();**

**}**

**if ("Tiramisu".equals(id)) {**

**return new Tiramisu();**

**}**

**if ("Margherita".equals(id)) {**

**return new MargheritaPizza();**

**}**

**throw new IllegalArgumentException("Unknown id = " + id);**

**}**

**}**

## **@Factory Annotation**

Guess what: We want to generate the **MealFactory** by using annotation processing. To be more general, we want to provide an annotation and a processor for generating factory classes.

Let’s have a look at the **@Factory** annotation:

**@Target(ElementType.TYPE) @Retention(RetentionPolicy.CLASS)**

**public @interface Factory {**

**/\*\***

**\* The name of the factory**

**\*/**

**Class type();**

**/\*\***

**\* The identifier for determining which item should be instantiated**

**\*/**

**String id();**

**}**

The idea is that we annotate classes which should belong to the same factory with the same **type()** and with **id()** we do the mapping from “**Calzone”** to **CalzonePizza** class. Let’s apply **@Factory** to our classes:

**@Factory(**

**id = "Margherita",**

**type = Meal.class**

**)**

**public class MargheritaPizza implements Meal {**

**@Override public float getPrice() {**

**return 6f;**

**}**

**}**

**@Factory(**

**id = "Calzone",**

**type = Meal.class**

**)**

**public class CalzonePizza implements Meal {**

**@Override public float getPrice() {**

**return 8.5f;**

**}**

**}**

**@Factory(**

**id = "Tiramisu",**

**type = Meal.class**

**)**

**public class Tiramisu implements Meal {**

**@Override public float getPrice() {**

**return 4.5f;**

**}**

**}**

You may ask yourself if we could just apply **@Factory** on the **Meal** interface. Annotations are not inherited. Annotating **class X** with an annotation does not mean that **class Y extends X**is automatically annotated. Before we start writing the processor code we have to specify some rules:

1. Only classes can be annotated with **@Factory** since interfaces or abstract classes can not be instantiated with the **new** operator.
2. Classes annotated with **@Factory** must provide at least one public empty default constructor (parameterless). Otherwise we could not instantiate a new instance.
3. Classes annotated with **@Factory** must inherit directly or indirectly from the specified **type** (or implement it if it’s an interface).
4. **@Factory** annotations with the same **type** are grouped together and one Factory class will be generated. The name of the generated class has “*Factory”* as suffix, for example **type = Meal.class** will generate **MealFactory**
5. **id** are limited to Strings and must be unique in it’s type group.

## **The Processor**

I will guide you step by step by adding line of code followed by an explanation paragraph. Three dots (…) means that code is omitted either was discussed in the paragraph before or will be added later as next step. Goal is to make the snipped more readable. As already mentioned above the complete code can be found [on github](https://github.com/sockeqwe/annotationprocessing101). Ok lets start with the skeleton of our **FactoryProcessor**:

//Using AutoService annotation which help to auto generate service configuration file at compile time.

**@AutoService(Processor.class)**

**public class FactoryProcessor extends AbstractProcessor {**

**private Types typeUtils;**

**private Elements elementUtils;**

**private Filer filer;**

**private Messager messager;**

**private Map<String, FactoryGroupedClasses> factoryClasses = new LinkedHashMap<String, FactoryGroupedClasses>();**

**@Override**

**public synchronized void init(ProcessingEnvironment processingEnv) {**

**super.init(processingEnv);**

**typeUtils = processingEnv.getTypeUtils();**

**elementUtils = processingEnv.getElementUtils();**

**filer = processingEnv.getFiler();**

**messager = processingEnv.getMessager();**

**}**

**@Override**

**public Set<String> getSupportedAnnotationTypes() {**

**Set<String> annotataions = new LinkedHashSet<String>();**

**annotataions.add(Factory.class.getCanonicalName());**

**return annotataions;**

**}**

**@Override**

**public SourceVersion getSupportedSourceVersion() {**

**return SourceVersion.latestSupported();**

**}**

**@Override**

**public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {**

**...**

**}**

**}**

In the first line you see **@AutoService(Processor.class)**. What’s that? It’s an annotation from another annotation processor. This **AutoService** annotation processor has been developed by Google and generates the **META-INF/services/javax.annotation.processing.Processor**file. Yes, you read correctly. We can use annotation processors in our annotation processor. Handy, isn’t it? In **getSupportedAnnotationTypes()** we specify that **@Factory** is processed by this processor.

