# Introduction programmation Java

## Cours 6: Map et Notion d'ordre

Stéphane Airiau

Université Paris-Dauphine

# Map

un `Map` représente une relation binaire surjective : chaque élément d'un `Map` est une paire qui met en relation une une clé à une valeur : chaque clé est unique, mais on peut avoir des doublons pour les valeurs.
⇨un tableau à deux colonnes !

**Interface Map<K,V>**

| | |
|---|---|
| `V` | `get(Object key)`<br>Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key. |
| `V` | `put(K key, V value)`<br>Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced by the specified value. |
| `V` | `remove(Object key)`<br>Removes the mapping for a key from this map if it is present. |
| `default V` | `replace(K key, V value)`<br>Replaces the entry for the specified key only if it is currently mapped to some value. Returns the value to which this map previously associated the key, or null if the map contained no mapping for the key. |

# Parcours d'un `Map`

Attention, `Map` n'est pas une sous interface de `Iterable`, donc on ne peut pas parcourir un `Map` avec une boucle `for each`!

On peut obtenir l'ensemble des clés, l'ensemble des valeurs, et l'ensemble des paires (clé,valeur) grace aux méthodes suivantes :

- `Set<K> keySet()` donne l'ensemble des clés (bien un `Set`)
- `Collection<V> values()` donne la collection des valeurs
- `Set<Map.Entry<K,V» entrySet()` donne l'ensemble des paires

# Classe `Map.Entry`

`Map.Entry` désigne une interface `Entry` qui est interne à l'interface `Map`. On peut créer des classes à l'intérieur de classes, mais je n'en parlerai pas plus aujourd'hui.

**`Interface Map.Entry<K,V>`**

| | |
|---|---|
| `K` | `getKey()`<br>Returns the key corresponding to this entry. |
| `V` | `getValue()`<br>Returns the value corresponding to this entry. |
| `V` | `setValue(V value)`<br>Replaces the value corresponding to this entry with the specified value (optional operation). |

# Exemple parcours d'une `Map`

```
1  Map<Personnage,Region> origines = new HashMap<>();
2  ...
3  for (Map.Entry<Personnage,Region> paire: origines.entrySet()){
4     Personnage p = paire.getKey();
5     Region r = paire.getValue();
6     if (r.getName().equals("Ibère"))
7       System.out.println(p);
8   }
```

Dans cet exemple, on part une `Map` qui associe à chaque personnage sa région d'origine et on affiche seulement les personnages qui sont des ibères.

# Performances sur des opérations

Pour de larges volume de données, les méthodes usuelles (add, remove, contains, size) devraient être rapides.

Implémentation de l'interface `Set`

| | `add` | `remove` | `contains` |
|---|---|---|---|
| `HashSet` | temps constant | temps constant | temps constant |
| `TreeSet` | $log(n)$ | $log(n)$ | $log(n)$ |

Implémentation de l'interface `List`

| | `get` | `set` | autre |
|---|---|---|---|
| `LinkedList` | $n$ | $n$ | |
| `ArrayList` | temps constant$^\star$ | temps constant | $n$ |

# Performances empirique

Je veux comparer les performances de la méthode`contains` pour les classes `ArrayList`, `LinkedList`, `HashSet` et `TreeSet`.

> J'ai un dictionnaire d'environs 25,000 mots à ma disposition
> Pour 100 exécutions
>
> 1- je tire au sort environ 10,000 mots du dictionnaire et je les places dans la structure
>
> 2- je tire au sort 1000 mots du dictionnaire et je les place dans un tableau de `String`
>
> 3- je mesure le temps qu'il faut pour savoir si chaque mot du tableau de `String` se trouve dans ma structure
>
> Je calcule le temps total pour chaque structure

| structure | `ArrayList` | `LinkedList` | `HashSet` | `TreeSet` |
|---|---|---|---|---|
| `contains` | 3.245 s | 5.286 s | 0.004 s | 0.024 s |
| `remove` | 3.155 s | 5.378 s | 0.003 s | 0.041 s |

