Recherche dans DEJ avec Google Rechercher

Copyright (C) 1999-2014 Jean-Michel DOUDOUX.

10. Les annotations

Chapitre 10

Niveau : Intermédiaire

Java SE 5 a introduit les annotations qui sont des métadonnées incluses dans le code source. Les annotations ont été spécifiées dans la JSR 175 : leur but est d'intégrer au langage Java des métadonnées.

Des métadonnées étaient déjà historiquement mises en oeuvre avec Java notamment avec Javadoc ou exploitées par des outils tiers notamment XDoclet : l'outil open source XDoclet propose depuis longtemps des fonctionnalités similaires aux annotations. Avant Java 5, seul l'outil Javadoc utilisait des métadonnées en standard pour générer une documentation automatique du code source.

Javadoc propose l'annotation @deprecated qui bien qu'utilisée dans les commentaires permet de marquer une méthode comme obsolète et de faire afficher un avertissement par le compilateur.

Le défaut de Javadoc est d'être trop spécifique à l'activité de génération de documentation même si le tag deprecated est aussi utilisé par le compilateur.

Depuis leur introduction dans Java 5, les annotations sont de plus en plus dans le développement d'applications avec les plate-formes Java SE et Java EE.

Ce chapitre contient plusieurs sections :

- La présentation des annotations
- La mise en oeuvre des annotations
- L'utilisation des annotations
- Les annotations standards
- Les annotations communes (Common Annotations)
- Les annotations personnalisées
- L'exploitation des annotations
- L'API Pluggable Annotation Processing
- Les ressources relatives aux annotations

10.1. La présentation des annotations

Les annotations de Java 5 apportent une standardisation des métadonnées dans un but généraliste. Ces métadonnées associés aux entités Java peuvent être exploitées à la compilation ou à l'exécution.

Java a été modifié pour permettre la mise en oeuvre des annotations :

- une syntaxe dédiée a été ajoutée dans Java pour permettre la définition et l'utilisation d'annotations.
- le bytecode est enrichi pour permettre le stockage des annotations.

Les annotations peuvent être utilisées avec quasiment tous les types d'entités et de membres de Java : packages, classes, interfaces, constructeurs, méthodes, champs, paramètres, variables ou annotations elles même.

Java 5 propose plusieurs annotations standard et permet la création de ses propres annotations.

Une annotation précède l'entité qu'elle concerne. Elle est désignée par un nom précédé du caractère @.

Il existe plusieurs catégories d'annotations :

- les marqueurs (markers) : ces annotations ne possèdent pas d'attribut (exemple : @Deprecated, @Override, ...)
- les annotations paramétrées (single value annotations) : ces annotations ne possèdent qu'un seul attribut(exemple : @MonAnnotation("test"))
- les annotations multi paramétrées (full annotations) : ces annotations possèdent plusieurs attributs (exemple : @MonAnnotation(arg1="test 3", arg2="test 2", arg3="test3"))

Les arguments fournis en paramètres d'une annotation peuvent être de plusieurs types : les chaînes de caractères, les types primitifs, les énumérations, les annotations, le type Class.

Les annotations sont définies dans un type d'annotation. Une annotation est une instance d'un type d'annotation. Les annotations peuvent avoir des valeurs par défaut.

La disponibilité d'une annotation est définie grâce à une retention policy.

Les usages des annotations sont nombreux : génération de documentions, de code, de fichiers, ORM (Object Relational Mapping), ...

Les annotations ne sont guère utiles sans un mécanisme permettant leur traitement.

Une API est proposée pour assurer ces traitements : elle est regroupée dans les packages com.sun.mirror.apt, com.sun.mirror.declaration, com.sun.mirror.type et com.sun.mirror.util.

L'outil apt (annotation processing tool) permet un traitement des annotations personnalisées durant la phase de compilation (compile time). L'outil apt permet la génération de nouveaux fichiers mais ne permet pas de modifier le code existant.

Il est important de se souvenir que lors du traitement des annotations le code source est parcouru mais il n'est pas possible de modifier ce code.

L'API reflexion est enrichie pour permettre de traiter les annotations lors de la phase d'exécution (runtime).

Java 6 intègre deux JSR concernant les annotations :

- Pluggable Annotation Processing API (JSR 269)
- Common Annotations (JSR 250)

L'Api Pluggable Annotation Processing permet d'intégrer le traitement des annotations dans le processus de compilation du compilateur Java ce qui évite d'avoir à utiliser apt.

Les annotations vont évoluer dans la plate-forme Java notamment au travers de plusieurs JSR qui sont en cours de définition :

- JSR 305 Annotations for Software Defect Detection
- JSR 308 Annotations on Java Types : doit permettre de mettre en oeuvre les annotations sur tous les types notamment les generics et sur les variables locales à l'exécution.

10.2. La mise en oeuvre des annotations

Les annotations fournissent des informations sur des entités : elles n'ont pas d'effets directs sur les entités qu'elles concernent.

Les annotations utilisent leur propre syntaxe. Une annotation s'utilise avec le caractère @ suivi du nom de l'annotation : elle doit obligatoirement précéder l'entité qu'elle annote. Par convention, les annotations s'utilisent sur une ligne dédiée.

Les annotations peuvent s'utiliser sur les packages, les classes, les interfaces, les méthodes, les constructeurs et les paramètres de méthodes.

Exemple :

```
1. @Override
2. public void maMethode() {
3. }
```

Une annotation peut avoir un ou plusieurs attributs : ceux ci sont précisés entre parenthèses, séparés par une virgule. Un attribut est de la forme clé=valeur.

Exemple :

```
1. @SuppressWarnings(value = "unchecked")
2. void maMethode() { }
```

Lorsque l'annotation ne possède qu'un seul attribut, il est possible d'omettre son nom.

Exemple :

```
1. @SuppressWarnings("unchecked")
2. void maMethode() { }
```

Un attribut peut être de type tableau : dans ce cas, les différentes valeurs sont fournies entre accolades, chaque valeur placée entre guillemets et séparée de la suivante par une virgule.

Exemple :

```
@SuppressWarnings(value={"unchecked", "deprecation"})
```

Le tableau peut contenir des annotations.

Exemple :

```
@TODOItems({
  @Todo(importance = Importance.MAJEUR,
      description = "Ajouter le traitement des erreurs",
      assigneA = "JMD",
      dateAssignation = "07-11-2007"),
  @Todo(importance = Importance.MINEURE,
      description = "Changer la couleur de fond",
      assigneA = "JMD",
      dateAssignation = "13-12-2007")
})
```

10.3. L'utilisation des annotations

Les annotations prennent une place de plus en plus importante dans la plate-forme Java et de nombreuses API open source.

Les utilisations des annotations concernent plusieurs fonctionnalités :

- Utilisation par le compilateur pour détecter des erreurs ou ignorer des avertissements
- Documentation
- Génération de code
- Génération de fichiers

10.3.1. La documentation

Les annotations peuvent être mises en oeuvre pour permettre la génération de documentations indépendantes de JavaDoc : listes de choses à faire, de services ou de composants, ...

Il peut par exemple être pratique de rassembler certaines informations mises sous la forme de commentaires dans des annotations pour permettre leur traitement.

Par exemple, il est possible de définir une annotation qui va contenir les métadonnées relatives aux informations sur une classe. Traditionnellement, une classe débute par un commentaire d'en-tête qui contient des informations sur l'auteur, la date de création, les modifications, ... L'idée est de fournir ces informations dans une annotation dédiée. L'avantage est de facilement extraire et manipuler ces données qui ne seraient qu'informatives sous leur forme de commentaires.

10.3.2. L'utilisation par le compilateur

Les trois annotations fournies en standard avec la plate-forme entrent dans cette catégorie qui consiste à faire réaliser par le compilateur quelques contrôles basiques.

10.3.3. La génération de code

Les annotations sont particulièrement adaptées à la génération de code source afin de faciliter le travail des développeurs notamment sur des tâches répétitives.

