Les collections en Java

L. Nerima
Université de Genève

Références

- Programmer en Java, Claude Delannoy, 11e édition, Eyrolles, 2020, Chapitre 22 « Les collections et les algorithmes »
  - Disponible à la bibliothèque du CUI, Battelle Bât D
- Conception objet en Java avec BlueJ, David Barnes et Michael Kölling, 4e édition, Pearson Education, 2009
- Documentation de l'API Java SE 11, package *java.util* https://docs.oracle.com/en/java/javase/11/docs/api/java.base/module-summary.html

Plan

- Collections de données (définition)
- Les collections en Java
- Les interfaces racine *Collection* et *Map*
- Digression 1: les interfaces Java
- Digression 2: les classes génériques
- Les collections de données:
  1. Les tableaux dynamiques: la classe ArrayList
  2. Les listes: la classe LinkedList
  3. Les ensembles: la classe HashSet (TreeSet)
  4. Les fonctions (*map*): la classe HashMap

Collection de données

- Définition: « Une collection de données est un conteneur d'éléments de même type qui possède un protocole particulier pour l'ajout, le retrait et la recherche d'éléments »
- Exemples: pile, queue (file d'attente), séquence, ensemble et multi-ensemble, fonction (tableau associatif, dictionnaire ou *map* en anglais)
- Ordre dans les collections:
  - Les ensembles et les multi-ensembles n'ont pas d'ordre
  - Les autres collections ont un ordre « naturel » lié à l'ordre dans lequel les éléments ont été insérés dans la collection
- Les classes collection sont définies dans le package *java.util*
- Définies à partir de deux *Interfaces* Java
  - *Collection*
  - *Map*

Les collections en Java - *Collection*

- Définies à partir de la racine Interface *Collection <E>*
  ou E est le type des éléments de la collection
- Les classes *collection* (qui implémentent l'interface *Collection*) sont nombreuses dans l'API Java:
  AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector
- Nous verrons les classes collection suivantes
  - ArrayList (Vector)
  - LinkedList
  - HashSet (TreeSet)

Les collections en Java – *Map (fonction)*

- Les collections de type *fonction* (*map*), *tableau associatif* ou *dictionnaire* en Java, sont définies à partir de la racine Interface *Map <K, V>* (et non *Collection <E>*)
- La raison est qu'une telle collection est un ensemble de **paires** d'objets, chaque paire associant un objet de l'ensemble de départ *K* à un objet de l'ensemble d'arrivée *V* ; on parle de **paires** (clé, valeur)
- Application: chaque fois qu'il faut retrouver une valeur en fonction d'une clé, p.e. dans un dictionnaire: mot -> définition du mot; dans un annuaire: nom de personne -> adresse et n° de tél; localisation: avion -> aéroport, etc.
- Les classes *Map*: AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap
- Nous verrons la classe *Map* suivante
  - HashMap

Digression n° 1: les Interfaces Java

- Ressemblent aux classes abstraites mais…
- Toutes les méthodes sont vides (non implémentées), seule la *signature* des méthodes est définie

  signature = entête de la méthode

  c-à-d le **nom** de la méthode et la liste des **paramètres formels**
- Aucun champ n'est défini dans une interface, hormis les constantes

- Côté client (classes implémentant une interface):
  - Les classes pourront implémenter plusieurs Interfaces (~ proche de l'héritage multiple)
  - Toutes les méthodes de l'interface devront être implémentées

Définition et implémentation d'une Interface Java

- Définition

```
public interface I1
{ void m(int i);  // signature de la méthode m
}

public interface I2
  char n();       // signature de la méthode n
}
```

- Implémentation de(s) l'interface(s)

```
class A implements I1, I2
{ // implémentation obligatoire des méthodes m et n de I1 et I2
}
```

Digression n° 2: la généricité en Java

- A partir de JDK 5.0, les collections sont définies par le biais de classes génériques
- Une classe générique est une classe qui définit ses méthodes de manipulation de structure de données sans préciser le type de ses éléments. Idée: un ensemble de X, une liste d'Y, …
- Par exemple, on pourrait définir une *classe Pile<E>* qui définit les méthodes *void empile(E e), E sommet(), void depile()*
- Le type *E* est définit un moment de la déclaration d'un objet de la classe *Pile*, p.e.
  - *Pile<int> p1* // une pile d'entiers
  - *Pile<String> p2* // une pile de chaîne de caractères
  - Instanciation
  - *p1 = new Pile<int>;*
  - *p2 = new Pile<String>;*

Pour chacune des 4 collections, nous étudierons

- les caractéristiques de la collection
- la déclaration / construction (initialisation) de la collection
- les opérations usuelles sur les éléments
  - *ajout*
  - *accès*
  - *suppression*
- la complexité en temps de chacune de ces opérations
  complexité en temps = efficacité de l'algorithme
- le parcours de la collection (visite de tous ses éléments)

