Généricité et Collections

Oumar Y. MAÏGA

ITMA 2025

Plan

- Généricité
- Collections

Généricité

- En programmation, la généricité consiste à définir des algorithmes identiques opérant sur des données de types différents.
- C'est une forme de polymorphisme, le polymorphisme de type dit aussi paramétrage de type.
- Depuis la version 5.0 Java autorise la définition de classes et d'interfaces contenant un (des) paramètre(s) représentant un type(s). Cela permet de d'écrire une structure qui pourra être personnalisée au moment de l'instanciation à tout type d'objet.

Problématique de la Généricité

- les versions de Java antérieures à 1.5 permettaient de créer des classes de structures contenant n'importe quels types d'objet :
  - les collections (classes implémentant l'interface Collection)
  - des classes créées par le programmeur et travaillant sur des instances d'Object
- Problèmes :
  - manipuler les objets référencés par ces structures oblige à faire du
  - Transtypage vers le bas (downcasting).
  - risque d'erreur de cast, repérables uniquement à l'exécution (ClassCastException)
  - impossibilité d'obliger à ce qu'une collection ne contienne qu'un seul type d'objet (comme avec les tableaux)

Exemple: Couple d'Objets

```
public class CoupleEntier {
    private int premier;
    private int second;
public CoupleEntier(int premier, int second) {
    this.premier = premier;
    this.second = second;
}
public int getPremier() {return premier;}
public void setPremier(int premier) { this.premier = premier;}
public int getSecond() {return second;}
public void setSecond(int second) {this.second = second;}
public void interchanger(){
    int temp=premier;
    this.premier=this.second;
    this.second=temp;
}
}
```

Limites de la solution

- Pour avoir des couples de réels ou de chaînes de caractères ou d'autres types, il faut créer de nouvelles classes représentant

Solution utilisant la classe Object

```
public class CoupleObjet {
    private Object premier;
    private Object second;
public CoupleObjet(Object premier, Object second) {
    this.premier = premier;
    this.second = second;
}
public Object getPremier() { return premier;}
public void setPremier(Object premier) {this.premier = premier;}
public Object getSecond() {return second;}
public void setSecond(Object second) {this.second = second;}
public void interchanger(){
    Object temp=premier;
    this.premier=this.second;
    this.second=temp;
}
}
```

Inconvénients avec Object

- Les attributs peuvent être des instances de classes différentes
- Possibles erreurs de transtypage détecter seulement à l'exécution.

- A et B étant deux classes, on peut avoir ce genre d'utilisation
- A a = new A ( ) ; B b = new B ( ) ;
- CoupleObjet p = new CoupleObjet (a , b) ;
- A a2 = (A) p.getPremier( ) ; // downcasting ok
- p.interchanger ( ) ;
- A a2 = (A) p . g e t P r e m i e r ( ) ; // e r r e u r

Solution Générique

```
public class Couple<T> {
    private T premier;
    private T second;
public Couple(T premier, T second) {
    this.premier = premier;
    this.second = second;
}
public T getPremier() {return premier;}
public void setPremier(T premier) {this.premier = premier;}
public T getSecond() {return second;}
public void setSecond(T second) {this.second = second;}
public void interchanger(){
    T temp=premier;
    this.premier=this.second;
    this.second=temp;
}
}
```

Avantages de la généricité

- La généricité permet d'écrire des algorithmes généraux opérant sur toute une série de types
- Cette définition nous a permis d’ écrire des couples de type uniforme mais arbitraire.
- La généricité augmente le niveau d'abstraction des programmes écrits dans un langage qui possède cette fonctionnalité.

Utilisation du paramètre de type

- Une classe générique doit être instanciée avant d'être utilisée.
- On ne peut pas utiliser un type primitif pour l'instanciation, il faut utiliser les classes enveloppantes.
- Les types paramètres peuvent être utilisés pour déclarer des variables (y compris des tableaux) sauf dans une méthode de classe.
- Les types paramètres ne peuvent être utilisés pour créer des objets (ni des tableaux)
- On ne peut pas utiliser un paramètre de type avec instanceof (car à l'exécution, le type n'est plus connu)
- On peut utiliser un paramètre de type pour instancier une classe générique dans la classe paramétrée

Généricité avec plusieurs paramètres

```
public class Couplet<T,U> {
    private T premier;
    private U second;
public Couplet(T premier, U second) {
    this.premier = premier;
    this.second = second;
}
public T getPremier() {
    return premier;}
public void setPremier(T premier) {
    this.premier = premier;}
public U getSecond() {
    return second;}
public void setSecond(U second) {
    this.second = second;}
}
```

