



Le langage Java

Java 5

Richard
BONNAMY



Objectifs Pédagogiques

À l'issue de cette formation, vous serez en mesure de :

- ✓ Utiliser les génériques
- ✓ Utiliser les annotations
- ✓ Utiliser les énumérations

Programme détaillé ou sommaire

Introduction

Les génériques

Les fonctionnalités usuelles

Les énumérations

Les imports static

Nombre variable d'arguments

Les annotations

Chapitre 1

Introduction

Introduction

La version 5 de Java Standard Edition (nom de code : Tiger) propose de nombreuses fonctionnalités en plus.

Améliorations majeures dans la qualité, la supervision et la gestion, la performance et la faculté de montée en charge, ainsi que la facilité de développement.

Ce cours se focalise sur les fonctionnalités les plus significatives et les plus attendues du J2SE 5.0

Chapitre 2

Les génériques

Présentation

Les Génériques sont très attendus par la communauté Java.

Il existait un comité de réflexion sur les génériques depuis plus de 5 ans.

Les Génériques devraient permettre d'éviter de nombreuses erreurs d'exécution.

Principe de base : typer les éléments d'une collection.

Exemple sans les génériques

Il n'est pas possible de spécifier le type d'élément que doit stocker la liste.

Si un mauvais type est employé lors de la récupération d'un élément de la liste, l'erreur n'est pas détectée en phase de compilation.

```
List voitureList = new ArrayList();  
voitureList.add(new Voiture());  
voitureList.add(new Pizza());  
// ...  
Voiture v1 = (Voiture) voitureList.get(0);  
Voiture v1 = (Voiture) voitureList.get(1);
```

OK à la compilation

KO à l'exécution



Cette méthode était la seule possible avant Java 5.0

Exemple avec les génériques

La liste est **typée** lors de sa **création**.

Elle ne peut recevoir qu'un type d'élément donné.

Plus besoin de faire de **cast** lors de la récupération d'un élément.

Une éventuelle erreur de type est détectée en phase de compilation.

```
List<Voiture> voitureList = new ArrayList<>();  
voitureList.add(new Voiture());  
voitureList.add(new Pizza());  
// ...  
Voiture v1 = voitureList.get(0);
```

KO à la compilation



Exemple de définition

```
public class Conteneur<T> {  
  
    private T[] tableaux;  
  
    public void ajouter(T element){  
        for (int i=0; i<tableaux.length; i++){  
            if (tableaux[i]==null){  
                tableaux[i]=element;  
            }  
        }  
    }  
}  
  
public class EssaiGenerique {  
  
    public static void main(String[] args) {  
  
        Conteneur<String> conteneur = new Conteneur<String>();  
        conteneur.ajouter("Coucou");  
  
        Conteneur<Pizza> conteneurPizza = new Conteneur<Pizza>();  
        conteneurPizza.ajouter(new Pizza("CAD", "Calzone", 12.5));  
    }  
}
```

Chapitre 3

Les fonctionnalités usuelles

Autoboxing/unboxing (1/4)

Jusqu'au JDK 1.4, pour insérer un élément de type primitif dans une liste :

```
public class TestAutoboxingOld {  
    public static void main(String[] args) {  
        List liste = new ArrayList();  
        int i = 12;  
        liste.add(new Integer(i));  
    }  
}
```

JDK 5.0 propose l'autoboxing et l'unboxing.

L'autoboxing : transformer automatiquement une variable de type primitif en un objet

Unboxing : opération inverse

Autoboxing/unboxing (2/4)

Avec Java 5.0, le compilateur effectue automatiquement la conversion

La liste peut être remplie indifféremment par des variables de type primitif (ex: int) ou de type 'wrapper' (ex: Integer).

```
List<Integer> tempList = new ArrayList<Integer>();  
  
int i1 = 1;  
int i2 = 2;  
Integer i3 = new Integer(3);  
  
tempList.add(i1);  
tempList.add(i2);  
tempList.add(i3);
```

Exemple de remplissage d'une liste par autoboxing

Autoboxing/unboxing (3/4)

Un même élément "entier" peut être récupéré sous forme d'un int (type primitif) ou d'un Integer (objet).