Attention, le traitement des annotations ne peut pas modifier le code existant mais simplement créer de nouveaux fichiers sources.

10.3.4. La génération de fichiers

Les API standards ou les frameworks open source nécessitent fréquemment l'utilisation de fichiers de configuration ou de déploiement généralement au format XML.

Les annotations peuvent proposer une solution pour maintenir le contenu de ces fichiers par rapport aux entités incluses dans le code de l'application.

La version 5 de Java EE fait un important usage des annotations dans le but de simplifier les développements de certains composants notamment les EJB, les entités et les services web. Pour cela, l'utilisation de descripteurs est remplacée par l'utilisation d'annotations ce qui rend le code plus facile à développer et plus clair.

10.3.5. Les API qui utilisent les annotations

De nombreuses API standards utilisent les annotations depuis leur intégration dans Java notamment :

- JAXB 2.0 : JSR 222 (Java Architecture for XML Binding 2.0)
- Les services web de Java 6 (JAX-WS) : JSR 181 (Web Services Metadata for the Java Platform) et JSR 224 (Java APIs for XML Web Services 2.0 API)
- Les EJB 3.0 et JPA : JSR 220 (Enterprise JavaBeans 3.0)
- Servlets 3.0, CDI
- ...

De nombreuses API open source utilisent aussi les annotations notamment JUnit, TestNG, Hibernate, ...

10.4. Les annotations standards

Java 5 propose plusieurs annotations standards.

10.4.1. L'annotation @Deprecated

Cette annotation a un rôle similaire au tag de même nom de Javadoc.

C'est un marqueur qui précise que l'entité concernée est obsolète et qu'il ne faudrait plus l'utiliser. Elle peut être utilisée avec une classe, une interface ou un membre (méthode ou champ)

Exemple :

```
01. public class TestDeprecated {
02.
03.   public static void main(String[] args) {
04.     MaSousClasse td = new MaSousClasse();
05.     td.maMethode();
06.   }
07. }
08.
09. @Deprecated
10. class MaSousClasse {
11.
12.   /**
13.    * Afficher un message de test
14.    * @deprecated methode non compatible
15.    */
16.   @Deprecated
17.   public void maMethode() {
18.     System.out.println("test");
19.   }
20. }
```

Les entités marquées avec l'annotation @Deprecated devraient être documentées avec le tag @deprecated de Javadoc en lui fournissant la raison de l'obsolescence et éventuellement l'entité de substitution.

Il est important de tenir compte de la casse : @Deprecated pour l'annotation et @deprecated pour Javadoc.

Lors de la compilation, le compilateur donne une information si une entité obsolète est utilisée.

Exemple :

```
1. C:\Documents and Settings\jmd\workspace\Tests>javac TestDeprecated.java
2. Note: TestDeprecated.java uses or overrides a deprecated API.
3. Note: Recompile with -Xlint:deprecation for details.
```

L'option -Xlint :deprecation permet d'afficher le détail sur les utilisations obsolètes.

Exemple :

```
01. C:\Documents and Settings\jmd\workspace\Tests>javac -Xlint:deprecation TestDepre
02. cated.java
03. TestDeprecated.java:7: warning: [deprecation] MaSousClasse in unnamed package ha
04. s been deprecated
05.     MaSousClasse td = new MaSousClasse();
06.     ^
07. TestDeprecated.java:7: warning: [deprecation] MaSousClasse in unnamed package ha
08. s been deprecated
09.     MaSousClasse td = new MaSousClasse();
10.                           ^
11. TestDeprecated.java:8: warning: [deprecation] maMethode() in MaSousClasse has be
12. en deprecated
13.     td.maMethode();
14.       ^
15. 3 warnings
```

Il est aussi possible d'utiliser l'option -deprecation de l'outil javac.

10.4.2. L'annotation @Override

Cette annotation est un marqueur utilisé par le compilateur pour vérifier la réécriture de méthodes héritées.

@Override s'utilise pour annoter une méthode qui est une réécriture d'une méthode héritée. Le compilateur lève une erreur si aucune méthode héritée ne correspond.

Exemple :

```
1. @Override
2. public void maMethode() {
3. }
```

Son utilisation n'est pas obligatoire mais recommandée car elle permet de détecter certains problèmes.

Exemple :

```
01. public class MaClasseMere {
02.
03. }
04.
05. class MaClasse extends MaClasseMere {
06.
07.   @Override
08.   public void maMethode() {
09.
10.   }
11. }
```

Ceci est particulièrement utile pour éviter des erreurs de saisie dans le nom des méthodes à redéfinir.

Exemple :

```
01. public class TestOverride {
02.   private String nom;
03.   private long id;
04.
05.   public int hasCode() {
06.     final int PRIME = 31;
07.     int result = 1;
08.     result = PRIME * result + (int) (id ^ (id >>> 32));
09.     result = PRIME * result + ((nom == null) ? 0 : nom.hashCode());
10.     return result;
11.   }
12. }
```

Dans l'exemple ci-dessous, le développeur souhaitait redéfinir la méthode hashCode() mais une faute frappe à simplement défini une nouvelle méthode nommée hasCode(). Cette classe se compile parfaitement mais elle comporte une erreur qui est signalée en utilisation l'annotation @Override

Exemple :

```
01. public class TestOverride {
02.   private String nom;
03.   private long id;
04.
05.   @Override
06.   public int hasCode() {
07.     final int PRIME = 31;
08.     int result = 1;
09.     result = PRIME * result + (int) (id ^ (id >>> 32));
10.     result = PRIME * result + ((nom == null) ? 0 : nom.hashCode());
11.     return result;
12.   }
13. }
```

Résultat :

```
1. C:\Documents and Settings\jmd\workspace\Tests>javac TestOverride.java
2. TestOverride.java:6: method does not override or implement a method from a super
3. type
4.   @Override
5.   ^
6. 1 error
```

10.4.3. L'annotation @SuppressWarning

L'annotation @SuppressWarning permet de demander au compilateur d'inhiber certains avertissements qui sont pris en compte par défaut.

La liste des avertissements utilisables dépend du compilateur. Un avertissement utilisé dans l'annotation non reconnu par le compilateur ne provoque par d'erreur mais éventuellement un avertissement.

Le compilateur fourni avec le JDK supporte les avertissements suivants :

Nom	Rôle
deprecation	Vérification de l'utilisation d'entités déclarées deprecated
unchecked	Vérification de l'utilisation des generics
fallthrough	Vérification de l'utilisation de l'instruction break dans les cases des instructions switch
path	Vérification des chemins fournis en paramètre du compilateur
serial	Vérification de la définition de la variable serialVersionUID dans les beans
finally	Vérification de l'absence d'instruction return dans une clause finally

Il est possible de passer en paramètres plusieurs types d'avertissements sous la forme d'un tableau

Exemple :

```
1. @SuppressWarnings(value={"unchecked", "fallthrough"})
```

L'exemple ci-dessous génère un avertissement à la compilation

Exemple :

```
01. import java.util.ArrayList;
02. import java.util.List;
03.
04. public class TestSuppresWarning {
05.   public static void main(String[] args) {
06.     List donnees = new ArrayList();
07.     donnees.add("valeur1");
08.   }
09. }
```

Résultat :

```
1. C:\Documents and Settings\jmd\workspace\Tests>javac TestSuppresWarning.java
2. Note: TestSuppresWarning.java uses unchecked or unsafe operations.
3. Note: Recompile with -Xlint:unchecked for details.
```

L'option -Xlint :unchecked permet d'obtenir des détails

Exemple :

```
1. C:\Documents and Settings\jmd\workspace\Tests>javac -Xlint:unchecked TestSuppres
2. Warning.java
3. TestSuppresWarning.java:8: warning: [unchecked] unchecked call to add(E) as a me
4. mber of the raw type java.util.List
5.     donnees.add("valeur1");
6.                ^
7. 1 warning
```

Pour supprimer cet avertissement, il faut utiliser les générics dans la déclaration de la collection ou utiliser l'annotation SuppressWarning.

