Annot@tions et Generics <>

Plan

- Annotations
- Generics

Annotations et Generics

Annotations

Exemples

Avez vous déjà rencontré ces éléments?

```
@Deprecated
@Override
@Before
@Test
```

Apparition

Historique

Les annotations sont apparues avec Java 1.5 en 2004, en même temps que les **Generics**, les **Enumerations**, les **Varargs**, …

Elles ont pour objectif de fournir des **Métadonnées** sur les éléments auxquelles elle se rapprochent : des éléments de définition ou d'usage complémentaire.

Les annotations sont des "entités", comme le sont les classes ou les énumérations.

Usage

Syntaxe

Une annotation se rapporte toujours à l'entité qu'elle précède. Elles peuvent être plusieurs à intéragir avec la même entité.

Une annotation se définit toujours grâce au caractère : **@**

Il existe 3 "catégories" d'annotation :

- Les **Markers** : Les annotations sans attribut
- Les **Single Value Annotations** : Les annotations possédant un attribut
- Les **Full Annotations** : les annotations possédant plusieurs attributs

Usage

Multiples annotations sur une entité

```
@EqualsAndHashCode
@NoArgsConstructor
@AllArgsConstructor
@NodeEntity(label = "test")
public class ClasseQuelconque{


}
```

Usage

Import

Comme n'importe quelle "entité" , toutes les annotations possèdent un **package**.

Pour être utilisées dans une autre entité, elles doivent être **importées** en précisant leur package complet.

IMPORT mon.package.mon.Annotation

Utilisation

Les annotations peuvent être apposées sur :

- Les classes
- Les interfaces
- Les constructeurs
- Les méthodes
- Les paramètres de méthode

Usage

Classe ou interface

L'annotation est à placer avant la déclaration de la classe ou de l'interface

```
import mon.Annotation
@MonAnnotation
public class MaClasse {
}
```

Usage

Méthode ou constructeur

L'annotation est à placer avant la déclaration de la méthode ou du constructeur

```
@MonAnnotation
public void maMethode() {
}
```

Usage

Paramètres de méthode

L'annotation est à placer avant la déclaration du paramètre

```
public void maMethode(@MonAnnotation String monParametre,
                      @MonAnnotation String monSecondParametre) {
}
```

Single Value Annotations

L'attribut de l'annotation se place comme un paramètre de méthode, entre parenthèses. Il est **typé** , comme n'importe quel attribut de classe.

```
@Query("SELECT e from Etudiant e")
```

```
@MonAnnotation(MonEnum.MaValeur)
```

Syntaxe

Full Annotations

Pour identifier les différents attributs, ils doivent être nommés.

Ils sont déclarés l'un derrière l'autre, séparés par des virgules.

```
@MonAnnotation(arg1="test 3", arg2="test 2", arg3="test3")
```

Attributs multi valués

Les attributs d'une annotation peuvent être multivalués.

La syntaxe reste la même pour chaque attribut simple. Les attributs multivalués prennent une syntaxe "tableau" pour leur valeur, avec des accolades.

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

Création d'une annotation

La création d'une annotation se fait comme celle d'une classe, dans un fichier qui lui est propre. C'est le mot clé **@interface** qui est utilisé.

```
package mon.package;


public @interface MonAnnotation {
}
```

Syntaxe

Attributs d'une annotation

A la différence des classes, les attributs d'une annotation sont toutes des méthodes.

Une valeur par défaut peut être spécifiée, avec le mot clé **default**.

```
public @interface MonAnnotation {

    public String attr1() default "";
    public int attr2() default 0;
}
```

Scope d'une annotation

Définir la portée et l'usage d'une annotation se font via des annotations elle même.

- `@Target` : spécifie la cible possible d'une annotation (classe, méthode,...)

- `@Retention` : donne la durée de vie d'une annotation. **Runtime** pour l'exécution, **Source** pour la compilation, ou **Class** le comportement par défaut.

