L3 SRI - COO

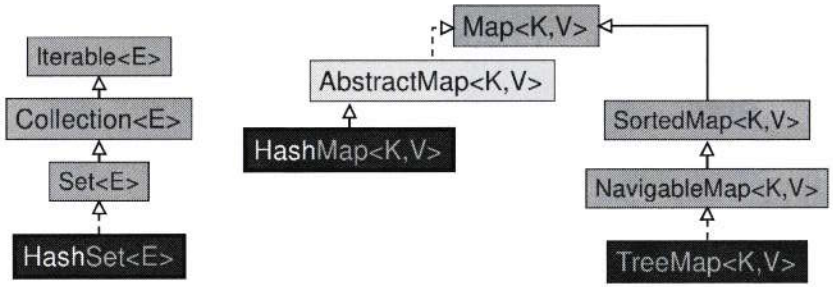
Cours 6 : Collections

Les tables de hachage
Les méthodes (equals, hashCode)

Auteur : CHAUDET Christelle

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* 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)

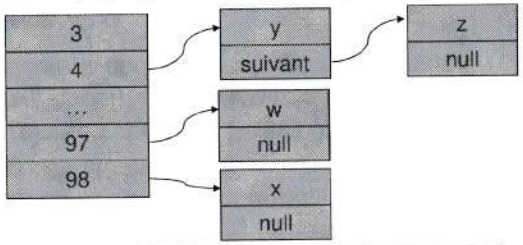
- Une table de hachage est constituée d'un tableau de listes chaînées. Chaque liste est appelée un **seau** (ou *panier* ou *bucket*).
- Pour trouver la place d'un élément il faut :
  - calculer son code de hachage,
  - réduire le résultat par un modulo du nombre total de seaux.
- Exemple :
  - String 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 (1/4)

- 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 (3/4)

- Reprenons l'exemple du musée, il contient des trophées correspondant aux équipements des romains rapportés au combat et sont dans un état plus ou moins correct.

```
public enum Etat {
   ETINCELANT("étincelant"), SALE("sale"), USE("usé"),
   CABOSSE("cabossé");

   private String name = "";

   Etat(String name) {
      this.name = name;
   }

   public String toString() {
      return name;
   }
}
```

Méthodes hashCode et equals (3/4)

- Le musée contient des trophées correspondant aux équipements des romains (casque, bouclier, glaive) rapportés au combat.

```
public abstract class Equipement {
  private final String NOM;
  private final Etat ETAT;

 public Equipement(final String NOM, final Etat ETAT) {
    this.NOM = NOM;
    this.ETAT = ETAT;
 }

  public String toString() {
    return NOM + "" + ETAT;
  }
}
```


Méthodes hashCode et equals (4/4)

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

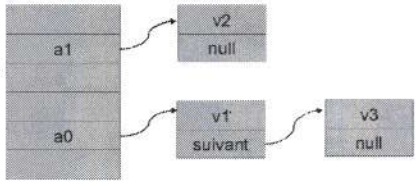
```
public boolean equals(Object objet) {
   if (objet instanceof Equipement) {
     Equipement equipementToCompare = (Equipement) objet;
     return NOM.equals(equipementToCompare.NOM);
   }
   return false;
  }
}
```

- La méthode *hashCode*

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

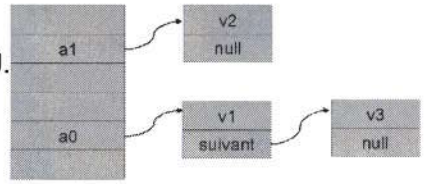

La classe HashSet<E>

- 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 leur contenu).
- 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.
    Ne vérifie les éléments que dans UN SEUL SEAU.
  - L'itérateur d'un set parcours TOUS LES SEAUX un par un dans un ordre aléatoire.



Exemple de HashSet (1/2)

```
public abstract class Equipement {
  private final String NOM;
  private final Etat ETAT;

  public Equipement(final String NOM, final Etat ETAT) {
    this.NOM = NOM;
    this.ETAT = ETAT;
  }

  public boolean equals(Object objet) {
    if (objet instanceof Equipement) {
      Equipement equipementToCompare = (Equipement) objet;
      return NOM.equals(equipementToCompare.NOM);
    }
    return false;
  }

  public String toString() {
    return NOM + "" + ETAT;
  }

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


Exemple de HashSet (2/2)

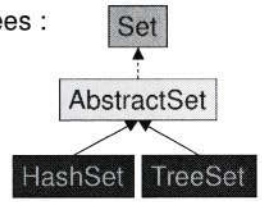
```
Glaive glaive1 = new Glaive(Etat.ETINCELANT);
Glaive glaive2 = new Glaive(Etat.SALE);
Casque casque = new Casque(Etat.USE);
Bouclier bouclier = new Bouclier(Etat.ETINCELANT);

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

System.out.println(typeTrophees +", "+typeTrophees.size());
//[bouclier étincelant, casque usé, glaive étincelant], 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 usé], 2
```


Conclusion

- TreeSet ou HashSet ?
  - Pas besoin que les données soient triées : HashSet
    - plus rapide
    - pas de définition d'ordre de tri (pas toujours évident)
  - Besoin que les données soient triées : TreeSet



- Inconvénient des set : nécessité de connaitre exactement l'élément, aucune méthode ne permet de récupérer un élément de l'ensemble.
- Propriété des set : ne contient pas de doublons.

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)

- Nous avons vu dans le musée que deux équipements sont considérés identiques s'ils ont le même nom indépendamment de son état.

```
public class Equipement {
 final private String NOM;
 final private Etat ETAT;
```

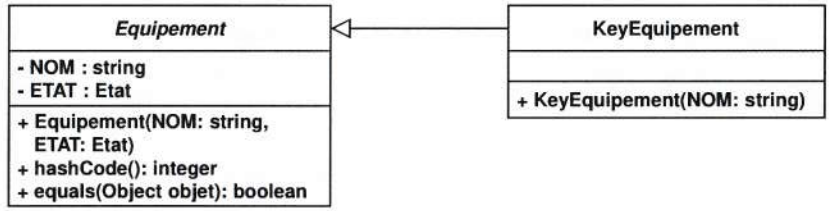
ni et ses enfants

```
public boolean equals(Object objet) {
  if (objet instanceof Equipement) {
    Equipement equipementToCompare = (Equipement) objet;
    return NOM.equals(equipementToCompare.NOM);
  }
  return false;
}
```

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


Classe dégénérée (3/5)

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




```
public class KeyEquipement extends Equipement {
   public KeyEquipement(String NOM) {
      super(NOM, Etat.ETINCELANT);
   }
}
```


Classe dégénérée (4/5)

- Précédemment nous avions créé l'ensemble typeTrophees de type « HashSet » et placé un glaive un casque et un bouclier.

```
Glaive glaive = new Glaive(Etat.ETINCELANT);
Casque casque = new Casque(Etat.USE);
Bouclier bouclier = new Bouclier(Etat.ETINCELANT);

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


Classe dégénérée (5/5)

- Comment savoir si un casque est dans l'ensemble ?
- Nous avons deux possibilités :
  - Utiliser un objet dégénéré comme nous avons appris à le faire dans le cours précédent.

```
Equipement casqueDegenere = new Casque(Etat.ETINCELANT);
System.out.println("Le musée possède-t-il un casque ? "
            + typeTrophees.contains(casqueDegenere));
```

  - Utiliser une classe dégénérée

```
Equipement equipementDegenere = new KeyEquipement("casque");
System.out.println("Le musée possède-t-il un casque ? "
            + typeTrophees.contains(equipementDegenere));
```