Aucun cast nécessaire

```
List<Integer> tempList = new ArrayList<Integer>();  
tempList.add(50);           // ajout d'un int  
tempList.add(new Integer(30)); // ajout d'un Integer  
  
Integer i1 = tempList.get(0);  
int i2 = tempList.get(1);
```

Autoboxing/unboxing (4/4)

Opérations sur des éléments

*Autoboxing: passage de **int** à **Integer** (automatique)*

*Unboxing: passage de **Integer** à **int** (automatique également)*

```
int i = 12;  
Integer j = 15;  
int k = i + j;
```


Itérations simplifiées

Dans la nouvelle forme de l'instruction **for**, l'utilisation d'un Iterator est transparente.

```
public static void main(String[] args) {  
    List<Integer> liste = new ArrayList<Integer>();  
    for (int i = 0; i < 10; i++) {  
        liste.add(i);  
    }  
  
    for (Integer i : liste) {  
        System.out.println(i);  
    }  
}
```

Chapitre 4

Les énumérations

Les énumérés

Java 5 propose une nouvelle catégorie d'éléments : les enums

Une **enum** n'est pas une classe

Écriture simplifiée

A la compilation, une classe énumérée implémentant les bonnes pratiques est générée.

Problématique

Comment limiter le nombre d'instances d'une classe en Java.

Exemple: j'ai une classe **JourSemaine** dont je veux limiter le nombre d'instances à 7:

Lundi

Mardi

Mercredi

Jeudi

Vendredi

Samedi

Dimanche

Solution sans les énums

```
public class JourSemaine {  
  
    public static final JourSemaine LUNDI = new JourSemaine("Lundi");  
    public static final JourSemaine MARDI = new JourSemaine("Mardi");  
    public static final JourSemaine MERCREDI = new JourSemaine("Mercredi");  
    public static final JourSemaine JEUDI = new JourSemaine("Jeudi");  
    public static final JourSemaine VENDREDI = new JourSemaine("Vendredi");  
    public static final JourSemaine SAMEDI = new JourSemaine("Samedi");  
    public static final JourSemaine DIMANCHE = new JourSemaine("Dimanche");  
  
    private String nom;  
  
    private JourSemaine(String nom) {  
        this.nom = nom;  
    }  
    public String getNom() {  
        return nom;  
    }  
    public void setNom(String nom) {  
        this.nom = nom;  
    }  
}
```

Instances fournies sous forme de constantes

Constructeur *private*

Utilisation sans les énums

```
public static void main(String[] args) {  
  
    JourSemaine jour = JourSemaine.LUNDI;  
  
    // Suite du code  
}
```

Je ne peux pas instancier un JourSemaine mais je peux en référencer un existant

Passage de classe à enum

```
public enum JourSemaine {  
  
    public static final JourSemaine LUNDI = new JourSemaine("Lundi"),  
    public static final JourSemaine MARDI = new JourSemaine("Mardi"),  
    public static final JourSemaine MERCREDI = new JourSemaine("Mercredi"),  
    public static final JourSemaine JEUDI = new JourSemaine("Jeudi"),  
    public static final JourSemaine VENDREDI = new JourSemaine("Vendredi"),  
    public static final JourSemaine SAMEDI = new JourSemaine("Samedi"),  
    public static final JourSemaine DIMANCHE = new JourSemaine("Dimanche");  
  
    private String nom;  
  
    private JourSemaine(String nom) {  
        this.nom = nom;  
    }  
    public String getNom() {  
        return nom;  
    }  
    public void setNom(String nom) {  
        this.nom = nom;  
    }  
}
```

1) class remplacé par enum

2) Les constantes sont séparées par des Virgules.

3) Simplification de l'instanciation:
On ne garde que le **nom de l'instance**
et **les paramètres**

Énumération équivalente

```
public enum JourSemaine {  
  
    LUNDI("Lundi"),  
    MARDI("Mardi"),  
    MERCREDI("Mercredi"),  
    JEUDI("Jeudi"),  
    VENDREDI("Vendredi"),  
    SAMEDI("Samedi"),  
    DIMANCHE("Dimanche");  
  
    private String nom;  
  
    private JourSemaine(String nom) {  
        this.nom = nom;  
    }  
    public String getNom() {  
        return nom;  
    }  
    public void setNom(String nom) {  
        this.nom = nom;  
    }  
}
```

Attention, les 7 constantes sont des instances de JourSemaine avec un attribut nom.

