Collections en Java (cours 2)

Virginia Aponte

CNAM-Paris

23 mars 2017

1. Parcourir tableaux/collections : boucle *for each*

Boucle *for each*

Boucle de parcours sur *collections* et *tableaux* :

```
for (T el : col) {
    «appliquer actions sur» el;
}
```

« *Pour chaque valeur* `el` *de type* `T` *dans* `col` *appliquer actions.* »

- `col` : nom de variable tableau ou collection.
- `T` : type des éléments ,
- `el` : nom locale à la boucle donné à chaque élément ;

Limitation

Accès à une copie de *chaque composante* (pas de modification collection).

Exemple (1) : parcours d'une collection

- `lc` : est une collection d'éléments `String` ;
- on veut parcourir cette collection en affichant ses composantes.

```
Collection<String> lc = new ArrayList<String>();
...// ajout de composantes dans lc
for (String n : lc) {
   System.out.println(n);
}
```

- `String` : type éléments,
- `lc` : nom de la collection que l'on parcoure
- `n` : nom local donné à chaque composante,

Exemple (2) : parcours d'un tableau

Employé ici pour « accès » à chaque case de `tab` :

```
int [] tab = {1, 2, 3};
// Additionnes ttes cases du tableau
int somme = 0;
for (int e : tab) {
    somme = somme + e;;
}
System.out.println("Somme = "+ somme);
```

Affiche : `Somme = 6`

Attention

Variable e n'est pas une case du tableau, mais une copie de son contenu !

Pas de modification de collection/tableau

Attention : `for` each

ne permet pas traverser + modifier le tableau/collection en même temps !

```
int [] tab = {1, 2, 3};
for (int e : tab){
    e = e*10;   // Tentative de modification
}
System.out.println("Tableau apres:");
for (int e : tab){
   System.out.print(" "+ e);
}
```

```
Tableau apres:
 1 2 3
```

⇒ Non modifié après la boucle !

2. Remove, itérateurs

Modification pendant *for-each*

Impossible modifier tableau/collection via un *for each* :

- si tableau $\Rightarrow$ pas de modification ;
- si collection $\Rightarrow$ lévée exception
  `ConcurrentModificationException`

Modification collection dans for-each

Tentons d'enlever les éléments qui commencent par A :

```
// Tentative modification sur collection
for (String s: union){
    if (s.charAt(0)=='A') union.remove(s);
}
```

```
Exception in thread "main"
  java.util.ConcurrentModificationException
```

Modification collection pendant itération

Seul moyen sûr de le faire : via un *iterator*

Itérateurs

Itérateur

Objet de type (interface) `Iterator`, permettant de « visiter » dans une boucle tous les éléments d'une collection. Ses méthodes :

- `T next()` : obtenir le « prochain » élément de la collection ;
- `boolean hasNext()` : true s'il reste des éléments non encore « visités » ;
- `boolean remove()` : enleve un élément de la collection pendant l'itération

Toute collection fornit la méthode

```
Iterator iterator();
```

permettant de fabriquer cet objet.

Utiliser un itérateur sur une collection

1. Fabriquer un objet itérateur sur les composantes de type T de la collection `c` :
   - via un appel à sa méthode `c.iterator();`
     ```
     Iterator<T> it = c.iterator();
     ```
2. boucler tant que `it.hasNext()` renvoie true ;
   ```
   while (it.hasNext()) { ... }
   ```
3. dans la boucle :
   - `it.next()` ⇒ pour obtenir prochain élément ;
   - `it.remove()` ⇒ pour supprimer dernier élément obtenu par un appel à `next();`.

Modifier collection via itérateur

Enlever de la collection lc les mots qui commencent par A :

```
// création objet itérateur
Iterator<String> itc = lc.iterator();
while (itc.hasNext()) {
    String element = itc.next();
    if (element.charAt(0) == 'A') {
        itc.remove(); // enlever le dernier visité
    }
}
for (String s: lc){  // affichage après
    System.out.println(s);
}
```

