Bases de la POO / Java

La généricité

TYPES GENERIQUES
MÉTHODES ET CLASSES GÉNÉRIQUES

Qu'est-ce que la généricité ?

- Dans une fonction « classique », les paramètres sont des valeurs
  - Au moment de sa définition, des valeurs sont inconnues (par. formels)
  - Au moment de l'appel, ces valeurs sont fixées (par. d'appel)
- Dans un générique, les paramètres sont des types
  - Dans sa définition, des types sont inconnus
  - Au moment d'utiliser le générique, ces types sont fixés
- Un générique est un modèle
  - Instanciation = création d'un élément à partir d'un modèle
  - Instancier un générique → fixer le type de ses paramètres
- Composants pouvant être génériques en Java
  - Classes, interfaces et méthodes
  - A partir du JDK 5.0

En programmation

- La généricité est un concept important car elle permet au programmeur de s'abstraire de détails inhérents au fonctionnement de la machine.
- Elle repose sur l'indépendance du code vis-à-vis du type des données manipulées.
- Elle augmente le niveau d'abstraction du langage.

Types génériques (1)

Java Specification Request (**JSR**) **014** :

« Add Generic Types to the Java Programming Language ».

Pouvoir décrire des comportements factorisés pour plusieurs types polymorphes de données.

Déléguer au compilateur des transtypages à réaliser de manière implicite et automatique.

Types génériques (2)

Java Specification Request (**JSR**) **014** (suite) :

Suppression du contrôle de type à l'exécution si contrôle préalable à la compilation.

Usage du couple de meta caractères **< --- >**

Domaines d'application

- Modules paramétrés
- Classes génériques (ou template C++)
  - Au lieu de définir plusieurs classes similaires pour décrire un même concept, appliqué à plusieurs types de données différents.
- Utilisée et **très utile** avec les collections

Liste sans typage générique

```
public static float moyenne (LinkedList notes)
{
    float somme=0.0f;
    Iterator i= notes.iterator();
    while (i.hasNext()) somme += (Float)i.next();
     return somme/notes.size();
}
```

Liste avec typage générique

```
public static float moyenne (LinkedList<Float) notes)
{
    float somme=0.0f;
    Iterator i= notes.iterator();
    while (i.hasNext()) somme += i.next();
     return somme/notes.size();
}
```

Syntaxe nouvelle boucle **for** (1)

for (<type> <var> **:** <collection>) {instructions}.

Usage du symbole **:**

Forme simplifiée pour le parcours des collections.

Permet de masquer l'usage des itérateurs.

Exemple :

```
for (float val : notes) somme+= val;
```

Syntaxe nouvelle boucle **for** (2)

Parcours d'une collection (Java.util.Iterable)

```
ArrayList lesValeurs= new ArrayList();
lesValeurs.add("bleu");
lesValeurs.add("vert");
lesValeurs.add("jaune");

   for (String v : lesValeurs)
      System.out.println(v);
```

Liste avec typage générique et nouvelle boucle for

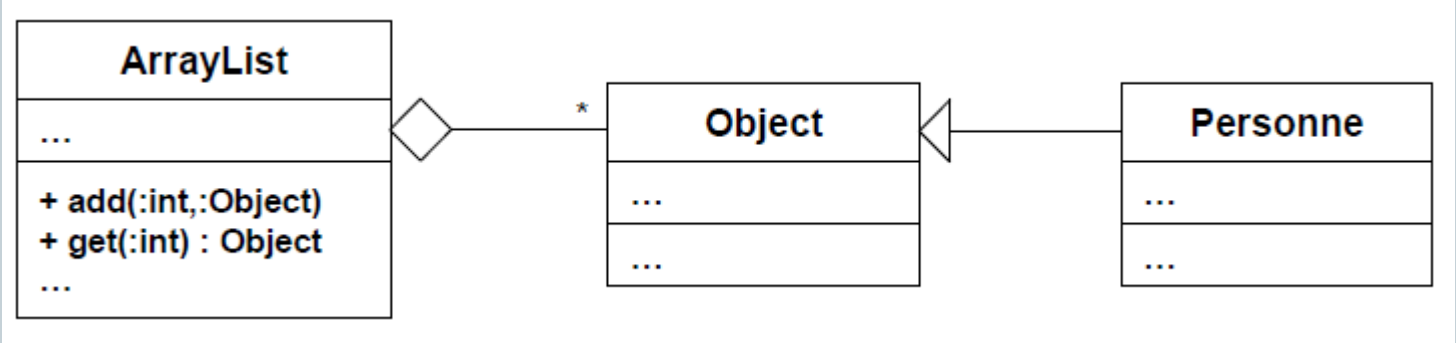
```
public static float moyenne (LinkedList<Float) notes)
{
    for (float val : notes) somme += val;
     return somme/notes.size();
}
```