Exemple :

```
01. import java.util.ArrayList;
02. import java.util.List;
03.
04. @SuppressWarnings("unchecked")
05. public class TestSuppresWarning {
06.   public static void main(String[] args) {
07.     List donnees = new ArrayList();
08.     donnees.add("valeur1");
09.   }
10. }
```

Il n'est pas recommandé d'utiliser cette annotation mais plutôt d'apporter une solution à l'avertissement.

10.5. Les annotations communes (Common Annotations)

Les annotations communes sont définies par la JSR 250 et sont intégrées dans Java 6. Leur but est de définir des annotations couramment utilisées et ainsi d'éviter leur redéfinition pour chaque outil qui en aurait besoin.

Les annotations définies concernent :

- la plate-forme standard dans le package javax.annotation (@Generated, @PostConstruct, @PreDestroy, @Resource, @Resources)
- la plate-forme entreprise dans le package javax.annotation.security (@DeclareRoles, @DenyAll, @PermitAll, @RolesAllowed, @RunAs).

10.5.1. L'annotation @Generated

De plus en plus d'outils ou de frameworks génèrent du code source pour faciliter la tâche des développeurs notamment pour des portions de code répétitives ayant peu de valeur ajoutée.

Le code ainsi généré peut être marqué avec l'annotation @Generated.

Exemple :

```
1. @Generated(
2.     value = "entite.qui.a.genere.le.code",
3.     comments = "commentaires",
4.     date = "12 April  2008"
5. )
6. public void toolGeneratedCode(){
7. }
```

L'attribut obligatoire value permet de préciser l'outil à l'origine de la génération

Les attributs facultatifs comments et date permettent respectivement de fournir un commentaire et la date de génération.

Cette annotation peut être utilisée sur toutes les déclarations d'entités.

10.5.2. Les annotations @Resource et @Resources

L'annotation @Resource définit une ressource requise par une classe. Typiquement, une ressource est par exemple un composant Java EE de type EJB ou JMS.

L'annotation @Resource possède plusieurs attributs :

Attribut	Description
authentificationType	Type d'authentification pour utiliser la ressource (Resource.AuthenticationType.CONTAINER ou Resource.AuthenticationType.APPLICATION)
description	Description de la ressource
mappedName	Nom de la ressource spécifique au serveur utilisé (non portable)
name	Nom JNDI de la ressource
shareable	Booléen qui précise si la ressource est partagée
type	Le type pleinement qualifié de la ressource

Cette annotation peut être utilisée sur une classe, un champ ou une méthode.

Lorsque l'annotation est utilisée sur une classe, elle correspond simplement à une déclaration des ressources qui seront requises à l'exécution.

Lorsque l'annotation est utilisée sur un champ ou une méthode, le serveur d'applications va injecter une référence sur la ressource correspondante. Pour cela, lors du chargement d'une application par le serveur d'applications, celui-ci recherche les annotations @Resource afin d'assigner une instance de la ressource correspondante.

Exemple :

```
@Resource(name="MaQueue",
    type = "javax.jms.Queue",
    shareable=false,
    authenticationType=Resource.AuthenticationType.CONTAINER,
    description="Queue de test"
)
private javax.jms.Queue maQueue;
```

L'annotation @Resources est simplement une collection d'annotation de type @Resource.

Exemple :

```
@Resources({
    @Resource(name = "maQueue" type = javax.jms.Queue),
    @Resource(name = "monTopic" type = javax.jms.Topic),
})
```

10.5.3. Les annotations @PostConstruct et @PreDestroy

Les annotations @PostConstruct et @PreDestroy permettent respectivement de désigner des méthodes qui seront exécutées après l'instanciation d'un objet et avant la destruction d'une instance.

Ces deux annotations ne peuvent être utilisées que sur des méthodes.

Ces annotations sont par exemple utiles dans Java EE car généralement un composant géré par le conteneur est instancié en utilisant le constructeur sans paramètre. Une méthode marquée avec l'annotation @PostConstruct peut alors être exécutée juste après l'appel au constructeur.

Une telle méthode doit respecter certaines règles :

- ne pas avoir de paramètres sauf dans des cas précis (exemple avec les intercepteurs des EJB)
- ne pas avoir de valeur de retour (elle doit renvoyer void)
- ne doit pas lever d'exceptions vérifiées
- ne doit pas être statique

L'annotation @PostConstruct est utilisée en général sur une méthode qui initialise des ressources en fonction du contexte.

Dans une même classe, chacune de ces annotations n'est utilisable que par une seule méthode.

10.6. Les annotations personnalisées

Java propose la possibilité de définir ses propres annotations. Pour cela, le langage possède un type dédié : le type d'annotation (annotation type).

Un type d'annotation est similaire à une classe et une annotation est similaire à une instance de classe.

10.6.1. La définition d'une annotation

Sur la plate-forme Java, une annotation est une interface lors de sa déclaration et est une instance d'une classe qui implémente cette interface lors de son utilisation.

La définition d'une annotation nécessite une syntaxe particulière utilisant le mot clé @interface. Une annotation se déclare donc de façon similaire à une interface.

Exemple : le fichier MonAnnotation.java

```
1. package com.jmdoudoux.test.annotations;
2.
3. public @interface MonAnnotation {
4.
5. }
```

Une fois compilée, cette annotation peut être utilisée dans le code. Pour utiliser une annotation, il faut importer l'annotation et l'appeler dans le code en la faisant précéder du caractère @.

Exemple :

```
1. package com.jmdoudoux.test.annotations;
2.
3. @MonAnnotation
4. public class MaCLasse {
5.
6. }
```

Si l'annotation est définie dans un autre package, il faut utiliser la syntaxe pleinement qualifiée du nom de l'annotation ou ajouter une clause import pour le package.

Il est possible d'ajouter des membres à l'annotation simplement en définissant une méthode dont le nom correspond au nom de l'attribut en paramètre de l'annotation.

Exemple :

```
01. package com.jmdoudoux.test.annotations;
02.
03. public @interface MonAnnotation {
04.   String arg1();
05.   String arg2();
06. }
07.
08. package com.jmdoudoux.test.annotations;
09.
10. @MonAnnotation(arg1="valeur1", arg2="valeur2")
11. public class MaCLasse {
12.
13. }
```

Les types utilisables sont les chaînes de caractères, les types primitifs, les énumérations, les annotations, les chaînes de caractères, le type Class.

Il est possible de définir un membre comme étant un tableau à une seule dimension d'un des types utilisables.

Exemple :

```
1. package com.jmdoudoux.tests.annotations
2.
3. public @interface MonAnnotation {
4.     String arg1();
5.     String[] arg2();
6.     String arg3();
7. }
```

Il est possible de définir une valeur par défaut, ce qui rend l'indication du membre optionnelle. Cette valeur est précisée en la faisant précéder du mot clé default.

Exemple :

```
1. package com.jmdoudoux.test.annotations;
2.
3. public @interface MonAnnotation {
4.   String arg1() default "";
5.   String[] arg2();
6.   String arg3();
7. }
```

La valeur par défaut d'un tableau utilise une syntaxe raccourcie.

Exemple :

```
1. package com.jmdoudoux.tests.annotations
2.
3. public @interface MonAnnotation {
4.     String arg1();
5.     String[] arg2() default {"chaine1", "chaine2" };
6.     String arg3();
7. }
```

Il est possible de définir une énumération comme type pour un attribut

Exemple :

```
01. package com.jmdoudoux.test.annotations;
02.
03. public @interface MonAnnotation {
04.   public enum Niveau {DEBUTANT, CONFIRME, EXPERT} ;
05.   String arg1() default "";
06.   String[] arg2();
07.   String arg3();
08.   Niveau niveau() default Niveau.DEBUTANT;
09.
10. }
```

10.6.2. Les annotations pour les annotations

La version 5 de Java propose quatre annotations dédiées aux types d'annotations qui permettent de fournir des informations sur l'utilisation.