## **Elements and TypeMirrors**

In **init()** we retrieve a reference to

* **Elements**: A utils class to work with **Element** classes (more information later).
* **Types**: A utils class to work with **TypeMirror** (more information later)
* **Filer**: Like the name suggests with Filer you can create files.

In annotation processing we are scanning java source code files. Every part of the source code is a certain type of **Element**. In other words: **Element** represents a program element such as a package, class, or method. Each element represents a static, language-level construct. In the following example I have added comments to clarify that:

**package com.example; // PackageElement**

**public class Foo { // TypeElement**

**private int a; // VariableElement**

**private Foo other; // VariableElement**

**public Foo () {} // ExecuteableElement**

**public void setA ( // ExecuteableElement**

**int newA // TypeElement**

**) {}**

**}**

You have to change the way you see source code. It’s just structured text. It’s not executable. You can think of it like a XML file you try to parse (or an abstract syntax tree in compiler construction). Like in XML parsers there is some kind of DOM with elements. You can navigate from Element to it’s parent or child Element.

For instance if you have a **TypeElement** representing **public class Foo** you could iterate over its children like that:

**TypeElement fooClass = ... ;**

**for (Element e : fooClass.getEnclosedElements()){ // iterate over children**

**Element parent = e.getEnclosingElement(); // parent == fooClass**

**}**

As you see Elements are representing source code. TypeElement represent type elements in the source code like classes. However, TypeElement does not contain information about the class itself. From TypeElement you will get the name of the class, but you will not get information about the class like the superclass. This is kind of information are accessible through a **TypeMirror**. You can access the TypeMirror of an Element by calling **element.asType()**.

## **Searching For @Factory**

So lets implement the **process()** method step by step. First we start with searching for classes annotated with **@Factory**:

**@AutoService(Processor.class)**

**public class FactoryProcessor extends AbstractProcessor {**

**private Types typeUtils;**

**private Elements elementUtils;**

**private Filer filer;**

**private Messager messager;**

**private Map<String, FactoryGroupedClasses> factoryClasses = new LinkedHashMap<String, FactoryGroupedClasses>();**

**...**

**@Override**

**public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {**

**// Itearate over all @Factory annotated elements**

**for (Element annotatedElement : roundEnv.getElementsAnnotatedWith(Factory.class)) {**

**...**

**}**

**}**

**...**

**}**

No rocket science here. **roundEnv.getElementsAnnotatedWith(Factory.class))** returnes a list of Elements annotated with **@Factory**. You may have noted that I have avoited saying “*returns list of classes annotated with @Factory”*, because it really returns list of **Element**. Remember: **Element** can be a class, method, variable etc. So what we have to do next is to check if the Element is a class:

**@Override**

**public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {**

**for (Element annotatedElement : roundEnv.getElementsAnnotatedWith(Factory.class)) {**

**// Check if a class has been annotated with @Factory**

**if (annotatedElement.getKind() != ElementKind.CLASS) {**

**...**

**}**

**}**

**...**

**}**

What’s going on here? We want to ensure that only elements of type class are processed by our processor. Previously we have learned that classes are **TypeElements**. So why don’t we check **if (! (annotatedElement instanceof TypeElement) )**. That’s a wrong assumption because interfaces are TypeElement as well. So in annoation processing you should avoid *instanceof* but rather us [ElementKind](http://docs.oracle.com/javase/7/docs/api/javax/lang/model/element/ElementKind.html) or [TypeKind](http://docs.oracle.com/javase/7/docs/api/javax/lang/model/type/TypeKind.html) with TypeMirror.

## **Error Handling**

In **init()** we also retrieve a reference to **Messager**. A Messager provides the way for an annotation processor to report error messages, warnings and other notices. It’s not a logger for you, the developer of the annotation processor (even thought it can be used for that during development of the processor). Messager is used to write messages to the third party developer who uses your annotation processor in their projects. There are different levels of messages described in the [official docs](http://docs.oracle.com/javase/7/docs/api/javax/tools/Diagnostic.Kind.html). Very important is [Kind.ERROR](http://docs.oracle.com/javase/7/docs/api/javax/tools/Diagnostic.Kind.html" \l "ERROR) because this kind of message is used to indicate that our annotation processor has failed processing. Probably the third party developer is misusing our **@Factory** annotation (i.e. annotated an interface with @Factory). The concept is a little bit different from traditional java application where you would throw an **Exception**. If you throw an exception in **process()** then the jvm which runs annotation processing crashs (like any other java application) and the third party developer who is using our FactoryProcessor will get an error from javac with a hardly understandable Exception, because it contains the stacktrace of FactoryProcessor. Therefore Annotation Processor has this **Messager** class. It prints a pretty error message. Additionaly, you can link to the element who has raised this error. In modern IDEs like IntelliJ the third party developer can click on this error message and the IDE will jump to the source file and line of the third party developers project where the error source is.

