Conception et Programmation Objet - GM3

Collection, égalité et types de données

Mathieu Bourgais

2023

Plan

Principes des collections

En Java, le terme **collection** fait référence à des objets permettant de manipuler des groupes d'objets homogènes, groupes dont la taille n'est pas toujours connue.

- Les collections héritent d'interfaces indiquant leur comportement
- Plusieurs implémentations de ces interfaces sont disponibles, on peut donc choisir son niveau de complexité
- Possibilité d'utiliser des algorithmes génériques
- Gourmand en mémoire

Les 3 collections principales

En Java, on compte principalement 3 types de collection :

- **Set** : représente un ensemble non ordonné d'objets. Hérite de l'interface *Collection* qui hérite de l'interface *Iterable*
- **List** : représente un ensemble ordonné d'objets. Hérite de l'interface *Collection* qui hérite de l'interface *Iterable*
- **Map** : Représente un ensemble de couples <Clé,Objet>. Chaque objet est associé à une clé (par défaut, cette clé doit être unique) du type que l'on souhaite

Interface Set

Liste des méthodes de l'interface Set :

- int **size()** : retourne la taille de l'ensemble
- boolean **isEmpty()** : indique si l'ensemble est vide
- boolean **contains(Object e)** ;
- boolean **add(Object e)** ;
- boolean **remove(Object e)** ;
- Iterator<E> **iterator()** : renvoi un itérateur sur les éléments de l'ensemble
- void **clear()** : vide l'ensemble

On compte aussi des méthodes containsAll(Collection), addAll(Collection), etc.

Interface List

Liste des méthodes supplémentaires de l'interface List :

- E **get(int i)** : retourne l'élément à l'emplacement i
- E **set(int i, Object e)** : modifie l'objet se trouvant à l'emplacement i
- void **add(int i, Object e)** ;
- E **remove(int i)** : enlève l'élément à l'emplacement i
- int **indexOf(Objet e)** ;
- int **lastIndexOf(Object e)** ;
- ListIterator<E> **listIterator()** ;
- List<E> **subList(int from, int to)** ;

On retrouve aussi les méthodes de l'interface **Set**

Interface Map

Liste des méthodes de l'interface Map :

- V **put(Key k, Value v)** : ajoute le couple (k,v)
- V **get(Key k)** : retourne la valeur associée à la clé k
- V **remove(Key k)** : enlève le couple (k,v) quelque soit la valeur v
- boolean **containsKey(Key k)** ;
- boolean **containsValue(Value v)** ;
- Set<K> **keySet()** : retourne l'ensemble des clés de la table
- Collection<V> **values()** : retourne l'ensemble des valeurs de la table

On retrouve aussi les classiques size(), isEmpty(), etc.

Implémentation des collections

Pour être utilisées, ces interfaces doivent être implémentées dans une classe concrète. En Java, il existe de multiples implémentations pour chacune de ces interfaces.

	Table de hachage	Tableau dynamique	Arbre équilibré	Liste chaînée
Set	HashSet		TreeSet	
List		ArrayList		LinkedList
Map	HashMap		TreeMap	

On peut aussi proposer sa propre implémentation en définissant une classe qui implémente l'interface désirée.

Collections de types génériques

- Toutes les collections sont définies de façon générique ; Elles fonctionnent quelque soit le type des objets que vous allez y mettre.
- Par exemple, la classe ArrayList est en réalité définie comme **ArrayList<E>**.
- Lors de l'utilisation, vous devrez alors spécifier le type précis qui sera contenu dans la collection.

Définitions génériques

De façon générale, on peut créer des classes génériques qui seront spécialisées par la suite.

```
public class UnePaire<G,D> {
    private G valeurGauche;
    private D valeurDroite;

    public UnePaire(G gauche, D droite){
        this.valeurGauche = gauche;
        this.valeurDroite = droite;
    }

    public G getValeurGauche(){
        return this.valeurGauche;
    }

}
```

Exemple d'utilisation des collections

```
import java.util.ArrayList;

public ArrayList<Personnage> listePerso = new
    ArrayList<Personnage>();
public Monstre m = new Monstre();
public Magicien magie = new Magicien();

listePerso.add(m);
listePerso.add(magie);
```

On peut faire la même chose avec une **LinkedList**, un **HashSet** ou un **TreeSet**. Pour utiliser une **HashMap** ou une **TreeMap** il faut manipuler des couples (clé,valeur).
Il faut penser à importer la classe souhaitée depuis *java.util* avant de déclarer votre classe.

Itération sur collection

- Pour itérer sur une *List* , on peut simplement avancer un compteur dans une boucle
- On peut aussi utiliser la forme **for (Object o : maListe) {...}**
- Pour itérer sur un *Set* (qui n'est pas ordonné), on doit déclarer un **itérateur**
- Pour itérer sur une *Map*, on peut extraire les clés, ou les valeurs, ou les deux (**entrySet()**) sous la forme d'un ensemble, et ensuite utiliser un itérateur.
- Dans le cas des *Map*, il existe de nombreuses solutions.