Mécanisme de la Généricité

- A la compilation, lors de l'instanciation d'une classe générique, les paramètres de type sont remplacés par le type spécifié
- Une seule classe compilée est produite, quelque soit le nombre d'instanciations
- Les variables de type sont oubliées (ne sont plus connues après compilation)
- Le compilateur doit parfois ajouter des cast pour respecter les contraintes de type

Généricité et Abstraction

- Les interfaces et classes abstraites peuvent être rendues génériques par paramétrage de type
- On peut instancier une classe (ou interface) générique par une classe abstraite ou une interface.
  - Les objets utilisés dans la classe instanciée seront par contre forcément des instances de classes concrètes
- Il est impossible d'implémenter deux interfaces qui sont des instanciations de la même interface générique

Héritage et Généricité

- On peut étendre une classe générique par une autre classe générique, à condition que les variables de type soient les mêmes
- Les règles de sous-typage des classes génériques sont :
  - Si C <T> extends D <T> alors C <T> est une sous-classe de D <T> (au sens où toute instance de l'une est instance de l'autre)
  - La relation d'héritage reste valide entre instanciations de C et D par un même type. Par exemple, C <String> est sous-classe de D <String>.
  - Cette relation d'héritage n'est pas valide entre instanciations de C et D par des types différents, même si ces types sont liés par héritage!
  - Une instanciation d'une classe générique n'est pas sous-classe de la classe générique : C <String> n'est pas sous-classe de C <T>

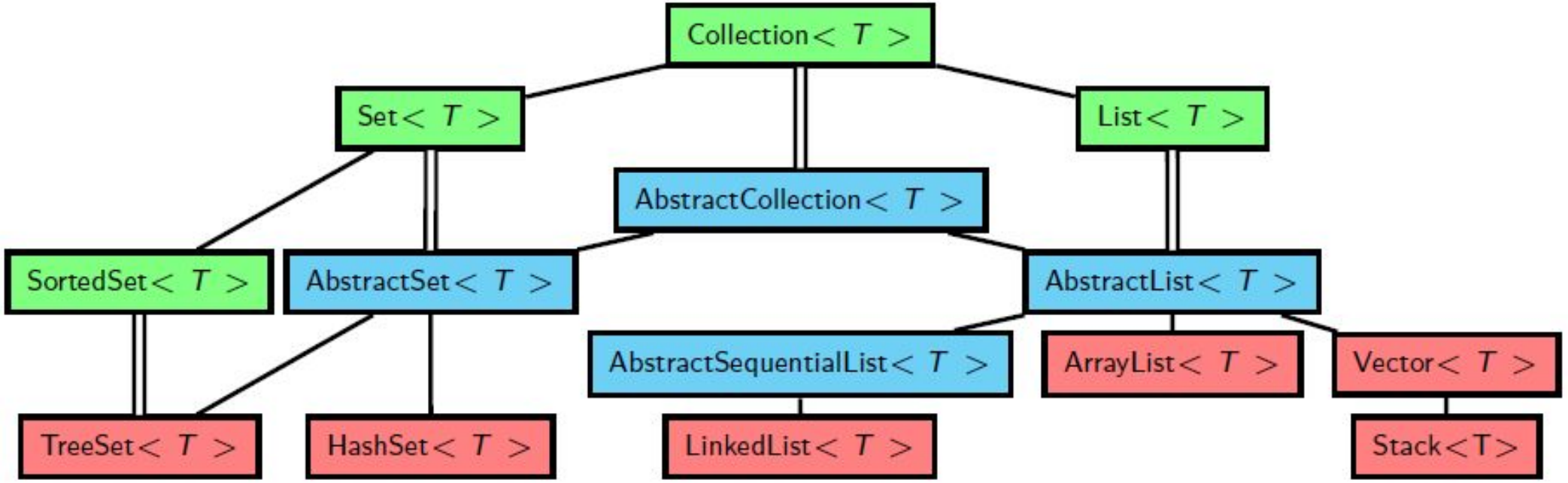
Méthodes Génériques

- Une méthode peut être paramétrée par un type, qu'elle soit dans une classe générique ou non
- L'appel de la méthode nécessite de l'instancier par un type, sauf si le compilateur peut réaliser une inférence de type

Collections

- Java propose plusieurs moyens de manipuler des ensembles d'objets : on a vu les tableaux dont l'inconvénient est de ne pas être dynamique vis à vis de leur taille. Java fournit des interfaces qui permettent de gérer des ensembles d'objets dans des structures qui peuvent être parcourues.
- Ce chapitre donne un aperçu de ces collections. Elles sont toutes génériques.
- Toutes les collections d'objets
  - sont dans le paquetage java.util
  - implémentent l'interface générique Collection
- L'interface Set< T > sert à implémenter les collections de type ensemble : les éléments n'y figurent qu'une fois et ne sont pas ordonnés.
- L'interface List< T > sert à implémenter les collections dont les éléments sont ordonnées et qui autorisent la répétition.

