

Les Annotations de Java 5.0

par [F. Martini \(adiGuba\)](#)

Date de publication : 05/07/2005

Dernière mise à jour :

Les Annotations permettent de marquer différents éléments du langage Java avec des attributs particuliers, dans le but d'automatiser certains traitements et même d'ajouter des traitements avant la compilation grâce au nouvel outil du JDK : apt...

Avant propos...

Introduction

1 - Qu'est-ce qu'une Annotation ?

1.1 - Les Annotations standards

`@Deprecated`

`@Override`

`@SuppressWarnings`

1.2 - Les Méta-Annotations

`@Documented`

`@Inherit`

`@Retention`

`@Target`

2 - Création d'une Annotation

2.1 - Une simple annotation : `@TODO`

2.2 - Utiliser des attributs

2.3 - Les membres et les valeurs par défauts

3 - Annotation Processing Tool (APT)

3.1 - `AnnotationProcessorFactory`

3.2 - `AnnotationProcessor`

3.3 - `DeclarationVisitor`

3.4 - Compilation et utilisation

3.5 - Un autre exemple : `@EmptyConstructor`

3.6 - Restriction à l'utilisation d'APT

4 - Les annotations et l'introspection

4.1 - Recherche dynamique des Annotations

4.2 - Exemple d'utilisation

Conclusion

Avant propos...

Je tiens à remercier l'équipe de rédacteurs Java de developpez.com pour leurs conseils avisés...

En particulier [vbrabant](#) et le responsable de l'équipe : [vedaer](#).



Ce tutoriel est également disponible en version PDF à l'adresse suivante :

<ftp://ftp-developpez.com/adiguba/tutoriels/java/tiger/annotations/annotations.pdf>

(miroir : <http://ftp-developpez.com/adiguba/tutoriels/java/tiger/annotations/annotations.pdf>).

Vous pouvez également télécharger le code source de certain des exemples de ce tutoriel :

<ftp://ftp-developpez.com/adiguba/tutoriels/java/tiger/annotations/annotations-src.zip>

(miroir : <http://ftp-developpez.com/adiguba/tutoriels/java/tiger/annotations/annotations-src.zip>).

<ftp://ftp-developpez.com/adiguba/tutoriels/java/tiger/annotations/annotations-src-rt.zip>

(miroir : <http://ftp-developpez.com/adiguba/tutoriels/java/tiger/annotations/annotations-src-rt.zip>).

Introduction

Tiger, la version **5.0** de Java, est sûrement la version de Java qui aura apporté le plus de changements dans le langage. Parmi ceux-ci, les Annotations (également appelées Métadonnées) sont peut-être les plus intéressantes ...

En effet, elles permettent non seulement de mieux documenter le code source, mais également d'utiliser ces informations pendant l'exécution et même d'interagir avec le compilateur grâce au nouvel utilitaire : **APT** (**A**nnotation **P**rocessing **T**ool).



*Avant de commencer, je vous invite à consulter l'article de **Lionel Roux** sur les principales nouveautés de Java 5.0, si ce n'est pas déjà fait :*

<http://lroux.developpez.com/article/java/tiger/>

*En effet, l'utilisation des Annotations et les codes sources présents dans ce tutoriel utilisent une syntaxe propre à Java 5.0 (notamment en ce qui concerne les Génériques et les boucles **for** étendues).*

Les Annotations de Java 5.0 offrent les mêmes possibilités qu'un outil tel que **XDoclet** (<http://xdoclet.sourceforge.net>) ou encore d'**EJBGen** (<http://www.beust.com/cedric/ejbgen/>), mais possèdent l'avantage de faire partie du langage lui-même ...

1 - Qu'est-ce qu'une Annotation ?

Une annotation permet de "marquer" certains éléments du langage Java afin de leur ajouter une propriété particulière. Ces annotations peuvent ensuite être utilisées à la compilation ou à l'exécution pour automatiser certaines tâches...

Une annotation se déclare comme une interface, mis à part le symbole **@** devant le mot-clef **interface** qui permet de spécifier qu'il s'agit d'une interface :

Une annotation simple

```
public @interface MonAnnotation {
}
```

Une annotation peut être utilisée sur plusieurs type d'éléments (package, class, interface, enum, annotation, constructeur, méthode, paramètre, champs d'une classe ou variable locale). Plusieurs annotations différentes peuvent être utilisées sur un même élément mais on ne peut pas utiliser deux fois la même annotation. En général l'annotation est placée devant la déclaration de l'élément qu'elle marque :

Exemple d'utilisation d'une annotation sur une classe

```
@MonAnnotation
public class MaClasse {
    /* ... */
}
```

1.1 - Les Annotations standards

L'API standard de Java 5.0 propose seulement trois annotations. Elle permettent d'interagir avec le compilateur Java.

@Deprecated

L'annotation **@Deprecated** vient remplacer le tag javadoc **@deprecated** afin de signaler au compilateur que l'élément marqué est déprécié et ne devrait plus être utilisé.

Le compilateur affichera un warning si l'élément est utilisé dans du code non-déprécié (ou une 'note', selon la configuration du compilateur).

Utilisation de @Deprecated

```
public class Maclasse {

    /**
     * Retourne l'année en cours.
     * @return L'année en cours.
     */
    @Deprecated
    public int getYear () {
        return year;
    }

}
```

Ainsi, il est désormais possible de savoir par réflexion si une méthode est dépréciée ou pas...



*Un petit bémol toutefois : contrairement au tag javadoc **@deprecated**, cette annotation ne permet pas de spécifier la raison de cette dépréciation...*

Je vous conseille donc de l'utiliser conjointement avec le tag **javadoc** `@deprecated` :

Utilisation de `@Deprecated` et `@deprecated`

```
public class Maclasse {

    /**
     * Retourne l'année en cours.
     * @return L'année en cours.
     * @deprecated Retourne en réalité le nombre d'années depuis 1900. Remplacée par
     * getFullYear().
     */
    @Deprecated
    public int getYear () {
        return year;
    }

}
```

`@Override`

L'annotation **`@Override`** ne peut être utilisée que sur une méthode afin de préciser au compilateur que cette méthode est redéfinie et doit donc 'cacher' une méthode héritée d'une classe parent. Si ce n'est pas le cas (par exemple si une faute de frappe s'est glissée dans le nom de la méthode), le compilateur doit afficher un erreur (et donc faire échouer la compilation).

Par exemple, si on redéfinit la méthode **`toString()`** de **`Object`** :

Exemple

```
@Override
public String toString() {
    return "Texte";
}
```



Cette annotation est doublement utile car non seulement elle permet d'éviter des erreurs bêtes, mais elle rend également le code plus lisible en distinguant les méthodes redéfinies.

`@SuppressWarnings`

L'annotation **`@SuppressWarnings`** indique au compilateur de ne pas afficher certains warnings. Le principal intérêt de cette annotation est de cacher des warnings sur des parties de code plus anciennes sans pour autant les cacher sur toute l'application.

Elle reste toutefois à utiliser avec parcimonie.

