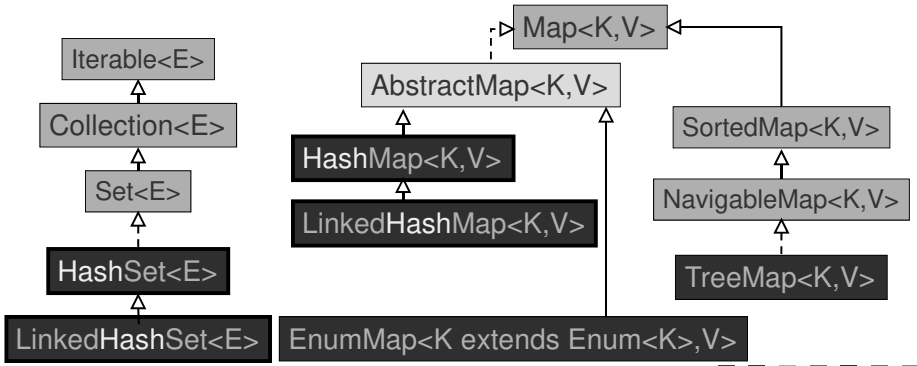
Cours 5 : Collections

Les tables de hachage
Les méthodes (equals, hashCode)
Les associations

Auteur : CHAUDET Christelle – Intervenants : BODEVEIX Jean-Paul, MIGEON Frédéric

Introduction

- Les tables de hachage
  - Principe
  - Collision de hachage
  - Code de hachage
- Les méthodes
  - hashCode
  - equals
- Les classes concrètes



Principe (1/3)

- Calcul d'un nombre entier appelé **code de hachage** pour chacun des éléments.
  - Exemple : la classe **String** (comme tous les enveloppeurs) possède une méthode *hashCode*, déclarée dans la classe Object, qui calcule le code de hachage d'une chaîne.

```
String w, x;
w= "a"; System.out.println(w.hashCode());    // 97
x="b";  System.out.println(x.hashCode());    // 98
```

- Ces codes de hachage :
  - sont calculés très rapidement,
  - ne dépendent que de l'état de l'objet à rechercher (et non des autres objets de la table).

Principe (2/3)

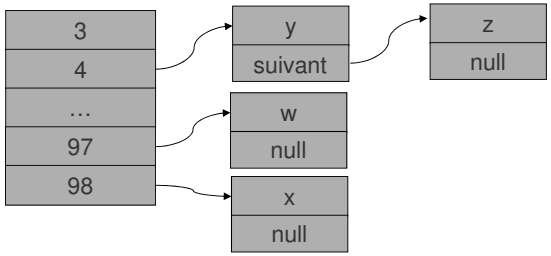
- Une table de hachage est constituée d'un tableau de listes chaînées. Chaque liste est appelée un **seau** (ou *panier* ou *bucket*).
- Pour trouver la place d'un élément il faut :
  - calculer son code de hachage,
  - réduire le résultat par un modulo du nombre total de seaux.
- Exemple :
  - String y;
    y="ba"; System.out.println(y.hashCode()); // 3 135
  - Nombre de seaux de la table : 101

  ⇒ L'objet est placé dans le seau 4 (3 135 modulo 101)

Principe (3/3)

- Insertion d'un élément dans une table :
  - si le seau est vide : on insère l'élément,
  - si le seau est plein : collision de hachage.

	Code de hachage	Numéro du seau
w = "a"; System.out.println(w.hashCode());	// 97	97
x = "b"; System.out.println(x.hashCode());	// 98	98
y = "ba"; System.out.println(y.hashCode());	// 3135	4
z = x.concat(w); System.out.println(z.hashCode());	// 3135	4



Facteur de charge

- Attention : trop d'éléments dans une table de hachage
  - ⇒ le nombre de collisions augmente,
  - ⇒ la performance de recherche baisse.

- Les constructeurs des collections utilisant une table de hachage proposent de choisir le facteur de charge.

  Si le nombre approximatif d'éléments :
  - est connu, alors prendre pour taille initiale de la table 150% du nombre d'élément,
  - est inconnu, alors ne pas modifier le facteur de charge par défaut. La table sera réorganisée automatiquement quand elle atteindra un facteur de charge de 0,75 en doublant le nombre de seaux.

Méthodes hashCode et equals (1/3)

- Dans la classe Object, les méthodes *hashCode* et *equals* sont toutes les deux définies et basées sur la référence mémoire de l'objet.