Back to implementing the **process()** method. We raise a error message if the user has annotated a non class with @Factory:

**public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {**

**for (Element annotatedElement : roundEnv.getElementsAnnotatedWith(Factory.class)) {**

**// Check if a class has been annotated with @Factory**

**if (annotatedElement.getKind() != ElementKind.CLASS) {**

**error(annotatedElement, "Only classes can be annotated with @%s",**

**Factory.class.getSimpleName());**

**return true; // Exit processing**

**}**

**...**

**}**

**private void error(Element e, String msg, Object... args) {**

**messager.printMessage(**

**Diagnostic.Kind.ERROR,**

**String.format(msg, args),**

**e);**

**}**

**}**

To get the message of the Messager displayed it’s **important that the annotation processor has to complete without crashing**. That’s why we **return** after having called **error()**. If we don’t return here **process()** will continue running since**messager.printMessage( Diagnostic.Kind.ERROR)** does not stop the process. So it’s very likely that if we don’t return after printing the error we will run in an internal *NullPointerException* etc. if we continue in **process()**. As said before, the problem is that if an unhandled exception is thrown in **process()** javac will print the stacktrace of the internal *NullPointerException* and NOT your error message of **Messager**.

## **Datamodel**

Before we continue with checking if classes annotated with @Factory observe our five rules (see above) we are going to introduce data structures which makes it easier for us to continue. Sometimes the problem or processor seems to be so simple that programmers tend to write the whole processor in a procedural manner. **But you know what? An Annotation Processor is still a java application. So use object oriented programming, interfaces, design patterns and anything else you would use in any other java application!**

Our FactoryProcessor is quite simple but there are some information we want to store as objects. With **FactoryAnnotatedClass** we store the annotated class data like qualified class name along with the data of the @Factory annotation itself. So we store the TypeElement and evaluate the @Factory annotation:

**public class FactoryAnnotatedClass {**

**private TypeElement annotatedClassElement;**

**private String qualifiedSuperClassName;**

**private String simpleTypeName;**

**private String id;**

**public FactoryAnnotatedClass(TypeElement classElement) throws IllegalArgumentException {**

**this.annotatedClassElement = classElement;**

**Factory annotation = classElement.getAnnotation(Factory.class);**

**id = annotation.id();**

**if (StringUtils.isEmpty(id)) {**

**throw new IllegalArgumentException(**

**String.format("id() in @%s for class %s is null or empty! that's not allowed",**

**Factory.class.getSimpleName(), classElement.getQualifiedName().toString()));**

**}**

**// Get the full QualifiedTypeName**

**try {**

**Class<?> clazz = annotation.type();**

**qualifiedSuperClassName = clazz.getCanonicalName();**

**simpleTypeName = clazz.getSimpleName();**

**} catch (MirroredTypeException mte) {**

**DeclaredType classTypeMirror = (DeclaredType) mte.getTypeMirror();**

**TypeElement classTypeElement = (TypeElement) classTypeMirror.asElement();**

**qualifiedSuperClassName = classTypeElement.getQualifiedName().toString();**

**simpleTypeName = classTypeElement.getSimpleName().toString();**

**}**

**}**

**/\*\***

**\* Get the id as specified in {@link Factory#id()}.**

**\* return the id**

**\*/**

**public String getId() {**

**return id;**

**}**

**/\*\***

**\* Get the full qualified name of the type specified in {@link Factory#type()}.**

**\***

**\* @return qualified name**

**\*/**

**public String getQualifiedFactoryGroupName() {**

**return qualifiedSuperClassName;**

**}**

**/\*\***

**\* Get the simple name of the type specified in {@link Factory#type()}.**

**\***

**\* @return qualified name**

**\*/**

**public String getSimpleFactoryGroupName() {**

**return simpleTypeName;**

**}**

**/\*\***

**\* The original element that was annotated with @Factory**

**\*/**

**public TypeElement getTypeElement() {**

**return annotatedClassElement;**

**}**

**}**

Lot of code, but the most important thing happens ins the constructor where you find the following lines of code:

**Factory annotation = classElement.getAnnotation(Factory.class);**

**id = annotation.id(); // Read the id value (like "Calzone" or "Tiramisu")**

**if (StringUtils.isEmpty(id)) {**

**throw new IllegalArgumentException(**

**String.format("id() in @%s for class %s is null or empty! that's not allowed",**

**Factory.class.getSimpleName(), classElement.getQualifiedName().toString()));**

**}**

Here we access the @Factory annotation and check if the id is not empty. We will throw an IllegalArgumentException if id is empty. You may be confused now because previously we said that we are not throwing exceptions but rather use **Messager**. That’s still correct. We throw an exception here internally and we will catch that one in **process()** as you will see later. We do that for two reasons:

1. I want to demonstrate that you should still code like in any other java application. Throwing and catching exceptions is considered as good practice in java.
2. If we want to print a message right from **FactoryAnnotatedClass** we also have to pass the **Messager** and as already mentioned in “Error Handling” (scroll up) the processor has to terminate successfully to make **Messager** print the error message. So if we would write an error message by using **Messager** how do we “notify” **process()** that an error has occurred? The easiest and from my point of view most intuitive way is to throw an Exception and let **process()** chatch this one.

Next we want to get the **type** field of the **@Factory** annotation. We are interessted in the full qualified name.

**try {**

**Class<?> clazz = annotation.type();**

**qualifiedGroupClassName = clazz.getCanonicalName();**

**simpleFactoryGroupName = clazz.getSimpleName();**

**} catch (MirroredTypeException mte) {**

**DeclaredType classTypeMirror = (DeclaredType) mte.getTypeMirror();**

**TypeElement classTypeElement = (TypeElement) classTypeMirror.asElement();**

**qualifiedGroupClassName = classTypeElement.getQualifiedName().toString();**

**simpleFactoryGroupName = classTypeElement.getSimpleName().toString();**

**}**

That’s a little bit tricky, because the type is **java.lang.Class**. That means, that this is a real Class object. Since annotation processing runs before compiling java source code we have to consider two cases:

1. **The class is already compiled**: This is the case if a third party .jar contains compiled .class files with **@Factory** annotations. In that case we can directly access the Class like we do in the **try-block**.
2. **The class is not compiled yet**: This will be the case if we try to compile our source code which has @Factory annotations. Trying to access the Class directly throws a **MirroredTypeException**. Fortunately MirroredTypeException contains a **TypeMirror**representation of our not yet compiled class. Since we know that it must be type of class (we have already checked that before) we can cast it to **DeclaredType** and access **TypeElement** to read the qualified name.

Alright, now we need one more datastructure named **FactoryGroupedClasses** which basically groups all **FactoryAnnotatedClasses** together.

**public class FactoryGroupedClasses {**

**private String qualifiedClassName;**

**private Map<String, FactoryAnnotatedClass> itemsMap =**

**new LinkedHashMap<String, FactoryAnnotatedClass>();**

**public FactoryGroupedClasses(String qualifiedClassName) {**

**this.qualifiedClassName = qualifiedClassName;**

**}**

**public void add(FactoryAnnotatedClass toInsert) throws IdAlreadyUsedException {**

**FactoryAnnotatedClass existing = itemsMap.get(toInsert.getId());**

**if (existing != null) {**

**throw new IdAlreadyUsedException(existing);**

**}**

**itemsMap.put(toInsert.getId(), toInsert);**

**}**

**public void generateCode(Elements elementUtils, Filer filer) throws IOException {**

**...**

**}**

**}**

As you see it’s basically just a **Map<String, FactoryAnnotatedClass>**. This map is used to map an @Factory.id() to FactoryAnnotatedClass. We have chosen **Map** because we want to ensure that each id is unique. That can be easily done with a map lookup. **generateCode()**will be called to generate the Factory code (discussed later).

