

Java-2

```
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() { }
```

```
}
```

```
@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>  
annoations, RoundEnvironment env) { }  
}
```

MyProcessor.jar

- com

- example

- MyProcessor.class

- META-INF

- services

- javax.annotation.processing.Processor
(содержимое:

- com.example.MyProcessor

- com.foo.OtherProcessor

- net.blabla.SpecialProcessor

-)

```
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;
    }
}
```

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

```
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());  
    }
```

```
}
```

```
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);  
    }  
}
```



```
@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;  
    }  
}
```

```

@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();
    }
    @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();
    }
    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        ...
    }
}

```

```
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
    ) {}

}
```

```
@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 boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        // Iterate over all @Factory annotated elements
        for (Element annotatedElement : roundEnv.getElementsAnnotatedWith(Factory.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) {
```

```
            ...  
        }
```

```
    }
```

```
    ...
```

```
}
```

```
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);
    }
}
```

```

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();
        }
    }
}

```



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

```
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 {
    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
    }
}
}

```

```
// 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;
}
```

```
@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;
```

```
}
```

```
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 (\"" + item.getId() + "\".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();
}
```