Cours 7 : Les associations

HashMap
TreeMap

Auteur : CHAUDET Christelle

Les cartes / associations

- Utilisation des cartes (ou map) : Si on ne connait pas exactement un objet, mais que l'on dispose seulement de certaines informations importantes à rechercher.
- Structure : enregistrement de paires clé / valeur
- Exemple :
  Les lecteurs doivent pouvoir donner une appréciation à un parchemin.
  - Les appréciations sont données à partir de l'énuméré « Appreciation »

```
public enum Appreciation {
  EXCELLENT, TRES_BIEN, BIEN, PASSABLE, MAUVAIS;
}
```


Les cartes / associations

- Nous voulons donc associer pour chaque parchemin une appréciation.
- Structure : enregistrement de paires clé / valeur
- la clef -> le parchemin, la classe « Parchemin » DOIT posséder les méthodes : **hashCode** et **equals**.
  la valeur -> une appréciation

```
Map<Parchemin, Appreciation> parcheminApprecie
   = new HashMap<>()
parcheminApprecie.put(
   new Parchemin("Commentaires sur la guerre des gaules",
   cesar, new Date(12, 07, -50)),
   Appreciation.PASSABLE
);
```


L'interface Map

Map<K,V>

- Définit les opérations de manipulation d'un ensemble d'association clé-valeur, dans lesquelles les clés sont uniques.

Methods

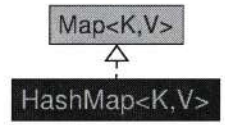
Modifier and Type	Method and Description
`void`	`clear()` Removes all of the mappings from this map (optional operation).
`boolean`	`containsKey(Object key)` Returns `true` if this map contains a mapping for the specified key.
`boolean`	`containsValue(Object value)` Returns `true` if this map maps one or more keys to the specified value.
`Set<Map.Entry<K,V>>`	`entrySet()` Returns a `Set` view of the mappings contained in this map.
`boolean`	`equals(Object o)` Compares the specified object with this map for equality.
`V`	`get(Object key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.

L'interface Map

Map<K,V>

V	**get**(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
int	**hashCode**() Returns the hash code value for this map.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Set<K>	**keySet**() Returns a Set view of the keys contained in this map.
V	**put**(K key, V value) Associates the specified value with the specified key in this map (optional operation).
void	**putAll**(Map<? extends K,? extends V> m) Copies all of the mappings from the specified map to this map (optional operation).
V	**remove**(Object key) Removes the mapping for a key from this map if it is present (optional operation).
int	**size**() Returns the number of key-value mappings in this map.
Collection<V>	**values**() Returns a Collection view of the values contained in this map.

Classe concrète : HashMap<K,V>



- La clé doit être unique,
- Redéfinition des méthodes :
  hashCode & *equals*
  dans la classe qui sera utilisée comme clé,
- L'interface
  - n'hérite pas de Collection donc vous ne pouvez pas utiliser *add* mais *put*.
  - n'hérite pas d'Iterable donc ne peut être directement utilisée dans un foreach.

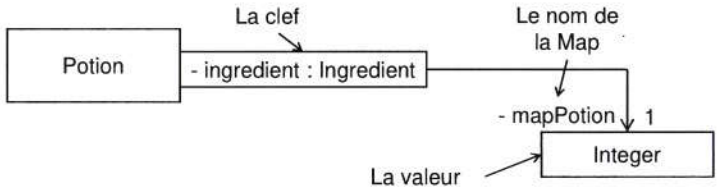
Les HashMap dans UML

- Il faut voir une map comme une association qualifiée, la clef étant le qualifieur.
- Exemple :

```
Map<Ingredient, Integer> mapPotion = new HashMap<>();
```

Quantité

- Représentation UML



Exemple : la bibliothèque (1/3)

- Dans la classe « GestionBibliotheque », nous avons ajouté la HashMap associant une liste de commentaires à un parchemin :

```
Map<Parchemin, List<Appreciation>> parcheminsApprecies = new HashMap<>();
```

La classe « Parchemin » DOIT posséder les méthodes : **hashCode** et **equals**

- La méthode *ajouterParchemin* crée le parchemin, puis **si** la map ne la contient pas déjà, l'ajoute avec une liste vide, qui pourra contenir les appréciations sur le parchemin, et retourne le parchemin créé.

```
public Parchemin ajouterParchemin(String titre, Personnage auteur,
                                  Date date) {
  Parchemin parchemin = new Parchemin(titre, auteur, date);
  if (!parcheminApprecie.containsKey(parchemin)) {
    List<Appreciation> liste = new ArrayList<>();
    parcheminApprecie.put(parchemin, liste);
  }
  return parchemin;
}
```

Exemple : la bibliothèque (2/3)

```
Map<Parchemin, List<Appreciation>> parcheminsApprecies = new HashMap<>();
```

