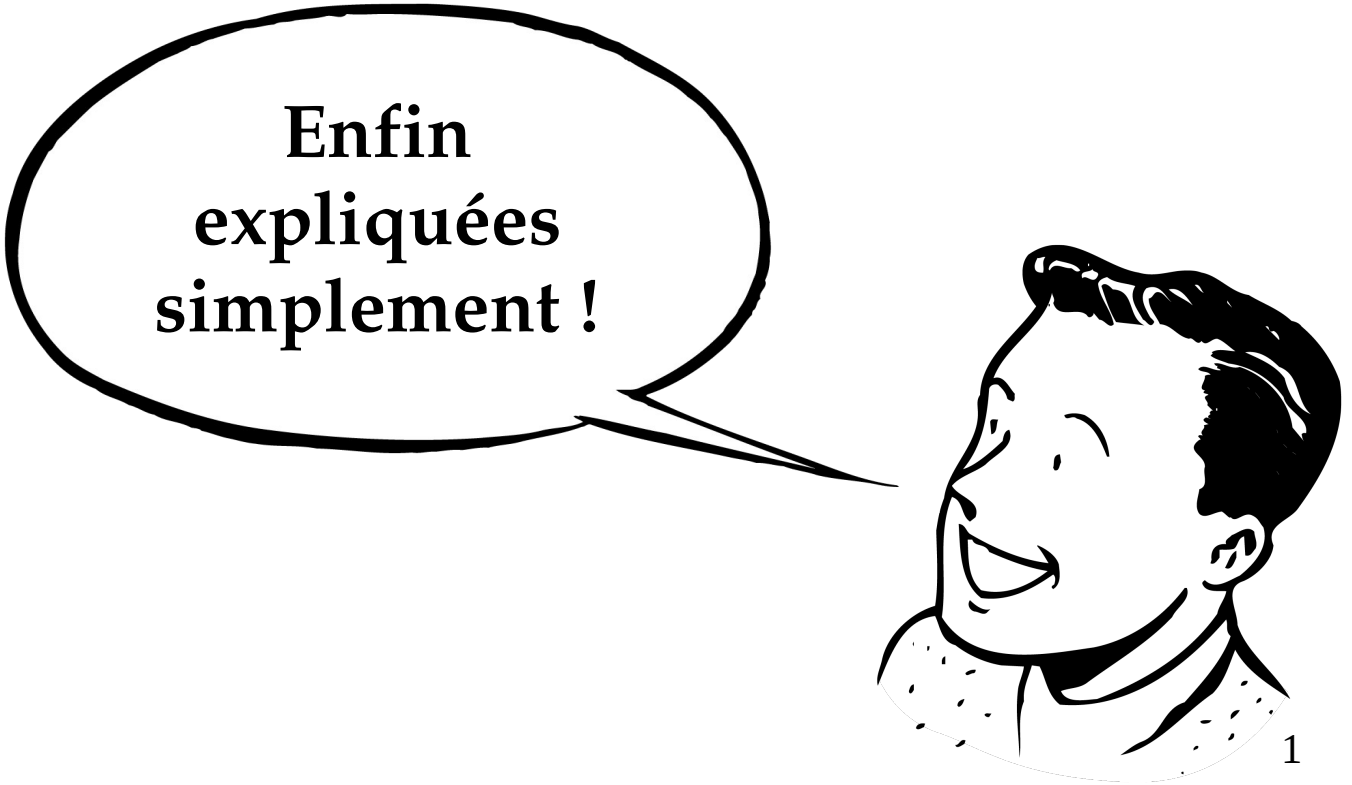
Les Annotations Java



Olivier Croisier

- Consultant Zenika
- Certifié Java 5.0 (100%)
- Certifié Spring Framework
- Formateur certifié JavaSpecialist (javaspecialists.eu)
- Formateur certifié Terracotta
- Blog : The Coder's Breakfast (thecodersbreakfast.net)
- Twitter : `@OlivierCroisier`

(Effet dramatisé en post-production)

Plan

- Présentation
- Annotations, mode d'emploi
- Annotations personnalisées
- Outillage compile-time
- Outillage runtime
- Injection d'annotations
- Conclusion

Plan

- **Présentation**
  - Historique
  - Où trouver des annotations ?
- Annotations, mode d'emploi
- Annotations personnalisées
- Outillage compile-time
- Outillage runtime
- Injection d'annotations
- Conclusion

Présentation
Historique

- Java a toujours proposé une forme ou une autre de méta-programmation

- Dès l'origine, l'outil "javadoc" permettait d'exploiter automatiquement des méta-données à but documentaire

```
/**
 * Méthode inutile
 * @param param Un paramètre (non utilisé)
 * @return Une valeur fixe : "foo"
 * @throws Exception N'arrive jamais (promis!)
 */
public String foo(String param) throws Exception {
   return "foo";
}
```

Présentation
Historique

- Ce système était flexible et a rapidement été utilisé / détourné pour générer d'autres artefacts : fichiers de configuration, classes annexes...
  - Voir le projet XDoclet (xdoclet.sourceforfge.net)

```
/**
 * @ejb.bean
 *     name="bank/Account"
 *     type="CMP"
 *     jndi-name="ejb/bank/Account"
 *     local-jndi-name="ejb/bank/LocalAccount"
 *     primkey-field="id"
 * @ejb.transaction
 *     type="Required"
 * @ejb.interface
 *     remote-class="test.interfaces.Account"
 */
```