Affiche :

```
Loulou Lucie Julien Thomas Remy
```

3. Les tables de Hash

Tables de Hash

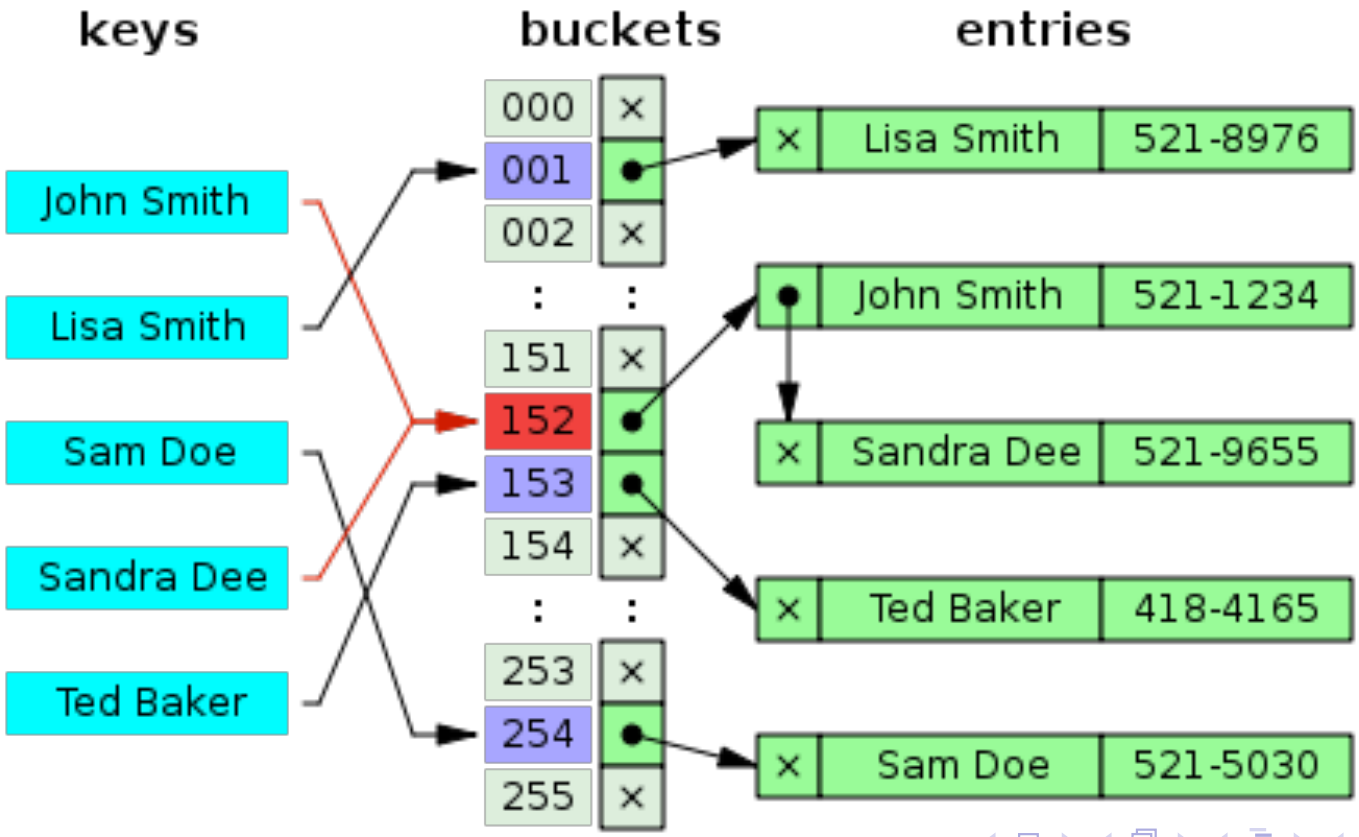
Table de Hash

Stocker des valeurs (String, Integer, Comptes) avec ajout/récupération de ces valeurs en temps quasi constant (presque comme pour un tableau), grace à une *fonction de hachage*.

- valeurs stockées dans une tableau (taille fixe) ;
- pour ajouter/rechercher une valeur $\Rightarrow$ appliquer *fonction de hachage* sur celle-ci $\Rightarrow$ donne son indice dans le tableau ;
- fonction de hachage peut attribuer même indice à plusieurs valeurs :
  - dans chaque case tableau $\Rightarrow$ *liste de collisions* avec toutes les valeurs « associées » à cet indice ;
  - chercher la valeur revient alors à chercher séquentiellement dans cette liste de collisions ;

Tables de Hash (2)$^{(*)}$

Si la fonction de hachage *distribue* bien les valeurs et si le tableau n'est pas trop petit, les listes de collisioin seront courtes $\Rightarrow$ chercher une valeur équivaut *en moyenne* à recupérer une case dans un tableau. (*) Source image : Wikipedia