Cette Annotation attend un attribut contenant le nom des warnings qu'il faut supprimer, par exemple :

Suppression du warning 'deprecation' pour la classe OldClass

```
@SuppressWarnings("deprecation")
public class OldClass {
    /* ... */
}
```

Suppression des warning 'deprecation' et 'unchecked' pour la méthode m()

```
@SuppressWarnings({ "deprecation", "unchecked" })
public int m () {
    /* ... */
}
```

Le nom de ces types de warning sont les mêmes que ceux de l'outil **javac** avec **-Xlint**, même si les compilateurs peuvent définir d'autres types de warning.

Si le compilateur ne connaît pas le type de warning indiqué, il doit l'ignorer ou afficher un message dans le pire des cas, mais en aucun cas il ne doit bloquer la compilation.



*Attention ! Aussi étrange que cela puisse paraître, le compilateur **javac 5.0** ne gère pas cette annotation...*

*Le 'bug' a déjà été signalé dans la base de données de Sun et cette fonctionnalité a été implémentée dans la version **b29** de **Mustang** (le prochain Java). Donc à moins d'une nouvelle mise à jours du JDK 5.0, il faudra encore attendre pour pouvoir l'utiliser.*

Pour plus d'information vous pouvez consulter la fiche du bug : [add support for jsr175's java.lang.SuppressWarnings](#).

1.2 - Les Méta-Annotations

Les Méta-Annotations sont simplement des Annotations destinées à marquer d'autres Annotations, généralement pour indiquer au compilateur comment il doit les traiter.

Elles vont donc nous servir pour la conception de nos Annotations. Chacune de ces Méta-Annotations répondent à un problème bien précis.

Elles appartiennent toutes au package [java.lang.annotation](#) :

@Documented

La méta-annotation **@Documented** indique à l'outil **javadoc** (ou à tout autre outil compatible) que l'Annotation doit être présente dans la documentation générée pour tous les éléments marqués.

Prenons comme exemple les deux annotations suivantes :

```
public @interface SimpleAnnotation {
}

import java.lang.annotation.Documented;

@Documented
public @interface DocAnnotation {
}
```

La classe suivante comprend deux méthodes marquées chacune avec une des deux précédentes annotations :

Exemple.java

```
public class Exemple {

    /**
     * Commentaire de la méthode 1.
     */
    @SimpleAnnotation
    public void method1 () {
        /* ... */
    }
}
```

Exemple.java

```

/**
 * Commentaire de la méthode 2.
 */
@DocAnnotation
public void method2 () {
    /* ... */
}

```

Ainsi, lorsqu'on consulte la documentation **javadoc** de cette classe, on ne visualise que l'annotation de la seconde méthode :

```

public void method1()

    Commentaire de la méthode 1.
-----
@DocAnnotation
public void method2()

    Commentaire de la méthode 2.

```



Les Annotations peuvent bien sûr être utilisées par **javadoc** grâce à son API Doclet. Pour plus d'informations sur le sujet je vous invite à consulter [la documentation de l'outil Javadoc 5.0](#) sur le site officiel de Sun.

@Inherit

La méta-annotation **@Inherit** indique que l'annotation sera héritée par tous les descendants de l'élément sur lequel elle a été posée. Par défaut, les annotations ne sont pas héritées par les éléments fils.

Prenons comme exemple les deux annotations suivantes :

```

public @interface SimpleAnnotation {
}

import java.lang.annotation.Inherit;

@Inherit
public @interface InheritAnnotation {
}

```

Et une classe marquée avec ces deux annotations :

ExempleInherited.java

```

@SimpleAnnotation
@InheritAnnotation
public class ExempleInherited {
}

```

Toutes les classes qui étendront **ExempleInherited** seront alors automatiquement marquées par l'annotation **@InheritAnnotation** mais pas par **@SimpleAnnotation**..



Attention ! L'API précise bien que cette méta-annotation est sans effet si l'annotation est utilisée sur autre chose qu'une classe. Ainsi les annotations ne peuvent pas être héritées d'une interface par exemple.

@Retention

La méta-annotation **@Retention** indique la "durée de vie" de l'annotation, c'est à dire de quelle manière elle doit être gérée par le compilateur.

Cette méta-annotation peut prendre une de ces trois valeurs :

Valeur	Description
RetentionPolicy.SOURCE	Les annotations ne sont pas enregistrées dans le fichier *.class. Elles ne sont donc accessibles que par des outils utilisant les fichiers sources (compilateur, javadoc, etc...).
RetentionPolicy.CLASS	Les annotations sont enregistrées dans le fichier *.class à la compilation mais elle ne sont pas utilisées par la machine virtuelle à l'exécution de l'application. Elles peuvent toutefois être utilisées par certains outils qui lisent directement les *.class. Il s'agit du comportement par défaut si la méta-annotation n'est pas présente.
RetentionPolicy.RUNTIME	Les annotations sont enregistrées dans le fichier *.class à la compilation et elle sont utilisées par la machine virtuelle à l'exécution de l'application. Elles peuvent donc être lues grâce à l'API de réflexion (plus de détails dans le chapitre sur l'introspection)...

Par exemple, dans le code suivant, la classe 'Main' possède trois annotations avec une "rétention" différente :

```
import java.lang.annotation.Annotation;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.SOURCE)
@interface SourceAnnotation { }

@Retention(RetentionPolicy.CLASS)
@interface ClassAnnotation { }

@Retention(RetentionPolicy.RUNTIME)
@interface RuntimeAnnotation { }

@SourceAnnotation
@ClassAnnotation
@RuntimeAnnotation
public class Main {

    public static void main(String[] args) {

        System.out.println ("Liste des annotations de la classe 'Main' :");
        System.out.println ();
        for (Annotation a : Main.class.getAnnotations() ) {
            System.out.println ("\t * Annotation : " +
a.annotationType().getSimpleName());
        }
    }
}
```

Lors de l'exécution de cette classe, on obtient donc l'affichage suivant :

```
Liste des annotations de la classe 'Main' :

    * Annotation : RuntimeAnnotation
```



RetentionPolicy.CLASS permet au compilateur d'accéder à l'annotation d'une classe dont il ne connaît pas la source, par exemple lorsqu'on hérite d'une classe d'une librairie.

@Target

La méta-annotation **@Target** permet de limiter le type d'éléments sur lesquels l'annotation peut être utilisée. Le tableau ci-dessous définit les différentes valeurs de **@Target** correspondant aux différents éléments du langage.

Si la méta-annotation **@Target** est absente de la déclaration d'une annotation, elle peut alors être utilisée sur tous ces éléments :

Valeur	Description
ElementType.ANNOTATION	L'annotation peut être utilisée sur d'autres annotations.
ElementType.CONSTRUCTOR	L'annotation peut être utilisée sur des constructeurs.
ElementType.FIELD	L'annotation peut être utilisée sur des champs d'une classe.
ElementType.LOCAL_VARIABLE	L'annotation peut être utilisée sur des variables locales.
ElementType.METHOD	L'annotation peut être utilisée sur des méthodes.
ElementType.PACKAGE	L'annotation peut être utilisée sur des packages
ElementType.PARAMETER	L'annotation peut être utilisée sur des paramètres d'une méthode ou d'un constructeur.
ElementType.TYPE	L'annotation peut être utilisée sur la déclaration d'un type : class , interface (annotation comprise) ou d'une énumération (mot-clef enum).