Interface Collection



Interface Collection: Méthodes

- boolean add(T e) ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- boolean addAll(Collection) ajoute à la collection tous les éléments de la collection fournie en paramètre
- void clear() supprime tous les éléments de la collection
- boolean contains(T e) indique si la collection contient au moins un élément identique à celui fourni en paramètre
- boolean containsAll(Collection) indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- boolean isEmpty() indique si la collection est vide
- Iterator iterator() renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Interface Collection: Méthodes

- boolean remove(T e) supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- boolean removeAll(Collection) supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- int size() renvoie le nombre d'éléments contenu dans la collection
- Object[] toArray() renvoie d'un tableau d'objets qui contient tous les éléments de la collection
- int hashCode()
- Remarque : en Java, chaque instance d'une classe a un hashCode fourni par la méthode hashCode() de la classe Object. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type int.

La Classe Collections

- La classe java.util.Collections contient des méthodes statiques qui opèrent sur des objets List, Set, Map etc.
  - void sort(List list) trie le paramètre list
  - void sort(List list, reverseOrder()) trie le paramètre list en ordre décroissant
  - Object max(Collection coll) renvoie le plus grand objet
  - Object min(Collection coll) renvoie le plus petit objet
- On peut utiliser ces méthodes statiques sur des objets de toutes les classes du schéma précédent.

Itérateurs

- Un itérateur est un objet utilisé avec une collection pour fournir un accès séquentiel aux éléments de cette collection.
- L'interface Iterator permet de fixer le comportement d'un itérateur.
- boolean hasNext() indique s'il reste au moins un élément à parcourir dans la collection
- T next() renvoie le prochain élément dans la collection
- void remove() supprime le dernier élément parcouru
- La méthode next() lève une exception de type NoSuchElementException si elle est appelée alors que la fin du parcours des éléments est atteinte.
- La méthode remove() lève une exception de type IllegalStateException si l'appel ne correspond à aucun appel à next(). Cette méthode est optionnelle (exception UnsupportedOperationException). On ne peut pas faire deux appels consécutifs à remove().

- A sa construction un itérateur doit être lié à une collection.
- A sa construction un itérateur se place tout au début de la collection.
- On ne peut pas ≪ réinitialiser ≫ un itérateur ; pour parcourir de nouveau la collection il faut créer un nouvel itérateur.
- Java utilise un itérateur pour implémenter la boucle for each

Méthode toString()

- Pour tout objet de type Collection, la méthode print (ou println) appelle itérativement la méthode toString() de chacun de ses éléments.

L’interface ListIterator<T>

- L’interface ListIterator<T> étend l’interface Iterator<T> et permet de parcourir la collection dans les deux sens.
- T previous() renvoie l’élément précédent dans la collection boolean hasPrevious() teste l’existence d’un élément précédent
- T next() renvoie l’élément suivant de la liste
- T previous() renvoie l’élément précédent de la liste
- int nextIndex() renvoie l’indice de l’élément qui sera renvoyé au prochain appel de next()
- int previousIndex() renvoie l’indice de l’élément qui sera renvoyé au prochain appel de previous()
- void add(T e) ajoute l’élément e à la liste en fin de liste
- void remove() supprime le dernier élément retourné par next() ou previous()
- void set(T e) remplace le dernier élément retourné par next() ou previous() par e

Interface ListIterator

- next() et previous() lèvent une exception de type NoSuchElementException si l'itérateur est en fin de liste alors nextIndex() renvoie la taille de la liste si l'itérateur est au début de la liste alors nextIndex() renvoie -1
- add(), remove() et set(T e) lèvent une exception de type IllegalStateException si l'appel ne correspond à aucun appel à next() ou previous(). Elles sont toutes les trois optionnelles.
- set(T e) lève une exception de type ClassCastException si le type de e ne convient pas.
- dans toutes les classes prédéfinies implémentant Iterator ou ListIterator, les méthodes next() et previous() renvoient les références des objets de la collection.

Classe ArrayList<T>

- Un ArrayList est un tableau d'objets dont la taille est dynamique.
- La classe ArrayList<T> implémente en particulier les interfaces Iterator, ListIterator et List.

Constructeurs

- public ArrayList(int initialCapacite) crée un arrayList vide avec la capacité initialCapacite (positif)
- public ArrayList() crée un arrayList vide avec la capacité 10
- public ArrayList(Collection<? extends T> c) crée un arrayList contenant tous les éléments de la collection c dans le même ordre avec une dimension correspondant à la taille réelle de c et non sa capacité ; le arrayList créé contient les références aux éléments de c.

Méthodes

- add et addAll ajoute à la fin du tableau
- void add(int index, T element) ajoute au tableau le paramètre element à l'indice index en décalant d'un rang vers la droite les éléments du tableau d'indice supérieur
- void ensureCapacity(int k) permet d'augmenter la capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre
- T get(int index) renvoie l'élément du tableau dont la position est précisée
- T set(int index, T element) renvoie l'élément à la position index et remplace sa valeur par celle du paramètre element

Méthodes

- int indexOf(Object o) renvoie la position de la première occurrence de l'élément fourni en paramètre
- int lastIndexOf(Object o) renvoie la position de la dernière occurrence de l'élément fourni en paramètre
- T remove(int index) renvoie l'élément du tableau à l'indice index et le supprime décalant d'un rang vers la gauche les éléments d'indice supérieur
- void removeRange(int j,int k) supprime tous les éléments du tableau de la position j incluse jusqu'à la position k exclue
- void trimToSize() ajuste la capacité du tableau sur sa taille actuelle

HashSet<T>

- La classe HashSet<T> implémente l'interface Set<T> et l'interface Iterator<T>.
- Elle permet de représenter un ensemble ; les éléments ne sont pas ordonnés par leur ordre d'insertion et chaque élément sera en un seul exemplaire : par conséquent la méthode boolean add(T e) n'ajoutera pas l'élément e s'il est déjà contenu dans le HashSet. Pour cela l'implémentation d'un HashSet s'appuie sur une table de hachage et sur les méthodes equals et hashCode de T.

Table de Hachage

- Une table de hachage est un tableau indexé par des entiers (en général) contenant les couples clef-valeur. L'indexation est réalisée par une fonction de hachage qui associe un indice du tableau à chaque clef.

hashCode()

- En Java, chaque instance d'une classe a un hashCode fourni par la méthode hashCode() de la classe Object. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type int.
- Dans certaines architectures, l'espace d'adressage est beaucoup plus grand que l'intervalle de valeur du type int ; il est donc possible que deux objets distincts aient le même hashCode. Donc on constate en pratique que cette fonction de hachage n'est pas injective.
- Dans le cas où l'on réécrit la méthode hashCode(), on peut toujours revenir à la valeur initiale du hashCode de la classe Object en utilisant la méthode statique System.identityHashCode(Object o).

Hashcode()

- Afin d'utiliser les instances d'une classe comme éléments d'une collection basée sur des tables de hachage (HashSet, HashMap, ...) il est indispensable de réécrire la méthode hashCode() de façon que :
  - elle soit compatible avec equals()
  - elle soit rapide
  - elle produise le même résultat pour un objet quelque soit le moment de l'appel
- ATTENTION : si on change la valeur d'un objet et si son hashCode réécrit à cause de equals est alors modifié on ne pourra pas le retrouver dans la table grâce à son code.

Méthodes

- Un HashSet peut contenir null. Les méthodes add, remove, contains, size sont exécutées en temps O(1).
- boolean add(T e) si e n'appartient pas à l'instance de HashSet alors e est ajouté et true est renvoyé ; sinon l'instance n'est pas modifiée et false est renvoyé.
- boolean addAll(Collection<? extends T> c) fait l'union de la collection c avec l'instance de HashSet ; si cette instance est modifiée true est renvoyé sinon false est renvoyé.
- boolean remove(Object e) si e appartient à l'instance de HashSet alors e est supprimé et true est renvoyé ; sinon false est renvoyé.
- boolean removeAll(Collection<? extends T> c) se comporte de même
- boolean retainAll(Collection<?> c) fait l'intersection de la collection c avec l'instance de HashSet ; si cette instance est modifiée true est renvoyé sinon false est renvoyé.
- void clear() supprime tous les éléments de la collection

TreeSet<T>

- La classe TreeSet<T> implémente l'interface Set<T> et l'interface SortedSet<T>.
- Un TreeSet s'appuie sur un arbre (rouge-noir) pour représenter un ensemble d'objets triés par ordre croissant (ordre naturel ou précisé par la méthode compareTo).
- Attention : la classe des éléments d'un TreeSet doit redéfinir la méthode compareTo et la equals de façon cohérente (equals vrai doit être équivalent à compareTo vaut 0).
- Les méthodes add, remove, contains, size sont exécutées en temps O(log n).