- La méthode *ajouterAppreciation* ajoute une appréciation à un parchemin.
  - si le parchemin n'est pas contenu dans la map alors elle crée une liste vide et place l'association parchemin / liste dans la map
  - sinon elle récupère la valeur (la liste d'appréciation) pour le parchemin souhaité,

Et dans les deux cas, ajoute l'appréciation à la liste.

```
public void ajouterAppreciation(
        Parchemin parchemin,
        Appreciation appreciation) {
    List<Appreciation> liste;
    if (!parcheminApprecie.containsKey(parchemin)) {
      liste = new ArrayList<>();
      parcheminApprecie.put(parchemin, liste);
    } else {
      liste = parcheminApprecie.get(parchemin);
    }
    liste.add(appreciation);
}
```

2

Exemple : la bibliothèque (3/3)

- La méthode *donnerAppreciation* retourne une chaine listant l'ensemble des appréciations d'un parchemin.

```
public String donnerAppreciation(Parchemin parchemin) {
  String chaine = "Les appreciations pour le parchemin \""
                + parchemin.getTitre()  + "\" sont : \n";
  List<Appreciation> listeAppreciations = parcheminApprecie.get(parchemin);
  for (Appreciation appreciation : listeAppreciations) {
    chaine += "- " + appreciation + "\n";
  }
  return chaine;
}
```

```
Les appreciations pour le parchemin "Mes plus grands succès" sont :
- PASSABLE
- BIEN
```

```
Parchemin bestOfMusic = new Parchemin("Mes plus grands succès",
                                      assurancetourix, new Date(30, 04, -45));
gestionBibliotheque.ajouterAppreciation(bestOfMusic,
                                        Appreciation.PASSABLE);
gestionBibliotheque.ajouterAppreciation(bestOfMusic, Appreciation.BIEN);
System.out.println(gestionBibliotheque.donnerAppreciation(bestOfMusic));
```

9

Les vues d'une Map (1/3)

- Possibilité d'obtenir une vue de carte : objet qui implémente l'interface Collection ou l'une de ses sous-interfaces.
- Avantage d'une vue : Mise à jour automatique de la vue sur l'ensemble.
- Il existe 3 vues :
  - l'ensemble de clés : Set keySet(),
  - l'ensemble des valeurs : Collection values(),
  - l'ensemble des paires clé/valeur : Set entrySet().

`Set<Map.Entry<K,V>>`	**`entrySet()`** Returns a `Set` view of the mappings contained in this map.
`Set<K>`	**`keySet()`** Returns a `Set` view of the keys contained in this map.
`Collection<V>`	**`values()`** Returns a `Collection` view of the values contained in this map.

Les vues d'une Map (2/3)

- Soit la méthode donnerParchemin qui retourne une vue sur la map parcheminApprecie

```
public Set<Parchemin> donnerParchemins(){
   return parcheminApprecie.keySet();
}
```

- Dans la classe TestMusee j'ajoute une méthode static qui me permet d'afficher un ensemble

```
private static void afficherVue(Set<Parchemin> vueParchemins) {
   System.out.println("Vue Parchemins :");
   for (Parchemin parchemin : vueParchemins) {
     System.out.println(parchemin);
   }
}
```

Les vues d'une Map (3/3)

- Le scénario de test fonctionnel est le suivant :

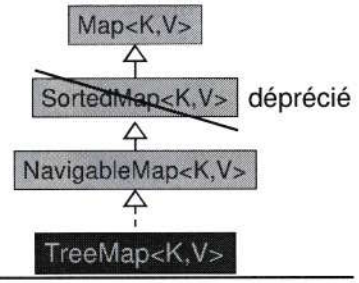
```
Set<Parchemin> vueParchemins = musee.donnerParchemins();
afficherVue(vueParchemins);
```
```
Vue parchemins :
```

```
Parchemin guerreDesGaules = musee.ajouterParchemin(
   "Commentaires sur la guerre des gaules", cesar, new Date(12, 07, -50));
musee.ajouterAppreciation(guerreDesGaules, Appreciation.PASSABLE);
afficherVue(vueParchemins);
```
```
Vue parchemins :
Commentaires sur la guerre des gaules, Cesar, 12/7/-50
```

```
Parchemin effetsPotion = musee.ajouterParchemin("Les effets
secondaires de la potion magique", druide, new Date(21, 12, -70));
musee.ajouterAppreciation(effetsPotion, Appreciation.EXCELLENT);
afficherVue(vueParchemins);
```
```
Vue parchemins :
Commentaires sur la guerre des gaules, Cesar, 12/7/-50
Les effets secondaires de la potion magique, Panoramix, 21/12/-70
```