Par exemple, pour indiquer qu'une annotation ne peut être utilisée que sur un constructeur :

```
@Target( ElementType.CONSTRUCTOR )
public @interface ConstructorAnnotation {
}
```

Et pour permettre l'utilisation de l'annotation à la fois sur les constructeurs et sur les méthodes :

```
@Target( { ElementType.CONSTRUCTOR, ElementType.METHOD } )
public @interface ConstructorAnnotation {
}
```



Si on utilise une annotation sur un élément non autorisé, la compilation générera une erreur...

2 - Création d'une Annotation

Afin de détailler pas à pas la création d'une annotation, nous allons prendre pour exemple une annotation **@TODO** qui permettra aux développeurs de marquer le code source des tâches restant à accomplir.

2.1 - Une simple annotation : @TODO

Nous allons donc commencer par créer notre annotation. Elle se résume pour commencer à la simple déclaration suivante :

TODO.java

```
public @interface TODO {
}
```

La seconde étape est d'utiliser les méta-annotations standard du chapitre précédent afin de modifier le comportement de notre annotation. Pour cela il faut 'répondre' aux questions suivantes :

Méta-Annotation	Question	Réponse pour @TODO
@Documented	L'annotation doit-elle être documentée par javadoc ?	Oui , afin d'avoir un aperçu des tâches restantes dans la documentation.
@Inherit	L'annotation doit-elle être héritée (seulement si l'annotation est posée sur une classe) ?	Non , on ne souhaite pas impacter les classes filles.
@Retention	Quel est la 'durée de vie' de l'annotation ?	SOURCE : Il n'est pas nécessaires de conserver cette annotation dans les fichiers *.class, on n'utilisera donc pas cette méta-annotation.
@Target	L'annotation doit-elle se restreindre à certains éléments ?	Non , on souhaite pouvoir annoter tous types d'éléments.

Ceci se traduira par le code suivant :

TODO.java

```
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.SOURCE;

@Documented
@Retention(SOURCE)
public @interface TODO {
}
```

Notre annotation peut désormais être utilisée de la manière suivante :

Exemple d'utilisation

```
@TODO
public void doSomething () {
    /* ... */
}
```

2.2 - Utiliser des attributs

Pour l'instant notre annotation ne permet que de positionner des marqueurs dans le code source, mais elle

n'apporte aucune information supplémentaire : c'est une annotation "marqueur". Nous allons maintenant lui ajouter un attribut permettant de décrire plus précisément la tâche restant à effectuer :

TODO.java

```
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.SOURCE;

@Documented
@Retention(SOURCE)
public @interface TODO {
    /** Message décrivant la tâche à effectuer. */
    String value();
}
```

La méthode **value()** représente l'attribut de notre annotation. Il est inutile de préciser la portée de **value()** : elle est implicitement **public** et ne peut pas prendre d'autre valeur. Notre annotation s'utilise désormais de la manière suivante :

Exemple d'utilisation

```
/**
 * Commentaire de la méthode...
 */
@TODO(value="La gestion des exceptions est incorrecte...")
public void doSomething () {
    /* ... */
}
```

Le type de retour de la méthode **value()** correspond au type attendu par le membre **value=** lors de l'utilisation de l'annotation.



Le nom **value()** n'est pas anodin. En effet, pour une annotation à membre unique dont le nom est **value()**, il est possible de ne pas spécifier le nom de l'attribut lors de l'utilisation de l'annotation. Ce qui nous donne :

Exemple d'utilisation

```
/**
 * Commentaire de la méthode...
 */
@TODO("La gestion des exceptions est incorrecte...")
public void doSomething () {
    /* ... */
}
```

Notre annotation devient un peu plus intéressante : lorsqu'on génère la **javadoc** de la classe contenant la méthode **doSomething()**, on obtient quelque chose du genre :

```
@TODO(value="La gestion des exceptions est incorrecte...")
public void doSomething()

    Commentaire de la méthode...
```

- Un type primitif (**boolean**, **int**, **float**, etc.).
- Une chaîne de caractères (**java.lang.String**).
- Une référence de classe (**java.lang.Class**).
- Une Annotation (**java.lang.annotation.Annotation**).
- Un type énuméré (**enum**).
- Un tableau à une dimension d'un des types ci-dessus.



Toutes les annotations héritent implicitement de l'interface **java.lang.annotation.Annotation**.

Si le type d'un membre correspond à un tableau, il faut utiliser une syntaxe proche de l'ellipse pour le déclarer. Par exemple, pour un membre **tab()** correspondant à un tableau de **int**, on peut utiliser les différentes formes suivantes :

Utilisation d'un membre avec un tableau de 5 éléments

```
@MonAnnotation (tab={1,2,3,4,5})
public class MaClasse {
}
```

Si le tableau ne possède qu'un seul et unique élément, les accolades sont alors inutiles :

Utilisation d'un membre avec un tableau contenant un seul élément

```
@MonAnnotation (tab=1)
public class MaClasse {
}
```

2.3 - Les membres et les valeurs par défauts

On va désormais enrichir notre annotation par un second membre permettant d'indiquer un niveau de priorité à la tâche **@TODO**. Ces niveaux de priorité seront définis via un type énuméré :

TODO.java

```
@Documented
@Retention(SOURCE)
public @interface TODO {

    /** Message décrivant la tâche à effectuer. */
    String value();
    /** Niveau de criticité de la tâche. */
    Level level();

    /** Énumération des différents niveaux de criticités. */
    public static enum Level { MINEUR, NORMAL, IMPORTANT };

}
```

Désormais, notre annotation sera obligatoirement utilisée de la manière suivante :

Exemple d'utilisation

```
@TODO(value="La gestion des exceptions est incorrecte...", level=TODO.Level.NORMAL)
public void doSomething () {
    /* ... */
}
```



L'ordre des différents attributs de l'annotation n'est pas important, mais il est obligatoire d'utiliser le nom de l'attribut.