Ces annotations sont définies dans le package java.lang.annotation

10.6.2.1. L'annotation @Target

L'annotation @Target permet de préciser les entités sur lesquelles l'annotation sera utilisable. Cette annotation attend comme valeur un tableau de valeurs issues de l'énumération ElementType

Valeur de l'énumération	Rôle
ANNOTATION_TYPE	Types d'annotation
CONSTRUCTOR	Constructeurs
LOCAL_VARIABLE	Variables locales
FIELD	Champs
METHOD	Méthodes hors constructeurs
PACKAGE	Packages
PARAMETER	paramètres d'une méthode ou d'un constructeur
TYPE	Classes, interfaces, énumérations, types d'annotation

Si une annotation est utilisée sur une entité non précisée par l'annotation, alors une erreur est émise lors de la compilation

Exemple :

```
01. package com.jmdoudoux.test.annotations;
02.
03. import java.lang.annotation.ElementType;
04. import java.lang.annotation.Target;
05.
06. @Target({ElementType.METHOD, ElementType.CONSTRUCTOR })
07. public @interface MonAnnotation {
08.   String arg1() default "";
09.   String arg2();
10. }
11.
12. package com.jmdoudoux.test.annotations;
13.
14. @MonAnnotation(arg1="valeur1", arg2="valeur2")
15. public class MaClasse {
16.
17. }
```

Résultat de la compilation :

```
1. C:\Documents and Settings\jmd\workspace\Tests>javac com/jmdoudoux/test/annotatio
2. ns/MaClasse.java
3. com\jmdoudoux\test\annotations\MaClasse.java:3: annotation type not applicable t
4. o this kind of declaration
5. @MonAnnotation(arg1="valeur1", arg2="valeur2")
6. ^
7. 1 error
```

10.6.2.2. L'annotation @Retention

Cette annotation permet de préciser à quel niveau les informations concernant l'annotation seront conservées. Cette annotation attend comme valeur un élément de l'énumération RetentionPolicy

Enumération	Rôle
RetentionPolicy.SOURCE	informations conservées dans le code source uniquement (fichier .java) : le compilateur les ignore
RetentionPolicy.CLASS	informations conservées dans le code source et le bytecode (fichier .java et .class)
RetentionPolicy.RUNTIME	informations conservées dans le code source et le bytecode et elles sont disponibles à l'exécution par introspection

Cette annotation permet de déterminer de quelle façon l'annotation pourra être exploitée.

Exemple :

```
1. @Retention(RetentionPolicy.RUNTIME)
```

10.6.2.3. L'annotation @Documented

L'annotation @Documented permet de demander l'intégration de l'annotation dans la documentation générée par Javadoc.

Par défaut, les annotations ne sont pas intégrées dans la documentation des classes annotées.

Exemple :

```
01. package com.jmdoudoux.test.annotations;
02.
03. import java.lang.annotation.Documented;
04.
05. @Documented
06. public @interface MonAnnotation {
07.   String arg1() default "";
08.   String arg2();
09. }
```

```
com.jmdoudoux.test.annotations
Class MaCLasse

java.lang.Object
  └ com.jmdoudoux.test.annotations.MaCLasse

@MonAnnotation(arg1="valeur1",
               arg2="valeur2")
public class MaCLasse
extends java.lang.Object

Author:
      JMD
```

10.6.2.4. L'annotation @Inherited

L'annotation @Inherited permet de demander l'héritage d'une annotation aux classes filles de la classe mère sur laquelle elle s'applique.

Si une classe mère est annotée avec une annotation elle-même annotée avec @Inherited alors toutes les classes filles sont automatiquement annotées avec cette annotation.

10.7. L'exploitation des annotations

Pour être profitables, les annotations ajoutées dans le code source doivent être exploitées par un ou plusieurs outils.

La déclaration et l'utilisation d'annotations sont relativement simples par contre leur exploitation pour permettre la production de fichiers est moins triviale.

Cette exploitation peut se faire de plusieurs manières

- en définissant un doclet qui exploite le code source
- en utilisant apt au moment de la compilation
- en utilisant l'introspection lors de l'exécution
- en utilisant le compilateur java à partir de Java 6.0

10.7.1. L'exploitation des annotations dans un Doclet

Pour des traitements simples, il est possible de définir un Doclet et de le traiter avec l'outil Javadoc pour utiliser les annotations.

L'API Doclet est définit dans le package com.sun.javadoc. Ce package est dans le fichier tools.jar fourni avec le JDK

L'API Doclet définit des interfaces pour chaque entité pouvant être utilisée dans le code source.

La méthode annotation() de l'interface ProgramElementDoc permet d'obtenir un tableau de type AnnotationDesc.

L'interface AnnotationDesc représente une annotation. Elle définit deux méthodes

Méthode	Rôle
AnnotationTypeDoc annotationType()	Renvoyer le type d'annotation
AnnotationDesc.ElementValuePair[] elementValues()	Renvoyer les éléments de l'annotation

L'interface AnnotationTypeDoc représente un type d'annotation. Elle ne définit qu'une seule méthode

Méthode	Rôle
AnnotationTypeElementDoc[] elements()	Renvoyer les éléments d'un type d'annotation

L'interface AnnotationTypeElementDoc représente un élément d'un type d'annotation. Elle ne définit qu'une seule méthode

Méthode	Rôle

AnnotationValue defaultValue()	Renvoyer la valeur par défaut de l'élément d'un type d'annotation

L'interface AnnotationValue représente la valeur d'un élément d'un type d'annotation. Elle définit deux méthodes

Méthode	Rôle
String toString()	Renvoyer la valeur sous forme d'une chaîne de caractères
Object value()	Renvoyer la valeur

Pour créer un Doclet, il faut définir une classe qui contienne une méthode ayant pour signature `public static boolean start (RootDoc rootDoc)`.

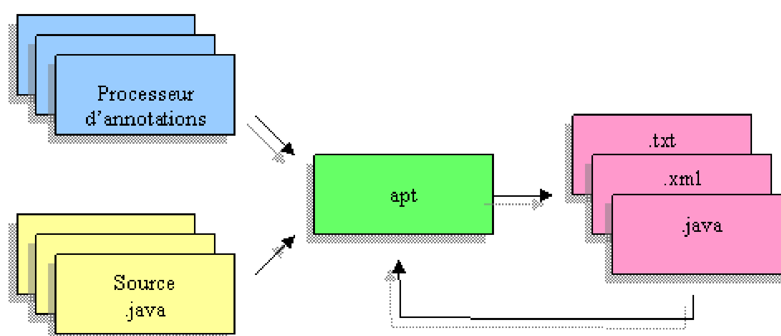
Pour utiliser le Doclet, il faut compiler la classe qui l'encapsule et utiliser l'outil javadoc avec l'option -doclet suivi du nom de la classe.

10.7.2. L'exploitation des annotations avec l'outil Apt

La version 5 du JDK fournit l'outil apt pour le traitement des annotations.

L'outil apt qui signifie annotation processing tool est l'outil le plus polyvalent en Java 5 pour exploiter les annotations.

Apt assure la compilation des classes et permet en simultané le traitement des annotations par des processeurs d'annotations créés par le développeur.



Les processeurs d'annotations peuvent générer de nouveaux fichiers sources, pouvant eux-mêmes contenir des annotations. Apt traite alors récursivement les fichiers générés jusqu'à ce qu'il n'y ait plus d'annotation à traiter et de classe à compiler.