Généricité et collections

- **Sans la généricité**
- Collection = collection d'objets de type **Object**
  - Tout objet de type **Object** peut être ajouté dans la collection
  - Aucun contrôle préalable ⇒ contenu **hétérogène**

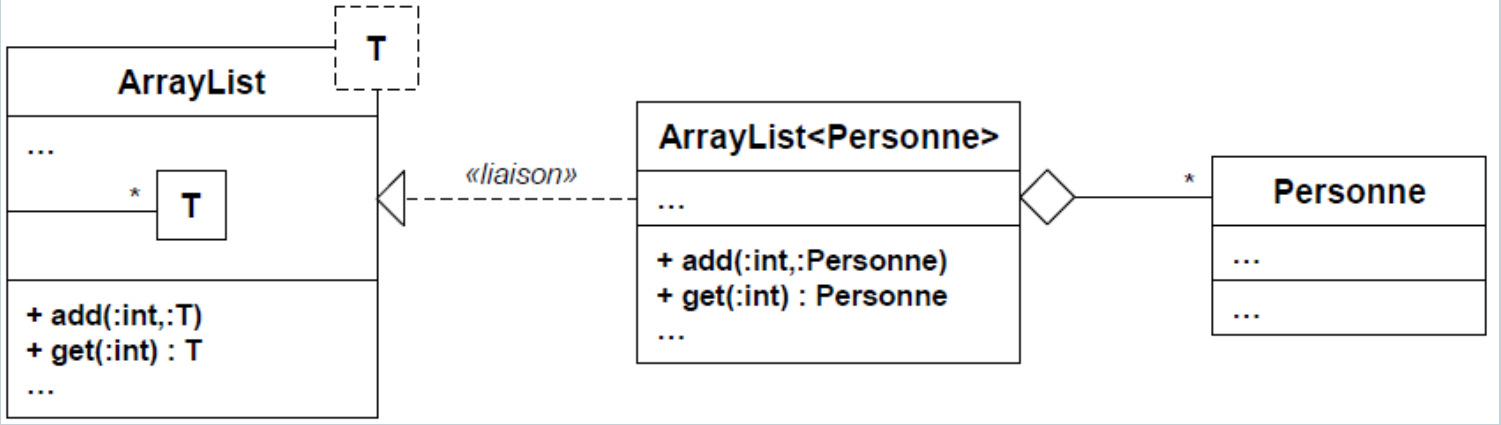
- **Avec la généricité**
- Collection **générique** = collection d'objets de type **T**
  - **T à définir**
  - Seuls des objets de type **T** peuvent être ajoutés dans la collection
  - Contrôle préalable automatique ⇒ contenu **homogène**

Sans la généricité



- Aucune précision du type des éléments contenus dans la collection
  - **ArrayList personnes = new ArrayList();**
- Ajout
  - Méthode add de ArrayList
  - **Personne p1 = new Personne("toto");**
  - **personnes.add(p1); // ok, Personne hérite de Object**
- Accès
  - Méthode get de ArrayList : retourne un **Object**
  - **Personne p2 = (Personne) personnes.get(0); // Transtypage obligatoire**
  - **Problème : en cas d'erreur, la détection ne se fera qu'à l'exécution**

Avec la généricité



- Définition du type précis des éléments contenus dans la collection
  - **ArrayList<Personne> personnes = new ArrayList<Personne>();**
- Toutes les vérifications sont faites à la compilation
  - **Personne p1 = new Personne("toto");**
  - **personnes.add (p1);** **// OK**
  - **Personne p2 = personnes.get (0);** **// OK, recast inutile**
  - **personnes add (new Rectangle()) ;** **// KO, erreur compilation**
- Intérêt
  - Vérification des types à la compilation
  - Moins de contrôle à l'exécution
  - Transtypage inutile

Classe ArrayList<E>

java.util

Class ArrayList<E>

java.lang.Object
  └ java.util.AbstractCollection<E>
    └ java.util.AbstractList<E>
      └ **java.util.ArrayList<E>**

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList