



IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

TP6 – Collections

Informatique – MAPD

Correction

Objectifs

— Choisir à bon escient le type de collection Java à utiliser dans un programme

La hiérarchie des collections

Exercice 1 (*Collections*)

Une des structures de données la plus couramment utilisée dans le développement logiciel est ce qu'on appelle une « *collection* ». Une collection en Java peut être une liste, un tableau, un arbre... Toutes les collections peuvent être *castées* sur le même type de base. En Java ce type est l'interface `Collection`. De plus, Java propose une autre structure de données appelée `Map`, qui n'étend pas l'interface `Collection` (pour la justification de ce choix, voir : <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/doc-files/coll-designfaq.html#a14>). Dans les exercices suivants, nous considérons `Map` comme un type de « *collection* ».

▷ Question 1.1 :

Explorez le framework de collections Java (c'est à dire, les hiérarchies `Collection` et `Map`). Par exemple, consultez <https://www.codejava.net/java-core/collections/overview-of-java-collections-framework-api-uml-diagram> et survolez les tutoriels proposés pour répondre aux questions suivantes :

1. Comment pouvez-vous créer un `NavigableSet` ?
2. Quelle/s est/sont la/les différence/s fondamentale/s entre un `Set` et une `List` ?
3. Combien de paramètres génériques dans une `Map` et pourquoi ?
4. Quelle/s est/sont la/les différence/s fondamentale/s entre `TreeMap` et `HashMap` ?

1. C'est une interface et il y a une seule implémentation donc créer une instance de `TreeSet`.

2. Dans une **List** il peut y avoir des doublons et les données sont insérées dans l'ordre d'insertion (l'index), dans un **Set** non.
3. Il y en a deux un pour la clé et un pour la valeur associée.
4. **TreeMap** est ordonnée par l'ordre naturel des valeurs des clés et ne permet pas d'avoir la valeur **null** pour les clés. Les **HashMap** ne sont pas ordonnées et permettent valeurs null et clé nulle. Mais ce sont toutes les deux des implémentations de **Map**.

	Set	List	Map	Queue
type	Any	Any	Key-Value	Any
ordonné	Non	Oui	Non	[F/L]IFO
index	N/A	int	Key	first/last
unicité	Oui	Non	de la clé	Non

Exercice 2 (Un stock de produits)

Dans cet exercice vous allez réaliser un stock qui gère un ensemble de produits. Le diagramme de classes UML simplifié est ci-dessous. Comme le montre le diagramme, la classe **Stock** gère un attribut (**contenue**) qui contient les produits du stock et la quantité disponible pour chacun (regroupés dans la classe **StockEntry**). Il a été décidé que, pour implanter cette collection, le type (apparent) utilisé doit être **Map** et que la clé de chaque élément sera la référence (**ref**) du produit dans le stock.



▷ Question 2.1 :

Map étant une interface de l'ensemble des collections Java, quelle est l'implémentation qu'il serait le plus judicieux d'utiliser pour les produits? Consultez <https://www.baeldung.com/java-treemap-vs-hashmap> pour vous aider.

Il y a 3 implémentations de **Map** : **HashMap**, **LinkedHashMap**, **TreeMap**. La seule qui ne permet pas d'avoir des clés à valeur **null** est la **TreeMap** mais dans cette implémentation les éléments (clé-valeur) sont stockés dans l'ordre des clés (dans notre cas ce serait l'ordre alphabétique). Couteux et pas nécessaire (voir <https://www.baeldung.com/java-treemap-vs-hashmap>). Parmi les deux autres implémentations, **LinkedHashMap** a aussi un ordre dans les éléments ... Donc, on choisit une **HashMap** et il faudra contrôler dans le programme qu'on n'ajoute aucun élément avec une clé **null**. Pour aller plus loin sur les hashcode et performance des hashtable : <https://www.baeldung.com/java-hashcode> et <https://dzone.com/articles/what-is-wrong-with-hashcode-in-javalangstring>.

▷ Question 2.2 :

Vous trouverez le squelette du projet Eclipse qui contient ces classes et interfaces sur Moodle et des tests unitaires (fichier MAPD-Sources-Collections.zip).