Cette section va créer un processeur pour l'annotation personnalisée Todo

Exemple : l'annotation personnalisée Todo

```
01. package com.jmdoudoux.test.annotations;
02.
03. import java.lang.annotation.Documented;
04.
05. @Documented
06. public @interface Todo {
07.
08.   public enum Importance {
09.     MINEURE, IMPORTANT, MAJEUR, CRITIQUE
10.   };
11.
12.   Importance importance() default Importance.MINEURE;
13.
14.   String[] description();
15.
16.   String assigneA();
17.
18.   String dateAssignation();
19. }
```

Apt et l'API à utiliser de concert ne sont disponibles qu'avec le JDK : ils ne sont pas fournis avec le JRE.

Les packages de l'API sont dans le fichier lib/tools.jar du JDK : cette bibliothèque doit donc être ajoutée au classpath lors de la mise en oeuvre d'apt.

L'API est composée de deux grandes parties :

- Modélisation du langage
- Interaction avec l'outil de traitement des annotations

L'API est contenue dans plusieurs sous-packages de com.sun.mirror notamment :

- com.sun.mirror.apt : contient les interfaces pour la mise en oeuvre d'apt

- com.sun.mirror.declaration : encapsule la déclaration des entités dans les sources qui peuvent être annotées (packages, classes, méthodes, ...) sous le forme d'interfaces qui héritent de l'interface Declaration
- com.sun.mirror.type : encapsule les types d'entités dans les sources sous la forme d'interfaces qui héritent de l'interface TypeMirror
- com.sun.mirror.util : propose des utilitaires

Un processeur d'annotations est une classe qui implémente l'interface com.sun.mirror.apt.AnnotationProcessor. Cette interface ne définit qu'une seule méthode process() qui va contenir les traitements à réaliser pour une annotation.

Il faut fournir un constructeur qui attend en paramètre un objet de type com.sun.mirror.apt.AnnotationProcessorEnvironment : ce constructeur sera appelé par une fabrique pour en créer une instance.

L'interface AnnotationProcessorEnvironment fournit des méthodes pour obtenir des informations sur l'environnement d'exécution des traitements des annotations et créer de nouveaux fichiers pendant les traitements.

L'interface Declaration permet d'obtenir des informations sur une entité :

Méthode	Rôle
<A extends Annotation> getAnnotation(Class<A> annotationType)	Renvoie une annotation d'un certain type associée à l'entité
Collection<AnnotationMirror> getAnnotationMirrors()	Renvoie les annotations associées à l'entité
String getDocComment()	Renvoie le texte des commentaires de documentations Javadoc associés à l'entité
Collection<Modifier> getModifiers()	Renvoie les modificateurs de l'entité
SourcePosition getPosition()	Renvoie la position de la déclaration dans le code source
String getSimpleName()	Renvoie le nom de la déclaration

De nombreuses interfaces héritent de l'interface Declaration : AnnotationTypeDeclaration, AnnotationTypeElementDeclaration, ClassDeclaration, ConstructorDeclaration, EnumConstantDeclaration, EnumDeclaration, ExecutableDeclaration, FieldDeclaration, InterfaceDeclaration, MemberDeclaration, MethodDeclaration, PackageDeclaration, ParameterDeclaration, TypeDeclaration, TypeParameterDeclaration

Chacune de ces interfaces propose des méthodes pour obtenir des informations sur la déclaration et sur le type concernée.

L'interface TypeMirror permet d'obtenir des informations sur un type utilisé dans une déclaration.

De nombreuses interfaces héritent de l'interface TypeMirror : AnnotationType, ArrayType, ClassType, DeclaredType, EnumType, InterfaceType, PrimitiveType, ReferenceType, TypeVariable, VoidType, WildcardType.

La classe com.sun.mirror.util.DeclarationFilter permet de définir un filtre des entités annotées avec les annotations concernées par les traitements du processeur. Il suffit de créer une instance de cette classe en ayant redéfini sa méthode match(). Cette méthode renvoie un booléen qui précise si l'entité fournie en paramètre sous la forme d'un objet de type Declaration est annotée avec une des annotations concernées par le processeur.

Exemple :

```
01. package com.jmdoudoux.test.annotations.outils;
02.
03. import java.util.Collection;
04.
05. import com.jmdoudoux.test.annotations.Todo;
06. import com.sun.mirror.apt.AnnotationProcessor;
07. import com.sun.mirror.apt.AnnotationProcessorEnvironment;
08. import com.sun.mirror.declaration.Declaration;
09. import com.sun.mirror.declaration.TypeDeclaration;
10. import com.sun.mirror.util.DeclarationFilter;
11.
12. public class TodoAnnotationProcessor implements AnnotationProcessor {
13.   private final AnnotationProcessorEnvironment env;
14.
15.   public TodoAnnotationProcessor(AnnotationProcessorEnvironment env) {
16.     this.env = env;
17.   }
18.
19.   public void process() {
20.     // Creation d'un filtre pour ne retenir que les déclarations annotées avec Todo
21.     DeclarationFilter annFilter = new DeclarationFilter() {
22.       public boolean matches(
23.           Declaration d) {
24.         return d.getAnnotation(Todo.class) != null;
25.       }
26.     };
27.
28.     // Recherche des entités annotées avec Todo
29.     Collection<TypeDeclaration> types = annFilter.filter(env.getSpecifiedTypeDeclarations());
30.     for (TypeDeclaration typeDecl : types) {
31.       System.out.println("class name: " + typeDecl.getSimpleName());
32.
33.       Todo todo = typeDecl.getAnnotation(Todo.class);
34.
35.       System.out.println("description : ");
36.       for (String desc : todo.description()) {
37.         System.out.println(desc);
38.       }
39.       System.out.println("");
40.     }
41.   }
42. }
```

Il faut créer une fabrique de processeurs d'annotations : cette fabrique est en charge d'instancier des processeurs d'annotations pour un ou plusieurs types d'annotations. La fabrique doit implémenter l'interface com.sun.mirror.apt.AnnotationProcessorFactory.

L'interface AnnotationProcessorFactory déclare trois méthodes :

Méthode	Rôle
AnnotationProcessor getProcessorFor(Set<AnnotationTypeDeclaration> atds, AnnotationProcessorEnvironment env)	Renvoyer un processeur d'annotations pour l'ensemble de types d'annotations fournis en paramètres.
Collection<String> supportedAnnotationTypes()	Renvoyer une collection des types d'annotations dont un processeur peut être instancié par la fabrique
Collection<String> supportedOptions()	Renvoyer une collection des options supportées par la fabrique ou par les processeurs d'annotations créés par la fabrique

Exemple :

```
01. package com.jmdoudoux.test.annotations.outils;
02.
03. import com.sun.mirror.apt.*;
04. import com.sun.mirror.declaration.*;
05.
06. import java.util.Collection;
07. import java.util.Set;
08. import java.util.Collections;
09. import java.util.Arrays;
10.
11. public class TodoAnnotationProcessorFactory implements AnnotationProcessorFactory {
12.   private static final Collection<String> supportedAnnotations =
13.     Collections.unmodifiableCollection(Arrays
14.       .asList("com.jmdoudoux.test.annotations.Todo"));
15.
16.   private static final Collection<String> supportedOptions     = Collections.emptySet();
17.
18.   public Collection<String> supportedOptions() {
19.     return supportedOptions;
20.   }
21.
22.   public Collection<String> supportedAnnotationTypes() {
23.     return supportedAnnotations;
24.   }
25.
26.   public AnnotationProcessor getProcessorFor(
27.       Set<AnnotationTypeDeclaration> atds,
28.       AnnotationProcessorEnvironment env) {
29.     return new TodoAnnotationProcessor(env);
30.   }
31. }
```

Dans l'exemple ci-dessus, aucune option n'est supportée et la fabrique ne prend en charge que l'annotation personnalisée Todo.

Pour mettre en oeuvre les traitements des annotations, il faut que le code source utilise ces annotations.