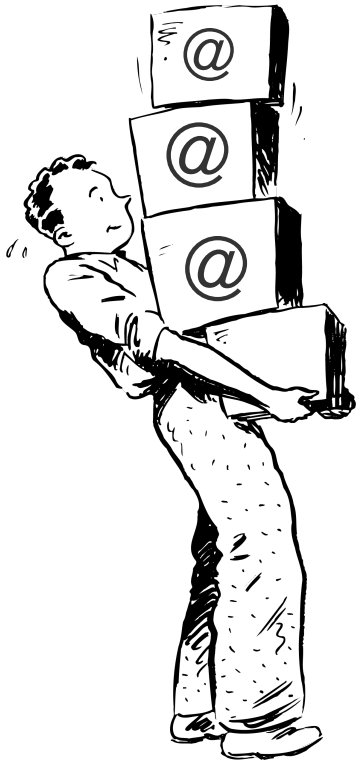
Présentation
Historique

- Reconnaissant le besoin d'un système de méta-programmation plus robuste et plus flexible, Java 5.0 introduit les Annotations
- Elles remplacent avantageusement les doclets dans tous les domaines – sauf bien sûr pour la génération de la Javadoc !

```
public class PojoAnnotation extends Superclass {

    @Override
    public void overridenMethod() {
        super.overridenMethod();
    }

    @Deprecated
    public void oldMethod(){
    }

}
```

Présentation
Où trouver des annotations ?

- Java SE propose assez peu d'annotations en standard
  - `@Deprecated`, `@Override`, `@SuppressWarnings`
  - 4 méta-annotations dans `java.lang.annotation`
  - Celles définies par JAXB et Commons Annotations
- Java EE en fournit une quantité impressionnante
  - Pour les EJB 3, les Servlets 3, CDI, JSF 2, JPA...
- Les frameworks modernes en tirent également parti
  - Spring, Hibernate, CXF, Stripes...
- Développez les vôtres !



Plan

- Présentation
- **Annotations, mode d'emploi**
  - Elements annotables
  - Annoter une classe, une méthode ou un champ
  - Annoter un package
  - Paramètres d'annotations
  - Restrictions et astuces
- Annotations personnalisées
- Outillage compile-time
- Outillage runtime
- Injection d'annotations
- Conclusion

Annotations, mode d'emploi
Elements annotables

- Les annotations peuvent être apposées sur les éléments suivants :
  - Les classes, interfaces et enums
  - Les propriétés de classes
  - Les constructeurs et méthodes
  - Les paramètres des constructeurs et méthodes
  - Les variables locales
  - … et d'autres éléments encore grâce à la JSR 308 (Java ~~7~~ **8**)

- Mais aussi sur...
  - Les packages
  - Les annotations elles-mêmes (méta-annotations) !

Annotations, mode d'emploi
Annoter une classe, méthode ou champ

zenika
ARCHITECTURE INFORMATIQUE

```
@Deprecated
public class Pojo {

    @Deprecated
    private int foo;

    @Deprecated
    public Pojo() {

        @Deprecated
        int localVar = 0;

    }

    @Deprecated
    public int getFoo() {
        return foo;
    }

    public void setFoo(@Deprecated int foo) {
        this.foo = foo;
    }
}
```

Annotations, mode d'emploi
Annoter un package

- Comment annoter un package ?

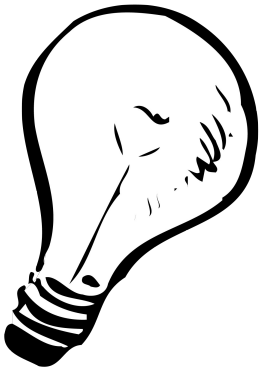
- La déclaration d'un package est présente dans toutes les classes appartenant à ce package. Impossible d'y placer une annotation : risque de conflit ou d'information incomplète !

```
@Foo
package com.zenika.presentation.annotations;
public class ClassA {}
```

```
@Bar
package com.zenika.presentation.annotations;
public class ClassB {}
```

Annotations, mode d'emploi
Annoter un package

- Solution : pour annoter un package, il faut créer un fichier spécial nommé `package-info.java`
- Il contient uniquement la déclaration du package, accompagné de sa javadoc et de ses annotations



```
/**
 * Ce package contient le code source de
 * la présentation sur les annotations.
 */
@Foo
@Bar
package com.zenika.presentation.annotations;
```

Annotations, mode d'emploi
Paramètres d'annotations

- Les annotations peuvent prendre des paramètres
- Ils peuvent être obligatoires ou facultatifs (ils prennent alors la valeur par défaut spécifiée par le développeur de l'annotation)
- Les paramètres se précisent entre parenthèses, à la suite de l'annotation ; leur ordre n'est pas important

```
@MyAnnotation( param1=value1, param2=value2... )
```

- Si l'annotation ne prend qu'un paramètre, il est souvent possible d'utiliser une notation raccourcie

```
@MyAnnotation( value )
```

Annotations, mode d'emploi
Paramètres d'annotations

zenika ARCHITECTURE INFORMATIQUE

javax.servlet.annotation

Annotation Type WebInitParam

```
@Target(value=TYPE)
@Retention(value=RUNTIME)
@Documented
public @interface WebInitParam
```

This annotation is used on a Servlet or Filter implementation class to specify an initialization parameter.

Required Element Summary	
java.lang.String	**name** Name of the initialization parameter
java.lang.String	**value** Value of the initialization parameter

Optional Element Summary	
java.lang.String	**description** Description of the initialization parameter

```
@WebInitParam( name="foo", value="bar" )
```

Annotations, mode d'emploi
Restrictions et astuces

- Il est interdit d'annoter plusieurs fois un élément avec la même annotation. Pour contourner le problème, on peut recourir à des annotations "*wrapper*"

```
@SecondaryTables({
   @SecondaryTable(name="city"),
   @SecondaryTable(name="country")
})
public class Address
{...}
```

- Il est possible de séparer l'@ du nom de l'annotation !

```
@
/** Javadoc */
Deprecated
```

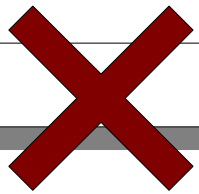
Annotations, mode d'emploi
Restrictions et astuces

- Pour les paramètres :
  - Les valeurs des paramètres doivent être des "*compile-time constants*"...

```
@MyAnnotation(answer = (true!=false) ? 42 : 0)
```

  - … mais ne peuvent pas être *null*
    (Et personne ne sait pourquoi !)