TreeMap<K,V>

- Carte triée selon l'ordre naturel ou l'ordre imposé de la clef.
- Constructeurs
  - TreeMap()
  - TreeMap(Comparator<? super K> comparator)
  - TreeMap(Map<? extends K, ? Extends V> m)
  - TreeMap(SortedMap<K, ? Extends V> m)



`Map.Entry<K,V>`	`ceilingEntry(K key)` Returns a key-value mapping associated with the least key greater than or equal to the given key, or `null` if there is no such key.
`K`	`ceilingKey(K key)` Returns the least key greater than or equal to the given key, or `null` if there is no such key.
`void`	`clear()` Removes all of the mappings from this map.
`Object`	`clone()` Returns a shallow copy of this `TreeMap` instance.
`Comparator<? super K>`	`comparator()` Returns the comparator used to order the keys in this map, or `null` if this map uses the natural ordering of its keys.
`boolean`	`containsKey(Object key)` Returns `true` if this map contains a mapping for the specified key.
`boolean`	`containsValue(Object value)` Returns `true` if this map maps one or more keys to the specified value.
`NavigableSet<K>`	`descendingKeySet()` Returns a reverse order `NavigableSet` view of the keys contained in this map.
`NavigableMap<K,V>`	`descendingMap()` Returns a reverse order view of the mappings contained in this map.
`Set<Map.Entry<K,V>>`	`entrySet()` Returns a `Set` view of the mappings contained in this map.

`Map.Entry<K,V>`	`firstEntry()` Returns a key-value mapping associated with the least key in this map, or `null` if the map is empty.
`K`	`firstKey()` Returns the first (lowest) key currently in this map.
`Map.Entry<K,V>`	`floorEntry(K key)` Returns a key-value mapping associated with the greatest key less than or equal to the given key, or `null` if there is no such key.
`K`	`floorKey(K key)` Returns the greatest key less than or equal to the given key, or `null` if there is no such key.
`V`	`get(Object key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`SortedMap<K,V>`	~~`headMap(K toKey)`~~ Returns a view of the portion of this map whose keys are strictly less than `toKey`.
`NavigableMap<K,V>`	`headMap(K toKey, boolean inclusive)` Returns a view of the portion of this map whose keys are less than (or equal to, if `inclusive` is true) `toKey`.
`Map.Entry<K,V>`	`higherEntry(K key)` Returns a key-value mapping associated with the least key strictly greater than the given key, or `null` if there is no such key.
`K`	`higherKey(K key)` Returns the least key strictly greater than the given key, or `null` if there is no such key.

`Set<K>`	`keySet()` Returns a `Set` view of the keys contained in this map.
`Map.Entry<K,V>`	`lastEntry()` Returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.
`K`	`lastKey()` Returns the last (highest) key currently in this map.
`Map.Entry<K,V>`	`lowerEntry(K key)` Returns a key-value mapping associated with the greatest key strictly less than the given key, or `null` if there is no such key.
`K`	`lowerKey(K key)` Returns the greatest key strictly less than the given key, or `null` if there is no such key.
`NavigableSet<K>`	`navigableKeySet()` Returns a `NavigableSet` view of the keys contained in this map.
`Map.Entry<K,V>`	`pollFirstEntry()` Removes and returns a key-value mapping associated with the least key in this map, or `null` if the map is empty.
`Map.Entry<K,V>`	`pollLastEntry()` Removes and returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.
`V`	`put(K key, V value)` Associates the specified value with the specified key in this map.
`void`	`putAll(Map<? extends K,? extends V> map)` Copies all of the mappings from the specified map to this map.
`V`	`remove(Object key)` Removes the mapping for this key from this TreeMap if present.
`int`	`size()` Returns the number of key-value mappings in this map.
`NavigableMap<K,V>`	`subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)` Returns a view of the portion of this map whose keys range from `fromKey` to `toKey`.
`SortedMap<K,V>`	`subMap(K fromKey, K toKey)` Returns a view of the portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive.
`SortedMap<K,V>`	`tailMap(K fromKey)` Returns a view of the portion of this map whose keys are greater than or equal to `fromKey`.
`NavigableMap<K,V>`	`tailMap(K fromKey, boolean inclusive)` Returns a view of the portion of this map whose keys are greater than (or equal to, if `inclusive` is true) `fromKey`.
`Collection<V>`	`values()` Returns a `Collection` view of the values contained in this map.