Méthodes

- Outre les méthodes similaires à celles de HashSet on peut citer :
- public SortedSet<T> subSet(T fromElement, T toElement) renvoie le sous-ensemble des éléments compris entre fromElement inclus jusqu'à toElement exclu
- public SortedSet<E> headSet(T toElement) renvoie le sous-ensemble des éléments strictement inférieurs à toElement
- public SortedSet<E> tailSet(E fromElement) renvoie le sous-ensemble des éléments supérieurs ou égal à fromElement
- public T first() renvoie le plus petit élément
- public T last() renvoie le plus grand élément
- public T floor(T e) renvoie le plus grand élément inférieur ou égal à e
- public T ceiling(T e) renvoie le plus petit élément supérieur ou égal à e

L’interface Map

- Un objet de type Map stocke des couples clef-valeur. On parle aussi de dictionnaire.
- La clef doit être unique mais une valeur peut avoir plusieurs clefs.
- L’interface Map<K,V> fixe les méthodes pour manipuler de tels couples. Les clefs sont de type K et sont associées aux valeurs de type V.
  - Toutes les collections de couples clef-valeur sont dans le paquetage java.util
  - implémentent l’interface générique Map<K,V>

Méthodes

- V put(K clef, V valeur) associe valeur à clef. Si clef est déjà associée à un objet V alors il est remplacé par le paramètre valeur puis il est renvoyé (null est renvoyé si clef était associée à rien)
- V get(Object clef) renvoie la valeur associée à clef si elle existe ou null sinon.
- boolean containsKey(Object clef) indique si un élément est associée au paramètre clef
- boolean containsValue(Object valeur) indique si le paramètre valeur est associée à au moins une clef
- boolean isEmpty() indique si la collection est vide
- Set<K> keySet() renvoie un ensemble constitué des clefs de l'objet
- Collection<V> values() renvoie la collection constituée des valeurs de l'objet
- Set<Map.Entry<K,V> > entrySet() renvoie un ensemble constitué des couples (clefs,valeurs)

Recommandation sur la classe K

- Pour toutes les classes ci-dessous il est fortement recommandé d'utiliser des objets immuables pour les clefs.
- Un objet immuable est un objet que l'on ne peut pas modifier une fois qu'il est créé.
- Pour rendre immuables les instances d'une classe il faut :
  - la marquer final pour qu'elle n'ait pas de classe fille
  - marquer private les attributs
  - marquer final les attributs (conseillé)
  - ne pas écrire d'accesseurs en écriture (set..)
  - écrire des accesseurs en lecture (get..) qui renvoient une nouvelle instance de l'attribut lu ou bien un objet immuable
  - ne pas implémenter Cloneable
  - Les classes enveloppantes (Byte, Short, Integer, Long, Float, Double, Character, Boolean) et la classe String sont immuables.

Les implémentations

- La classe HashMap<K,V> implémente l'interface Map<K,V> avec une table de hachage. null peut être une clef.
- La classe Hashtable<K,V> implémente l'interface Map<K,V> avec une table de hachage. null ne peut pas être une clef.
- La classe TreeMap<K,V> implémente l'interface Map<K,V> avec un arbre rouge noir pour un ordre naturel sur les clefs ou bien avec un Comparator sur les clefs. Cette dernière classe implémente l'interface SortedMap et a des méthodes spécifiques s'appuyant sur l'ordre des clefs.

Itération sur Map

- On peut utiliser plusieurs méthodes :
  - utiliser une boucle foreach sur l'ensemble des clefs obtenu par la méthode keySet()
  - utiliser un itérateur sur l'ensemble des clefs obtenu par la méthode keySet()
  - utiliser une boucle foreach sur l'ensemble des couples (clefs,valeurs) obtenu par la méthode entrySet()
  - utiliser un itérateur sur l'ensemble des couples (clefs,valeurs) obtenu par la méthode entrySet()

Références