Exemple : une classe annotée avec l'annotation Todo

```
01. package com.jmdoudoux.test;
02.
03. import com.jmdoudoux.test.annotations.Todo;
04. import com.jmdoudoux.test.annotations.Todo.Importance;
05.
06. @Todo(importance = Importance.CRITIQUE,
07.       description = "Corriger le bug dans le calcul",
08.       assigneA = "JMD",
09.       dateAssignation = "11-11-2007")
10. public class MaClasse {
11.
12. }
```

Exemple : une autre classe annotée avec l'annotation Todo

```
01. package com.jmdoudoux.test;
02.
03. import com.jmdoudoux.test.annotations.Todo;
04. import com.jmdoudoux.test.annotations.Todo.Importance;
05.
06. @Todo(importance = Importance.MAJEUR,
07.       description = "Ajouter le traitement des erreurs",
08.       assigneA = "JMD",
09.       dateAssignation = "07-11-2007")
10. public class MaClasse1 {
```

```
11.
12. }
```

Pour utiliser apt, il faut que le classpath contienne la bibliothèque tools.jar fournie avec le JDK et les classes de traitements des annotations (fabrique et processeur d'annotations).

L'option -factory permet de préciser la fabrique à utiliser.

Résultat de l'exécution d'apt

```
01. C:\Documents and Settings\jmd\workspace\Tests>apt -cp ".;./bin;C:/Program Files/
02. Java/jdk1.5.0_07/lib/tools.jar" -factory com.jmdoudoux.test.annotations.outils.T
03. odoAnnotationProcessorFactory com/jmdoudoux/test/*.java
04. class name: MaClasse
05. description :
06. Corriger le bug dans le calcul
07.
08. class name: MaClasse1
09. description :
10. Ajouter le traitement des erreurs
```

A partir de l'objet de type AnnotationProcessorEnvironment, il est possible d'obtenir un objet de type com.sun.miror.apt.Filer qui encapsule un nouveau fichier créé par le processeur d'annotations.

L'interface Filer propose quatre méthodes pour créer différents types de fichiers :

Méthode	Rôle
OutputStream createBinaryFile(Filer.Location loc, String pkg, File relPath)	Créer un nouveau fichier binaire et renvoyer un objet de type Stream pour écrire son contenu
OutputStream createClassFile(String name)	Créer un nouveau fichier .class et renvoyer un objet de type Stream pour écrire son contenu
PrintWriter createSourceFile(String name)	Créer un nouveau fichier texte contenant du code source et renvoyer un objet de type Writer pour écrire son contenu
PrintWriter createTextFile(Filer.Location loc, String pkg, File relPath, String charsetName)	Créer un nouveau fichier texte et renvoyer un objet de type Writer pour écrire son contenu

L'énumération Filter.Location permet de préciser si le nouveau fichier est créé dans la branche source (SOURCE_TREE) ou dans la branche compilée (CLASS_TREE).

Exemple :

```
01. package com.jmdoudoux.test.annotations.outils;
02.
03. import java.io.File;
04. import java.io.IOException;
05. import java.io.PrintWriter;
06. import java.util.Collection;
07.
08. import com.jmdoudoux.test.annotations.Todo;
09. import com.sun.mirror.apt.AnnotationProcessor;
10. import com.sun.mirror.apt.AnnotationProcessorEnvironment;
11. import com.sun.mirror.apt.Filer;
12. import com.sun.mirror.declaration.Declaration;
13. import com.sun.mirror.declaration.TypeDeclaration;
14. import com.sun.mirror.util.DeclarationFilter;
15.
16. public class TodoAnnotationProcessor implements AnnotationProcessor {
17.   private final AnnotationProcessorEnvironment env;
18.
19.   public TodoAnnotationProcessor(AnnotationProcessorEnvironment env) {
20.     this.env = env;
21.   }
22.
23.   public void process() {
24.     // Creation d'un filtre pour ne retenir que les déclarations annotées avec
25.     // Todo
26.     DeclarationFilter annFilter = new DeclarationFilter() {
27.       public boolean matches(
28.           Declaration d) {
29.         return d.getAnnotation(Todo.class) != null;
30.       }
31.     };
32.
33.     Filer f = this.env.getFiler();
34.     PrintWriter out;
35.     try {
36.       out = f.createTextFile(Filer.Location.SOURCE_TREE, "", new File("todo.txt"), null);
37.
38.       // Recherche des entités annotées avec Todo
39.       Collection<TypeDeclaration> types = annFilter.filter(env.getSpecifiedTypeDeclarations());
```

```
40.       for (TypeDeclaration typeDecl : types) {
41.         out.println("class name: " + typeDecl.getSimpleName());
42.
43.         Todo todo = typeDecl.getAnnotation(Todo.class);
44.
45.         out.println("description : ");
46.         for (String desc : todo.description()) {
47.           out.println(desc);
48.         }
49.         out.println("");
50.       }
51.
52.       out.close();
53.     } catch (IOException e) {
54.       e.printStackTrace();
55.     }
56.   }
57. }
```

Résultat de l'exécution

```
01. C:\Documents and Settings\jm\workspace\Tests>apt -cp ".;./bin;C:/Program Files/
02. Java/jdk1.5.0_07/lib/tools.jar" -factory com.jmdoudoux.test.annotations.outils.T
03. odoAnnotationProcessorFactory com/jmdoudoux/test/*.java
04.
05. C:\Documents and Settings\jm\workspace\Tests>dir
06.  Volume in drive C has no label.
07.  Volume Serial Number is 1D31-4F67
08.
09.  Directory of C:\Documents and Settings\jmd\workspace\Tests
10.
11. 19/11/2007  08:39    <DIR>          .
12. 19/11/2007  08:39    <DIR>          ..
13. 16/11/2007  08:15               433 .classpath
14. 31/10/2006  14:06               381 .project
15. 14/09/2007  12:45    <DIR>          .settings
16. 16/11/2007  08:15    <DIR>          bin
17. 02/10/2007  15:22               854 build.xml
18. 29/06/2007  07:12    <DIR>          com
19. 15/11/2007  13:01    <DIR>          doc
20. 19/11/2007  08:39               148 todo.txt
21.                8 File(s)          1 812 bytes
22.                 6 Dir(s)  66 885 595 136 bytes free
23.
24. C:\Documents and Settings\jm\workspace\Tests>type todo.txt
25. class name: MaClasse
26. description :
27. Corriger le bug dans le calcul
28.
29. class name: MaClasse1
30. description :
31. Ajouter le traitement des erreurs
```

Concernant les entités à traiter, l'API Mirror fournit de nombreuses autres fonctionnalités qui permettent de rendre très riche le traitement des annotations. Parmi ces fonctionnalités, il y a le parcours des sources par des classes mettant en oeuvre le motif de conception visiteur.

10.7.3. L'exploitation des annotations par introspection

Pour qu'une annotation soit exploitée à l'exécution, il est nécessaire qu'elle soit annotée avec une RetentionPolicy à la valeur RUNTIME.

Exemple :

```
01. package com.jmdoudoux.test.annotations;
02.
03. import java.lang.annotation.Documented;
04. import java.lang.annotation.Retention;
05. import java.lang.annotation.RetentionPolicy;
06.
07. @Documented
08. @Retention(RetentionPolicy.RUNTIME)
09. public @interface Todo {
10.
11.   public enum Importance {
12.     MINEURE, IMPORTANT, MAJEUR, CRITIQUE
13.   };
14.
15.   Importance importance() default Importance.MINEURE;
16.
17.   String[] description();
18.
19.   String assigneA();
20.
```

```
21.   String dateAssignation();
22. }
```

L'interface java.lang.reflect.AnnotatedElement définit les méthodes pour le traitement des annotations par introspection :

Méthode	Rôle
<T extends Annotation> getAnnotation(Class<T>)	Renvoyer l'annotation si le type fourni en paramètre est utilisé sur l'entité, sinon null
Annotation[] getAnnotations()	Renvoyer un tableau de toutes les annotations de l'entité. Renvoie un tableau vide si aucune annotation n'est concernée
Annotation[] getDeclaredAnnotations()	Renvoyer un tableau des annotations directement associées à l'entité (en ignorant donc les annotations héritées). Renvoie un tableau vide si aucune annotation n'est concernée
boolean isAnnotationPresent(Class< ? extends Annotation>)	Renvoyer true si l'annotation dont le type est fourni en paramètre est utilisé sur l'entité. Cette méthode est particulièrement utile dans le traitement des annotations de type marqueur.