Temps total

# Un peu d'algorithmique

- On veut stocker un ensemble de $k$ éléments qui ont une clé unique.
- Les clés peuvent prendre valeur dans un univers $U$.
- Si $U$ est grand, il n'est pas pratique, voir impossible, de faire un tableau de taille $U$ pour stocker un élément qui a pour clé $n$ dans la case $n^{ième}$ case du tableau.
- Si $k << |U|$, on peut utiliser une **fonction de hachage** $h$ et l'élément qui a pour clé $i$ sera stocké dans la case $h(i)$.
  On dit aussi que $h(i)$ est la valeur de hachage de i.

⇨ on cherche à minimiser la taille du tableau qui stocke les éléments.

- **problème :** deux clés peuvent avoir la même valeur de hachage (inévitable)
  on a alors une "collision"

# Résoudre les collisions avec chainage

**idée** : on stocke tous les éléments qui ont une même valeur de hash dans une liste chaînée.
⇨la case $i$ contient une référence vers la tête de la liste.
Supposons qu'on ait $k$ valeurs de hachage et $n$ éléments

- si les $n$ éléments ont la même valeur de hachage, on aura simplement une longue liste chainée ⇨ **aucun** intérêt !
- la situation idéale est une répartition uniforme des $n$ éléments sur les $k$ valeurs de hachage.

- l'insertion d'un élément se fait donc en $\mathcal{O}(1)$
- enlever un élément se fait en $\mathcal{O}(1)$ si la liste est doublement chaînée
- cherche un élément se fait en $\Theta(1+\alpha)$ où $\alpha$ est le nombre moyen d'éléments stockés dans une chaîne.

# Fonction de Hachage

Supposons qu'on ait $k$ valeurs de hachage.
On veut une fonction qui distribue de manière uniforme toutes les clés dans chacun des $k$.
Le problème, c'est qu'on ne connait que rarement la probabilité qu'on ait un élément avec une certaine clé.
Il existe des techniques pour générer de bonnes fonctions de hachage, mais ce n'est pas le problème dans ce cours.

# Retour sur `HashCode()` de la classe `Object`

Pour rappel, la classe `Object` possède une méthode `hashCode()` qui retourne un `int`.
⇨ la méthode `hashCode` est la fonction de hachage pour la classe!

⇨ **Attention!** si vous utilisez un `HashSet` ou un `HashMap`, l'implémentation de `hashCode()` est **très** important.

- si vous pouvez vous ramener à des fonctions de hachage codé dans `Java`, cela pourra avoir de bonnes performances (ex `String`).
- si votre fonction de hachage est constante, vous perdez tout bénéfice d'utiliser ces structures
- si votre fonction de hachage n'est pas cohérente par rapport à votre méthode `equals`, vous risquez de ne pas trouver un élément pourtant présent!
  - si `this.equals(obj)` retourne vrai, on doit aussi avoir `this.hashCode()==obj.hashCode()`.

# Retour sur `HashCode()` de la classe `Object`

Il existe une méthode `Objects.hash(varargs)` qui calcule la valeur de hachage des ses arguments et les combine automatiquement.

La classe `Object` possède une méthode `hashCode`, mais qui utilise des méthodes qui dépendent de l'implémentation locale. La valeur peut être dérivée de l'adresse mémoire, d'un nombre aléatoire, ou d'une combinaison des deux.

# Un mot sur la boucle `for each`

```
for (Personnage p : villageois)
      System.out.println(p);
```

- Facilité syntaxique pour écrire plus rapidement une boucle.
- Lors de la compilation, `Javava` traduire cette ligne en utilisant un `Iterator`.

**Attention** A l'intérieur de la boucle, on **ne** peut **pas** modifier la collection (par exemple avec `remove()`).
⇨ sinon, cela cause une `ConcurrentModificationException`.