Tables de Hash en Java : très utilisées

- Utilisées pour implanter `HashSet` et `HashMap`, et d'autres encore...
- *tout* objet Java hérite d'une fonction de hachage `hashCode()` définie par défaut dans la classe `Object`. Elle permet :
  - calculer position objet dans une table de hash. Donc, on peut construire tables de hash sur tous types d'objets !
  - utilise l'adresse de l'objet pour ce calcul $\Rightarrow$ même adresse, même résultat de hachage.
  - doit être redéfinie si on veut distiguer les objets (i.e. leur attribuer des cases différentes) selon leur contenu et non pas selon leurs adresses ;
  - si on redéfinit `hashCode()`, faire de même avec `equals()`.

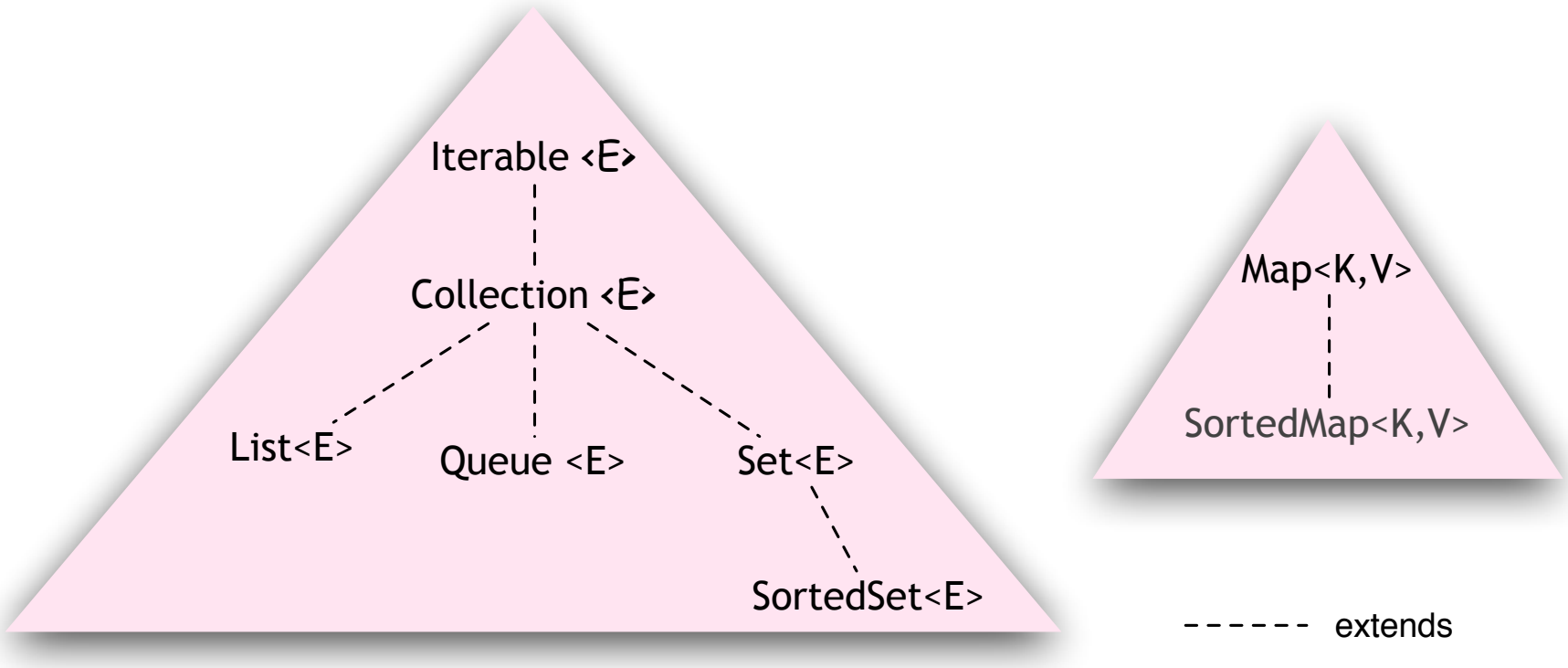
Pour en savoir plus $\Rightarrow$ voir transparents sur site de cours.

Table des hash utilisée $\Rightarrow$ probablement nécesaire de redéfinir `hashCode()` et `equals()`. Leurs définitions doivent être cohérentes : si o1.equals(o2) alors ils doivent avoir la même valeur de hachage !