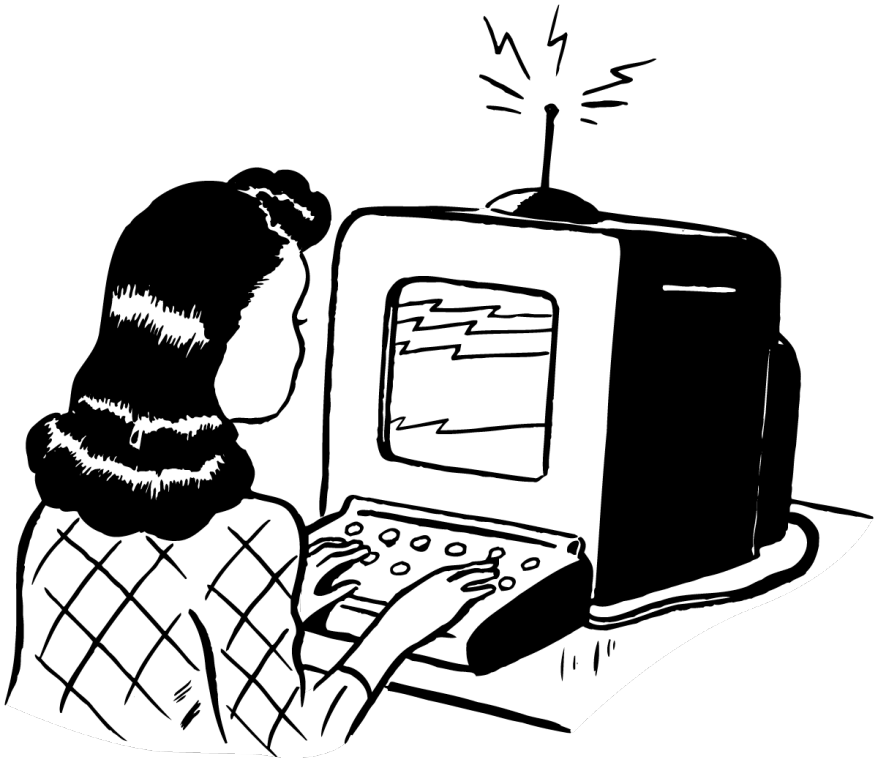
```
@MyAnnotation(param = null)
```



Annotations, mode d'emploi
Démos

Démos

- *Mode d'emploi*



Plan

- Présentation
- Annotations, mode d'emploi
- **Annotations personnalisées**
  - Use-cases
  - Méta-annotations
  - Paramètres d'annotations
- Outillage compile-time
- Outillage runtime
- Injection d'annotations
- Conclusion

Annotations personnalisées
A quoi ça sert ?

- Pourquoi développer des annotations personnalisées ?
  - Pour remplacer / compléter des fichiers de configuration XML
  - Pour simplifier ou générifier une portion de code en recourant à la méta-programmation
  - Pour appliquer des règles de compilation supplémentaires, grâce aux Annotation Processors

  - … parce que c'est fun !

Annotations personnalisées
Syntaxe

- Une annotation se déclare comme un type spécial d'interface
- Elle sera compilée sous la forme d'une interface héritant de `java.lang.annotation.Annotation`

```
public @interface MyAnnotation {
}
```

- Il est ensuite possible de compléter l'annotation avec :
  - Des paramètres
    - *Pour véhiculer des données supplémentaires*
  - Des méta-annotations
    - *Pour spécifier les conditions d'utilisation de l'annotation*

Annotation personnalisées
Paramètres d'annotations

- Chaque paramètre se déclare sous la forme d'une méthode (non générique, sans paramètres et sans exceptions)

- Il peut posséder une valeur par défaut (*compile-time constant*), déclarée grâce au mot-clé "`default`" ; il est alors optionnel.

```
public @interface MyAnnotation {
   String message();
   int answer() default 42;
}
```

```
@MyAnnotation( message="Hello World" )
@MyAnnotation( message="Hello World", answer = 0 )
```

Annotation personnalisées
Paramètres d'annotations

- Si l'annotation ne possède qu'un seul paramètre nommé "value", il est possible d'utiliser une syntaxe raccourcie

```
public @interface MyAnnotation {
   String value();
}
```

```
@MyAnnotation( "Hello World" )
@MyAnnotation( value = "Hello World" )
```

Annotation personnalisées
Paramètres d'annotations

- Comme dans toute interface, il est possible de définir des classes, interfaces ou enums internes

```
public @interface MyAnnotation {

   int defaultAnswer = 42;
   int answer() default defaultAnswer;

   enum Season {SPRING, SUMMER, FALL, WINTER};
   Season season();

}
```

```
@MyAnnotation( season = MyAnnotation.Season.WINTER )
```

Annotation personnalisées
Les Méta-Annotations : @Target

- La méta-annotation `@Target` indique sur quels éléments de code l'annotation peut être apposée :

```
@Target( ElementType[] )
public @interface MyAnnotation {
}
```

- Exemple :

```
@Target( {ElementType.TYPE, ElementType.METHOD} )
public @interface MyAnnotation {
}
```

Annotation personnalisées
Les Méta-Annotations : @Target

- Valeurs possibles pour l'enum `ElementType` :
  - `TYPE`
  - `CONSTRUCTOR`
  - `FIELD`
  - `METHOD`
  - `PARAMETER`
  - `LOCAL_VARIABLE`
  - `ANNOTATION_TYPE` (méta-annotation)
  - `PACKAGE`
  - `TYPE_PARAMETER` et `TYPE_USE` (JSR 308, Java ~~7~~ 8?)
    Ex: `@English String @NonEmpty []`