1. Les tableaux dynamiques: ArrayList

- Tableaux dynamiques (anciennement Vector, classe qui existe toujours)
- Dynamique = la taille (nombres d'éléments) du tableau n'est pas fixe et peut varier en cours d'exécution
- L'accès à ses éléments est direct, comme dans un tableau (complexité en temps *O(1)*)
- L'opération d'ajout et de suppression est en *O(N)*, car nécessitant éventuellement un réarrangement des éléments pour qu'ils soient toujours contigus (comme dans un tableau)

ArrayList - construction et méthodes usuelles

- **Déclaration / construction**
  ArrayList <E> v1 = new ArrayList <E> (); // vecteur dynamique vide
  ou
  ArrayList <E> v2 = new ArrayList <E> (c); /* vecteur dynamique contenant tous les éléments de la collection c */
- **Ajout** d'un élément en fin de vecteur:
  v1.add(elem);
- **Accès** au ième élément
  e = v1.get(3); // accès au 3ème élément du vecteur v1
- **Suppression** du ième élément du vecteur (avec retour dans e)
  E e = v1.remove(3); // suppression du 3ème élément

ArrayList - parcours

- Le parcours de la liste se fait avec la boucle for
  for (E e : v)
- Exemple: afficher tous les éléments d'une liste

```
public static void affiche (ArrayList <E> v) {
    for (E e : v) System.out.print ( e + « » );
    System.out.println();
}
```

2. Les listes: LinkedList

- Listes doublement chaînées
- La liste peut être parcourue par un itérateur bidirectionnel *ListIterator*
- Ajout et supression d'un élément à une position donnée: complexité *O(1)*
- Accès d'un élément en fonction de sa valeur: complexité *O(N)* car nécessite le parcours de la liste
- Utilisation
  - la classe LinkedList se prête bien à l'implémentation des collections ordonnées, c'est-à-dire
    - pile
    - queue (file d'attente)
    - séquence

LinkedList - construction et méthodes usuelles

- **Déclaration / construction**
  LinkedList<E> l1 = new LinkedList<E> (); // liste vide
  ou
  LinkedList<E> l2 = new LinkedList<E> (c); /* liste contenant tous les éléments de la collection c */
- **Ajout** d'un élément E e au début / fin de la liste
  l1.addFirst(elem); l1.addLast(elem);
- **Accès** au premier / dernier élément de la liste
  E e = l1.getFirst(); e = l1.getLast();
- **Suppression** du premier / $i^{ème}$ / dernier élément
  e = l1.removeFirst(); e = l1.remove(i); e = l1.removeLast()

LinkedList - parcours

- Le parcours d'une LinkedList se fait avec un itérateur *ListIterator*
- Notion de **position courante** dans la liste avec l'itérateur de liste
  ListIterator <E> iter = l1.listIterator() //iter désigne le début de la liste
- Avancer / reculer d'un élément et retourner l'élément
  e = iter.next();
  e = iter.previous();
- **Ajout** d'un élément à la position courante
  iter.add(elem); // si fin de liste: ajout à la fin
- **Suppression** de l'élément à la position courante
  iter.remove(); // suppres. dernier élément retourné par *next* ou *previous*
- **Parcours**: exemple, afficher tous les éléments de la liste

```
ListIterator <E> iter = l1.listIterator();
while (iter.hasNext()) {
    E elem = iter.next();
    System.out.println(elem);
}
```

3. Les ensembles: HashSet (et TreeSet)

- Un ensemble est une collection non ordonnée d'éléments de type *E*, aucun élément ne peut apparaître plus d'une fois dans un ensemble
- Problème: comme deux objets distincts ont des références différentes, on ne pourra jamais avoir deux objets égaux même si toutes leurs valeurs sont identiques -> Il faudra définir un comparateur qui sera capable de tester l'égalité de deux objets (*equals* et *compareTo*)
- Même s'il n'existe pas d'ordre dans un ensemble, l'implémentation informatique s'appuie sur une organisation des éléments afin de garantir un accès efficace. Au lieu de *O(N)* nous aurons
  - HashSet -> *O(1)* (implémentation avec une table hachage)
  - TreeSet -> O(log N) (implémentation avec un arbre de recherche binaire)
- L'utilisateur devra définir, pour l'utilisation d'un
  - HashSet -> les méthodes *hashCode* et *equals* dans la classe des éléments *E*
  - TreeSet -> la méthode *compareTo* dans la classe *E*

HashSet - construction et méthodes usuelles

- **Déclaration / construction**
  HashSet<E> e1 = new HashSet<E> (); // ensemble vide
  ou
  HashSet<E> e2 = new HashSet<E> (c); /* ensemble contenant tous les éléments de la collection c */
- **add** – ajout d'un élément s'il n'appartient pas encore à l'ensemble (sinon état de l'ensemble inchangé)
  HashSet<E> e = new HashSet<E> ();
  E elem = new E();
  boolean nouveau = e.add(elem); // true si elem a été ajouté
  if (nouveau) System.out.println(elem + « ajouté à l'ensemble »);
  else System.out.println(elem + « existait déjà dans l'ensemble »);
- **contains** – test d'appartenance
  boolean appartient = e.contains(elem); // *elem* appartient-il à *e* ?
- **remove** – suppression d'un élément
  boolean trouve = e.remove(elem); //false si elem n'appartient pas à *e*