## **Matching Criteria**

Let’s proceed with the implementation of **process()**. Next we want to check if the annotated class has at least one public constructor, is not an abstract class, inherits the specified type and is a public class (visibility):

**public class FactoryProcessor extends AbstractProcessor {**

**@Override**

**public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {**

**for (Element annotatedElement : roundEnv.getElementsAnnotatedWith(Factory.class)) {**

**...**

**// We can cast it, because we know that it of ElementKind.CLASS**

**TypeElement typeElement = (TypeElement) annotatedElement;**

**try {**

**FactoryAnnotatedClass annotatedClass =**

**new FactoryAnnotatedClass(typeElement); // throws IllegalArgumentException**

**if (!isValidClass(annotatedClass)) {**

**return true; // Error message printed, exit processing**

**}**

**} catch (IllegalArgumentException e) {**

**// @Factory.id() is empty**

**error(typeElement, e.getMessage());**

**return true;**

**}**

**...**

**}**

**private boolean isValidClass(FactoryAnnotatedClass item) {**

**// Cast to TypeElement, has more type specific methods**

**TypeElement classElement = item.getTypeElement();**

**if (!classElement.getModifiers().contains(Modifier.PUBLIC)) {**

**error(classElement, "The class %s is not public.",**

**classElement.getQualifiedName().toString());**

**return false;**

**}**

**// Check if it's an abstract class**

**if (classElement.getModifiers().contains(Modifier.ABSTRACT)) {**

**error(classElement, "The class %s is abstract. You can't annotate abstract classes with @%",**

**classElement.getQualifiedName().toString(), Factory.class.getSimpleName());**

**return false;**

**}**

**// Check inheritance: Class must be childclass as specified in @Factory.type();**

**TypeElement superClassElement =**

**elementUtils.getTypeElement(item.getQualifiedFactoryGroupName());**

**if (superClassElement.getKind() == ElementKind.INTERFACE) {**

**// Check interface implemented**

**if (!classElement.getInterfaces().contains(superClassElement.asType())) {**

**error(classElement, "The class %s annotated with @%s must implement the interface %s",**

**classElement.getQualifiedName().toString(), Factory.class.getSimpleName(),**

**item.getQualifiedFactoryGroupName());**

**return false;**

**}**

**} else {**

**// Check subclassing**

**TypeElement currentClass = classElement;**

**while (true) {**

**TypeMirror superClassType = currentClass.getSuperclass();**

**if (superClassType.getKind() == TypeKind.NONE) {**

**// Basis class (java.lang.Object) reached, so exit**

**error(classElement, "The class %s annotated with @%s must inherit from %s",**

**classElement.getQualifiedName().toString(), Factory.class.getSimpleName(),**

**item.getQualifiedFactoryGroupName());**

**return false;**

**}**

**if (superClassType.toString().equals(item.getQualifiedFactoryGroupName())) {**

**// Required super class found**

**break;**

**}**

**// Moving up in inheritance tree**

**currentClass = (TypeElement) typeUtils.asElement(superClassType);**

**}**

**}**

**// Check if an empty public constructor is given**

**for (Element enclosed : classElement.getEnclosedElements()) {**

**if (enclosed.getKind() == ElementKind.CONSTRUCTOR) {**

**ExecutableElement constructorElement = (ExecutableElement) enclosed;**

**if (constructorElement.getParameters().size() == 0 && constructorElement.getModifiers()**

**.contains(Modifier.PUBLIC)) {**

**// Found an empty constructor**

**return true;**

**}**

**}**

**}**

**// No empty constructor found**

**error(classElement, "The class %s must provide an public empty default constructor",**

**classElement.getQualifiedName().toString());**

**return false;**

**}**

**}**

We have added a method **isValidClass()** and it checks if our rules are complied:

* Class must be public: **classElement.getModifiers().contains(Modifier.PUBLIC)**
* Class can not be abstract: **classElement.getModifiers().contains(Modifier.ABSTRACT)**
* Class must be subclass or implement the Class as specified in **@Factoy.type()**. First we use **elementUtils.getTypeElement(item.getQualifiedFactoryGroupName())** to create a Element of the passed **Class** (**@Factoy.type()**). Yes you got it, you can create **TypeElement**(with TypeMirror) just by knowing the qualified class name. Next we check if it’s an interface or a class: **superClassElement.getKind() == ElementKind.INTERFACE**. So we have two cases: If it’s an interfaces then **classElement.getInterfaces().contains(superClassElement.asType())**. If it’s a class, then we have to scan the inheritance hierarchy with calling **currentClass.getSuperclass()**. Note that this check could also be done with **typeUtils.isSubtype()**.
* Class must have a public empty constructor: So we iterate over all enclosed elements **classElement.getEnclosedElements()** and check for **ElementKind.CONSTRUCTOR**, **Modifier.PUBLIC** and **constructorElement.getParameters().size() == 0**

If these conditions are fulfilled **isValidClass()** returns true, otherwise it prints a error message and returns false.