Note : l'utilisation des **import static** de Java 5.0 devient ici intéressante car il permet de simplifier la syntaxe :

Exemple d'utilisation avec 'import static' :

```
import static com.developpez.adiguba.annotation.TODO.Level.*;

/* ... */

@TODO(value="La gestion des exceptions est incorrecte...", level=NORMAL)
public void doSomething () {
    /* ... */
}
```

Exemple d'utilisation avec 'import static' :

}

Afin de permettre une utilisation plus souple, nous allons associer une valeur par défaut à l'attribut **level**. Cela est possible grâce au mot-clef **default** placé derrière la déclaration :

TODO.java

```
@Documented
@Retention(SOURCE)
public @interface TODO {

    /** Message décrivant la tâche à effectuer. */
    String value();
    /** Niveau de criticité de la tâche (défaut : NORMAL). */
    Level level() default Level.NORMAL;

    /** Énumération des différents niveaux de criticités. */
    public static enum Level { MINEUR, NORMAL, IMPORTANT };
}
```

Le membre **level** est désormais optionnel et sera implicitement positionné à **NORMAL** s'il n'est pas précisé. Ainsi on peut désormais utiliser notre annotation de différentes manières :

Exemple d'utilisation

```
@TODO("La gestion des exceptions est incorrecte...")
public void doSomething () {
    /* ... */
}

@TODO(value="La gestion des exceptions est incorrecte...")
public void doSomething () {
    /* ... */
}

@TODO(value="La gestion des exceptions est incorrecte...", level=NORMAL)
public void doSomething () {
    /* ... */
}

@TODO(level=NORMAL, value="La gestion des exceptions est incorrecte...")
public void doSomething () {
    /* ... */
}
```



*Il n'est pas possible de marquer plusieurs fois le même élément avec la même annotation. Pour pallier à ce problème, il faut ruser en utilisant une autre annotation conteneur. Par exemple, afin de pouvoir utiliser notre annotation **@TODO** plusieurs fois sur un même élément, on utilisera une annotation **@TODOs** qui comporte un tableau d'annotation :*

TODOs.java

```
@Documented
@Retention(SOURCE)
public @interface TODOs {
    /** Le Tableau des différentes annotations TODO. */
    TODO[] value();
}
```

Qui sera donc utilisée de la manière suivante :

Exemple d'utilisation :

```
@TODOs( {
    @TODO("La gestion des exceptions est incorrecte..."),
    @TODO(value="NullPointerException possible dans certain cas.", level=IMPORTANT)
})
```

Exemple d'utilisation :

```
public void doSomething () {  
    /* ... */  
}
```

Notre annotation est désormais terminée. On pourrait éventuellement y ajouter des membres supplémentaires, comme le nom du développeur qui devra effectuer la tâche, ou la date à laquelle elle devra être réalisée ... Mais on se contentera de ce simple exemple dans ce tutoriel.

Toutefois, certains EDIs proposent une fonctionnalité similaire. Par exemple **Eclipse** propose des **TaskTag** qui permet de reporter certains mots-clés présents dans les commentaires dans un onglet de son interface...

Il est également possible d'utiliser l'API Doclets de **javadoc** afin de traiter le tag **@TODO** à l'intérieur des commentaires lors de la génération de la documentation...

Mais notre annotation possède deux avantages :

- Elle est indépendante de l'outil de développement.
- Elle permet de modifier le cours de la compilation grâce au nouvel outil du JDK 5.0 : **APT** !

3 - Annotation Processing Tool (APT)

Le **JDK 5.0** propose un nouvel outil : **APT (Annotation Processing Tool)**. Il s'agit d'un outil permettant d'effectuer des traitements sur les annotations avant la compilation effective des sources Java. Il propose ainsi une vision de la structure du code Java avant la compilation.

Il permet ainsi :

- De générer des messages (note, warning et error).
- De générer des fichiers (texte, binaire, source et classe Java).

Ceci avant de réellement compiler les fichiers *.java.

Pour cela, **APT** utilise les mêmes options de la ligne de commande que **javac**, en plus des suivantes :

- **-s dir** : Spécifie le répertoire de base où seront placés les fichiers générés (par défaut dans le même répertoire que **javac**).
- **-nocompile** : Ne pas effectuer la compilation après le traitement des annotations.
- **-print** : Affiche simplement une représentation des éléments annotés (sans aucun traitement ni compilation).
- **-A[key=val]** : Permet de passer des paramètres supplémentaires destinés aux "fabriques".
- **-factorypath path** : Indique le(s) chemin(s) de recherche des "fabriques". Si absent, c'est le **classpath** qui sera utilisé.
- **-factory classname** : Indique le nom de la classe Java qui servira de "fabrique".

On peut voir que l'outil **APT** utilise des "fabriques" (**factory** en anglais) qui permettent de créer les différents processeurs d'annotations dont on a besoin. En effet, pour le traitement des codes sources, nous avons besoins de trois classes principales :

- **AnnotationProcessorFactory** : La fabrique qui sera utilisée par **APT**.
- **AnnotationProcessor** créé par la fabrique et utilisé par **APT** pour traiter les fichiers sources.
- Les **Visitors** qui permettent de visiter simplement les différentes déclarations/types d'un fichier source.

Ces différents objets sont définis dans l'**API mirror** de Sun qui fait partie de la librairie **tools.jar** du JDK. Elle doit donc faire partie du **classpath** pour pouvoir l'utiliser.



Pour plus de détails vous pouvez consulter le guide d'**APT** sur le site de Sun :

<http://java.sun.com/j2se/1.5.0/docs/guide/apt/>

Ainsi que la documentation de l'**API mirror** :

<http://java.sun.com/j2se/1.5.0/docs/guide/apt/mirror/>

Comment ça marche ?

C'est à la fois simple et complexe : **APT** s'utilise comme **javac** pour compiler des fichiers Java, mis à part le fait qu'il effectue un traitement avant de lancer la compilation.

Il a besoin pour cela d'une "fabrique" qui lui donnera une instance d'**AnnotationProcessor** qui effectuera une

'analyse' des sources Java (plus exactement de la vue qu'**APT** propose de ces sources).

L'outil **APT** utilisera la "*fabrique*" spécifiée par le paramètre **-factory**. Toutefois si ce paramètre est absent, **APT** recherchera dans le **classpath** (ou le **factorypath** si il est précisé) des fichiers nommés **com.sun.mirror.apt.AnnotationProcessorFactory** dans le répertoire **META-INF/services** des différents répertoires et archives jar...

Il s'agit d'un simple fichier texte contenant le nom complet des différentes "*fabriques*" qui seront utilisées. Il est ainsi possible d'utiliser plusieurs "*fabriques*" de plusieurs librairies simplement...

Nous allons donc maintenant créer notre propre "*fabrique*" pour notre annotation **@TODO** afin d'afficher des messages pendant la compilation avec **APT**...

3.1 - AnnotationProcessorFactory

L'**AnnotationProcessorFactory** est une interface à implémenter pour créer une "*fabrique*" utilisable par **APT**. Elle nécessite l'implémentation de trois méthodes :

- **public** Collection<String> supportedAnnotationTypes();

Fournit la liste des annotations supportées en retournant une Collection contenant le nom complet des annotations. Il est également possible d'utiliser le caractère "*" tout comme dans les **import**.

- **public** Collection<String> supportedOptions();

Fournit la liste des options que la fabrique accepte et qui peuvent être passées par la ligne de commande avec **-Akey[=value]** (le préfixe "-A" doit être présent dans les chaînes retournées par cette méthode). Pour l'instant, cette méthode n'est pas réellement utilisée par **APT**, mais dans le futur il pourrait retourner une erreur si une option est passé mais qu'aucune fabrique ne l'utilise...