- Si le paramètre n'est pas spécifié, l'annotation peut être utilisée partout (Ex : `@Deprecated`)

Annotation personnalisées
Les Méta-Annotations : @Retention

- La méta-annotation `@Retention` indique la durée de vie de l'annotation

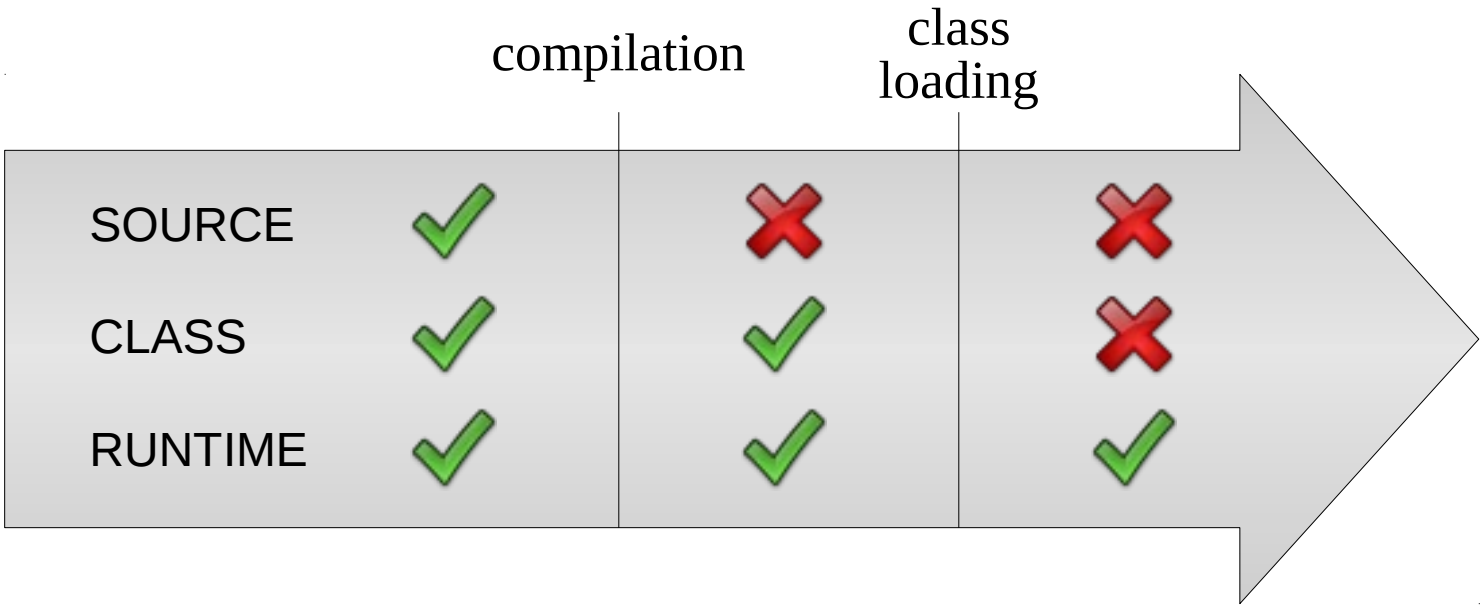
```
@Retention( RetentionPolicy )
public @interface MyAnnotation {
}
```

- Exemple

```
@Retention( RetentionPolicy.RUNTIME )
public @interface MyAnnotation {
}
```

Annotation personnalisées
Les Méta-Annotations : @Retention

- Valeurs possibles pour l'enum `RetentionPolicy` :
  - `SOURCE`
  - `CLASS` (par défaut)
  - `RUNTIME`



Annotation personnalisées
Les Méta-Annotations : @Inherited

- La méta-annotation `@Inherited` indique que l'annotation est héritée par les classes filles de la classe annotée

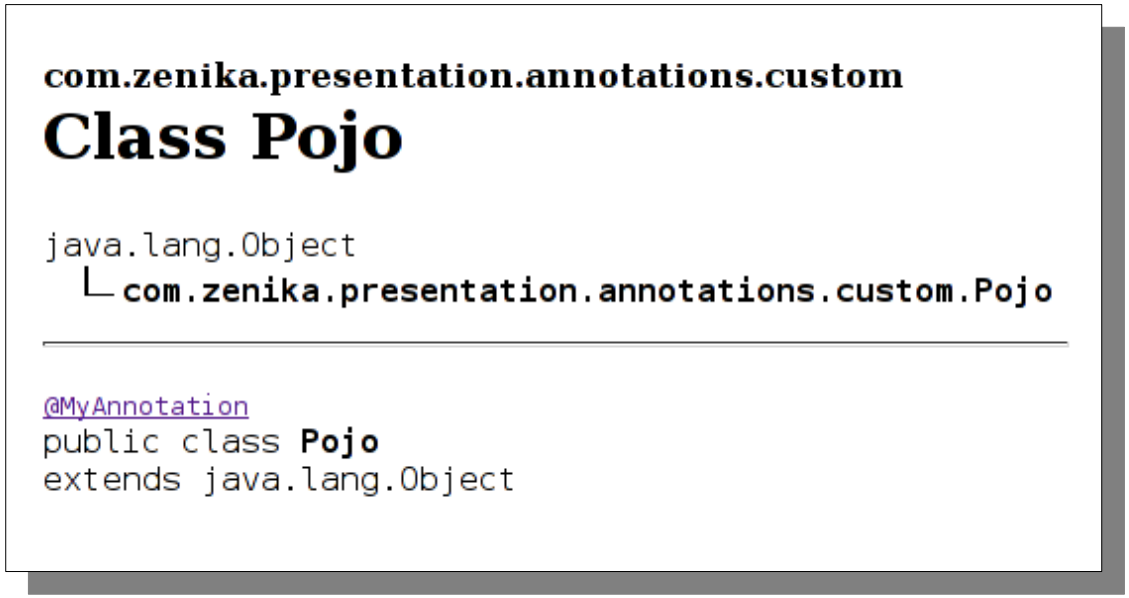
```
@Inherited
public @interface MyAnnotation {
}
```

- Restrictions
  - Seules les annotations portées sur les classes sont héritées
  - Les annotations apposées sur une interface ne sont pas héritées par les classes implémentant cette interface ; idem pour les packages.

Annotation personnalisées
Les Méta-Annotations : @Documented

- La méta-annotation `@Documented` indique que l'annotation doit apparaître dans la javadoc

```
@Documented
public @interface MyAnnotation {
}
```


com.zenika.presentation.annotations.custom

Class Pojo

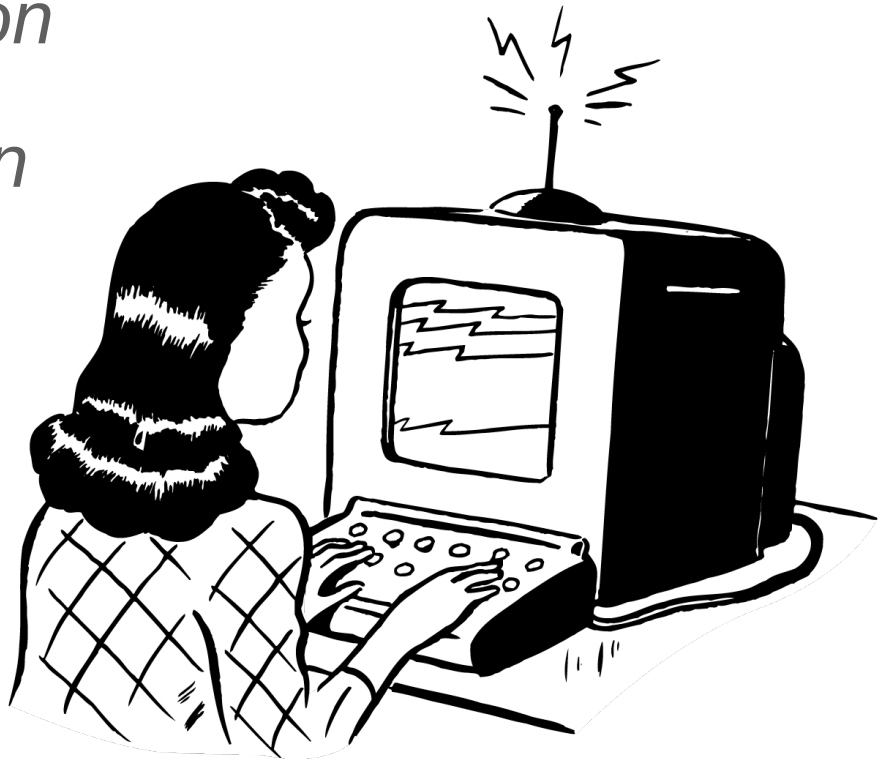
java.lang.Object
└ com.zenika.presentation.annotations.custom.Pojo

@MyAnnotation
public class Pojo
extends java.lang.Object

Annotation personnalisées
Démos

Démos

- *Développer une annotation*
- *Décompiler une annotation*



Plan

- Présentation
- Anatomie
- Mode d'emploi
- Annotations personnalisées
- **Outillage compile-time**
  - Historique et use-cases
  - Processus de compilation
  - Anatomie d'un processeur
  - Limitations
- Outillage runtime
- Injection d'annotations
- Conclusion

Outillage compile-time
Historique et use-cases

zenika ARCHITECTURE INFORMATIQUE

Historique

- Java 5.0 : APT (Annotation Processing Tool)
  - Devait être lancé en plus du compilateur javac
- Java 6.0 : Pluggable Annotation Processors
  - Intégrés au processus de compilation standard
  - Paramètre `-processor` ou utilisation du SPI (Service Provider Interface)

Pour plus d'informations sur le SPI, voir :
- Un article sur The Coder's Breakfast
- La Javadoc de `java.util.ServiceLoader`

Outillage compile-time
Historique et use-cases

Use-cases

- Génération de ressources : "XDoclet ++"
  - Fichiers de configuration
  - Classes annexes
    - *Ex: Proxies, PropertyEditors...*
  - Documentation
    - *Ex: Matrice des rôles JavaEE, cartographie d'IOC...*
- Amélioration du compilateur
  - Vérification de normes de codage
    - *Ex: Les classes "modèle" doivent être Serializable...*
  - Messages d'alerte ou d'erreur supplémentaires

Outillage compile-time
Historique et use-cases

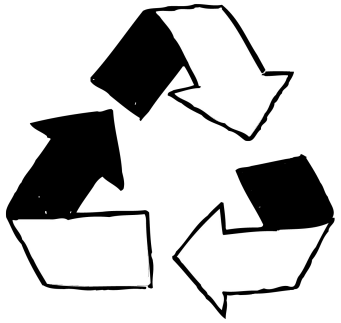
Mention spéciale : Lombok (projectlombok.org)

- Améliore "magiquement" les classes
  - *Génère les getters/setters, equals/hashCode, toString*
- Viole la règle de non-modification des ressources existantes
- Dépend des implémentations internes des compilateurs pour manipuler les AST des classes
  - *com.sun.tools.javac.**
  - *org.eclipse.jdt.internal.compiler.**
- Danger : la compilation de votre projet dépend désormais du support de votre compilateur cible par Lombok.
  Peser le rapport gain / risques !
  - *Votre IDE propose également ces fonctionnalités*

Outillage compile-time
Processus de compilation

1. Le compilateur découvre les processeurs d'annotations
   - *Paramètre -processor sur la ligne de commande*
   - *ou via le Service Provider Interface (SPI)*

2. Un round de compilation est lancé
   - *Le compilateur et les processeurs s'exécutent*
   - *Si de nouvelles ressources sont créées lors de ce round, un nouveau round est lancé*



