JAVA
La généricité

LICENCE 3 Sciences pour l'Ingénieur Parcours Informatique

Problématique

- On veut créer une classe avec des attributs dont on ne connais pas encore le type : des chaînes de caractères ? Des entiers ? Des réels ?
- Quelles solutions s'offrent à nous ?
  - Mettre des Object ? (utilisation du polymorphisme)
    - Intérêt : on met ce qu'on veut dedans
    - Limitation : une fois l'objet utilisé, on ne sais le type de données utilisé : c'est des Object, mais après ?
  - La généricité ?
    - Intérêt : on peut contraindre le type de données utilisé

Exemple

```
public class ClasseGenerique<E>{
    private E contenu;
    public ClasseGenerique(E contenu){
        this.contenu = contenu;
    }
    public E getContenu(){
        return contenu;
    }
}
```

La généricité

- Paramétrer les classes ou interfaces par un type
  - abstraction sur les types lors de la conception des classes
  - les comportements communs sont définis quel que soit le type des objets
- Un type générique, ou type paramétré peut prendre différentes valeurs
  - L'instanciation se fait suivant un ou plusieurs types de données
- Pour utiliser un type générique
  - spécifier un type pour chacun des paramètres de type demandé,
  - le type générique est alors contraint à ne prendre en compte que les objets des types que vous avez spécifié

Exemple : la classe ArrayList<Integer>

- Remplacer le paramètre de type `<E>` par un type concret, comme `<Integer>` ou `<String>` ou `<VotreType>`
- Une `ArrayList<Integer>` ne peut stocker que des instances d'Integer ou instances de filles de Integer (sous-type)
- Exemple :

```
ArrayList<Integer> li = new
    ArrayList<Integer>();
li.add(new Integer(0));
Integer i = li.iterator().next();
```

Raisons à l'introduction de la généricité

- Liées à la gestion des collections d'objet
- Avant la version 1.5 les collections manipulent des types Object
  - On peut donc avoir des collections hétérogènes
  - Il est impossible de contraindre la collection à ne manipuler que des objets d'un type prédéfini
  - Grande souplesse, mais limitation du contrôle des éventuelles erreurs
- Exemple :
  - Vous souhaitez stocker des chaînes de caractère dans un ArrayList.
  - Par erreur, vous intégrez un Integer
  - Le compilateur ne pas détecter l'erreur

Raisons à l'introduction de la généricité

- Problème
  - Le compilateur ne peut vérifier dans les collections d'objets la cohérence des types
  - Utilisation systématique du forçage de type (Cast)
  - Levée d'exceptions du type ClassCastException lors de l'exécution
- Solution offerte par la généricité
  - Vérification de la compatibilité des types à la compilation
  - Le cast est effectué par le compilateur
  - L'erreur précédente est évitée...

Exemple : Cast automatique

- Avant la généricité (cast obligatoire)

```
List myIntList = new ArrayList();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.iterator().next();
```

- Après la généricité (pas de cast !)

```
List<Integer> myIntList = new ArrayList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

Définir une classe générique

- Faire suivre le nom du type (classe ou interface) d'un ou plusieurs paramètres de type (par convention une lettre majuscule)
- Si T est un paramètre qui désigne un type inconnu au moment de la compilation, il peut intervenir dans les déclarations de variables, méthodes, classes, interfaces, collections
- T est un paramètre de type formel

```
public class ClasseGenerique<T,E,...>{...}
```

Conventions de nommage des paramètres de type

- E – Element
  - Elément (utilisé en particulier dans le cadre des collections)
- K - Key
  - Clé (table de hachage)
- N – Number
- T – Type
- V - Value

Méthodes générique

- Les méthodes peuvent, comme les classes et les interfaces être paramétrées par un type
  - Méthodes d'instances, de classe et constructeurs
- Une méthode générique peut être incluse dans une classe générique (si elle utilise un autre paramètre que les paramètres de type formels de la classe) ou dans une classe non générique
- Syntaxe :

```
public interface Collection<E> extends Iterable<E>{
     ...
     public abstract <T> T[] toArray(T[] a);
}
```

Utilisation, invocation

- Les méthodes génériques s'invoquent comme les méthodes "classiques"
- Le compilateur déduit du contexte le type qui doit être utilisé
- ```
  public interface Collection<E> extends
                                  Iterable<E>{
      public abstract <T> T[] toArray(T[] a);
  }
  ```
- ```
  ArrayList<String> liste;
  String[] result = liste.toArray(new String[0]);
  //Implicitement liste.<String>toArray(...)
  ```

La généricité avec plusieurs types ?

- A l'intérieur de la collection les relations de sous-typage sont toujours effectives

```
ArrayList<Number> an = new ArrayList<Number>();
an.add(new Integer(5));          //OK
an.add(new Long(1000L));         //OK
an.add(new String("Hello"));     //Erreur de compilation
```

- Problème : comment écrire une méthode permettant d'afficher tous les éléments d'une collection quels qu'ils soient ?
- Solution : utiliser les **wildcard types** ou **types joker**

```
static void printCollection(Collection<?> c){
     for (Object o : c) System.out.println(o);
}
```

Généricité : les types joker

- **<?>** type fixé mais inconnu
- **<? extends A>** désigne un type fixé mais inconnu qui est A ou un sous type de A, une classe fille
- **<? super A>** désigne un type fixé mais inconnu qui est A ou un sur type de A, une classe mère
- **A** peut être une classe, une interface, ou même un paramètre de type formel