Plusieurs classes du package java.lang implémentent l'interface AnnotatedElement : AccessibleObject, Class, Constructor, Field, Method et Package

Exemple :

```
01. package com.jmdoudoux.test;
02.
03. import java.lang.reflect.Method;
04.
05. import com.jmdoudoux.test.annotations.Todo;
06. import com.jmdoudoux.test.annotations.Todo.Importance;
07.
08. @Todo(importance = Importance.CRITIQUE,
09.       description = "Corriger le bug dans le calcul",
10.       assigneA = "JMD",
11.       dateAssignation = "11-11-2007")
12. public class TestInstrospectionAnnotation {
13.
14.   public static void main(
15.       String[] args) {
16.     Todo todo = null;
17.
18.     // traitement annotation sur la classe
19.     Class classe = TestInstrospectionAnnotation.class;
20.     todo = (Todo) classe.getAnnotation(Todo.class);
21.     if (todo != null) {
22.       System.out.println("classe "+classe.getName());
23.       System.out.println("  ["+todo.importance()+"]"+" ("+todo.assigneA()
24.         +" le "+todo.dateAssignation()+")");
25.       for(String desc : todo.description()) {
26.         System.out.println("     _ "+desc);
27.       }
28.     }
29.
30.     // traitement annotation sur les méthodes de la classe
31.     for(Method m : TestInstrospectionAnnotation.class.getMethods()) {
32.       todo = (Todo) m.getAnnotation(Todo.class);
33.       if (todo != null) {
34.         System.out.println("methode "+m.getName());
35.         System.out.println("  ["+todo.importance()+"]"+" ("+todo.assigneA()
36.           +" le "+todo.dateAssignation()+")");
37.         for(String desc : todo.description()) {
38.           System.out.println("     _ "+desc);
39.         }
40.       }
41.     }
42.   }
43.
44.   @Todo(importance = Importance.MAJEUR,
45.         description = "Implementer la methode",
46.         assigneA = "JMD",
47.         dateAssignation = "11-11-2007")
48.   public void methode1() {
49.
50.   }
51.
52.   @Todo(importance = Importance.MINEURE,
53.         description = {"Completer la methode", "Ameliorer les logs"},
54.         assigneA = "JMD",
55.         dateAssignation = "12-11-2007")
56.   public void methode2() {
57.
58.   }
59.
60. }
```

Résultat d'exécution :

```
01. classe com.jmdoudoux.test.TestInstrospectionAnnotation
02.   [CRITIQUE] (JMD le 11-11-2007)
03.      _ Corriger le bug dans le calcul
04. methode methode1
05.   [MAJEUR] (JMD le 11-11-2007)
06.      _ Implementer la methode
07. methode methode2
08.   [MINEURE] (JMD le 12-11-2007)
09.      _ Completer la methode
10.      _ Ameliorer les logs
```

Pour obtenir les annotations sur les paramètres d'un constructeur ou d'une méthode, il faut utiliser la méthode getParameterAnnotations() des classes Constructor ou Method qui renvoie un objet de type Annotation[][]. La première dimension du tableau concerne les paramètres dans leur ordre de déclaration. La seconde dimension contient les annotations de chaque paramètre.

10.7.4. L'exploitation des annotations par le compilateur Java

Dans la version 6 de Java SE, la prise en compte des annotations est intégrée dans le compilateur : ceci permet un traitement à la compilation des annotations sans avoir recours à un outil tiers comme apt.

Une nouvelle API a été définie par la JSR 269 (Pluggable annotations processing API) et ajoutée dans la package javax.annotation.processing.

Cette API est détaillée dans la section suivante.

10.8. L'API Pluggable Annotation Processing

La version 6 de Java apporte plusieurs améliorations dans le traitement des annotations notamment l'intégration de ces traitements directement dans le compilateur javac grâce à une nouvelle API dédiée.

L'API Pluggable Annotation Processing est définie dans la JSR 269. Elle permet un traitement des annotations directement par le compilateur en proposant une API aux développeurs pour traiter les annotations incluses dans le code source.

Apt et son API proposaient déjà une solution à ces traitements mais cette API standardise le traitement des annotations au moment de la compilation. Il n'est donc plus nécessaire d'utiliser un outil tiers post compilation pour traiter les annotations à la compilation.

Dans les exemples de cette section, les classes suivantes seront utilisées

Exemple : MaClasse.java

```
01. package com.jmdoudoux.tests;
02.
03. import com.jmdoudoux.tests.annotations.Todo;
04. import com.jmdoudoux.tests.annotations.Todo.Importance;
05.
06. @Todo(importance = Importance.CRITIQUE,
07.       description = "Corriger le bug dans le calcul",
08.       assigneA = "JMD",
09.       dateAssignation = "11-11-2007")
10. public class MaClasse {
11.
12. }
```

Exemple : MaClasse1.java

```
01. package com.jmdoudoux.tests;
02.
03. import com.jmdoudoux.tests.annotations.Todo;
04. import com.jmdoudoux.tests.annotations.Todo.Importance;
05.
06. @Todo(importance = Importance.MAJEUR,
07.       description = "Ajouter le traitement des erreurs",
08.       assigneA = "JMD",
09.       dateAssignation = "07-11-2007")
10. public class MaClasse1 {
11.
12. }
```

Exemple : MaClasse3.java

```
1. package com.jmdoudoux.tests;
2.
3. @Deprecated
4. public class MaClasse3 {
5.
6. }
```

Un exemple de mise en oeuvre de l'API est aussi fourni avec le JDK dans le sous-répertoire sample/javac/processing

10.8.1. Les processeurs d'annotations

La mise en oeuvre de cette API nécessite l'utilisation des packages javax.annotation.processing, javax.lang.model et javax.tools.

Un processeur d'annotations doit implémenter l'interface Processor. Le traitement des annotations se fait en plusieurs passes (round). A chaque passe le processeur est appelé pour traiter des classes qui peuvent avoir été générées lors de la précédente passe. Lors de la première passe, ce sont les classes fournies initialement qui sont traitées.

L'interface javax.annotation.processing.Processor définit les méthodes d'un processeur d'annotations. Pour définir un processeur, il est possible de créer une classe qui implémente l'interface Processor mais le plus simple est d'hériter de la classe abstraite javax.annotation.processing.AbstractProcessor.

La classe AbstractProcessor contient une variable nommée processingEnv de type ProcessingEnvironment. La classe ProcessingEnvironment permet d'obtenir des instances de classes qui permettent des interactions avec l'extérieur du processeur ou fournissent des utilitaires :

- Filer : classe qui permet la création de fichiers
- Messager : classe qui permet d'envoyer des messages affichés par le compilateur
- Elements : classe qui fournit des utilitaires pour les éléments
- Types : classe qui fournit des utilitaires pour les types

La méthode getRootElements() renvoie les classes Java qui seront traitées par le processeur dans cette passe.