Outillage compile-time
Anatomie d'un Processeur

- Un processeur est une classe implémentant l'interface `javax.annotation.processing.Processor`
  - Généralement, on sous-classe `AbstractProcessor`

- L'annotation `@SupportedAnnotationTypes` permet d'indiquer quelles annotations le processeur sait traiter

- La méthode `init()` permet d'initialiser le processeur

- La méthode principale `process()` reçoit un paramètre de type `RoundEnvironment` représentant l'environnement de compilation. Elle renvoie un booléen indiquant si l'annotation est définitivement "consommée" par ce processeur

Outillage compile-time
Anatomie d'un Processeur

- Des utilitaires sont accessibles via la propriété `processingEnv` fournie par `AbstractProcessor` :
  - `Types` et `Elements`, permettant d'introspecter le code
  - `Messager`, pour lever des erreurs de compilation et afficher des messages dans la console
  - `Filer`, autorisant la création de nouvelles ressources (classes, fichiers de configuration...)

```
Types      types     = processingEnv.getTypeUtils();
Elements   elts      = processingEnv.getElementUtils();
Messager   messager  = processingEnv.getMessager();
Filer      filer     = processingEnv.getFiler();
```

Outillage compile-time
Anatomie d'un Processeur

zenika ARCHITECTURE INFORMATIQUE

```
@SupportedAnnotationTypes("com.zenika.*")
public class MyProcessor extends AbstractProcessor {

    Types types;
    Elements elements;
    Messager messager;
    Filer filer;

    public void init(ProcessingEnvironment processingEnv) {
        super.init(processingEnv);
        types    = processingEnv.getTypeUtils();
        elements = processingEnv.getElementUtils();
        messager = processingEnv.getMessager();
        filer    = processingEnv.getFiler();
    }

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

Outillage compile-time
Anatomie d'un Processeur

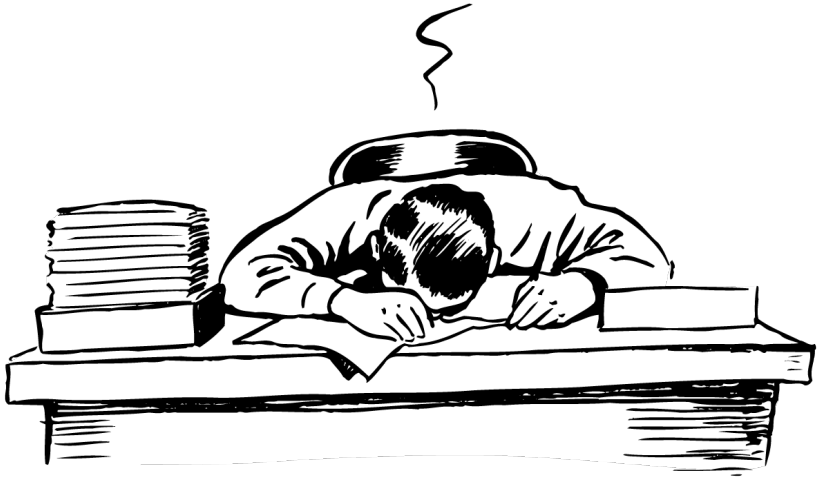
```
public boolean process(
    Set<? extends TypeElement> annotations,
    RoundEnvironment roundEnv) {

    // Pour chaque annotation traitée...
    for (TypeElement annotation : annotations) {

        // Trouver les éléments portant cette annotation
        for (Element e :
            roundEnv.getElementsAnnotatedWith(annotation)) {

            messager.printMessage(
                Diagnostic.Kind.NOTE,
                e.getSimpleName());
        }
    }

    return false;
}
```

Outillage compile-time
Limitations

- Développer un processeur est complexe !
  - Les Types et les Elements offrent deux vues différentes sur le code compilé
    - *Les Elements représentent l'AST brut du code*
    - *Les Types offrent une interface davantage typée "java"*
    - *Il existe des ponts entre ces deux univers*
  - Le pattern Visiteur est beaucoup utilisé

- Un processeur ne peut pas modifier du code existant !

- Quelques bugs encore, et un faible support dans les IDE



Outillage compile-time
Démos

Démos

- *ListingProcessor*
- *MessageHolderProcessor*
- *SerializableClassesProcessor*

Plan

- Présentation
- Annotations, mode d'emploi
- Annotations personnalisées
- Outillage compile-time
- **Outillage runtime**
  - Use-cases
  - Récupération des paramètres
  - Une fois les annotations récupérées...
- Injection d'annotations
- Conclusion

Outillage runtime
Use-cases

- Au runtime, il est possible d'utiliser la Réflexion pour découvrir les annotations présentes sur les classes, champs, méthodes et paramètres de méthodes.
- Uniquement si `@Retention(RetentionPolicy.RUNTIME)`

- Use-cases :
  - Mapping Java ↔ ?
  - Programmation orientée POJO
  - Configuration de containers / frameworks

- Exemples :
  - Hibernate, Apache CXF, XStream...
  - Spring, Guice, Java EE 5/6 (CDI, EJB 3.0, Servlets 3.0...)

Outillage runtime
Récupération des annotations

- Les classes `Package`, `Class`, `Constructor`, `Field` et `Method` implémentent l'interface `AnnotatedElement` :

```
public interface AnnotatedElement {
   Annotation[] getAnnotations();
   Annotation[] getDeclaredAnnotations();
   boolean isAnnotationPresent(Class annotationClass);
   Annotation getAnnotation(Class annotationClass);
}
```

  - `getAnnotations()` renvoie toutes les annotations applicables à l'élément, y compris les annotations héritées
  - `getDeclaredAnnotations()` ne renvoie que les annotations directement apposées sur l'élément

Outillage runtime
Récupération des annotations

- Exemple : lecture des annotations d'une classe et de celles de son package