## **Grouping The Annotated Classes**

Once we have checked **isValidClass()** we continue with adding **FactoryAnnotatedClass** to the corresponding **FactoryGroupedClasses** as follows:

**public class FactoryProcessor extends AbstractProcessor {**

**private Map<String, FactoryGroupedClasses> factoryClasses =**

**new LinkedHashMap<String, FactoryGroupedClasses>();**

**@Override**

**public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {**

**...**

**try {**

**FactoryAnnotatedClass annotatedClass =**

**new FactoryAnnotatedClass(typeElement); // throws IllegalArgumentException**

**if (!isValidClass(annotatedClass)) {**

**return true; // Error message printed, exit processing**

**}**

**// Everything is fine, so try to add**

**FactoryGroupedClasses factoryClass =**

**factoryClasses.get(annotatedClass.getQualifiedFactoryGroupName());**

**if (factoryClass == null) {**

**String qualifiedGroupName = annotatedClass.getQualifiedFactoryGroupName();**

**factoryClass = new FactoryGroupedClasses(qualifiedGroupName);**

**factoryClasses.put(qualifiedGroupName, factoryClass);**

**}**

**// Throws IdAlreadyUsedException if id is conflicting with**

**// another @Factory annotated class with the same id**

**factoryClass.add(annotatedClass);**

**} catch (IllegalArgumentException e) {**

**// @Factory.id() is empty --> printing error message**

**error(typeElement, e.getMessage());**

**return true;**

**} catch (IdAlreadyUsedException e) {**

**FactoryAnnotatedClass existing = e.getExisting();**

**// Already existing**

**error(annotatedElement,**

**"Conflict: The class %s is annotated with @%s with id ='%s' but %s already uses the same id",**

**typeElement.getQualifiedName().toString(), Factory.class.getSimpleName(),**

**existing.getTypeElement().getQualifiedName().toString());**

**return true;**

**}**

**}**

**...**

**}**

## **Code Generation**

We have collected all classes annotated with **@Factory** stored as **FactoryAnnotatedClass**and grouped into **FactoryGroupedClasses**. Now we are going to generate java files for each Factory:

**@Override**

**public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {**

**...**

**try {**

**for (FactoryGroupedClasses factoryClass : factoryClasses.values()) {**

**factoryClass.generateCode(elementUtils, filer);**

**}**

**} catch (IOException e) {**

**error(null, e.getMessage());**

**}**

**return true;**

**}**

Writing a java file is pretty the same as writing any other file in java. We use an **Writer**provided by **Filer**. We could write our generated code as concatination of Strings. Fortunately, Square Inc. well known for plenty awesome open source projects gives us with [JavaWriter](https://github.com/square/javawriter) a high level library for generating Java Code:

**public class FactoryGroupedClasses {**

**/\*\***

**\* Will be added to the name of the generated factory class**

**\*/**

**private static final String SUFFIX = "Factory";**

**private String qualifiedClassName;**

**private Map<String, FactoryAnnotatedClass> itemsMap =**

**new LinkedHashMap<String, FactoryAnnotatedClass>();**

**...**

**public void generateCode(Elements elementUtils, Filer filer) throws IOException {**

**TypeElement superClassName = elementUtils.getTypeElement(qualifiedClassName);**

**String factoryClassName = superClassName.getSimpleName() + SUFFIX;**

**JavaFileObject jfo = filer.createSourceFile(qualifiedClassName + SUFFIX);**

**Writer writer = jfo.openWriter();**

**JavaWriter jw = new JavaWriter(writer);**

**// Write package**

**PackageElement pkg = elementUtils.getPackageOf(superClassName);**

**if (!pkg.isUnnamed()) {**

**jw.emitPackage(pkg.getQualifiedName().toString());**

**jw.emitEmptyLine();**

**} else {**

**jw.emitPackage("");**

**}**

**jw.beginType(factoryClassName, "class", EnumSet.of(Modifier.PUBLIC));**

**jw.emitEmptyLine();**

**jw.beginMethod(qualifiedClassName, "create", EnumSet.of(Modifier.PUBLIC), "String", "id");**

**jw.beginControlFlow("if (id == null)");**

**jw.emitStatement("throw new IllegalArgumentException(\"id is null!\")");**

**jw.endControlFlow();**

**for (FactoryAnnotatedClass item : itemsMap.values()) {**

**jw.beginControlFlow("if (\"%s\".equals(id))", item.getId());**

**jw.emitStatement("return new %s()", item.getTypeElement().getQualifiedName().toString());**

**jw.endControlFlow();**

**jw.emitEmptyLine();**

**}**

**jw.emitStatement("throw new IllegalArgumentException(\"Unknown id = \" + id)");**

**jw.endMethod();**

**jw.endType();**

**jw.close();**

**}**

**}**

**Tipp:** Since JavaWriter is very very popular there are many other processors, libraries and tools depending on JavaWriter. This maybe will cause problems if you use dependency management tools like maven or gradle if one library depends on a newer version of JavaWriter as another one. Therefore I recommend to copy and repackage JavaWriter directly into your annotation processor code base (it’s just one java file).

**Update:** Use [JavaPoet](https://github.com/square/javapoet) instead of JavaWriter.

## **Processing Rounds**

Annotation Processing maybe takes more than one processing round. The official javadoc define processing like as follows:

Annotation processing happens in a sequence of rounds. On each round, a processor may be asked to process a subset of the annotations found on the source and class files produced by a prior round. The inputs to the first round of processing are the initial inputs to a run of the tool; these initial inputs can be regarded as the output of a virtual zeroth round of processing.

A simpler definition: A processing round is calling **process()** of an annotation processor. To stick with our factory sample: **FactoryProcessor** is instantiated once (new Processor object is **not** created for each round), but **process()** can be called multiple times, if new source files has been created. Sounds a little bit strange, doesn’t it? The reason is that, the generated source code files could contain **@Factory** annotated classes as well, which would be processed by **FactoryProcessor**.

For example our **PizzaStore** sample will be processed in 3 rounds:

|  |  |  |
| --- | --- | --- |
| Round | Input | Output |
| 1 | CalzonePizza.java  Tiramisu.java MargheritaPizza.java Meal.java PizzaStore.java | MealFactory.java |
| 2 | MealFactory.java | --- none --- |
| 3 | --- none --- | --- none --- |

There is another reason why I’m explaining processing rounds. If you look at our **FactoryProcessor** code you see that we collect data and store them in private field **Map<String, FactoryGroupedClasses> factoryClasses**. In first round we detect MagheritaPizza, CalzonePizza and Tiramisu and we generate a file MealFactory.java. In the second round we take MealFactory as input. Since there is no **@Factory** annotation in MealFactory no data will be collected, we would not expect to get an error. However we get one:

**Attempt to recreate a file for type com.hannesdorfmann.annotationprocessing101.factory.MealFactory**

The problem is that we never clear **factoryClasses**. That means, that in round two **process()**has still stored data from the first round and wants to generate the same file as round 1 already did which will cause this error. In our case, we know that only in the first round we will detect **@Factory** annotated classes and therefore we can simply fix it like that:

**@Override**

**public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {**

**try {**

**for (FactoryGroupedClasses factoryClass : factoryClasses.values()) {**

**factoryClass.generateCode(elementUtils, filer);**

**}**

**// Clear to fix the problem**

**factoryClasses.clear();**

**} catch (IOException e) {**

**error(null, e.getMessage());**

**}**

**...**

**return true;**

**}**

I know there are other ways to deal with that problem i.e. we could also set a boolean flag etc. The point is: Keep in mind that annotation processing is done in multiple processing rounds and you can not override or recreate already generated source files.