4. L'interface `Set` de `Collection`

Rappel : hiérarchie d'interfaces pour les collections

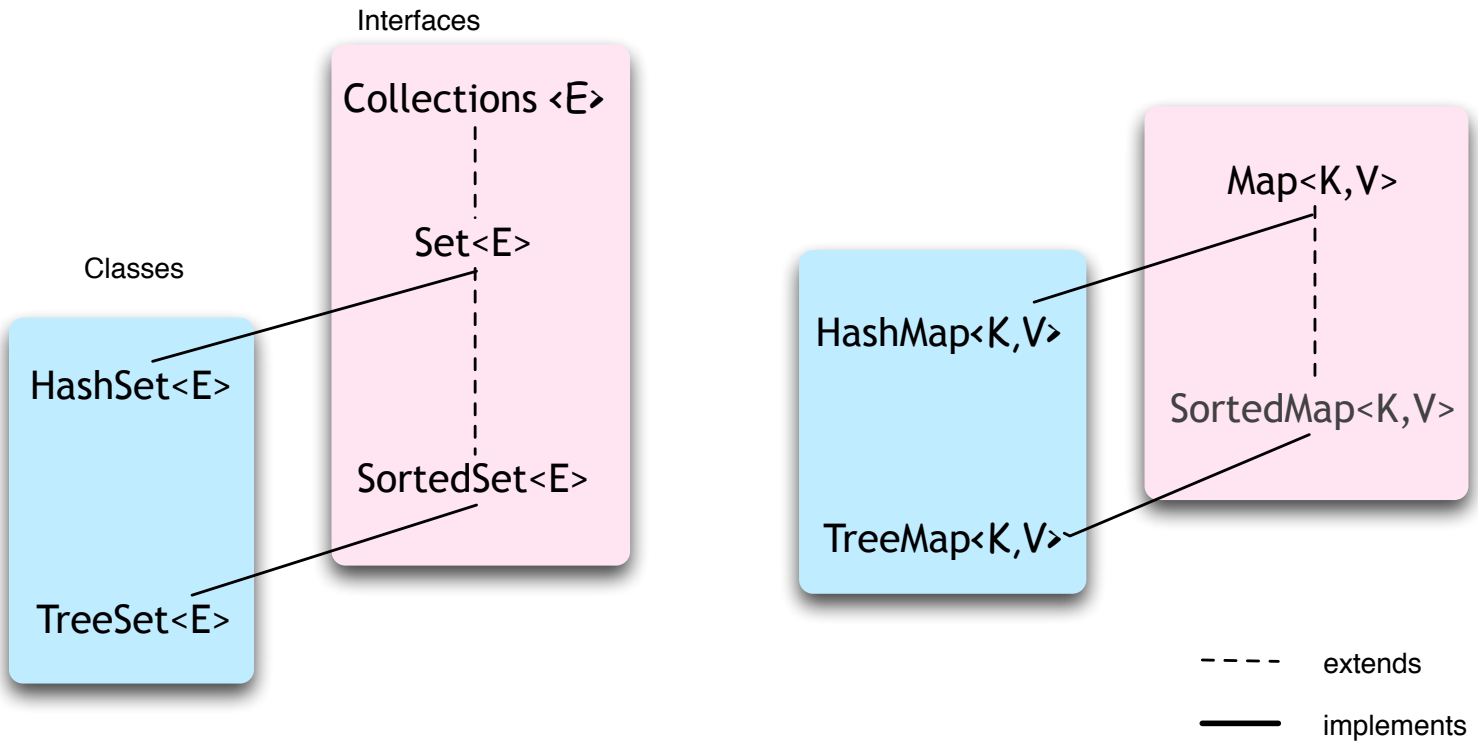
Extrait des hiérarchies d'interfaces :



<E> : type générique E pour les composantes

Rappel : Interfaces + classes ⇒ ensembles + mappes

(Extrait)



<E> : type générique E pour les composantes

Interface `Set`

Correspond à la structure des données « Ensemble »

- pas de doublons autorisés ;
- pas de position pour les éléments, pas d'élément « suivant » ;
  - pas de `get(i)` !
  - accès (indirect) aux éléments via *for each* ou *iterator()* ;

Garantie d'une structure sans doublons : très utile !