```
// Annotations sur la classe
Annotation[] clsAnnots = Pojo.class.getAnnotations();
for (Annotation annot : clsAnnots) {
   System.out.println(annot);
}

// Annotations sur le package
Package pkg = Pojo.class.getPackage();
Annotation[] pkgAnnots = pkg.getDeclaredAnnotations();
for (Annotation annot : pkgAnnots) {
   System.out.println(annot);
}
```

Outillage runtime
Récupération des annotations

- Les classes Constructor et Method permettent également de récupérer les annotations présentes sur leurs paramètres
  - `Annotation[][] getParameterAnnotations()`
  - Renvoie un tableau d'annotations par paramètre, dans l'ordre de leur déclaration dans la signature de la méthode

```
public void printParamAnnots(Method method) {

   Annotation[][] allAnnots =
      method.getParameterAnnotations();

   for (Annotation[] annots : paramAnnots) {
      System.out.println(Arrays.toString(annots));
   }

}
```

Outillage runtime
Une fois les annotations récupérées...

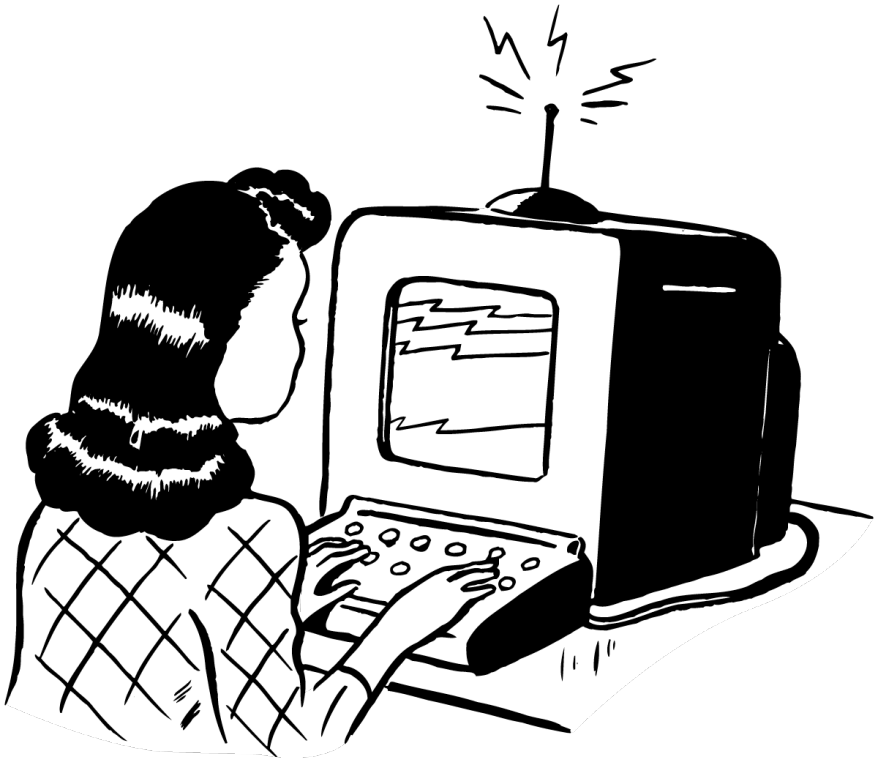
- Que peut-on faire avec les annotations récupérées par réflexion ?
  - On peut découvrir leur type dynamiquement
    `Class<? extends Annotation> annotationType()`
  - On peut lire leurs paramètres (cast nécessaire)

```
Annotation annot = … ;

// Détermination du type réel de l'annotation
Class<? extends Annotation> annotClass =
   myAnnotation.annotationType();

// Affichage du message de MyAnnotation
if (annot instanceof MyAnnotation) {
   MyAnnotation myAnnot = (MyAnnotation) annot;
   System.out.println(myAnnot.message());
}
```

Outillage runtime
Démos

Démos

- *Démonstration*
- *Exemple : CSVReader*



Plan

- Présentation
- Mode d'emploi
- Annotations personnalisées
- Outillage compile-time
- Outillage runtime
- **Injection d'annotations**
  - Au coeur de la classe Class
  - Instancier dynamiquement une annotation
  - Injection dans une classe
  - Injection sur une méthode ou un champ
- Conclusion

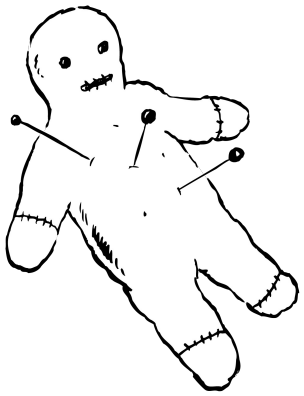
Injection d'annotations
Au coeur de la classe Class

- Dans la classe Class, les annotations sont représentées sous la forme de Maps :

```
private transient Map<Class, Annotation> annotations;
private transient Map<Class, Annotation> declaredAnnotations;
```

- Ces maps sont initialisées lors du premier appel à `getAnnotations()` ou `getDeclaredAnnotations()`
- L'initialisation est réalisée en décodant le `byte[]` renvoyé par la méthode native `getRawAnnotations()`

- En modifiant ces maps par Réflexion, il est possible d'injecter arbitrairement des annotations sur une Classe au runtime !



Injection d'annotations
Instancier une annotation

- Pour obtenir une instance de l'annotation à injecter, il suffit d'instancier une classe anonyme implémentant son "interface"

```
MyAnnotation myAnnotation = new MyAnnotation() {
    @Override
    public String message() {
        return MyAnnotation.defaultMessage;
    }

    @Override
    public int answer() {
        return MyAnnotation.defaultAnswer;
    }

    @Override
    public Class<? extends Annotation> annotationType() {
        return MyAnnotation.class;
    }
};
```

Injection d'annotations
Injection sur une Classe

- Il ne reste plus qu'à utiliser la Réflexion pour injecter notre annotation dans la classe cible