- **public** AnnotationProcessor getProcessorFor(Set<AnnotationTypeDeclaration> atds, AnnotationProcessorEnvironment env)

Cette méthode sera appelée par **APT** afin d'obtenir un objet **AnnotationProcessor** qui s'occupera du traitement sur les annotations. Le paramètre **atds** comporte la liste des annotations trouvées et **env** permet d'interagir avec l'environnement d'**APT**.

Notre fabrique pour l'annotation **@TODO** pourrait ressembler à cela :

SimpleAnnotationProcessorFactory.java

```
public class SimpleAnnotationProcessorFactory implements AnnotationProcessorFactory {

    /** Collection contenant le nom des Annotations supportées. */
    protected Collection<String> supportedAnnotationTypes =
        Arrays.asList( TODO.class.getName(), TODOs.class.getName() );

    /** Collection des options supportées. */
    protected Collection<String> supportedOptions =
        Collections.emptyList();

    /**
     * Retourne la liste des annotations supportées par cette Factory.
     */
    public Collection<String> supportedAnnotationTypes() {
        return supportedAnnotationTypes;
    }

    /**
     * Retourne la liste des options supportées par cette Factory.
     */
}
```

SimpleAnnotationProcessorFactory.java

```

public Collection<String> supportedOptions() {
    return supportedOptions;
}

/**
 * Retourne l'AnnotationProcessor associé avec cette Factory...
 */
public AnnotationProcessor getProcessorFor(Set<AnnotationTypeDeclaration> atds,
    AnnotationProcessorEnvironment env) {
    // Si aucune annotation n'est présente on retourne un processeur "vide"
    if (atds.isEmpty())
        return AnnotationProcessors.NO_OP;
    return new SimpleAnnotationProcessor(env);
}
}

```

La classe **SimpleAnnotationProcessor** représente notre processeur. Nous allons donc voir ce qu'elle doit contenir.

3.2 - AnnotationProcessor

AnnotationProcessor est une simple interface qui ne comporte qu'une seule méthode :

```
public void process();
```

Cette méthode sera appelée une fois par **APT** pour le traitement des annotations. Cette méthode ne possède aucun paramètre, il faut donc que l'environnement d'**APT** lui soit passé par la fabrique :

SimpleAnnotationProcessor.java

```

public class SimpleAnnotationProcessor implements AnnotationProcessor {

    /** L'environnement du processeur d'annotation. */
    protected final AnnotationProcessorEnvironment env;

    /**
     * Constructeur.
     * @param env L'environnement du processeur d'annotation.
     */
    public SimpleAnnotationProcessor (AnnotationProcessorEnvironment env) {
        this.env = env;
    }

    /**
     * Traitement des fichiers sources.
     */
    public void process() {
        // Instanciation du Visitor
        TODOVisitor todoVisitor = new TODOVisitor(env);

        // On boucle sur toutes les Annotations :
        for ( Declaration d : env.getTypeDeclarations() ) {

            // On "visite" chacune des déclarations trouvées :
            d.accept( DeclarationVisitors.getSourceOrderDeclarationScanner(
                todoVisitor, DeclarationVisitors.NO_OP ) );
        }
    }
}

```

La méthode **process()** parcourt la liste des déclarations de l'environnement d'**APT**, et utilise la méthode **accept()** de ces déclarations pour les visiter. On utilise pour cela la méthode **DeclarationVisitors.getSourceOrderDeclarationScanner()** qui permet de scanner totalement une déclaration de type grâce au visitor.

3.3 - DeclarationVisitor

L'interface **DeclarationVisitor** définit 16 méthodes **visit***()** permettant chacune de visiter une déclaration particulière. Lorsqu'on passe une instance de **DeclarationVisitor** à la méthode **accept()** d'une **Declaration**, la méthode **visit***()** la plus appropriée est appelée.

Dans notre **AnnotationProcessor**, nous utilisons la méthode statique **DeclarationVisitors.getSourceOrderDeclarationScanner()** qui permet de totalement scanner une déclaration et toutes ses sous-déclarations en utilisant une instance spécifique de **DeclarationVisitor**.

Notre instance de **DeclarationVisitor** utilisera seulement la méthode **visitDeclaration()** qui sera appelée pour n'importe quel type de déclaration. Pour simplifier l'implémentation de l'interface **DeclarationVisitor**, la classe **SimpleDeclarationVisitor** implémente toutes ces méthodes. Du coup il suffit de redéfinir les méthodes dont on a réellement besoin.

Dans un premier temps, notre visiteur se contentera d'afficher une note dans la console contenant le libellé de l'annotation **@TODO**:

TODOVisitor.java

```
public class TODOVisitor extends SimpleDeclarationVisitor{

    protected final AnnotationProcessorEnvironment env;

    public TODOVisitor (AnnotationProcessorEnvironment env) {
        this.env = env;
    }

    /**
     * Pour tout type de déclaration, on affiche un message si
     * l'Annotation @TODO est présente...
     * De même, on affiche un message pour tous les @TODO
     * contenu dans l'annotation @TODOs
     */
    @Override
    public void visitDeclaration(Declaration decl) {

        // On regarde si la déclaration possède une annotation TODO
        TODO todo = decl.getAnnotation(TODO.class);
        // Et on l'affiche éventuellement :
        if (todo!=null)
            printMessage(decl, todo);

        // On fait la même chose pour l'annotation TODOs :
        TODOs todos = decl.getAnnotation(TODOs.class);
        if (todos!=null) {
            // On affiche les message pour tout les TODOs :
            for ( TODO t : todos.value() )
                printMessage(decl,t);
        }

    }

    /**
     * Affiche dans la console l'annotation TODO.
     * @param decl
     * @param todo
     */
    public void printMessage (Declaration decl, TODO todo) {
        m.printNotice(decl.getSimpleName() + " : " + todo.value() );
    }
}
```

Dans la méthode **visitDeclaration()** (qui sera appelée pour toutes les déclarations), on recherche la présence de l'annotation **@TODO** et **@TODOs** afin d'afficher un message lors du traitement par **APT**.

3.4 - Compilation et utilisation

Nous pouvons désormais compiler notre *"fabrique"* ainsi que nos annotations. Il ne faut pas oublier d'ajouter la librairie **tools.jar** du répertoire */lib* du JDK !

Enfin, nous allons créer un fichier jar contenant toutes nos classes afin de faciliter l'utilisation de ces annotations et y ajouter le fichier **com.sun.mirror.apt.AnnotationProcessorFactory** dans le répertoire **META-INF/services** du jar afin de ne pas avoir besoin de spécifier la *"fabrique"* dans la ligne de commande d'**APT**. Ce fichier contiendra le nom complet de notre classe **Factory**.

