Collections et généricité

Pratique de la programmation orientée-objet
Michel Schinz – 2014-03-03

Collection

On appelle **collection**, ou **structure de données abstraite** (*abstract data structure*), un objet servant de conteneur à d'autres objets. Exemples :

- les tableaux,
- les listes et leurs variantes (piles, queues),
- les ensembles,
- les tables associatives,
- etc.

Chaque genre de collection a ses caractéristiques propres, ses forces et ses faiblesses. Le choix de la collection à utiliser dans un cas particulier dépend donc de ce qu'on veut en faire.

Collection : exemples

Vous connaissez déjà deux types de collections : les tableaux et les piles (d'entiers).
Un tableau a une taille fixe et permet l'accès à n'importe quel élément – étant donnée sa position – en temps constant, c-à-d en O(1).
Une pile d'entiers, par contre, a une taille variable mais seul un élément – celui se trouvant au sommet – est accessible en O(1).

Limitation des piles

Une autre différence cruciale entre les tableaux et les listes d'entiers est qu'on peut définir des tableaux contenant n'importe quel type d'objets alors qu'on est limité aux entiers pour les piles…
Comment résoudre ce problème et généraliser les piles à des types d'éléments quelconques ?

Généricité

Pile d'entiers

On a vu précédemment comment écrire une interface très simple pour les piles d'entiers :

```
public interface IntStack {
  // Ajoute l'élément au sommet de la pile.
  public void addFirst(int value);

  // Enlève et retourne l'élément au sommet
  // de la pile. Lève IllegalStateException
  // si la pile est vide.
  public int removeFirst();

  // Retourne vrai ssi la pile est vide.
  public boolean isEmpty();
}
```


Piles arbitraires

L'interface `IntStack` et les classes qui la mettent en œuvre ne peuvent représenter que des piles d'entiers.
Comment faire pour représenter des piles d'un type arbitraire ? Idées :

- écrire autant d'interfaces et de classes qu'il existe de types (spécialisation), ou
- faire des piles de `Object`, ou
- utiliser la généricité.

Spécialisation

La spécialisation consiste à écrire une interface par type d'éléments que la pile peut contenir. Cela devient vite irréaliste…

```
interface IntStack {
  void addFirst(int value); … }
interface BooleanStack {
  void addFirst(boolean value); … }
interface StringStack {
  void addFirst(String value); … }
interface IntStackStack {
  void addFirst(IntStack value); … }
interface ObjectStack {
  void addFirst(Object value); … }
```

Piles de type `Object`

Une solution plus réaliste que la spécialisation est l'utilisation du type `Object`, qui en Java est un super-type de tous les types, sauf les types de base :

```
interface Stack {
  public void addFirst(Object value);
  public Object removeFirst();
  public boolean isEmpty();
}
```

Cette solution était utilisée en Java avant l'introduction de la généricité.

Piles de type Object

Malheureusement, l'utilisation de telles piles implique une grande quantité de transtypages (*casts*) :

```
Stack s = new …;
s.addFirst("hello");
char c = ((String)l.removeFirst()).charAt(0);
```

et comporte des risques...

```
Stack s = new …;
s.addFirst(new Object());
char c = ((String)l.removeFirst()).charAt(0);
```


Généricité

En raison des problèmes posés par la spécialisation et la solution basée sur le type `Object`, la notion de **généricité** (*genericity*), aussi appelée **polymorphisme paramétrique** (*parametric polymorphism*) a été introduite dans la version 5 de Java.

Au moyen de la généricité, il est possible de définir des piles génériques, c'est-à-dire capables de contenir des éléments d'un type arbitraire.

Pile générique

Une interface pour les piles de valeurs d'un type arbitraire peut se définir ainsi :

```
interface Stack<E> {
  public void addFirst(E value);
  public E removeFirst();
  public boolean isEmpty();
}
```

Cette interface est **générique**, et E est un **paramètre de type** de cette interface. Il s'agit d'une variable (de type !) représentant le type des éléments de la pile.