- ensembles de noms de contacts ;
- ensembles de comptes dans une banque
- ensemble de numéros de page où figure un terme (index) ;

Extrait de l'interface `Set`

```
public interface Set<E> extends Collection <E> {
 boolean add(E e);
 boolean addAll(Collection<?> c);
 boolean remove(Object o);
 boolean contains(Object o);
 boolean removeAll(Collection<?> c);
 ...
```

- `c.add(e)` : ajoute e dans c uniquement si absent.
- `addAll(c)`, union avec c ; `retainAll(c)`, intersection avec c ;
- ajout/suppression $\Rightarrow$ retournent true si effectuée.

Implantations principales de « Set<E> »

- `HashSet<E>` :
  - basée sur des *tables de hash* ;
  - ordre éléments <E> « non determiné » ;
- `TreeSet<E>` :
  - basée sur des *arbres de recherche* ;
  - éléments <E> ordonnées
    - ⇒ implanter `Comparable<E>` ou `Comparator<E>`

Contraintes

1. La classe <E> *doit rédéfinir* : `equals()` et `hashCode()`.
2. Parcours uniquement via `for each` ou `iterator()`.

5. La classe HashSet (implantation de Set)

Utiliser `HashSet<E>`

1. opérations d'ajout ⇒ n'ajoutent pas de doublons :
   - comparaison par égalité, si égal, pas d'ajout ! (retourne false)
   - pour que cela fonctionne bien :
     - classe <E> ⇒ doit rédéfinir `equals()` et `hashCode()`.
2. parcours uniquement via : `for each` ou `iterator()`.

Bon à savoir

- certaines classes prédéfinies (String, Integer, bibliothèque), rédéfinissent `equals()`, `hashCode()` de manière à considérer « égales » les objets avec même contenu.
- . . . et implantent `Comparable`.

⇒ faciles à utiliser avec Set !

Exemple : `HashSet<String>`

Rappel : `String` rédéfini `equals` et `hashCode()` pour comparer/calculer avec les contenus des chaînes et non leurs adresses !

```
  // Creation
  HashSet<String> col = new HashSet<String> ();
  col.add("A");  col.add("F");      // ajout A
  boolean doublon = col.add("A");  // Ajout A?
  System.out.println("A dedans? "+col.contains("A"));
  System.out.println("Ajoute 2 fois? "+doublon);
  for (String s: col){ System.out.print(s);}
```

Exemple (2) : affichages

```
  HashSet<String> col = new HashSet<String> ();
  col.add("A"); col.add("F");       // ajout A, ajout F
  boolean doublon = col.add("A");   // Ajout A?
  System.out.println("A dedans? "+col.contains("A"));
  System.out.println("Ajoute 2 fois? "+doublon);
  for (String s: col){ System.out.print(s);}
```

Affichages :

```
A dedans? true
Ajoute 2 fois? false
AF
```

5. Egalité en Java

Egalité en Java

Nombreuses méthodes des bibliothèques Java testent l'égalité d'objets :

- `add(E o)`, `remove(Object o)`
- `contains(Object o)`, `containsAll(Collection c)`, etc.
- $\Rightarrow$ parcours+comparaison pour recherche/suppression d'éléments :

La plupart utilise `boolean equals(Object o)` :

- prédéfinie dans la classe `Object` :
  - elle teste l'égalité d'adresses !

$$\Rightarrow \boxed{\texttt{o1.equals(o2)} \approx \texttt{o1 == o2}}$$

- présente dans *tout* objet (héritée d'Object) ;

Quelle notion d'égalité ?

L'égalité d'adresses n'est pas forcément adaptée à nos besoins :

- (oui) comparer deux Comptes ⇒ égalité d'adresses !
- (non) comparer deux String ⇒ égalité des caractères
- (non) comparer deux Dates ⇒ égalité jour, mois, année

Selon le cas ⇒ redéfinir la méthode equals

- de manière adaptée, dans la classe des objets à comparer.
- Il faut alors aussi rédéfinir `hashCode()` !

Si redéfinition equals ⇛ rédéfinir aussi hashCode() ! !

Egalité et types prédéfinis

`equals` ⇒ rédéfini pour majorité des types prédéfinis/bibliothèques Java.

Exemple : objets `String` :

- teste l'égalité des caractères !

  `s1.equals(s2)` ⇒ true

  si s2 non nul, et si tous leurs caractères 2 à 2 sont égaux.

Si collection avec String ⇒ ajout/recherche/suppression compare caractères car `equals` redéfini !.