- Ces deux méthodes sont COMPATIBLES, c'est-à-dire que deux objets égaux ont le même hashCode.
  Si x.equals(y) = true alors x.hashCode() = y.hashCode()

- Donc si vous souhaitez que vos objets soient comparés selon leur état et non sur leur référence mémoire il vous faut redéfinir les méthodes *hashCode* **et** *equals* pour les objets que vous insérez dans une table de hachage

Méthodes hashCode et equals (2/3)

- Reprenons l'exemple du musée, il contient des trophées correspondant aux équipements des romains (casque, bouclier, glaive) rapportés au combat.

```
public abstract class Equipement {
 private String etat;
 private String nom;

 public Equipement(String etat, String nom) {
   this.etat = etat;
   this.nom = nom;
 }

 public String toString() {
   return nom + " " + etat;
 }
}
```

Méthodes hashCode et equals (3/3)

- Deux équipements sont considérés identiques s'ils ont le même nom.
- La méthode *equals*

```
public boolean equals(Object objet) {
   if (objet =! null && getClass() == objet.getClass()) {
     Equipement equipementToCompare = (Equipement) objet;
     return nom.equals(equipementToCompare.nom);
   }
   return false;
 }
```

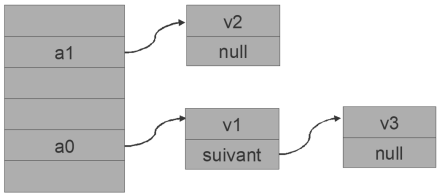
- La méthode *hashCode*

```
public int hashCode() {
   return 31 * nom.hashCode();
}
```

Remarque :
le hashCode() pourrait aussi prendre en compte le getClass()

La classe HashSet<E>

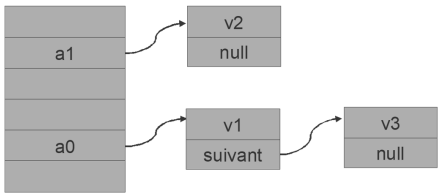
- C'est l'implémentation de l'interface Set la plus utilisée.
- Cette classe implémente un ensemble à partir d'une table de hachage (tableau dans lequel les éléments sont stockés à un emplacement déduit de leurs contenus).
- Le constructeur par défaut HashSet génère une table de hachage :
  - de 101 seaux
  - avec un facteur de charge de 0,75



- Possibilité de modifier ces données :
  - HashSet(int initialCapacity)
  - HashSet(int initialCapacity, float loadFactor)

Les fonctionnalités de HashSet

- Méthode add : ajoute des éléments dans la liste
- Il est plus efficace d'utiliser la méthode contains que de passer par un itérateur :
  - Méthode contains : redéfinie pour effectuer une recherche rapide et vérifier si un élément fait déjà partie du set.
    N'accède qu'aux éléments d'UN SEUL SEAU.
  - L'itérateur d'un set parcourt TOUS LES SEAUX un par un dans un ordre aléatoire.

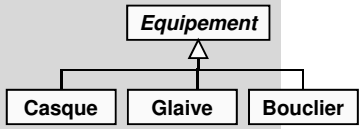


Exemple de HashSet (1/2)

```
public abstract class Equipement {
  String etat;
  String nom;

  public Equipement(String etat, String nom) {
    this.etat = etat;
    this.nom = nom;
  }

  public String toString() {
    return nom + " " + etat;
  }

  public boolean equals(Object objet) {
    if (objet =! null && getClass() == objet.getClass()) {
      Equipement equipementToCompare = (Equipement) objet;
      return nom.equals(equipementToCompare.nom);
    }
    return false;
  }

  public int hashCode() {
     return 31 * nom.hashCode();
  }
}
```

Exemple de HashSet (2/2)

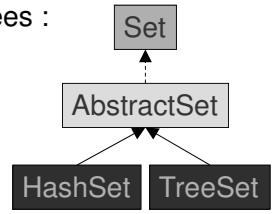
```
Glaive glaive1 = new Glaive("étincelant");
Glaive glaive2 = new Glaive("sale");
Casque casque = new Casque ("usée");
Bouclier bouclier = new Bouclier("étincelant");
```




```
Set<Equipement> typeTrophees = new HashSet<>();
typeTrophees.add(glaive1);      typeTrophees.add(glaive2);
typeTrophees.add(casque);    typeTrophees.add(bouclier);

System.out.println(typeTrophees +", "+typeTrophees.size());
//[bouclier, glaive, casque], 3
System.out.println("Le type de trophée contient : ");
System.out.println(" - le glaive1 ? " + typeTrophees.contains(glaive1));
System.out.println(" - le glaive2 ? " + typeTrophees.contains(glaive2));
//Le type de trophée contient :
// - le glaive1 ? true
// - le glaive2 ? true
typeTrophees.remove(glaive2);
System.out.println(typeTrophees + ", " + typeTrophees.size());
//[bouclier  étincelant, casque usée], 2
```