Instanciation

Pour utiliser un type générique comme `Stack`, il faut spécifier le type concret à utiliser pour le paramètre de type.
Exemples :

- `Stack<Object>`
- `Stack<String>`
- `Stack<Stack<String>>`

Tous ces types sont des **instanciations** du type générique `Stack`.

Utilisation

Contrairement aux piles basées sur `Object`, les piles génériques n'impliquent aucun transtypage :

```
Stack<String> s = new …;
s.addFirst("hello");
char c = s.removeFirst().charAt(0);
```

et ne comportent pas les mêmes risques :

```
Stack<String> s = new …;
s.addFirst(new Object());
char c = s.removeFirst().charAt(0);
```

interdit !

Piles chaînées génériques

La classe des piles chaînées de valeurs arbitraires doit également être générique, de même que celle des nœuds :

```
public final class LinkedStack<E>
    implements Stack<E> {
  private Node<E> first = null;
  // … isEmpty, addFirst, removeFirst
  private final static class Node<E> {
    public final E value;
    public final Node<E> next;
    public Node(E value, Node<E> next) {
      this.value = value;
      this.next = next;
    }
  }
}
```


Exercice

Donnez les définitions des méthodes `isEmpty`, `addFirst` et `removeFirst`.

Utilisation

Une fois la classe des piles chaînées génériques définie, on peut p.ex. l'utiliser pour écrire un programme imprimant une chaîne en sens inverse :

```
String p = "Esope reste ici et se repose";
Stack<Character> s =
  new LinkedStack<Character>();
for (int i = 0; i < p.length(); ++i)
    s.addFirst(p.charAt(i));
while (!s.isEmpty())
    System.out.print(s.removeFirst());
System.out.println();
```


Inférence de types

Depuis la version 7 de Java, il n'est pas nécessaire de spécifier explicitement les paramètres de type dans un énoncé `new`. Ceux-ci sont alors inférés (devinés) par Java. On utilise pour cela les crochets vides `<>`, aussi appelés le diamant (*diamond*).
La création de la pile du programme précédent peut donc se récrire ainsi :

```
Stack<Character> s = new LinkedStack<>();
```

Paramètres multiples

Un type générique peut bien entendu avoir plusieurs paramètres de type. Par exemple, une classe modélisant les paires de valeurs quelconques pourrait être définie ainsi :

```
public final class Pair<F, S> {
  private final F fst;
  private final S snd;

  public Pair(F fst, S snd) {
    this.fst = fst;
    this.snd = snd;
  }
  public F fst() { return fst; }
  public S snd() { return snd; }
}
```


Méthodes génériques

Il n'y a pas que les classes et les interfaces qui puissent être génériques, les méthodes le peuvent aussi. Exemple :

```
public final class Stacks {
  static <E> Stack<E> newLinkedStack(E e) {
    Stack<E> s = new LinkedStack<>();
    s.addFirst(e);
    return s;
  }
}
```

Ici, `E` est le paramètre de type de la méthode générique `newLinkedStack`. Il est déclaré *avant* le type de retour de la méthode, car ce dernier peut y faire référence – comme c'est le cas ici.

Inférence des types (bis)

Lorsqu'on appelle une méthode générique, il n'est généralement pas nécessaire de spécifier explicitement les arguments de type car ils sont **inférés** (c-à-d devinés).
Par exemple, dans l'appel ci-dessous, l'information que E vaut `String` est déterminée automatiquement en fonction de la valeur passée :

```
Stacks.newLinkedStack("hello")
```

mais il est également possible – et parfois nécessaire – de le spécifier explicitement, au moyen de la syntaxe suivante :

```
Stacks.<String>newLinkedStack("hello")
```

Attention : le paramètre de type est placé *après* le point !

Généricité et types de base

Types de base

Rappel : Java possède 8 types dits *de base* qui ne sont pas des objets (`boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, `double`).
Malheureusement, les types de base ne peuvent pas être utilisés comme paramètres de type d'un type générique.
Dès lors, le code suivant est erroné :

```
Stack<int> s = …; // interdit !
s.add(2);
s.add(3);
```

Que faire si l'on désire créer une pile d'entiers ?