Une fois le fichier jar créé, nous pouvons compiler un code source java qui utilise notre annotation **@TODO** :

Test.java

```
import com.developpez.adiguba.annotation.TODO;
import static com.developpez.adiguba.annotation.TODO.Level.*;

public class Test {

    @TODO("Utiliser une annotation a la place d'une String")
    protected String day;

    @TODO(value="Ecrire le code de la methode", level=IMPORTANT)
    public void method() {
    }

}
```

Lorsqu'on compile avec **APT** grâce à la ligne de commande suivante (où *tutoriel-annotations.jar* correspond à notre fichier jar contenant les annotations et la factory) :

En compilant notre classe de test avec **APT**, on obtient le résultat suivant :

apt -cp tutoriel-annotations.jar Test.java

```
Note: day : Utiliser une annotation a la place d'une String
Note: method : Ecrire le code de la methode
```

On est désormais informé lors de la compilation des différentes annotations **@TODO**. On peut encore pousser plus loin le principe en générant différents types de message selon le niveau de l'annotation **@TODO**. On peut également utiliser l'option *"-Arelease"* pour encore augmenter l'importance de ces messages lorsqu'on souhaite compiler une version stable de notre application.

Ainsi notre classe **TODOVisitor** devient :

TODOVisitor.java

```
public class TODOVisitor extends SimpleDeclarationVisitor{

    protected final AnnotationProcessorEnvironment env;
    /** Indique si l'option -Arelease fait partie de la ligne de commande. */
    protected final boolean isRelease;

    public TODOVisitor (AnnotationProcessorEnvironment env) {
        this.env = env;
        isRelease = env.getOptions().containsKey(SimpleAnnotationProcessorFactory.RELEASE);
    }

    /**
     * Pour tout type de déclaration, on affiche un message si
     * l'Annotation @TODO est présente...
     * De même, on affiche un message pour tous les @TODO
     * contenu dans l'annotation @TODOs
     */
    @Override
    public void visitDeclaration(Declaration decl) {
```

TODOVisitor.java

```

// On regarde si la déclaration possède une annotation TODO
TODO todo = decl.getAnnotation(TODO.class);
// Et on l'affiche éventuellement :
if (todo!=null)
    printMessage(decl, todo);

// On fait la même chose pour l'annotation TODOs :
TODOs todos = decl.getAnnotation(TODOs.class);
if (todos!=null) {
    // On affiche les message pour tout les TODOs :
    for ( TODO t : todos.value() )
        printMessage(decl,t);
}

}

/**
 * Affiche dans la console l'annotation TODO.
 * @param decl
 * @param todo
 */
public void printMessage (Declaration decl, TODO todo) {
    if (isRelease)
        printReleaseError (decl, todo);
    else
        printDebuggingInfo (decl, todo);
}

/**
 * Affiche le message pendant la phase de developpement.
 * Les messages IMPORTANT sont affichées comme des 'warnings'.
 * Les messages NORMAL sont affichées comme des 'notes'.
 * Les messages MINEUR sont affichées comme des 'notes' simples (sans position dans le
code).
 * @param decl La déclaration qui contient l'annotation.
 * @param todo L'Annotation affichée.
 */
public void printDebuggingInfo (Declaration decl, TODO todo) {
    Messenger m = env.getMessenger();

    switch (todo.level()) {
        case IMPORTANT:
            m.printWarning(decl.getPosition(),
                decl.getSimpleName() + " : " + todo.value() );
            break;
        case NORMAL:
            m.printNotice(decl.getPosition(),
                decl.getSimpleName() + " : " + todo.value() );
            break;
        case MINEUR:
            m.printNotice(decl.getSimpleName() + " : " + todo.value() );
            break;
    }
}

/**
 * Affiche le message pendant la compilation en mode release.
 * Les messages IMPORTANT sont affichées comme des 'erreurs'.
 * Les messages NORMAL sont affichées comme des 'warning'.
 * Les messages MINEUR sont affichées comme des 'notes'.
 * @param decl La déclaration qui contient l'annotation.
 * @param todo L'Annotation affichée.
 */
public void printReleaseError (Declaration decl, TODO todo) {
    Messenger m = env.getMessenger();

    switch (todo.level()) {
        case IMPORTANT:
            m.printError(decl.getPosition(),
                decl.getSimpleName() + " : " + todo.value() );
            break;
        case NORMAL:
            m.printWarning(decl.getPosition(),
                decl.getSimpleName() + " : " + todo.value() );
            break;
        case MINEUR:
            m.printNotice(decl.getPosition(),
                decl.getSimpleName() + " : " + todo.value() );
            break;
    }
}

```

TODOVisitor.java

```

    }
}

```

Ce qui permettra d'obtenir les sorties suivantes selon que l'on utilise l'option **-Arelease** ou non :

apt -cp tutoriel-annotations.jar Test.java

```

Test.java:7: Note: day : Utiliser une annotation a la place d'une String
protected String day;
               ^

Test.java:11: warning: method : Ecrire le code de la methode
public void method() {
           ^

1 warning

```

apt -cp tutoriel-annotations.jar Test.java -Arelease

```

Test.java:7: warning: day : Utiliser une annotation a la place d'une String
protected String day;
               ^

Test.java:11: method : Ecrire le code de la methode
public void method() {
           ^

1 error
1 warning

```



Si **APT** trouve des annotations qui ne sont traitées par aucune fabrique, il générera un warning du style :

```
warning: Annotation types without processors
```

Ce message sera affiché notamment pour les annotations standards de l'API Java. Pour remédier a cela il suffit d'ajouter dans le fichier **META-INF/services/com.sun.mirror.apt.AnnotationProcessorFactory** le nom d'une fabrique qui traitera ces annotations (même si elle ne fait rien), par exemple :

StandardAnnotationProcessorFactory.java

```

public class StandardAnnotationProcessorFactory implements AnnotationProcessorFactory {

    /** Collection contenant le nom des Annotations supportées. */
    protected Collection<String> supportedAnnotationTypes =
        Arrays.asList(
            // Annotation Standard
            "java.lang.*",
            // Meta-Annotation
            "java.lang.annotation.*"
        );

    /**
     * Retourne la liste des annotations supportées par cette Factory.
     */
    public Collection<String> supportedAnnotationTypes() {
        return supportedAnnotationTypes;
    }

    /**
     * Retourne la liste des options supportées par cette Factory.
     */
    public Collection<String> supportedOptions() {
        return Collections.emptyList();
    }

    /**
     * Retourne AnnotationProcessors.NO_OP (Pas de traitement).
     */
    public AnnotationProcessor getProcessorFor(Set<AnnotationTypeDeclaration> atds,
        AnnotationProcessorEnvironment env) {

```

StandardAnnotationProcessorFactory.java

```

        return AnnotationProcessors.NO_OP;
    }
}

```

3.5 - Un autre exemple : @EmptyConstructor

Un autre exemple de l'utilisation de ce procédé : de nombreux frameworks Javainstancient dynamiquement des objets qui implémentent une interface particulière. La présence d'un constructeur vide est souvent requise mais il était impossible jusqu'à présent de vérifier cela à la compilation...