Conclusion

- TreeSet ou HashSet ?
  - Pas besoin que les données soient triées : HashSet
    - plus rapide
    - pas besoin de définir un ordre (pas toujours évident)
  - Besoin que les données soient triées : TreeSet



- Inconvénient des set : nécessité de connaitre exactement l'élément

Classe dégénérée (1/5)

- Nous avons précisé que l'inconvénient d'un HashSet et qu'il fallait connaitre exactement l'élément recherché.
- Nous avons vu dans le cours précédent (exemple d'utilisation de l'interface NavigableSet), que lorsque nous n'avons pas l'objet nous pouvons utiliser un objet dégénéré.

```
Gaulois gaulois = ensemble.ceiling(new Gaulois("O", 1));
System.out.println(gaulois);
```

Objet dégénéré

- Si l'utilisation est occasionnelle un objet dégénéré suffit, mais si on s'en sert régulièrement (par exemple dans une méthode) alors il faut créer une classe dégénérée.

Classe dégénérée (2/5)

- Admettons que deux parchemins sont maintenant considérés identiques s'ils ont le même titre et le même auteur, indépendamment de la date.

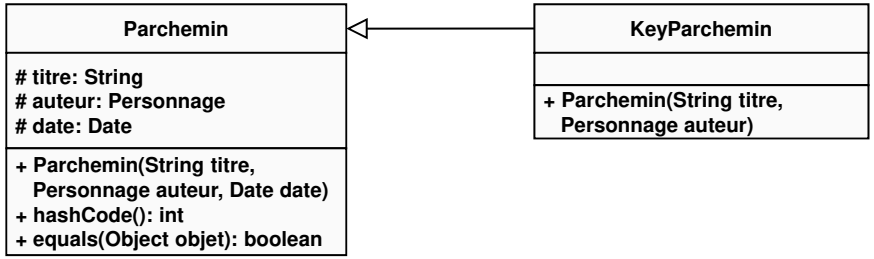
```
public class Parchemin {
 private String titre;
 private Personnage auteur;
 private Date date;
```

```
public boolean equals(Object obj) {
   if (obj instanceof Parchemin) {
     Parchemin parchemin = (Parchemin) obj;
     return (titre.equals(parchemin.titre)
         && auteur.equals(parchemin.auteur));
   }
   return false;
 }
```

```
public int hashCode() {
   return 31*(titre.hashCode() + auteur.hashCode());
 }
```

Classe dégénérée (3/5)

- Une classe dégénérée hérite de la classe que l'on veut imiter.




```
public class KeyParchemin extends Parchemin {
   public KeyParchemin(String titre, Personnage auteur) {
      super(titre, auteur, new Date(0,0,0));
   }
}
```


Classe dégénérée (4/5)

- Dans le musée nous ajoutons l'attribut parchemins de type HashSet.

```
Set<Parchemin> parchemins = new HashSet<>();
```

- Pour ajouter un parchemin on crée la méthode *ajouterParchemin*

```
public void ajouterParchemin(String titre, Personnage auteur,
                                                    Date date) {
   Parchemin parchemin = new Parchemin(titre, auteur, date);
   parchemins.add(parchemin);
}
```


Classe dégénérée (5/5)

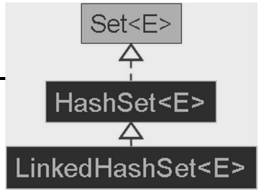
- Comment savoir si le parchemin « La guerre des Gaules » écrit par Jules César est dans l'ensemble ?
- Nous avons deux possibilités :
  - Utiliser un objet dégénéré comme nous avons appris à le faire dans le cours précédent.

```
public boolean existanceParchemin(String titre, Personnage auteur) {
   Parchemin parchemin = new Parchemin(titre, auteur, new Date(0,0,0));
   return parchemins.contains(parchemin);
   }
}
```