La méthode la plus importante est la méthode process() : c'est elle qui va contenir les traitements exécutés par le processeur. Elle possède deux paramètres :

- Un ensemble des annotations qui seront traitées par le processeur
- Un objet qui encapsule l'étape courante des traitements

Deux annotations sont dédiées aux processeurs d'annotations et doivent être utilisées sur la classe du processeur :

- @SupportedAnnotationTypes : cette annotation permet de préciser les types d'annotations traitées par le processeur. La valeur « * » permet d'indiquer que tous seront traités.
- @SupportedSourceVersion : cette annotation permet de préciser la version du code source traité par le processeur

Exemple :

```
01. package com.jmdoudoux.tests.annotations.outils;
02.
03. import java.util.Set;
04.
05. import javax.annotation.processing.AbstractProcessor;
06. import javax.annotation.processing.Messager;
07. import javax.annotation.processing.RoundEnvironment;
08. import javax.annotation.processing.SupportedAnnotationTypes;
09. import javax.annotation.processing.SupportedSourceVersion;
10. import javax.lang.model.SourceVersion;
11. import javax.lang.model.element.Element;
12. import javax.lang.model.element.TypeElement;
13. import javax.tools.Diagnostic.Kind;
14.
15. import com.jmdoudoux.tests.annotations.Todo;
16.
17. @SupportedAnnotationTypes(value = { "*" })
18. @SupportedSourceVersion(SourceVersion.RELEASE_6)
19. public class TodoProcessor extends AbstractProcessor {
20.
21.   @Override
22.   public boolean process(
23.       Set<? extends TypeElement> annotations,
24.       RoundEnvironment roundEnv) {
25.
26.     Messager messager = processingEnv.getMessager();
27.
28.     for (TypeElement te : annotations) {
29.       messager.printMessage(Kind.NOTE, "Traitement annotation "
30.         + te.getQualifiedName());
31.
32.       for (Element element : roundEnv.getElementsAnnotatedWith(te)) {
33.         messager.printMessage(Kind.NOTE, "  Traitement element "
34.           + element.getSimpleName());
```

```
35.         Todo todo = element.getAnnotation(Todo.class);
36.
37.         if (todo != null) {
38.           messager.printMessage(Kind.NOTE, "  affecte le " + todo.dateAssignation()
39.             + " a " + todo.assigneA());
40.         }
41.       }
42.     }
43.
44.     return true;
45.   }
46. }
```

10.8.2. L'utilisation des processeurs par le compilateur

Le compilateur javac est enrichi avec plusieurs options concernant le traitement des annotations :

Option	Rôle
-processor	permet de préciser le nom pleinement qualifié du processeur à utiliser
-proc	vérifie si le traitement des annotations et/ou la compilation sont effectués
-processorpath	classpath des processeurs d'annotations
-A	permet de passer des options aux processeurs d'annotations sous la forme de paires cle=valeur
-XprintRounds	option non standard qui permet d'afficher des informations sur le traitement des annotations par les processeurs
-XprintProcessorInfo	option non standard qui affiche la liste des annotations qui seront traitées par les processeurs d'annotations

Le compilateur fait appel à la méthode process() du processeur en lui passant en paramètre l'ensemble des annotations trouvées par le compilateur dans le code source.

Résultat :

```
01. C:\Documents and Settings\jm\workspace\TestAnnotations>javac -cp ".;./bin;C:/Pr
02. ogram Files/Java/jdk1.6.0/lib/tools.jar" -processor com.jmdoudoux.tests.annotati
03. ons.outils.TodoProcessor com/jmdoudoux/tests/*.java
04. Note: Traitement annotation com.jmdoudoux.tests.annotations.Todo
05. Note:   Traitement element MaClasse
06. Note:   affecte le 11-11-2007 a JMD
07. Note:   Traitement element MaClasse1
08. Note:   affecte le 07-11-2007 a JMD
09. Note: Traitement annotation java.lang.Deprecated
10. Note:   Traitement element MaClasse3
```

10.8.3. La création de nouveaux fichiers

La classe Filer permet de créer des fichiers lors du traitement des annotations.

Exemple :

```
01. package com.jmdoudoux.tests.annotations.outils;
02.
03. import java.io.IOException;
04. import java.io.PrintWriter;
05. import java.util.Set;
06.
07. import javax.annotation.processing.AbstractProcessor;
08. import javax.annotation.processing.Filer;
09. import javax.annotation.processing.Messager;
10. import javax.annotation.processing.RoundEnvironment;
11. import javax.annotation.processing.SupportedAnnotationTypes;
12. import javax.annotation.processing.SupportedSourceVersion;
13. import javax.lang.model.SourceVersion;
14. import javax.lang.model.element.Element;
15. import javax.lang.model.element.TypeElement;
16. import javax.lang.model.util.Elements;
17. import javax.tools.StandardLocation;
18. import javax.tools.Diagnostic.Kind;
19.
20. import com.jmdoudoux.tests.annotations.Todo;
21.
22. @SupportedAnnotationTypes(value = { "*" })
23. @SupportedSourceVersion(SourceVersion.RELEASE_6)
```

```
24. public class TodoProcessor2 extends AbstractProcessor {
25.
26.   @Override
27.   public boolean process(
28.       Set<? extends TypeElement> annotations,
29.       RoundEnvironment roundEnv) {
30.
31.     Filer filer = processingEnv.getFiler();
32.     Messager messager = processingEnv.getMessager();
33.     Elements eltUtils = processingEnv.getElementUtils();
34.     if (!roundEnv.processingOver()) {
35.       TypeElement elementTodo =
36.         eltUtils.getTypeElement("com.jmdoudoux.tests.annotations.Todo");
37.       Set<? extends Element> elements = roundEnv.getElementsAnnotatedWith(elementTodo);
38.       if (!elements.isEmpty())
39.         try {
40.           messager.printMessage(Kind.NOTE, "Creation du fichier Todo");
41.           PrintWriter pw = new PrintWriter(filer.createResource(
42.               StandardLocation.SOURCE_OUTPUT, "", "Todo.txt")
43.               .openOutputStream());
44.           // .createSourceFile("Todo").openOutputStream());
45.           pw.println("Liste des todos\n");
46.
47.           for (Element element : elements) {
48.             pw.println("\nelement:" + element.getSimpleName());
49.             Todo todo = (Todo) element.getAnnotation(Todo.class);
50.             pw.println("  affecte le " + todo.dateAssignation()
51.               + " a " + todo.assigneA());
52.             pw.println("  description : ");
53.             for (String desc : todo.description()) {
54.               pw.println("    " + desc);
55.             }
56.           }
57.
58.           pw.close();
59.         } catch (IOException ioe) {
60.           messager.printMessage(Kind.ERROR, ioe.getMessage());
61.         }
62.       else
63.         messager.printMessage(Kind.NOTE, "Rien a faire");
64.     } else
65.       messager.printMessage(Kind.NOTE, "Fin des traitements");
66.
67.     return true;
68.   }
69. }
```

Résultat :

```
01. C:\Documents and Settings\jmd\workspace\TestAnnotations>javac -cp ".;./bin;C:/Pr
02. ogram Files/Java/jdk1.6.0/lib/tools.jar" -processor com.jmdoudoux.tests.annotati
03. ons.outils.TodoProcessor2 com/jmdoudoux/tests/*.java
04. Note: Creation du fichier Todo
05. Note: Fin des traitements
06.
07. C:\Documents and Settings\jmd\workspace\TestAnnotations>type Todo.txt
08. Liste des todos
09.
10.
11. element:MaClasse
12.   affecte le 11-11-2007 a JMD
13.   description :
14.     Corriger le bug dans le calcul
15.
16. element:MaClasse1
17.   affecte le 07-11-2007 a JMD
18.   description :
19.     Ajouter le traitement des erreurs
```

10.9. Les ressources relatives aux annotations

La JSR 175 A Metadata Facility for the JavaTM Programming Language

La JSR 269 Pluggable Annotation Processing API

La JSR 250 Common Annotations

La page des annotations dans la documentation du JDK

La page des annotations dans le tutorial Java

La page d'utilisation de l'outil APT dans la documentation du JDK

Le projet open source XDoclet qui propose la génération de code à partir d'attributs dans le code

Développons en Java v 2.00
Copyright (C) 1999-2014 Jean-Michel DOUDOUX.