

Java-2

ANNOTATIONS IN COMPILE TIME


```
package com.example;
```

```
public class MyProcessor extends AbstractProcessor {
```

```
    @Override
```

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

```
    @Override
```

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

```
    @Override
```

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

```
    @Override
```

```
    public SourceVersion getSupportedSourceVersion() { }
```

```
}
```

init

`init(ProcessingEnvironment env):`

- У каждого `annotation processor` обязан быть пустой конструктор.
- Однако есть специальный `init()` метод, который вызывается с `ProcessingEnvironment` в качестве параметра.
- `ProcessingEnvironment` предоставляет доступ к некоторым утилитарным классам `Elements`, `Types` и `Filer`.

process

- `process(Set<? extends TypeElement> annotations, RoundEnvironment env):`
- Это своего рода метод `main()` каждого процессора. Здесь пишется код для сканирования, вычисления и обработки аннотаций. А также для генерации java файлов. `RoundEnvironment` передаваемый в качестве параметра используется для перечисления всех элементов имеющих определенную аннотацию.

getSupportedAnnotationTypes

- `getSupportedAnnotationTypes()`:
- Здесь необходимо перечислить аннотации, на которые «подписывается» наш процессор.

getSupportedSourceVersion

- `getSupportedSourceVersion()`:
- Используется для определения поддерживаемой версии Java. Обычно возвращается `SourceVersion.latestSupported()`. Хотя можно возвращать, например, `SourceVersion.RELEASE_6`

Java 8 style

```
@SupportedSourceVersion(SourceVersion.latestSupported())
@SupportedAnnotationTypes({
    // Set of full qualified annotation type names
})
public class MyProcessor extends AbstractProcessor {

    @Override
    public synchronized void init(ProcessingEnvironment env){ }

    @Override
    public boolean process(Set<? extends TypeElement> annotations,
        RoundEnvironment env) { }
}
```


Для работы необходимо собрать jar

MyProcessor.jar

- com

- example

- MyProcessor.class

- META-INF

- services

- javax.annotation.processing.Processor

- (содержимое:

- com.example.MyProcessor

- com.foo.OtherProcessor

- net.blabla.SpecialProcessor

-)

Пример

FACTORY PATTERN

Различные виды пиццы

```
public interface Meal {  
    public float getPrice();  
}  
  
public class MargheritaPizza implements Meal {  
    public float getPrice() {  
        return 6.0f;  
    }  
}  
  
public class CalzonePizza implements Meal {  
    public float getPrice() {  
        return 8.5f;  
    }  
}  
  
public class Tiramisu implements Meal {  
    public float getPrice() {  
        return 4.5f;  
    }  
}
```

Магазин пиццы

```
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());  
    }  
}
```

Вынесем фабрику в отдельный класс

```
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());  
    }  
}
```

MealFactory

```
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

```
@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();
}
```

```
@Factory(  
    id = "Margherita",  
    type = Meal.class  
)  
public class MargheritaPizza implements Meal {  
    @Override public float getPrice() {  
        return 6f;  
    }  
}
```


Договоримся о правилах

1. Только классы (не интерфейсы и не абстрактные классы) могут быть проаннотированы `@Factory`
2. Классы проаннотированные этой аннотацией должны иметь публичный конструктор без параметров (иначе мы не сможем их создавать)
3. Должна быть возможность привести аннотированные классы к «типу» указанному в параметре аннотации
4. Классы проаннотированные `@Factory` с одинаковым `type` будут собраны в единый класс-фабрику. Например, для `type Meal` будет создан класс `MealFactory`
5. `Id` – это строка. Она должна быть уникальна для классов с одинаковым `type`

Приступаем

```
@AutoService(Processor.class)
public class FactoryProcessor extends AbstractProcessor {
    @Override
    public Set<String> getSupportedAnnotationTypes() {
        Set<String> annotations = new LinkedHashSet<String>();
        annotations.add(Factory.class.getCanonicalName());
        return annotations;
    }
    @Override
    public SourceVersion getSupportedSourceVersion() {
        return SourceVersion.latestSupported();
    }
    .....
}
```

- `@AutoService(Processor.class)`
- Это специальная аннотация, которая автоматически генерирует манифест-файл

Приступаем

```
@AutoService(Processor.class)
public class FactoryProcessor extends AbstractProcessor {
    private Types typeUtils; private Elements elementUtils;
    private Filer filer;      private Messenger messenger;
    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();
        messenger = processingEnv.getMessenger();
    }
    public Set<String> getSupportedAnnotationTypes() {...}
    public SourceVersion getSupportedSourceVersion() {...}
    ...
}
```

Init

- В методе `init` мы получаем ссылки на:
 - `Elements` – утилитарный класс для работы с классами `Element`
 - `Type` – утилитарный класс для работы с `TypeMirror`
 - `Filer` – класс позволяющий нам удобно создавать файлы.

Init

Annotation Processor сканирует исходный код.
Каждая часть исходного кода представляет из себя
какой-то конкретный Element.