Methods inherited

`equals, hashCode, isEmpty, toString` **java.util.AbstractMap**

`finalize, getClass, notify, notifyAll, wait, wait, wait` **java.lang.Object**

`equals, hashCode, isEmpty` **java.util.Map**

Exemple : les potions (1/4)

- Rappel : un énuméré possède d'office un ordre naturel

```
public enum Necessite {
  INDISPENSABLE, AU_CHOIX, OPTIONNEL;
}
```

Dans cette énuméré l'ordre sera de la plus petite valeur à la plus grande : INDISPENSABLE, AU_CHOIX, OPTIONNEL

Exemple : les potions (2/4)

- La classe Potion devient :

```
public class Potion {
 Map<Necessite, List<Ingredient>> listeIngredients =
                                          new TreeMap<>();

 public Potion() {
  listeIngredients.put(Necessite.OPTIONNEL,
                       new ArrayList<Ingredient>());
  listeIngredients.put(Necessite.INDISPENSABLE,
                       new ArrayList<Ingredient>());
  listeIngredients.put(Necessite.AU_CHOIX,
                       new ArrayList<Ingredient>());
 }
}
```

L'ensemble listeIngrédients est trié selon l'ordre naturel des clés de type Necessite

listeIngredients = {INDISPENSABLE=[], AU_CHOIX=[], OPTIONNEL=[]}

Exemple : les potions (3/4)

- La classe Potion devient :

```
public void ajouterIngredient(Ingredient ingredient,
                              Necessite necessaire) {
  List<Ingredient> liste = listeIngredients.get(necessaire);
  if (!liste.contains(ingredient)){
    liste.add(ingredient);
  }
};

public String toString() {
  String chaine = "Les ingrédients de la potion sont :\n";
  chaine += listeIngredients;
  return chaine;
}
```

```
listeIngredients = {INDISPENSABLE=[], AU_CHOIX=[], OPTIONNEL=[]}
```

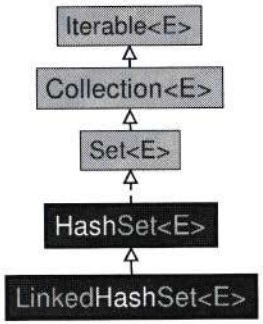
Exemple : les potions (4/4)

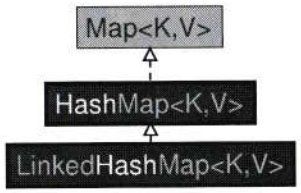
```
Potion potion = new Potion();
potion.ajouterIngredient(fraises, Necessite.OPTIONNEL);
potion.ajouterIngredient(huile, Necessite.AU_CHOIX);
potion.ajouterIngredient(hydromel, Necessite.INDISPENSABLE);
potion.ajouterIngredient(miel, Necessite.INDISPENSABLE);
potion.ajouterIngredient(trefle, Necessite.INDISPENSABLE);
potion.ajouterIngredient(homard, Necessite.OPTIONNEL);
potion.ajouterIngredient(carottes, Necessite.INDISPENSABLE);
potion.ajouterIngredient(betterave, Necessite.AU_CHOIX);
potion.ajouterIngredient(gui, Necessite.INDISPENSABLE);
potion.ajouterIngredient(sel, Necessite.INDISPENSABLE);
potion.ajouterIngredient(poisson, Necessite.INDISPENSABLE);
System.out.println(potion);
```

```
Les ingrédients de la potion sont :
{INDISPENSABLE=[hydromel, miel, trèfle à quatre feuilles, carottes, gui, sel, poisson],
AU_CHOIX=[huile de roche, betterave], OPTIONNEL=[fraises, homard]}
```

HashCode

- Le hashCode ne sert qu'aux Collections ?





Non, le hashCode est utile pour d'autres fonctions. Par exemple, depuis Java 7 il est possible de faire un switch sur un type String …

Structure de choix : cas où (1/2)

- Jusqu'à présent le switch ne pouvait s'utiliser que sur certains types primitifs (char , byte, short, int), leurs enveloppeurs (Character , Byte , Short, Integer), ou sur un type énuméré (enum type).
- **Java 7** permet d'utiliser un **switch** sur une **String**, afin de comparer une chaîne de caractères avec des valeurs constantes (définies dès la compilation).
- Tout comme le **switch(enum)**, le **switch(String)** n'accepte pas de valeur **null** (cela provoque une exception).

Structure de choix : cas où (2/2)

- La comparaison se fait sur le hashCode de la chaine.

```
String action = ...

switch(action) {
case "save":
    save();
    break;
case "save-as":
    saveAs();
    break;
case "update":
    update();
    break;
case "delete":
    delete();
    break;
case "delete-all":
    deleteAll();
    break;
default:
    unknownAction(action);
}
```

```
switch (action.hashCode()) {
case 3522941: // "save".hashCode()
    if ("save".equals(action))
        switchIndex = 1;
    break;
case 1872766594: // "save-as".hashCode()
    if ("save-as".equals(action))
        switchIndex = 2;
    break;
                    ...
switch (switchIndex) {
case 1: // "save"
    save();
    break;
case 2: // "save-as"
    saveAs();
    break;
                    ...
```