Emballage

Solution : stocker chaque valeur de type `int` dans un objet de type `java.lang.Integer`, et créer une pile de valeurs de ce type. L'exemple devient :

```
Stack<Integer> s = …;
s.add(new Integer(2));
s.add(new Integer(3));
```

On dit alors que les entiers ont été **emballés** (*wrapped* ou *boxed*) dans des objets de type `Integer`.
Le paquetage `java.lang` contient une classe d'emballage par type de base (`Boolean` pour `boolean`, `Character` pour `char`, `Double` pour `double`, etc.)

Déballage

Bien entendu, lorsqu'on ressort les valeurs emballées de la liste, il faut les déballer (*unwrap* ou *unbox*) avant de pouvoir les utiliser.

Dans le cas des entiers, cela se fait au moyen de la méthode `intValue` de la classe `Integer` :

```
Stack<Integer> s = …;
…
int sum = s.removeFirst().intValue() +
  s.removeFirst().intValue();
```


Emballage automatique

L'emballage et le déballage manuels étant lourds à l'usage, le code nécessaire peut être produit automatiquement. On appelle cela l'emballage – et le déballage – automatique (*autoboxing*).

L'exemple précédent peut donc également s'écrire ainsi :

```
Stack<Integer> s = …;
s.addFirst(2);
s.addFirst(3);
int sum = s.removeFirst() + s.removeFirst();
```

et est automatiquement transformé afin que les entiers soient emballés avant d'être passés à `addFirst` puis déballés avant l'addition.

Limitations de la généricité en Java

Limitations de la généricité

Pour des raisons historiques, la généricité en Java possède les limitations suivantes :

- la création de tableaux dont les éléments ont un type générique est interdite,
- les tests d'instance impliquant des types génériques sont interdits,
- les transtypages (*casts*) sur des types génériques ne sont pas sûrs, c-à-d qu'ils produisent un avertissement lors de la compilation et un résultat éventuellement incorrect à l'exécution,
- la définition d'exceptions génériques est interdite.

Généricité et tableaux

Limitation n°1 : la création de tableaux dont les éléments ont un type générique est interdite.
Par exemple, le code suivant est refusé :

```
static <T> T[] newArray(T x) {
  return new T[]{ x };
}
```

interdit !

Test d'instance générique

Limitation n°2 : les tests d'instance impliquant des types génériques sont interdits.
Par exemple, le code suivant est refusé :

```
<T> void clearIntStack(Stack<T> s) {
  if (s instanceof Stack<Integer>)
    while (!s.isEmpty())
      s.removeFirst();
}
```

interdit !

Transtypage générique

Limitation n°3 : les transtypages impliquant des types génériques ne sont pas sûrs.
Par exemple, le code suivant ne lève pas d'exception à l'exécution, alors qu'il devrait en lever une :

```
Stack<Integer> s = new …;
Object o = s;
Stack<String> s2 = (Stack<String>)o;
```

aucune exception levée

Un avertissement est toutefois produit.

Exceptions génériques

Limitation n°4 : une classe définissant une exception ne peut pas être générique. En d'autres termes, aucune sous-classes de `Throwable` ne peut avoir de paramètres de type.
Par exemple, la définition suivante est refusée :

```
class BadException<T> extends Exception {}
```

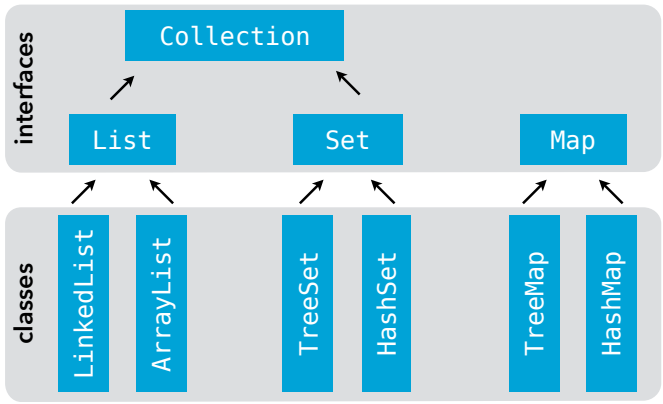
Collections génériques de Java

Collections de l'API Java

L'API Java fournit un certain nombre de collections dans le *Java collections framework*. Tout son contenu se trouve dans le (très mal nommé) paquetage `java.util`.
Pour chaque collection existe :