Votre travail consiste à compléter la classe **Stock** et **StockTest**. Pour la classe **StockTest**, vous devrez initialiser les attributs **items** et **quantities** en fonction du type de collection que vous avez choisi dans la question précédente.

- Vous pouvez choisir une implantation de **Map** (voir question précédente.)
- Ou un couple de **List**; l'une qui stocke les références (**ArrayList<String>** par exemple), et l'autre qui stocke **content** (par exemple **ArrayList<StockEntry>**). La référence et l'entrée associée (**StockEntry**) seront stockées au même index dans les 2 listes.

Pour la classe **Stock**, quelques méthodes vous sont fournies. Inspirez vous de celles-ci pour réaliser le code des méthodes manquantes. Commencez par le constructeur puis la méthode **getStock** et exécutez les tests fournis avant de passer à une autre méthode.

```
Version Map...  
package articles.implementation;  
  
import java.util.Map;  
import java.util.HashMap;  
import java.util.Iterator;  
import java.util.Map.Entry;  
  
import articles.interfaces.IItem;  
import articles.interfaces.IStock;  
import exceptions.InvalidItem;  
import exceptions.InvalidQuantity;  
import exceptions.InvalidStock;  
  
/**  
 * The Stock Class.
```

```

*
*/
public class Stock implements IStock {

    /** The contents */
    private Map<String, StockEntry> content;

    /**
     * Default constructor .
     */
    public Stock() {
        // TODO : initialize the stock attribute with the chosen collection
        // implementation
        content = new HashMap<String, StockEntry>();
    }

    @Override
    public Map<String, IItem> getStock() {
        // TODO
        Map<String, IItem> res = new HashMap<String, IItem>();
        for(String k:content.keySet())
            res.put(k, content.get(k).getItem());
        return res;
    }

    @Override
    public void setStock(Map<String, IItem> stock) throws InvalidStock {
        // TODO
        if (stock==null) throw new InvalidStock("stock null");
        for(String k:stock.keySet()) {
            if (k==null) throw new InvalidStock("key null");
            if (stock.get(k)==null ) throw new InvalidStock("value null");
            if (!content.containsKey(k)) {
                IItem i = stock.get(k);
                content.put(k, new StockEntry(i, 0));
            }
        }
    }

    @Override
    public Map<String, Integer> getQuantities() {
        // TODO
        Map<String, Integer> res = new HashMap<String, Integer>();
        for(String k:content.keySet())
            res.put(k, content.get(k).getQuantity());
        return res;
    }

    @Override
    public void setQuantities(Map<String, Integer> quantities) throws InvalidQuantity {

```

```

// TODO
if (quantities==null) throw new InvalidQuantity("map null");
for(String k:quantities.keySet()) {
    if (k==null) throw new InvalidQuantity("key null");
    if (quantities.get(k)==null ) throw new InvalidQuantity("value null");
    if (!content.containsKey(k)) { throw new InvalidQuantity("inexistent ref");}
    Integer i = quantities.get(k);
    if (i<0) throw new InvalidQuantity("negative value");
    content.replace(k, new StockEntry(content.get(k).getItem(), i));
}
}

@Override
public Integer getItemQuantity(String ref) throws InvalidItem {
    if (!content.containsKey(ref))
        throw new InvalidItem("The reference does not exist in the stock.");

    return this.content.get(ref).getQuantity();
}

@Override
public boolean enoughItemQuantity(String ref, int required) throws InvalidItem {
    return required <= getItemQuantity(ref);
}

@Override
public void add(String ref, IItem item, int qt) throws InvalidItem, InvalidQuantity {
    if ((item == null) || (ref == null))
        throw new InvalidItem("Invalid item; null.");
    if (qt <= 0)
        throw new InvalidQuantity("Quantity <= 0.");

    StockEntry entryToModify = null;
    if (!this.content.containsKey(ref)) {
        entryToModify = new StockEntry(item, qt);
    } else {
        entryToModify = this.content.get(ref);
        entryToModify.addQuantity(qt);
    }
    this.content.put(ref, entryToModify);
}

@Override
public void display() {
    System.out.println("Stock contents:");
    for(Entry<String, StockEntry> entry: this.content.entrySet()) {
        System.out.println(entry.getKey() + "; item = " + entry.getValue().getItem() +
            + entry.getValue().getQuantity().shortValue());
    }
}
}