C'est désormais possible avec les annotations :

EmptyConstructor

```

@Inherited
@Documented
@Target(ElementType.TYPE)
public @interface EmptyConstructor {
}

```

Cette simple annotation vide permet de signaler que la présence d'un constructeur vide est requis. Il suffit d'utiliser une classe **Visitor** adaptée qui recherchera pour chaque classe la présence de cette annotation et d'un constructeur vide :

EmptyConstructorVisitor.java

```

public class EmptyConstructorVisitor extends SimpleDeclarationVisitor {

    /** L'environnement du processeur d'annotation. */
    protected final AnnotationProcessorEnvironment env;

    /**
     * Constructeur.
     * @param env L'environnement du processeur d'annotation.
     */
    public EmptyConstructorVisitor (AnnotationProcessorEnvironment env) {
        this.env = env;
    }

    /**
     * Méthode appelé pour chaque déclaration d'une classe.
     */
    @Override
    public void visitClassDeclaration(ClassDeclaration classDecl) {

        // Si on n'a pas de constructeur vide et
        // que l'annotation EmptyConstructor a été trouvé
        // --> On affiche un message d'erreur :
        if ( !hasEmptyConstructor(classDecl) &&
            hasAnnotation(classDecl, EmptyConstructor.class) ) {
            env.getMessenger().printError( classDecl.getPosition(),
                "Un constructeur vide est requis par @EmptyConstructor.");
        }
    }

    /**
     * Cette méthode indique si la déclaration de classe passée en
     * paramètre possède un constructeur vide.
     * @param classDecl La déclaration de classe à analyser.
     * @return true si la classe possède un constructeur vide, false sinon.
     */
    public boolean hasEmptyConstructor (ClassDeclaration classDecl) {
        // On parcourt la liste des constructeurs :
        for ( ConstructorDeclaration c : classDecl.getConstructors() ) {
            if ( c.getParameters().isEmpty() )
                return true; // On a trouvé un constructeur vide
        }
        return false; // Pas de constructeur vide
    }
}

```

EmptyConstructorVisitor.java

```

/**
 * Recherche récursivement dans les interfaces si une Annotation est présente.<br/>
 * En effet, les annotations présentes dans les interfaces ne sont pas hérité...
 * @param typeDecl Le type de base de la recherche.
 * @param annotationClass La classe de l'annotation à rechercher
 * @return <b>true</b> si une annotation de type <b>annotationClass</b> est trouvée,
 * <b>false</b> sinon.
 */
public boolean hasAnnotation (TypeDeclaration typeDecl, Class<? extends Annotation>
annotationClass) {

    // Pour chaque interface directement implémenté :
    for ( InterfaceType iType : typeDecl.getSuperinterfaces() ) {
        InterfaceDeclaration superInterface = iType.getDeclaration();

        // On regarde si l'interface possède l'annotation
        if ( superInterface.getAnnotation(EmptyConstructor.class)!=null )
            return true; // On en a trouvé une : pas la peine de continuer ; )

        // Sinon on regarde dans les interfaces parentes :
        if ( hasAnnotation(superInterface,annotationClass) )
            return true; // On en a trouvé une : pas la peine de continuer ; )

    }
    // Si on arrive ici cela signifie que l'on n'a pas trouvé l'Annotation.
    return false;
}
}

```

Les possibilités offertes par **APT** sont nombreuses et permettent aux interfaces de proposer une conception par contrat plus élaborée.



Le fichier **annotations-src.zip** (16 Ko) comporte tous les sources de cette section ainsi que l'archive jar contenant les annotations et les fabriques pour **APT** :
[ftp://ftp-developpez.com/adiguba/tutoriels/java/tiger/annotations/annotations-src.zip](http://ftp-developpez.com/adiguba/tutoriels/java/tiger/annotations/annotations-src.zip)

(mirroir : <http://ftp-developpez.com/adiguba/tutoriels/java/tiger/annotations/annotations-src.zip>).

3.6 - Restriction à l'utilisation d'APT

L'utilisation d'**APT** apporte de nombreuses possibilités très intéressantes, en particulier lorsqu'on utilise des librairies externes. On peut en effet imaginer que les futures librairies puissent utiliser cela pour ajouter des contraintes à la compilation.

Surtout que nous n'avons survolé qu'une petite partie des possibilités d'**APT**. Il est en effet également possible de générer des fichiers, des sources et même des classes Java pendant le traitement.

Toutefois cela nécessite de compiler avec **APT** à la place de **javac**. Si cela ne pose pas vraiment de problème pour de petits projets compilés "à la main", le problème reste posé lorsqu'on utilise un EDI.



Si vous utilisez **ant**, vous pouvez utiliser dès à présent **APT** malgré le fait qu'il ne propose pas de tâche **<apt/>**. Il suffit en effet d'utiliser la tâche **<javac/>** avec les attributs **fork="yes"** et **executable="apt"**. Par exemple :

build.xml

```

<javac fork="yes" executable="apt" srcdir="${src}" destdir="${build}">
  <classpath>
    <pathelement path="tutoriel-annotations.jar"/>
  </classpath>
  <compilerarg value="-Arelease"/>
</javac>

```



```
build.xml
```

```
</javac>
```

Toutefois, à ma connaissance, les principaux EDI actuels ne semblent pas supporter les fonctionnalités apportées par **APT** ... ce qui risque de retarder l'utilisation des annotations pendant la compilation.



*Je parle ici d'**Eclipse 3.1**, de **NetBeans 4.1** et de **JBuilder 2005 Foundation** dans leurs versions de base sans plugin et sans configuration avancée.*

Il y a de fortes chances que ceci soit prochainement implémenté.


Malgré cela, l'utilisation d'**APT** a de fortes chances de se généraliser, et donc de se retrouver dans les futures versions de ces EDI.

En effet, sur le forum **Mustang** (nom de code du prochain Java), on peut trouver un message où il est question de déprécier **javac** (plus précisément, d'intégrer les fonctionnalités de **APT** dans **javac**) :

<http://forums.java.net/jive/thread.jspa?messageID=1402&tstart=0>

4 - Les annotations et l'introspection

L'introspection permet de manipuler des objets sans pour autant connaître leurs type exact. On utilise pour cela l'API de réflexion (package **java.lang.reflect**) qui permet d'accéder dynamiquement aux diverses informations d'une classe (constructeurs, méthodes, champs, classe parent, interface implémentée, etc.).


 Cette section nécessite d'avoir quelques notions de l'API de Réflexion de Java. Si ce n'est pas le cas, je vous conseille de consulter le tutoriel de **Ricky81** :

[Introspection en JAVA, présentation de l'API Réflexion.](#)

4.1 - Recherche dynamique des Annotations

L'API de Réflexion a bien entendu évolué avec le langage afin de supporter les nouveautés de Java 5.0, et notamment les Annotations. Pour cela, les classes **Package**, **Class**, **Constructor**, **Method** et **Field** possèdent quatre nouvelles méthodes décrite dans l'interface **AnnotatedElement** :

- **getAnnotation(Class<T>)** qui retourne l'annotation dont le type est passé en paramètre (ou **null** si cette annotation n'est pas présente).
- **getAnnotations()** qui retourne un tableau comportant une instance de toutes les annotations de l'élément.
- **getDeclaredAnnotations()** qui retourne un tableau comportant une instance de toutes les annotations directement présentes sur l'élément (c'est à dire sans les annotations héritées de la classe parente). Seule les **Class** peuvent hériter d'une annotation.
- **isAnnotationPresent(Class<T>)** qui retourne un booléen indiquant si l'annotation dont le type est passé en paramètre est présente sur l'élément ou non. Cette méthode est surtout utile pour les annotations marqueurs (sans attribut).