- une interface,
- une ou plusieurs mises en œuvre, sous la forme de classes.

Collections de l'API Java



(vue très partielle)

Listes

Une **liste** (*list*) est une collection ordonnée d'éléments, de taille variable.
Dans l'API Java, l'interface générique `List<E>` décrit les listes d'éléments de type `E`. Parmi les mises en œuvre de cette interface, on peut citer :

- `ArrayList<E>`, qui stocke les éléments dans un tableau qui est « redimensionné » par copie au besoin,
- `LinkedList<E>`, qui stocke les éléments dans des nœuds chaînés entre-eux.

Interface `List`

Version très simplifiée de l'interface `java.util.List<E>`, où E est le type des éléments de la liste :

```
public interface List<E> {
  boolean isEmpty(); // Vrai ssi vide.
  int size();        // Taille de la liste.
  // Ajoute un élément en fin de liste
  void add(E newElem);
  // Supprime l'élément d'index donné.
  void remove(int index);
  // Retourne l'élément d'index donné.
  E get(int index);
  // Modifie l'élément d'index donné.
  void set(int index, E elem);
}
```


Cas d'utilisation

Au vu de leur forte ressemblance avec les tableaux, les listes sont souvent utilisées dans des situations similaires, surtout lorsque la taille fixe des tableaux est handicapante.
On peut ainsi imaginer stocker les étudiants inscrits à un cours dans une liste, éventuellement ordonnée selon l'ordre alphabétique des noms de famille, ou encore les messages électroniques reçus par un utilisateur, ordonnés chronologiquement.

Exemple de code

```
import java.util.List;
import java.util.ArrayList;

List<String> workingDays = new ArrayList<>();
workingDays.add("lundi");
workingDays.add("mardi");
workingDays.add("mercredi");
workingDays.add("jeudi");
workingDays.add("vendredi");

System.out.println("nb. de jours ouvrés : " +
                   workingDays.size());
System.out.println("2e jour ouvré : " +
                   workingDays.get(1));
```


Listes et boucle *for-each*

Tout comme les éléments d'un tableau, ceux d'une liste peuvent être parcourus (dans l'ordre) au moyen de la boucle *for-each*. Exemple :

```
List<Integer> somePrimes = …;
for (int prime: somePrimes)
  System.out.println(prime);
```

Ensembles

Un **ensemble** (*set*) est une collection non ordonnée d'éléments tous différents, c-à-d que les duplicatas ne sont pas admis – comme dans les ensembles mathématiques. Dans l'API Java, l'interface générique `Set<E>` décrit les ensembles d'éléments de type `E`. Parmi les mises en œuvre de cette interface, on peut citer :

- `HashSet<E>`, qui organise les éléments par la technique du hachage que nous étudierons plus tard,
- `TreeSet<E>`, qui organise les éléments dans un arbre binaire, en fonction d'un ordre, selon une technique que nous verrons plus tard.

Interface Set

Version très simplifiée de l'interface `java.util.Set<E>`, où `E` est le type des éléments de l'ensemble :

```
public interface Set<E> {
  boolean isEmpty();// Vrai ssi vide.
  int size();       // Taille de l'ensemble.
  // Ajoute l'élément à l'ensemble si pas
  // présent.
  void add(E newElem);
  // Supprime l'élément de l'ensemble.
  void remove(E elem);
  // Vrai ssi l'élément est présent.
  boolean contains(E elem);
}
```


Cas d'utilisation

Les ensembles sont utilisés lorsque les duplicatas sont indésirables, ou alors lorsqu'il doit être possible de tester rapidement la présence d'un élément dans la collection. On peut ainsi p.ex. imaginer utiliser un ensemble pour stocker tous les nombres premiers dans un certain intervalle afin de rapidement tester si un nombre est premier.