Utilisation avec les énums

```
public static void main(String[] args) {  
  
    JourSemaine jour = JourSemaine.LUNDI;  
  
    // Suite du code  
}
```

*Identique à la version
précédente*

Souvenez-vous qu'une énumération fournit des instances de classes

Les avantages (1/4)

De très nombreux avantages :

Tout d'abord la déclaration des instances est **simplifiée**.

```
public static final JourSemaine LUNDI = new JourSemaine("Lundi");  
↓  
LUNDI("Lundi");
```

Les avantages (2/4)

Des méthodes déjà fonctionnelles :

Les méthodes **equals(...)** et **compareTo(...)** sont automatiquement implémentées

Se basent sur l'ordre de déclaration des valeurs possibles :

Par défaut : DIMANCHE > SAMEDI > ... > MARDI > LUNDI

```
JourSemaine j1 = JourSemaine.LUNDI;  
JourSemaine j2 = JourSemaine.LUNDI;  
  
System.out.println(j1.equals(j2));  
System.out.println(j1.compareTo(j2));
```

Affiche:
true
0

Les avantages (3/4)

Une méthode **implicite** qui retourne un tableau de toutes les instances:

La méthode static **values()** renvoie le tableau des valeurs possibles

```
JourSemaine[] jours = JourSemaine.values();  
for (JourSemaine jour : jours) {  
    System.out.println(jour.getNom());  
}
```

Les avantages (4/4)

Une méthode **implicite** qui retourne une instance en fonction de son **nom** :

La méthode static **valueOf(String name)** renvoie une instance en fonction de son nom :

```
JourSemaine jour = JourSemaine.valueOf("MARDI");  
System.out.println(jour);
```

Atelier (TP)

Objectifs du TP: apprendre à utiliser les énumérations.

Description du TP:

Dans ce TP, vous allez apprendre à utiliser une énumération.

Chapitre 5

Les imports static

Les imports statiques

Jusqu'à la version 1.4 de Java, pour utiliser un membre statique d'une classe, il faut obligatoirement préfixer ce membre par le nom de la classe qui le contient.

Par exemple, pour utiliser la constante **PI** définie dans la classe **java.lang.Math**, il est nécessaire d'utiliser **Math.PI**

```
public class TestAncienneManiere {  
    public static void main(String[] args) {  
  
        double rayon = 2.5;  
        double superficieCercle = Math.PI * rayon * rayon;  
  
        System.out.println(superficieCercle);  
    }  
}
```



Les imports statiques

En Java 5, si une classe est importée statiquement, il n'est plus nécessaire de préciser le type

```
import static java.lang.Math.PI;
public class TestNouvelleManiere {
    public static void main(String[] args) {

        double rayon = 2.5;
        double superficieCercle = PI * rayon * rayon;
        System.out.println(superficieCercle);
    }
}
```

L'utilisation de l'importation statique s'applique à tous les membres statiques constantes et méthodes statiques de l'élément importé

 Attention à ne pas abuser des imports statiques.
Ils peuvent provoquer des conflits d'espaces de nommage.

Chapitre 6

Nb d'arguments variables

Nombre variable d'arguments

Cette nouvelle fonctionnalité va permettre de passer un nombre non défini d'arguments d'un même type à une méthode. Ceci va éviter de devoir encapsuler ces données dans une collection.

Elle implique une nouvelle notation pour préciser la répétition d'un type d'argument. Cette nouvelle notation utilise trois points de suspension : ...

Nombre variable d'arguments

Paramètre traité comme un tableau

```
public class TestVarargs {  
    public static void main(String[] args) {  
        int[] valTab = { 1, 2, 3, 4 };  
        System.out.println("valeur = " + additionner(2, 5, 6, 8, 10));  
        System.out.println("valeurTab = " + additionner(valTab));  
    }  
  
    public static int additionner(int... valeurs) {  
        int total = 0;  
        for (int val : valeurs) {  
            total += val;  
        }  
        return total;  
    }  
}
```

Chapitre 7

Les annotations

Présentation

Dans les versions précédentes de Java, les métadonnées étaient traitées comme de simples commentaires par le compilateur.

Un interpréteur externe lisait les métadonnées.

Le compilateur de Java 5.0 peut stocker des métadonnées dans des classes.

Les métadonnées sont maintenant directement placées dans le code

```
@Override  
public String toString()  
{ //... }
```

Exemple de métadonnée Java 5.0

Définir / Utiliser une annotation

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

```
@MonAnnotation  
public void crediter(double montant) throws CompteException {  
    ...  
}
```

Annotations existantes

Le package `java.lang` définit deux annotations standard pour les méthodes :

@Override : définit qu'une méthode redéfinit une méthode de la classe mère.

Si la méthode 'marquée' ne redéfinit pas une méthode de la classe mère, la compilation échoue.

@Deprecated : Lance un avertissement si un membre annoté "deprecated" est utilisé.

@SuppressWarnings : indique au compilateur de ne pas afficher certains warnings.

Possibilité de créer ses propres annotations



Pour les tags ci-dessus, attention aux majuscules.

Méta-Annotations (1)

Les méta-annotations sont des annotations destinées à marquer d'autres annotations. Elles appartiennent toutes au package `java.lang.annotation`.

Les méta-annotations :

@Documented : indique à l'outil javadoc que l'annotation doit être présente dans la documentation générée pour tous les éléments marqués.

@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.

Méta-Annotations (2)

Les méta-annotations (suite) :

@Retention : indique de quelle manière elle doit être gérée par le compilateur. Elle peut prendre 3 valeurs :

RetentionPolicy.SOURCE : les annotations ne sont accessibles uniquement dans le fichier source (donc absent des classes compilées .class)

RetentionPolicy.CLASS (par défaut) : les annotations sont bien enregistrées dans les classes compilées mais ne sont pas prises en compte par la machine virtuelle lors de l'exécution.

RetentionPolicy.RUNTIME : les annotations peuvent être utilisées lors de l'exécution

Méta-Annotations (3)

Les méta-annotations (suite) :

@Target : permet de limiter le type d'éléments sur lesquels l'annotation peut être utilisée. Par défaut, une annotation peut être utilisée sur tous les éléments. Valeurs possibles :

ElementType.ANNOTATION_TYPE

ElementType.CONSTRUCTOR

ElementType.FIELD

ElementType.LOCAL_VARIABLE

ElementType.METHOD

ElementType.PACKAGE

ElementType.PARAMETER

ElementType.TYPE

Exemple

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface ToString {
```

```
    String separateur() default "";
    boolean upperCase() default false;
}
```

```
class Pizza {

    @ToString(separateur="->", upperCase=false)
    private String nom;

    @ToString(upperCase=true)
    private String libelle;
    |
    private float prix;
```

Annotation @ToString:

- avec 2 attributs **separateur** et **upperCase**.
- Ne peut se placer que sur les variables d'instance.

Exploitation des annotations

Pour exploiter les annotations, il faut savoir sur quels attributs, classes ou méthodes elles ont été positionnées.

Problématique : Comment connaître la structure d'une classe ?

Réponse : Grace à la méthode getClass().

```
Ville v = new Ville("10", "Centre", "37", "37000", "Tours", 132700);
```

```
Class<?> classe = v.getClass();
```

```
Field[] fields = classe.getDeclaredFields();
```

Permet de récupérer la liste des attributs de la classe (6 au total)

Package : java.lang.reflect

Accéder à la valeur d'un attribut

```
Ville v = new Ville("10", "Centre", "37", "37000", "Tours", 132700);
```

```
Class<?> classe = v.getClass();
```

```
Field[] fields = classe.getDeclaredFields();
```

```
for (Field f : fields) {
```

```
    // permet de forcer la lecture d'un attribut privé
```

```
    f.setAccessible(true);
```

Permet d'accéder en lecture à un attribut privé

```
    // Permet d'afficher le nom de l'attribut
```

```
    System.out.println(f.getName());
```

Permet d'afficher le nom de l'attribut. Exemple : population

```
    // Permet d'afficher la valeur de l'attribut pour l'instance v
```

```
    System.out.println(f.get(v));
```

Permet d'afficher la valeur de l'attribut. Exemple : 132700

```
}
```

Savoir si un attribut porte une annotation

```
Ville v = new Ville("10", "Centre", "37", "37000", "Tours", 132700);
```

```
Class<?> classe = v.getClass();
```

```
Field[] fields = classe.getDeclaredFields();
```

```
for (Field f : fields) {
```

```
    // permet de forcer la lecture d'un attribut privé
```

```
    f.setAccessible(true);
```

```
    if (f.isAnnotationPresent(ToString.class)) {
```

```
    }
```

```
}
```



Permet de savoir si l'attribut porte l'annotation ToString

Atelier (TP)

Objectifs du TP: apprendre à utiliser les annotations.

Description du TP:

Dans ce TP, vous allez apprendre à utiliser une annotation dans le cadre de la mise en place d'un mécanisme.