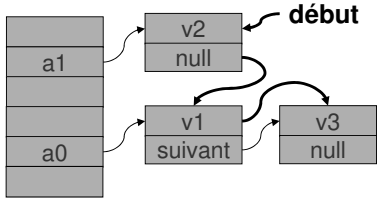
  - Utiliser une classe dégénérée

```
public boolean existanceParchemin(String titre, Personnage auteur) {
   Parchemin parchemin = new KeyParchemin(titre, auteur);
   return parchemins.contains(parchemin);
   }
}
```


LinkedHashSet<E>



- Affine le contrat de sa super-classe sur un seul point : garantit que les itérateurs retournent les éléments dans leur ordre d'insertion par une liste chaînée (en rouge).



- Exemple :

```
Set<Character> ens = new LinkedHashSet<>(6);
Collections.addAll(ens, 'a', 'j', 'b');
assert ens.toString().equals("[a, j, b]");
```

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 permettant de le retrouver.
- Structure : enregistrement de paires clé / valeur
- Exemple :
  Les lecteurs doivent pouvoir donner une appréciation sur 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 associer à chaque parchemin une appréciation.
- Structure : enregistrement de paires clé / valeur
- la clef -> le parchemin
  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'associations clé-valeur, dans lequel les clés sont uniques.

Methods

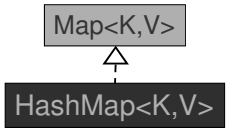
Modifier and Type	Method and Description
`void`	**`clear()`** Removes all of the mappings from this map (optional operation).
`boolean`	**`containsKey(Object key)`** Returns `true` if this map contains a mapping for the specified key.
`boolean`	**`containsValue(Object value)`** Returns `true` if this 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.

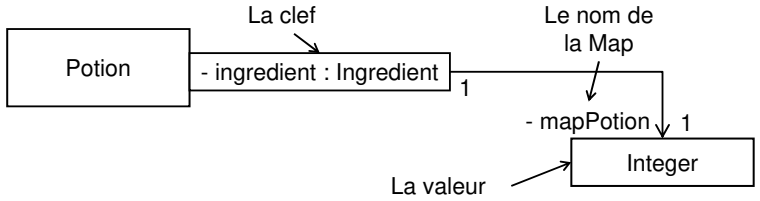
Les HashMap dans UML

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

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

Quantité

- Représentation UML



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 des 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 Parchemin :");
   for (Parchemin parchemin : vueParchemins) {
     System.out.println(parchemin);
    }
 }
```

Les vues d'une Map (3/3)

- Le scénario de test fonctionnel est le suivant :

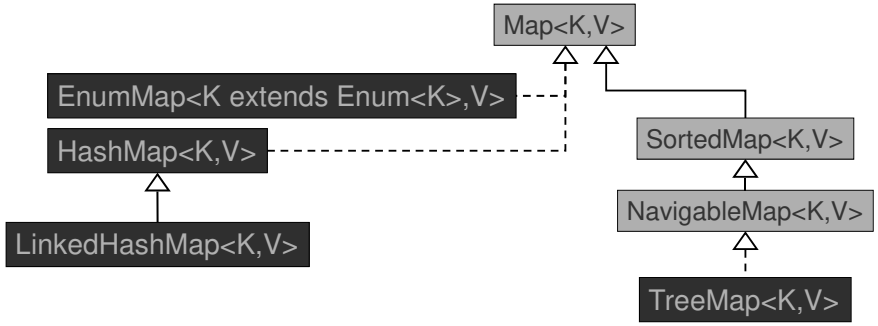
```
Set<Parchemin> vueParchemins = musee.donnerParchemins();
afficherVue(vueParchemins);
```
```
Vue parchemins :
```

```
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
```

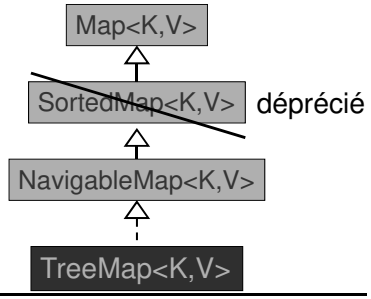

Autres Map

- Les classes concrètes ci-dessous implémentent l'interface Map. Elles ne sont pas exhaustives, mais ont été choisies car vous pouvez en déduire leur fonctionnement par leurs noms et vos connaissances.



TreeMap<K,V> (1/3)

- 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(NavigableMap<K, ? extends V> m)



TreeMap<K,V> (2/3)

- //Ajouter des données dans la TreeMap

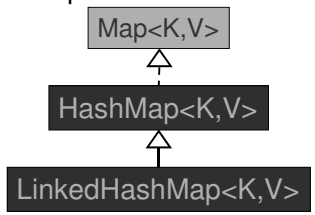
```
tMap.put(1, "Lundi");       tMap.put(2, « Mardi");
tMap.put(3, "Mercredi");    tMap.put(4, "Jeudi");
tMap.put(5, "Vendredi");    tMap.put(6, "Samedi");
tMap.put(7, "Dimanche");