Exemple d'itération sur collections

```
for (Personnage p : listePerso){
    System.out.println(p);
}
for (int i=0 ; i< listePerso.size() ; i++){
    System.out.println(listePerso.get(i));
}

public HashSet mesObjets;
for (Iterator i=mesObjets.iterator();i.hasNext();)
   {
    System.out.println(i.next().getNom);
}
```

Tableaux

En Java, les tableaux de valeurs sont des types complexes, de la classe **Array**.

```
int[] tableauEntier = {1,2,3};
float[] tableauFlottant = new float[taille];
tableauFlottant[x]=maValeur; // -1 < x < taille
```

La classe *Array* possède des méthodes comme **getLength()** qui retourne la taille d'un tableau

Chaînes de caractères

- Le type **String** est en réalité un type complexe, équivalent à un tableau de caractères (char [])
- La classe String propose de nombreuses méthodes pour manipuler les chaînes de caractères :
  - int indexOf(String s) ;
  - int length() ;
  - String[] split(String regex) ;
  - String substring(int begin,int end) ;
- De nombreuses méthodes sont à retrouver dans la documentation.

Plan

La méthode equals(Object o)

- Les test d'égalités pour les types simples se font avec ==
- Pour tester l'égalité entre deux types complexes (deux classes), on doit indiquer à la machine comment faire.
- On va alors redéfinir la méthode **equals(Object o)** de la classe *Object*.
- La méthode *equals(Object o)* est utilisée par défaut dans les méthodes *contains()* des collections par exemple.
- Il faut utiliser la méthode *equals(Object o)* pour tester l'égalité entre deux éléments *String*.

Exemple avec la classe Personnage

```
public boolean equals(Object o){
    if(!(o instanceOf Personnage)){
        return false
    }

    Personnage persoTemp = (Personnage) o;
    if (!persoTemp.getNom().equals(this.getNom()))
        {
        return false;
    }

    return true;
}
```

Principe de la méthode hashCode()

- La classe *Object* propose une méthode **int hashCode()** qui fonctionne de paire avec *equals(Object o)*.
- *hashCode()* renvoie une valeur entière qui définit une instance d'une classe.
- La seule règles est que deux instances égales (au sens de *equals()*) doivent renvoyer le même hashCode.
- Le hashCode est utilisé par certaines implémentations (*HashSet* et *HashMap* par exemple) pour optimiser les calculs.

Exemple avec la classe Personnage

```
public int hashCode(){
    return 13 + 3*this.nom.hashCode();
}
```

- Puisqu'on ne teste l'égalité que sur la valeur "nom", il n'y a que cette valeur qui rentre en compte dans le calcul du hashCode
- Puisque l'attribut "nom" est de la classe String, on utilise son hashcode (les attributs de types simples n'ont pas de méthode hashCode(), on doit les utiliser intelligemment)
- Les valeurs "13" et "3" sont complètement arbitraires

Plan

Modifications de l'exemple

- Méthodes equals() et hashCode() au niveau de Personnage
- Les PJ possèdent un inventaire sous la forme d'une liste d'objets. On leur ajoute une méthode pour présenter cet inventaire
- Création d'une classe Objet et ObjetMagique
- Création d'une classe Marchand qui possédera un ensemble d'objets à vendre
- Création d'un donjon qui contient une table associant des salles à une clé (le numéro des salles)

Code des PJ

```
private ArrayList<Objet> inventaire;

public PJ(){
    super();
    this.nomJoueur=" ";
    inventaire = new ArrayList<Objet>();
}

public void presenterInventaire(){
    for(Objet o : this.inventaire){
        System.out.println(o);
    }
}
```

Code du Marchand

```
private HashSet<Objet> objetsAVendre;

public Marchand(){
    super();
    this.objetsAVendre = new HashSet<Objet>();
}

public void presenterProduits(){
    for(Iterator<Objet> i=this.objetsAVendre.
        iterator();i.hasNext();){
        System.out.println(i.next());
    }
}
```

Le main sur le choix de la salle

```
Scanner scan = new Scanner(System.in);
System.out.println("Quelle salle voulez-vous
    explorer ?");
String couleur = scan.next();
System.out.println(hermione + " fait face a " +
    monDonjon.getMesSalles().get(couleur).getPerso
    ());

scan.close();
```

Résultat de l'éxécution

```
Bonjour, je m'appelle Bob
ObjetMagique [nom=anneau poids=0.2 effet=transparent]
Objet [nom=épée, poids=10.0]
Bonjour, je m'appelle Dudley Dursley et je suis joué par Aurore
ObjetMagique [nom=anneau poids=0.2 effet=transparent]
Objet [nom=épée, poids=10.0]
Quelle salle voulez-vous explorer ?
vert
nom=Hermione Granger fait face à nom=Docteur Quinn
```

L'ensemble des codes menant à ce résultat est à retrouver sur Moodle.