```
package com.example;      // PackageElement

public class Foo {        // TypeElement

    private int a;        // VariableElement
    private Foo other;    // VariableElement

    public Foo () {}      // ExecutableElement

    public void setA (    // ExecutableElement
        int newA // TypeElement
    ) {}
}
```

Как обходить элементы?

- Например:

```
TypeElement fooClass = ... ;  
// iterate over children  
for (Element e : fooClass.getEnclosedElements()){  
    // parent == fooClass  
    Element parent = e.getEnclosingElement();  
}
```


- `Element` – это описание элемента исходного кода. Т.е. имея в руках `TypeElement` мы не можем, например, получить информацию о родительском классе и т.д.
- Для всего этого есть `TypeMirror` – т.е. сопоставление элементов и реальных типов языка.
- `element.asType()`

Поиск аннотации @Factory

```
@AutoService(Processor.class)
public class FactoryProcessor extends AbstractProcessor {
    ...
    @Override
    public boolean process(Set<? extends TypeElement> annotations,
        RoundEnvironment roundEnv) {
        // Iterate over all @Factory annotated elements
        for (Element annotatedElement :
            roundEnv.getElementsAnnotatedWith(Factory.class)) {
            ...
        }
    }
    ...
}
```

Проверим, что аннотация там, где нужно

```
@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) {
            ...
        }
    }
    ...
}
```

```
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) {
        messenger.printMessage (
            Diagnostic.Kind.ERROR, String.format(msg, args), e);
    }
}
```

Лирическое отступление

- **FactoryAnnotatedClass**
- Это класс описывающий один проаннотированный класс
- **FactoryGroupedClasses**
- Это класс описывающий набор проаннотированных классов сгруппированных по type.

Лирическое отступление

```
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(); // 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()));
        }
        // 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();
        }
    }
}
```

Лирическое отступление

```
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();  
}  
}
```

Здесь рассматриваются два случая:

- Класс `Factory` уже откомпилирован. Тогда можем добраться до него уже напрямую.
- Класс `Factory` еще не откомпилирован. Тогда необходимо использовать `TypeMirror`.

FactoryGroupedClasses

```
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 {
        ...
    }
}
```


Возвращаемся к Processor

```
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;
            }
            ...
        }
    }
}
```

Как проверять класс на валидность?

```
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;  
    }  
}
```

Как проверять класс на валидность?

```
// 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 {
    TypeElement currentClass = classElement; // Check subclassing
    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())) {
            break; // Required super class found
        }
        currentClass = (TypeElement) typeUtils.asElement(superClassType); // Moving up in inheritance tree
    }
}
```

Как проверять класс на валидность?

```
// 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;
}
```

Что мы проверяем

- 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 `@Factory.type()`.
First we use `elementUtils.getTypeElement(item.getQualifiedFactoryGroupName())` to create a `Element` of the passed Class (`@Factory.type()`).
Yes 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`

Продолжаем Processor

```
try {
    FactoryAnnotatedClass annotatedClass = new FactoryAnnotatedClass(typeElement);
    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);
    }
    factoryClass.add(annotatedClass);
} catch (IllegalArgumentException e) {
    // @Factory.id() is empty --> printing error message
} catch (IdAlreadyUsedException e) {
    // Already existing
}
}
```

Code generation

```
@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;
}
```

Как генерировать код?

```
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();
}
```

Processing Rounds

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.

Processing Rounds

Один раунд – вызов метода `process()` у `annotation processor`.

При этом `FactoryProcessor` создается один раз (а не каждый раунд)!

А почему вообще могут вызывать наш процессор несколько раз? Дело все в том, что в сгенерированном коде могут быть классы проаннотированные `@Factory`!

Чтобы не было проблем необходимо в конце каждого раунда чистить необходимые данные. В нашем случае – это `map factoryClasses`.