## **Separation of processor and annotation**

If you have looked at the [git repository](https://github.com/sockeqwe/annotationprocessing101/tree/master/factory) of our factory processor you will see that we have organized our repository in two maven modules. We did that, because we want to give the user of our Factory example the possibility to compile just the annotation in his own project and to include the processor module just for compilation. Confused? I.e. if we would have just one single artifact another developer who wants to use our factory processor in his own project would include both the **@Factory** annotation and the whole **FactoryProcessor** code (incl. FactoryAnnotatedClass and FactoryGroupedClasses) in his project build. I’m pretty sure that the other one won’t have the processor class in his compiled project. If you are an android developer maybe you have heard of the 65k method limit (a android .dex file can only address 65.000 methods). If we would have used guava in FactoryProcessor and would provide just a single artifact containing annotation and processor code, then the android apk would not only contain FactoryProcessor code but the whole guava code as well. Guava has about 20.000 methods. Therefore a separation of annotation and processor makes sense.

## **Instantiation Of Generated Classes**

As you have seen in **PizzaStore** sample the generated class **MealFactory** is a normal java class as any other handwritten class. Furthermore, you have to instantiate it by hand (like any other java object).

**public class PizzaStore {**

**private MealFactory factory = new MealFactory();**

**public Meal order(String mealName) {**

**return factory.create(mealName);**

**}**

**...**

**}**

If you are an android developer you should be familiar with a great annotation processors called [ButterKnife](http://jakewharton.github.io/butterknife/). In ButterKnife you annotate android Views with **@InjectView**. The ButterKnifeProcessor generates a class **MyActivityViewInjector()** but you can use **Butterknife.inject(activity)**. ButterKnife internally uses reflections to instantiate **MyActivity$$ViewInjector()**:

**try {**

**Class<?> injector = Class.forName(clsName + "$$ViewInjector");**

**} catch (ClassNotFoundException e) { ... }**

But is reflection not slow and didn’t we tried to get rich of reflections performance issues by using annotation processing which generates native code? Yes, reflection brings perfomance issues. However, it speeds up development because the developer has not to instantiate objects by hand. ButterKnife uses a HashMap to “cache” the instantiated objects. So **MyActivityViewInjector** is needed it will be retrieved from the HashMap.

[FragmentArgs](https://github.com/sockeqwe/fragmentargs) works similar to ButterKnife. It uses reflection to instantiate things that otherwise the developer who uses FragmentArgs has to do manually. FragmentArgs generates a special “lookup” class while annotation processing which is kind of HashMap. So the whole FragmentArgs library executes only one reflection call at the very first time to instantiate this special HashMap class. Once this class is instantiated with **Class.forName()**fragment arguments injection runs in native java code.

All in all it’s up to you (the developer of annotation processor) to find a good compromise between reflection and usability for other users of your annotation processor.

# **Conclusion**

I hope you have a deeper understanding of annotation processing now. I have to say it once again: Annotation processing is a very powerful tool and helps reducing writing boiler plate code. I also want to mention that with annotation processors you can do much more complex things than I show in this simple factory sample, for example type erasure on Generics, because annotation processing happens before type erasure. As you have seen there are two common problems you have to deal when writing: First, if you want to use ElementUtils, TypeUtils and Messager in other classes then you have to pass them as parameters somehow. In [AnnotatedAdapter](https://github.com/sockeqwe/AnnotatedAdapter), one of my annotation processors for android, I tried to solve that problem with Dagger (Dependency Injection). It feels a little bit to much overhead for such a simple processor, however it has worked well. The second thing you have to deal is you have to make “queries” for **Elements**. As I said before, working with elements can be seen as parsing XML or HTML. For HTML you can use jQuery. Something similar to jQuery for annotation processing would be really awesome. Please comment below if you know any similar library.

**Please note that parts of the code of FactoryProcessor has some edges and pitfalls. These “mistakes” are placed explicit by me to struggle through them as I explain common mistakes while writing annotation processors (like “Attempt to recreate a file”). If you start writing your own annotation processor based on FactoryProcessor DON’T copy and paste this pitfalls. Instead you should avoid them from the very beginning.**

In a future blog post (Annotation Processing 102) I will write about unit testing annotation processors. However, my next blog post will be about software architecture in android. Stay tuned.

### **Update**

I gave a talk about Annotation Processing at droidcon Germany 2015. The video of my talk can be found at [youtube](https://www.youtube.com/watch?v=43FFfTyDYEg):

*Author*

# **HANNES DORFMANN**

Enthusiastic Android Developer, Sports Maniac, Java Hacker, Web Engineer and Open Source Lover.

# **COMMENTS**

[Hannes Dorfmann](http://hannesdorfmann.com/) • All rights reserved