Exemple rédéfintion equals (String)

Si s1, s2 de type `String`, `s1.equals(s2)` teste l'égalité des caractères !

```
HashSet<String> sc = new HashSet<String> ();
sc.add("A");                    // ajout de A
sc.add("A");                    // pas d'ajout
```

`sc.add("A")` avant l'ajout :

$\Rightarrow$ cherche dans `sc` un objet o t.q. `o.equals("A")`

- si trouvé, pas d'ajout ;
- si non, on l'ajoute.

Comportement de `add` si `equals` *non redéfini* pour String ?

Exemple : la classe `AdressePostale`

```
public class AdressePostale {
  private int numero;
  private String rue;
  private String ville;

  public AdressePostale(String n, String r, String v){
     numero=n; rue = r; ville =v;
  }
  public String toString(){
     return (""+ville+"-"+rue+","+numero);
  }
}
```

Quelques adresses

On ajoute 2 fois les adresses « 2, rue Conté » et « 6, rue Conté » :

```
public static void main(String[] args) {
  AdressePostale a1= new AdressePostale(2,"rue␣Conte","Pari
  AdressePostale a1bis= new AdressePostale(2,"rue␣Conte","P
  AdressePostale a2bis= new AdressePostale(6,"rue␣Conte","P
  AdressePostale a2= new AdressePostale(6,"rue␣Conte","Pari
  AdressePostale a3= new AdressePostale(4,"rue␣du␣Paradis",
  AdressePostale a4= new AdressePostale(120,"rue␣de␣la␣Gare
  HashSet<AdressePostale> col = new HashSet<AdressePostale>
  col.add(a1bis); col.add(a1);col.add(a2);
  col.add(a2bis); col.add(a3); col.add(a4);
  afficheSet(col); // applique toString() aux objets de col
```

Combien de fois apparaît l'adresse « 2, rue Conte » dans `col` ?

Les adresses affichées

```
Paris-rue Conte,2
Paris-rue Conte,6
Lyon-rue de la Gare,120
Paris-rue Conte,6
Paris-rue du Paradis,4
Paris-rue Conte,2
```

Combien de fois apparaît l'adresse « 2, rue Conte » dans `col` ? 2 fois !

Commet faire ?

Dans la classe `AdressePostale` :

1. rédéfinir `equals`,
2. mais aussi `hashCode`.

- A faire avec précaution !
- Astuce :
  - utiliser equals et hashCode sur des variables d'instance de types prédéfinis (p.e., String, Integer, etc)
  - ou demander à Eclipse de générer ces deux méthodes

Contraintes pour rédéfinir `equals`

Pour bien fonctionner `equals` doit satisfaire les propriétés suivantes :

- *réflexive* ⇒ `o1.equals(o1)` doit renvoyer true ;
- *symétrique* ⇒ `o1.equals(o2)` doit donner le même résultat que `o2.equals(o1)`
- *transitive*
- rédéfinir également `hashCode` et en cohérence avec `equals` : si 2 objets sont égaux par `equals`, alors ils renvoient le même hashCode !
- cohérence avec `compareTo` (par égalité) ;
- de préférence :
  - employer les mêmes variables pour calculer equals et hashCode ;
  - n'employer que des variables *inmuables* pendant la vie de l'application ;

Exemple de rédéfintion de `equals` et `hashCode`

Dans l'exemple qui suit :

- `equals(o)` :
  - utilise `equals` (du type String) sur la chaîne obtenue via `toString()` appliqué sur o et sur this.
  - un try catch traite le cas de pointeur o nul et celui de types différents.
- `hashCode()` :
  - utilise `hashCode` (du type String) appliqué sur la chaîne obtenue via `toString()` appliqué sur o et sur this.

Rédéfintion de `equals` et `hashCode`

```
public class AdressePostale {
  public boolean equals(Object o){
    try {
       AdressePostale a = (AdressePostale) o;
       String autre = a.toString().toLowerCase();
       String ici   = this.toString().toLowerCase();
       return ici.equals(autre);
    } catch (Exception e) {
       // Traite les cas o==null et
       // o n'est pas de type AdressePostale
       return false;
    }
  }
  public int hashCode() {
    String s = this.toString().toLowerCase();
    return s.hashCode();
  }
```

Affichages du main après rédéfinitions

```
Paris-rue Conte,2
Paris-rue Conte,6
Lyon-rue de la Gare,120
Paris-rue du Paradis,4
```