```

```
}  
Version LinkedList...  
package articles.implementation;  
  
import java.util.HashMap;  
import java.util.Iterator;  
import java.util.LinkedList;  
import java.util.List;  
import java.util.Map;  
import java.util.Map.Entry;  
  
import articles.interfaces.IItem;  
import articles.interfaces.IStock;  
import exceptions.InvalidItem;  
import exceptions.InvalidQuantity;  
import exceptions.InvalidStock;  
  
public class StockList implements IStock {  
    private List<StockEntry> content;  
    private List<String> refs;  
    public StockList() {  
        content = new LinkedList<>();  
        refs = new LinkedList<>();  
    }  
  
    @Override  
    public Map<String, IItem> getStock() {  
        Map<String, IItem> res = new HashMap<String, IItem>();  
        for(int i =0;i<refs.size();i++)  
            res.put(refs.get(i),content.get(i).getItem());  
        return res;  
    }  
  
    @Override  
    public void setStock(Map<String, IItem> stock) throws InvalidStock {  
        if (stock==null) throw new InvalidStock("stock null");  
        for(String k:stock.keySet()) {  
            if (k==null) throw new InvalidStock("key null");  
            if (stock.get(k)==null ) throw new InvalidStock("value null");  
            if (!refs.contains(k)) {  
                refs.add(k);  
                content.add(new StockEntry(stock.get(k), 0));  
            }  
        }  
    }  
  
    @Override  
    public Map<String, Integer> getQuantities() {  
        Map<String, Integer> res = new HashMap<String, Integer>();  
        for(int i =0;i<refs.size();i++)
```

```

        res.put(refs.get(i), content.get(i).getQuantity());
    }
    return res;
}

@Override
public void setQuantities(Map<String, Integer> quantities) throws InvalidQuantity {
    if (quantities==null) throw new InvalidQuantity("map null");
    for(String k:quantities.keySet()) {
        if (k==null) throw new InvalidQuantity("key null");
        if (quantities.get(k)==null ) throw new InvalidQuantity("value null");
        if (!refs.contains(k)) { throw new InvalidQuantity("inexistent ref");}
        Integer i = quantities.get(k);
        if (i<0) throw new InvalidQuantity("negative value");
        int index = refs.indexOf(k);
        content.set(index, new StockEntry(content.get(index).getItem(), i));
    }
}

@Override
public Integer getItemQuantity(String ref) throws InvalidItem {
    if (!refs.contains(ref))
        throw new InvalidItem("The reference does not exist in the stock.");
    return this.content.get(refs.indexOf(ref)).getQuantity();
}

@Override
public boolean enoughItemQuantity(String ref, int required) throws InvalidItem {
    return required <= getItemQuantity(ref);
}

@Override
public void add(String ref, IItem item, int qt) throws InvalidItem, InvalidQuantity {
    if ((item == null) || (ref == null))
        throw new InvalidItem("Invalid item; null.");
    if (qt <= 0)
        throw new InvalidQuantity("Quantity <= 0.");

    StockEntry entryToModify = null;
    if (!this.refs.contains(ref)) {
        entryToModify = new StockEntry(item, qt);
        refs.add(ref);
        content.add(entryToModify);
    } else {
        entryToModify = this.content.get(refs.indexOf(ref));
        entryToModify.addQuantity(qt);
        content.set(refs.indexOf(ref), entryToModify);
    }
}

@Override

```

```
public void display() {  
    System.out.println("Stock contents:");  
    for(int i =0;i<refs.size();i++){  
        System.out.println(refs.get(i) + "; item = " + content.get(i).getItem() + "; q  
            + content.get(i).getQuantity().shortValue());  
    }  
}  
  
}
```

▷ **Question 2.3 :**

Comparez les 2 solutions. Vous pouvez changer d'implantation de **Map** ou de **List** et comparer les temps d'exécution des tests. (Avec un stock de plusieurs milliers d'item.)

Construire un tableau... L'accès par hash ou array doit donner des performances meilleures qu'avec une liste chaînée...