```
public static void injectClass
    (Class<?> targetClass, Annotation annotation) {

    // Initialisation des maps d'annotations
    targetClass.getAnnotations();

    // Récupération de la map interne des annotations
    Field mapRef =  Class.class.getDeclaredField("annotations");
    mapRef.setAccessible(true);

    // Modification de la map des annotations
    Map<Class, Annotation> annots =
        (Map<Class, Annotation>) mapRef.get(targetClass);
    if (annots==null || annots.isEmpty()) {
        annots = new HashMap<Class, Annotation>();
    }
    pojoAnnotations.put(annotation.annotationType(), annotation);
    mapRef.set(targetClass, pojoAnnotations);
}
```

Injection d'annotations
Injection sur une Méthode ou un Champ

- L'injection d'annotations sur les classes Constructor, Field et Method est plus problématique

- Au sein de la classe Class, les méthodes `getConstructor()`, `getField()`, ou `getMethode()` renvoient des copies des objets correspondants

```
Method m1 = Pojo.class.getDeclaredMethod("foo", null);
Method m2 = Pojo.class.getDeclaredMethod("foo", null);
System.out.println(m1==m2); // false
```

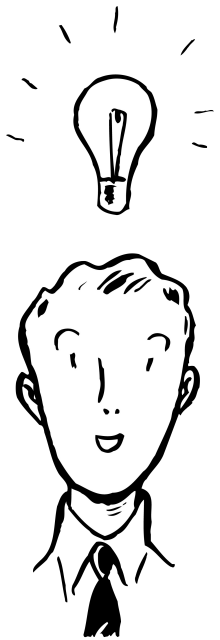
- Ces copies sont initialisées directement à partir du *bytecode* de la classe, pas à partir d'une instance préexistante
- Les modifications apportées aux maps d'annotations sur une instance sont donc strictement locales à cette instance

Injection d'annotations
Injection sur une Méthode ou un Champ

- Alors, comment faire ?
  - AOP pour intercepter les méthodes `getConstructor()`, `getMethod()`, `getField()` ?
  - Modifier directement la portion de bytecode correspondant à ces objets ?
  - Si vous trouvez une solution, je suis preneur !

Injection d'annotations
Use-cases

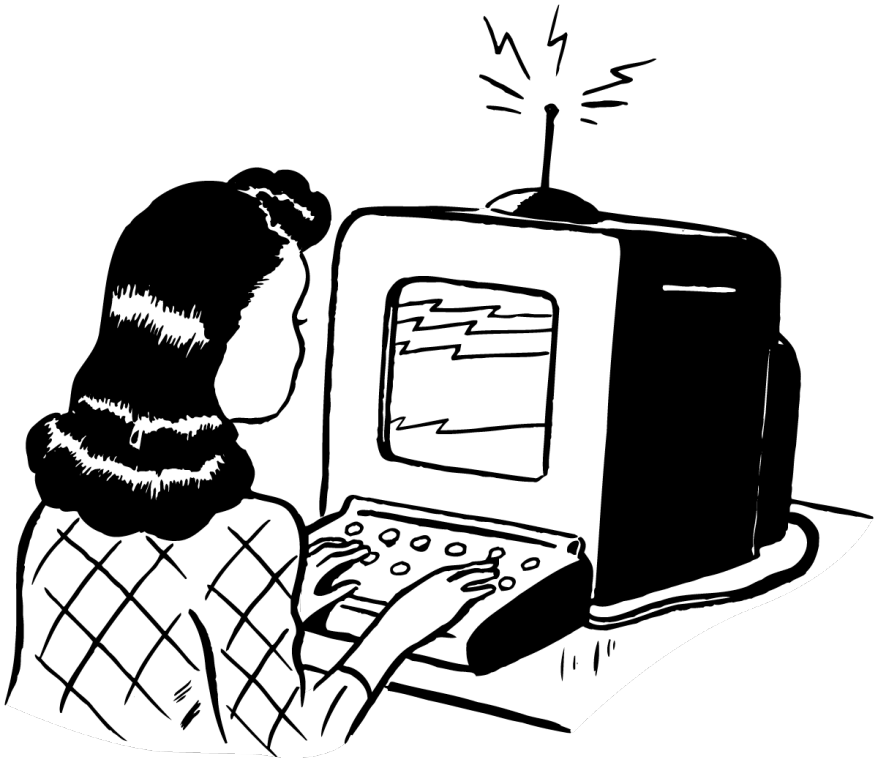
- Use-cases
  - Les annotations @Inherited posées sur les interfaces ne sont pas héritées par les classes implémentant ces interfaces.
    Il est possible d'y remédier en réinjectant manuellement ces annotations dans les classes. Mais attention aux conflits !
  - … d'autres idées ?



Injection d'annotations
Démos

Démos

- *Injection dans une classe*
- *Injection dans un champ*



Plan

- Présentation
- Mode d'emploi
- Annotations personnalisées
- Outillage compile-time
- Outillage runtime
- Injection d'annotations
- **Conclusion**

Conclusion

- Les doclets ont ouvert la voie à la méta-programmation ; Java 5.0 a standardisé et démocratisé les annotations

- Tous les frameworks modernes utilisent les annotations
- Elles complètent parfaitement les fichiers de configuration XML

- Il est facile de développer des annotations personnalisées
- Java fournit des outils pour les exploiter lors de la compilation et au runtime

- Mais attention à la complexité !

Conclusion

Questions / Réponses

Références

- Annotation processors
  - La documentation du SDK et la JLS
  - "Enforcing design rules with Pluggable Annotation Processors" sur http://thecodersbreakfast.net
  - La newsletter Javaspecialists (http://javaspecialists.eu)
- JSR 308 et le framework Checkers
  - http://types.cs.washington.edu/jsr308/
- Lombok
  - http://projectlombok.org/