System.out.println("Les clés sont : " + tMap.keySet());
//Les clés sont : [1, 2, 3, 4, 5, 6, 7]

System.out.println("Les valeurs sont : " + tMap.values());
//Les valeurs sont : [Lundi, Mardi, Mercredi, Jeudi, Vendredi,
//Samedi, Dimanche]

System.out.println("Clé : 5, Valeur : " + tMap.get(5)+ "\n");
//Clé : 5, Valeur : Vendredi
```

TreeMap<K,V> (3/3)

- System.*out*.println("Première clé : " + tMap.firstKey() + "
  Valeur : "
  + tMap.get(tMap.firstKey()) + "\n");
  //Première clé : 1 Valeur : Lundi
  System.*out*.println("Dernière clé : " + tMap.lastKey() + "
  Valeur : "
  + tMap.get(tMap.lastKey()) + "\n");
  //Dernière clé : 7 Valeur : Dimanche
  System.*out*.println("Retrait de la première donnée : " +
  tMap.remove(tMap.firstKey()));
  System.*out*.println("Les clés : " + tMap.keySet());
  System.*out*.println("Les valeurs: " + tMap.values() + "\n");
  //Retrait de la première donnée : Lundi
  //Les clés : [2, 3, 4, 5, 6, 7]
  //Les valeurs: [Mardi, Mercredi, Jeudi, Vendredi, Samedi,
  //Dimanche]

Cartes de hachage liées : LinkedHashMap

- Intérêt : se souvient de l'ordre dans lequel les paires clé / valeur sont insérées dans la carte.
  - Évite l'ordre aléatoire des clés dans *HashMap*.
  - Ne subit pas les dépendances d'un *TreeMap*.



- Fonctionnement : au fur et à mesure que les entrées sont insérées elles sont doublement chaînées dans une liste.
- Utilisation : les méthodes *keySet*, *entrySet* et *values* produisent des collections dont les itérateurs suivent les liens de cette liste.

EnumMap (1/2)

- Reprenons l'implémentation de la potion, nous souhaitons pouvoir lister les ingrédients dans l'ordre : indispensable, au choix puis optionnel.
- Pour cela nous allons créer une map qui aura pour clef l'ensemble des valeurs de l'énuméré et pour valeur une liste d'ingrédients

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

```
Map<Necessite, List<Ingredient>> listeIngredients =
                                    new EnumMap<>(Necessite.class);
```

Remarque :
Les méthodes equals et haschCode ne sont pas nécessaires dans un enum.

EnumMap (2/2)

- La méthode *ajouterIngredient* devient beaucoup plus simple !

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

- Dans le main nous aurons :

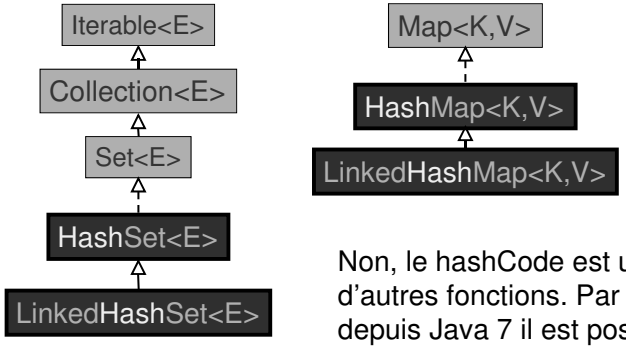
```
Ingredient fraises = new Ingredient("fraises");
Ingredient huile = new Ingredient("huile de roche");
…
Potion potion = new Potion();
potion.ajouterIngredient(fraises, Necessite.OPTIONNEL);
potion.ajouterIngredient(huile, Necessite.AU_CHOIX);
…
```

- L'affichage est clair :

```
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 une expression de 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 chaîne, 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;
                    …
```