Syntaxe

Exemple complet

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MonAnnotation {

    public String attr1() default "";
    public int attr2() default 0;

}
```

Syntaxe

Appel

```
@MonAnnotation(attr1 = "tata")
MaClasse faitQuelqueChose{
    …
}
```

Usage encore

Dans la vraie vie

En pratique il est **rare** de devoir redéfinir une annotation, on se sert généralement de celles fournies par le langage ou par les frameworks.

Ajouter du comportement à la compilation ou à l'exécution est **compliqué** (introspection, types paramétrables, etc), et est donc à réserver à des cas extrêmes ou très spécifiques.

Annotations et Generics

Generics<>

Historique

Encore du 1.5!

Comme pour les annotations, les **Generics** sont apparus avec Java 1.5.

L'objectif de cette feature est de fournir un niveau d'abstraction supplémentaire à la notion d'Objet et d'apporter des informations sur une classe qui ***pourrait*** avoir besoin d'une autre classe pour son traitement.

En français, on utilise souvent le terme de **"type paramétrable"**.

Problème original

Avant Java 1.5, la manipulation d'une liste d'entiers se faisait comme suit :

```
List list = new ArrayList();
list.add(new Integer(1));
Integer i = (Integer) list.get(0);
```

Le compilateur était incapable de comprendre le type de l'Objet dans la liste.

Il faut "**caster**" le résultat en précisant sa classe entre parenthèses.

Problème original

Comme il est possible de caster un Objet en n'importe quoi (tout hérite d'objet), cette ligne compilait également :

```
List list = new ArrayList();

list.add(new Integer(1));

Integer i = (List) list.get(0);
```

Erreur garantie à l'exécution! Indetectable à la compilation !

Diamond

Solution

Java 1.5 ajoute l'opérateur **Diamond <>** à la liste qui lui permet de stocker le type qu'elle contient. Ainsi le code devient :

```
List<Integer> list = new ArrayList();

list.add(new Integer(1));

Integer i = list.get(0);
```

Le Cast disparaît au profit du type paramétrable de la List.

Solution

Ainsi cette fois, c'est cette syntaxe qui ne compile plus :

```
List<Integer> list = new ArrayList();
list.add("toto");
Integer i =  list.get(0);
```

Type Erasure

Définition

La notion de type paramétrable n'existe pas dans le bytecode Java.

A la compilation, le compilateur remplace tous les types paramétrés par Object !

```
List<Integer> list = new ArrayList();
```

Devient ainsi

```
List<Object> list = new ArrayList();
```

On appelle ce phénomène le **Type Erasure.**

Les types paramétrés ne peuvent pas être des types primitifs!

Ils n'héritent pas d'Object!

```
List<int> list = new ArrayList();
```

Ne compile pas !

Une solution par type primitif

Chaque type primitif possède un **Wrapper** qui hérite d'Object

```
int => Integer
double => Double
char => Character
boolean => Boolean
...
```

Pièges de type erasure

```
public class MaClasse implements
Comparator<String>, Comparator<Integer>{ … }
```

Ne compile pas!

Raison :

```
Duplicate class:  'java.util.Comparator'
```

Déclaration

Avec la classe

Une classe avec un type paramétrable se déclare comme ceci :

```
public class MaClasse<T> {
}
```

En général, on utilise la lettre **T**, mais c'est plus une convention.

Au sein de la classe

La classe T est ensuite accessible comme n'importe quelle autre classe au sein de la notre.

```
public class MaClasse<T> {
    private T element;
    public T get(){
        return element;
    }
}
```

Utilisation

Un Generic peut être paramétré par plusieurs classes

```
public class MaClasse<T,P> {
    private T element;
    private P element2;

    public T getT(){return element;}

    public P getP(){return element2;}
}
```

Héritage et Generics

Supposons une classe Abstraite Produit comme ceci :

```
public abstract class Produit{
    public abstract Integer getPrix();
}
```

Et deux classes en héritant : Fruit et Legume

```
public class Fruit extends Produit {
}
public class Legume extends Produit {
}
```

Héritage et Generics

Nous aimerions créer une Classe qui, à partir d'une liste de Fruits ou de Legume retourne le Produit le plus cher.