HashSet - parcours

- **Parcours** à l'aide d'un itérateur

```
HashSet<E> e = new HashSet<E> ();
… // ajouts d'éléments à l'ensemble
Iterator <E> iter = e.iterator();
while (iter.hasNext()) {
    E elem = iter.next();
    System.out.println(elem);
}
```

- Remarques
  - Les éléments d'un ensemble n'étant pas ordonnées, aucun ordre d'itération n'est assuré
  - L'ordre d'itération peut varier dans le temps

4. Les fonctions (*map*): HashMap

- ***Fonction*** (*Map*) = ensemble de paires (*clé*, *valeur*)
- Notion proche de la *fonction* au sens mathématique
- En informatique, la ***fonction*** est aussi appelée ***tableau associatif*** ou encore ***dictionnaire***
- Rappel: en Java, les collections de type *fonction*, sont définies à partir de la racine Interface *Map <K, V>* (et non *Collection <E>*)
- Accès rapide *O(1)* à une valeur en fonction d'une clé
- Techniquement réalisé par une table de hachage sur le domaine des clés
- Tout comme pour un *HashSet*, l'utilisateur doit définir les méthodes *hashCode* et *equals* dans la classe des clés *K*
- remarque: si *K* est *String*, *hashCode* et *equals* sont déjà définies dans la classe *String* -> l'utilisateur n'a pas besoin de les définir ☺

HashMap - construction et méthodes usuelles

- **Déclaration / construction**
  HashMap <K, V> m = new HashMap <K, V> (); // map vide
- Ex: HashMap <String,Integer> m = new HashMap <String,Integer>();
- **put** - ajout d'une paire (ou mise à jour de la valeur si la clé existe)
  m.**put**(cle, val); // où *cle* objet de K, *val* objet de V
- Cas particulier: *val* peut être de type primitif int, float, char,…(≠ objet)
  exemple: m.put("occident", **1**);
- **get** - accès la valeur associée à une clé. Exemple:
  String cle = "occident";
  Integer val = m.**get**(cle);
  if (val == null) System.out.println("cette clé n'existe pas ! ")
- **remove** - suppression d'une paire en fonction de la clé. Exemple:
  m.**remove**(cle); // p. ex. m.remove("occident")
  ou
  Integer val = m.**remove**(cle); // suppr. avec retour de la valeur ou *null*

HashMap - parcours

- En théorie une *map* ne dispose pas d'itérateur
- En pratique, on utilise la méthode *entrySet()* définie dans la classe HashMap pour créer un ensemble à partir du map (l'ensemble des paires de la map); puis on crée un itérateur sur cet ensemble
- HashMap <K, V> m = new HashMap <K, V> ();
  Set <Map.Entry<K,V>> paires = m.**entrySet**(); // ensemble de paires
  Iterator <Map.Entry<K,V>> iter = paires.iterator(); // itérateur
  while (iter.hasNext()) {
  Map.Entry paire = iter.next(); // paire courante
  System.out.println(paire); // affichage de la paire courante
  K cle = paire.getKey(); // accès à la clé
  V val = paire.getValue(); // accès à la valeur
  …
  }
- Alternative: construire l'ensemble des clés avec la méthode *keySet()*; itérer sur cet ensemble et accéder aux éléments de la *map* avec *get*

HashMap – un exemple complet

```
import java.util.*; // HashMap, Map, Iterator, Set
public class HashMapDemo{
    public static void main(String args[]) {

        /* Déclaration du HashMap persAge */
        HashMap<String, Integer> persAge = new HashMap<String, Integer>();

        /* Ajouter des entrées dans le HashMap persAge*/
        persAge.put("Mélisande", 28);
        persAge.put("Carine", 33);
        persAge.put("Paul", 45);

        /* Afficher le contenu de persAge en utilisant un itérateur */
        Set paires = persAge.entrySet(); // crée en ensemble de paires à partir du HashMap
        Iterator <Map.Entry<String,Integer>> iter = paires.iterator();
        while (iter.hasNext()) {
            Map.Entry paire = iter.next();
            System.out.println( paire.getKey() + " âge " + paire.getValue());
        }
        /* Retrouver une valeur en donnant la clé */
        System.out.println( "Carine a " + persAge.get( "Carine") + " ans");
        /* Mettre à jour la valeur associée à une clé */
        persAge.put("Paul", 40);
        System.out.println(  "Paul à rajeuni ! Nouvel âge: " + persAge.get("Paul"));
        /* Supprimer une entrée */
        persAge.remove("Paul"); // Paul is dead…
    }
} // code sur le site du séminaire Java :  http://cui.unige.ch/isi/cours/javalettres/HashMapDemo.zip
```