Exemple de code

```
Set<Integer> primes = new HashSet<>();
for (int i = 0; i <= 1000; ++i) {
  if (isPrime(i))
    primes.add(i);
}

System.out.println("Il y a "+ primes.size()
  +" nombres premiers entre 0 et 1000");
System.out.println("N.p. inférieurs à 20 :");
for (int i = 0; i < 20; ++i) {
  if (primes.contains(i))
    System.out.println(i)
}
```

Ensembles et boucle *for-each*

Tout comme les éléments d'un tableau ou d'une liste, ceux d'un ensemble peuvent être parcourus au moyen de la boucle *for-each*. Exemple :

```
Set<Integer> somePrimes = …;
for (int prime: somePrimes)
  System.out.println(prime);
```

Attention toutefois : les éléments d'un ensemble n'étant pas ordonnés, l'ordre de parcours est quelconque et peut même changer entre deux exécutions du programme !

Tables associatives

Une **table associative** ou **dictionnaire** (*map* ou *dictionary*) est une collection d'éléments indexés par des clefs de type arbitraire.

Dans l'API Java, l'interface générique `Map<K,V>` décrit les tables associant des valeurs de type `V` à des clefs de type `K`.

Parmi les mises en œuvre de cette interface, on peut citer :

- `HashMap<K,V>`, qui organise les clefs par la technique du hachage,
- `TreeMap<K,V>`, qui organise les clef dans un arbre binaire, en fonction d'un ordre.

Interface Map

Version très simplifiée de l'interface `java.util.Map<K,V>` où `K` est le type des clefs et `V` celui des valeurs :

```
public interface Map<K, V> {
  boolean isEmpty(); // Vrai ssi vide.
  int size();        // Taille de la table.
  // Associe la valeur à la clef.
  void put(K key, V value);
  // Supprime la clef et sa valeur.
  void remove(K key);
  // Retourne la valeur associée à la clef.
  V get(K key);
  // Vrai ssi la clef existe dans la table.
  boolean containsKey(K key);
}
```


Cas d'utilisation

Les tables associatives sont utilisées chaque fois que l'on désire associer des données à des clefs.

On peut ainsi imaginer utiliser une table associative pour représenter :

- une encyclopédie, associant une définition (valeur) à un mot (clef),
- un annuaire téléphonique, associant un numéro de téléphone (valeur) à un nom (clef),
- une fonction mathématique, associant une valeur du codomaine (valeur) à une valeur du domaine (clef).

A noter qu'un tableau peut être vu comme une table associative dont les valeurs sont les éléments du tableau et les clefs les index entiers.

Exemple de code

En faisant l'hypothèse qu'il existe une classe **PN** (*phone number*) modélisant les numéros de téléphone, on peut utiliser une table associative pour gérer un annuaire ainsi :

```
Map<String, PN> pBook = new HashMap<>();
pBook.put("Jean", new PN("078 123 45 69"));
pBook.put("Marie", new PN("079 157 78 89"));
// …
pBook.put("Ursule",new PN("026 688 87 98"));
System.out.println("Le numéro de Jean est "
                   + pBook.get("Jean"));
pBook.put("Marie", new PN("077 554 12 55"));
pBook.remove("Ursule");
```


Exercice

Quel(s) type(s) de collection – liste, ensemble ou table associative – serait-il raisonnable d'utiliser pour stocker les éléments suivants :

1. la date de naissance de scientifiques célèbres,
2. les messages échangés par les participants à une discussion en ligne (*chat*),
3. le nom de la capitale de chaque pays du monde,
4. le nom de toutes les substances dopantes interdites aux sportifs,
5. la totalité des nombres premiers entre 0 et 10000,
6. les « diapositives » d'un programme comme PowerPoint.