 L'API de Réflexion ne peut accéder qu'aux annotations dont la rétention est **RUNTIME**. Toutes les autres annotations seront perdues à l'exécution...

Les Annotations renvoyées par les méthodes ci-dessus permettent bien sûr d'accéder aux informations de leurs attributs directement par leurs méthodes. Ainsi, pour récupérer la valeur d'une Annotation posé sur une classe, on peut utiliser le code suivant :

Exemple d'accès aux Annotations

```
// Instanciation de l'objet
Exemple objet = new Exemple();

// On récupère la classe de l'objet :
Class<Exemple> classInstance = objet.getClass();

// On regarde si la classe possède une annotation :
MonAnnotation annotation = classInstance.getAnnotation(MonAnnotation.class);
if (annotation!=null) {
    System.out.println ("Annotation TODO : " + annotation.value() );
}
```

Dans cette exemple, **MonAnnotation** correspond au nom d'une annotation et **Exemple** au nom d'une classe marqué par cette annotation...

Il est ainsi très facile d'accéder aux annotations d'un élément. Cela permet une automatisation simple et aisée de certains traitements.

4.2 - Exemple d'utilisation

Prenons le cas d'une application paramétrée par un ou plusieurs fichiers de configuration. Chacun des modules de l'application doit pouvoir récupérer les propriétés dont il a besoin.

Pour cela, on utilise généralement une classe qui s'occupe de charger les différents fichiers de configuration, et les modules de l'application utilisent cette classe afin d'accéder aux valeurs des propriétés.

Généralement, on obtient un code dans le genre suivant :

MaClasse.java

```
public MaClasse {
    private String config1 = Manager.getInstance().getProperty("property_1");
    private String config2 = Manager.getInstance().getProperty("property_2");
    private String config3 = Manager.getInstance().getProperty("property_3");
    private String config4 = Manager.getInstance().getProperty("property_4");

    public MaClasse () {
    }
}
```

Où **Manager.getInstance()** permet d'accéder à la classe qui gère les fichiers de configuration. Nous allons désormais utiliser une annotation qui indiquera le nom de la propriété :

@Property

```
/**
 * Permet d'indiquer la propriété associée à un champ ou une méthode.
 * @author adiGuba
 */
@Documented
@Retention(RUNTIME)
@Target(FIELD)
public @interface Property {

    /** Nom de la propriété liée à cet élément. */
    String value ();
    /** Valeur si l'élément est absent (optionnel / "" par défaut). */
    String missing() default "";
}
```

@Retention(RUNTIME) permet d'accéder à l'annotation pendant l'exécution et **@Target(FIELD)** limite son utilisation aux champs d'une classe. notre classe devient ainsi :

MaClasse.java

```
public MaClasse {
    @Property("property_1") private String config1;
    @Property("property_2") private String config2;
    @Property("property_3") private String config3;
    @Property("property_4") private String config4;

    public MaClasse () {
        Manager.getInstance().initialize(this);
    }
}
```

Les différents champs de notre classe sont désormais annotés et on utilise **Manager.getInstance().initialize(this)** pour les initialiser en utilisant l'API de Réflexion.

Ainsi, la méthode **initialize()** recherche tous les champs de la classe qui possède l'annotation. Pour chacun de ces champs, on récupère la valeur de l'annotation et on l'utilise comme clef de la propriété du fichier de configuration. Par exemple :

initialize() recherche des champs annotées pour les initialiser

```

public void initialize (Object pObjectInstance)
    throws PropertyManagerException {

    // On récupère la classe de l'objet.
    Class<?> lClassInstance = pObjectInstance.getClass();

    // Pour tous les champs de l'objet :
    for ( Field f : pClassInstance.getDeclaredFields() ) {
        // On recherche l'annotation @Property :
        Property property = f.getAnnotation(Property.class);
        if (property!=null) {
            // On récupère la valeur dans le Properties
            String value = lProp.getProperty(lPrefix+property.value());
            // Si la propriété n'existe pas on met la valeur par défaut:
            if (value==null)
                value= property.missing();

            try {
                // Si on n'a pas accès aux champs, on force son accessibilité
                // Cela permet de modifier les champs firendly, protected et private

                boolean pAccessible = f.isAccessible();
                if (!pAccessible)
                    f.setAccessible(true);
                f.set(pObjectInstance, value);
            } catch (Exception lException) {
                throw new PropertyManagerException(
                    "Impossible d'assigner le champ '" +
                    value + "'.", lException );
            }
        } // fin property!=null
    } // fin for getDeclaredFields()
}

```

On pourrait même utiliser une seconde annotation à poser sur la classe qui spécifierait le fichier de configuration à utiliser, et/ou un préfixe à utiliser pour le nom de la clef de la propriété...



Le fichier **annotations-src-rt.zip** (14 Ko) comporte tous les sources de cette section ainsi que l'archive jar contenant les annotations et les classes compilées :

[ftp://ftp-developpez.com/adiguba/tutoriels/java/tiger/annotations/annotations-src-rt.zip](http://ftp-developpez.com/adiguba/tutoriels/java/tiger/annotations/annotations-src-rt.zip)

(miroir : <http://ftp-developpez.com/adiguba/tutoriels/java/tiger/annotations/annotations-src-rt.zip>).

Il est également possible d'améliorer ceci en gérant le transtypage des propriétés selon le type réel du champs de la classe, ainsi qu'avec une meilleure gestion des exceptions... mais le principe général reste le même...

Conclusion

Si les Annotations sont d'ores et déjà utilisables dans n'importe quelle application **Java 5.0**, l'utilisation d'**APT** souffre pour le moment d'un support inexistant dans les principaux EDI Java du marché, ce qui risque de retarder son adoption.

Malgré cela, il y a de grande chances que l'utilisation des Annotations se généralise et apporte un nouvel horizon dans la conception par contrat. En effet, pour le moment cela se limitait principalement à implémenter une interface ou étendre une classe ... Il est désormais possible d'étendre cela à la plupart des éléments du langage. Cela apporte une multitude de possibilités. De plus, en couplant l'introspection et **APT**, il est possible à la fois d'automatiser plusieurs tâches tout en conservant une sécurité maximum en effectuant des vérifications lors de la compilation.

Ainsi, la prochaine version de la plateforme J2EE (**J2EE 1.5**) utilisera les annotations pour simplifier l'utilisation des **EJB** (en générant automatiquement les interfaces **Remote**). **JDBC 4.0** les utilisera également afin d'associer une classe Java avec une table de la base de données et pour automatiser les requêtes. **JUnit 4** devrait également les utiliser alors que **TestNG** (un autre outil de test inspiré de JUnit) les utilise déjà ...

Il serait donc vraiment dommage que les prochaines versions des EDI ne permettent pas le traitement des Annotations lors de la compilation comme avec **APT**...