```
public class CalculateurDePrixSimple {

    public Produit getPlusCher (List<Produit> produits){

        Produit plusCher = produits.get( 0);

        for(Produit produit : produits){

            if( produit.getPrix() > plusCher.getPrix()){

                plusCher = produit;

            }

        }

        return plusCher;

    }
```

Héritage et Generics

Pour l'utilisation, nous serions tentés de faire quelque chose comme ceci :

```
CalculateurDePrixSimple calc= new CalculateurDePrixSimple();
List<Fruit> fruits = new ArrayList();
Produit fruit = calc.getPlusCher(fruits);
```

Cela ne compile pas!

La méthode getPlusCher attend une liste de Produit et pas de ses sous types!

Extension

Il est possible lors de la définition d'un **type paramétré** de dire qu'il hérite d'une classe.

```
public class MaClasse <T  extends MaSecondeClasse> {

}
```

Ici le type T hérite nécessairement de la classe MaSecondeClasse grâce au mot clé **extends**. Si T n'hérite pas de la classe, cela ne compile pas!

Toutes les méthodes de MaSecondeClasse sont donc accessibles sur les instances de T.

SUPER

Il existe le même mécanisme avec le mot clé **super**.

Auquel cas T est une “superclasse” de MaSecondeClasse.

Son usage est très rare.

Héritage et Generics

```
public class CalculateurDePrix<T  extends Produit> {
    public T getPlusCher (List<T> produits){
        T plusCher = produits.get(0);
        for(T produit : produits){
            if( produit.getPrix() > plusCher.getPrix()){
                plusCher = produit;
            }
        }
        return plusCher;
    }
}
```

Usage V2

```
CalculateurDePrix<Fruit> calculateurDePrix = new CalculateurDePrix();
List<Fruit> fruits = new ArrayList();
Fruit fruit = calculateurDePrix.getPlusCher(fruits);
```

Compile parfaitement!

Usage V2

```
CalculateurDePrix<Fruit> calculateurDePrix = new CalculateurDePrix();
List<Legume> legumes = new ArrayList();
Legume legume = calculateurDePrix.getPlusCher(legumes);
```

Ne Compile pas!

La méthode attend une liste de Fruit, pas de Legume!

Wildcard

En java, l'opérateur **?**, aussi appelé **Wildcard**, permet de définir un type inconnu.

Ainsi `? extends MaSecondeClasse` définit n'importe quelle classe héritant de MaSecondeClasse.

? vs T

A la différence du nommage du classe paramétrable en T, ? ne peut pas être réutilisé en tant qu'objet dans un retour ou pour le typage d'un objet :

```
public class MaClasse <? extends MaSecondeClasse> {
    private ? test;
}
```

Ne compile pas du tout !

? vs T

L'intérêt est en général d'utiliser ? dans les paramètres d'une méthode.

Il permet de conserver le type du paramètre d'entrée et surtout d'utiliser une classe fille de MaSecondeClasse

```
public MaSecondeClasse maMethode(List<?  extends MaSecondeClasse> list){
    return list.get(0);
}
```

Héritage et Generics

```
public class CalculateurDePrixWildCard {
    public Produit getPlusCher (List<? extends Produit> produits){
        Produit plusCher = produits.get(0);
        for(Produit produit : produits){
            if( produit.getPrix() > plusCher.getPrix()){
                plusCher = produit;
            }
        }
        return plusCher;
    }
```

Usage V2

```
CalculateurDePrixWildCard calc= new CalculateurDePrixWildCard();
List<Fruit> fruits = new ArrayList();
List<Legume> legumes = new ArrayList();
Produit fruit = calc.getPlusCher(fruits);
Produit leg = calc.getPlusCher(legumes);
```

Compile parfaitement!

Usage V2

```
CalculateurDePrixWildCard calc= new CalculateurDePrixWildCard();
List<Fruit> fruits = new ArrayList();
Fruit fruit = calc.getPlusCher(fruits);
```

Ne compile pas du tout !

La méthode getPlusCher retourne un Produit!