# Ordre

L'interface `Comparable` contient une seule méthode :

```
public int compareTo(T o)
```

Cette méthode retourne

- un entier négatif si l'objet est plus petit que l'objet passé en paramètre
- zéro s'ils sont égaux
- un entier positif si l'objet est plus grand que l'objet passé en paramètre.

les classes `String`, `Integer`, `Double`, `Date`, `GregorianCalendar` et beaucoup d'autres implémentent toutes l'interface `Comparable`.

# Exemple

```
1  public class Gaulois extends Personnage
2                  implements Comparable<Gaulois>{
3    String nom;
4    int quantiteSanglier;
5      ...
6
7    public int compareTo(Gaulis ixis) {
8      return this.quantiteSanglier - ixis. quantiteSanglier;
9    }
10 }
```

# interface `Comparator`

Une classe qui implémente l'interface comparator représente une notion d'ordre / un critère d'ordre.
A priori, on n'a besoin d'implémenter une seule méthode, la méthode pour comparer deux éléments.

```
1 public interface Comparator<T> {
2    int compare(T o1, T o2);
3    boolean equals(Object obj);
4 }
```

Pour comparer des `Gaulois`, et même tous les `Personnage` selon leur taille, on peut écrire la classe suivante :

```
1 public class OrdreHauteur implements Comparator<Personnage> {
3     public int compare(Personnage gauche, Personnage droit){
4       return gauche.hauteur < droite.hauteur ? -1:
5             (gauche.hauteur == droite.hauteur ? 0 : 1);
5     }
6   }
```

# Trier les éléments d'une collection

La méthode `sort` de l'interface `List` effectue le tri.

```
void sort(Comparator<? super E> c)
```

Sorts this list according to the order induced by the specified Comparator. All elements in this list must be mutually comparable using the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the list).
If the specified comparator is null then all elements in this list must implement the Comparable interface and the elements' natural ordering should be used.

- appel avec le paramètre `null` : il vaudrait mieux que la liste contienne des instances d'une classe implémentant l'interface `Comparable`, sinon, `Java`ne va pas savoir comment comparer deux éléments !
  on utilise le critère d'ordre implémenté dans la classe des éléments de la liste
- appel avec une instance d'une classe implémentant l'interface `Comparator` : on donne directement le critère pour comparer !

# Trier les éléments d'une collection : autre solution

Deux méthodes de tri sont implémentées dans `Java`et se trouvent dans la classe `Collections` (attention avec un **s**).

- La première méthodea un seul argument : une collection d'instance d'une classe qui implémente l'interface `Comparable`. Le tri se fait donc en utilisant la méthode `compareTo` codée dans la classe `T`.
- La seconde méthode nécessite deux arguments : la collection d'instance d'une classe `T` et une notion d'ordre sur la classe `T`. Le tri se fera donc en utilisant la méthode `compare` codée dans la classe implémentant `Comparator`.

bon exercice pour les méthodes **`static`** avec paramètre de type !

```
   // classe Collections
1  public static <T extends Comparable<? super T»
2                  void sort(List<T> list)
3  public static <T> void sort
4                  (List<T> list, Comparator<? super T> c)
```

# Exemple

```
1  public static void main(String[] args){
2    List<Personnage> personnages = new ArrayList<>();
3    personnages.add(new IrreductibleGaulois("Obelix", 1.81));
4    personnages.add(new IrreductibleGaulois("Astérix", 1.60));
5    personnages.add(new Personnage("César", 1.75));
6
7    for (Personnage p: personnages)
8      System.out.println(p.presentation());
9
10   personnages.sort(null);
11     // ou bien Collections.sort(personnages);
12   for (Personnage p: personnages)
13     System.out.println(p.presentation());
14
15   Comparator<Personnage> ordre = new OrdreHauteur();
16   personnages.sort(ordre);
17     // ou bien Collections.sort(personnages, ordre);
18   System.out.println(personnages);
```

L'utilisation d'une instance d'une classe implémentant l'interface `Comparator` est quelque peu lourde. On va introduire des notions pour permettre une écriture plus élégante.