Introduction à la programmation en JAVA

technofuturTIC

WAVENET
Intelligent IT Solutions

Table des matières

technofuturTIC

© Wavenet 2015

WAVENET
Intelligent IT Solutions

Les collections et les énumérations

Aperçu du chapitre

I . Les collections

Les **collections** sont des objets utilisés pour stocker, récupérer et manipuler des ensembles de données.

A la différence des tableaux, les collections permettent de stocker un **nombre variable** d'objets de types homogènes et hétérogènes.

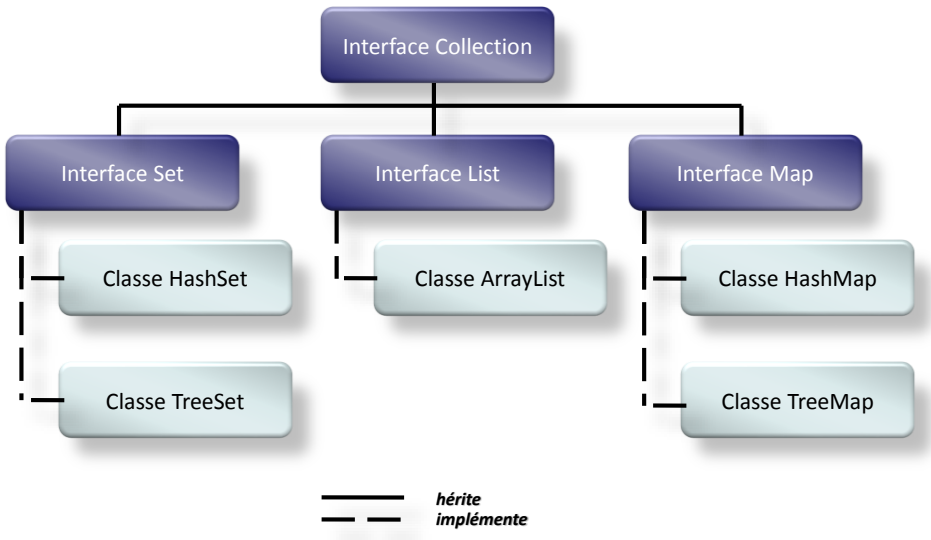
Trois types de collections sont importants :

- Les listes (**List**) : contiennent des objets accessibles séquentiellement.
- Les ensembles (**Set**) : contiennent des éléments non dupliqués dont l'accès reste très performant.
- Les maps (**Map**) : tableaux associatifs qui permettent d'associer un objet clé à un autre objet valeur. L'accès aux objets est donc effectué par une clé unique.

technofuturTIC

WAVENET Intelligent IT Solutions

I . Les collections – *Hiérarchie*



technofuturTIC

WAVENET Intelligent IT Solutions

Aperçu du chapitre

II . Interface Collection

Cette interface est implémentée par un certain nombre de collections et garantit que ces classes implémenteront l'ensemble des méthodes qu'elle contient.

Les principales méthodes sont :

boolean add (**Object** o)	Ajouter un objet à la collection
boolean remove (**Object** o) *	Retirer un objet de la collection
boolean contains (**Object** o) *	Tester si la collection contient l'objet indiqué
boolean addAll (**Collection** c)	Ajouter tous les objets d'une autre collection à celle-ci
boolean removeAll (**Collection** c) *	Retirer tous les objets d'une autre collection de celle-ci
boolean retainAll (**Collection** c) *	Retirer tous les objets qui ne sont pas dans la collection spécifiée de celle-ci.
boolean containsAll (**Collection** c) *	Tester si la collection contient tous les objets de la collection indiquée
void clear()	Vider la collection
boolean isEmpty()	Tester si la collection est vide
Iterator iterator()	Retourne un itérateur permettant de faire une boucle sur tous les objets contenus dans la collection
int size()	Retourne le nombre d'objets de la collection
Object[] toArray()	Convertit la collection en tableau d'objets
Object[] toArray (**Object**[] a)	Convertit la collection en tableau d'objets de classe spécifiée

* Pour garantir un fonctionnement correct, la méthode *equals* des objets de la liste doit être définie

II . Interface Collection

Attention, il ne faut pas confondre l'interface Collection avec la **classe Collections**.

Cette classe Collections contient des **méthodes statiques** qui manipulent ou retournent des collections :

`Collection sort(List c)`	Tri la collection selon l'ordre naturel de ses éléments
`Collection sort(List c, Comparator comp) **`	Tri la collection à l'aide du comparateur spécifié
`Collection reverse()`	Inverse l'ordre actuel des éléments
`int binarySearch (Collection c, Object o) *`	Retourne l'indice de l'élément recherché dans la collection
`int binarySearch (Collection c, Object o, Comparator comp) * **`	Retourne l'indice de l'élément recherché dans la collection en se basant sur un comparateur
`Object min(Collection c)`	Retourne le plus petit élément de la collection
`Object min(Collection c, Comparator comp) **`	Retourne le plus petit élément de la collection
`Object max(Collection c)`	Retourne le plus grand élément de la collection en se basant sur un comparateur
`Object max(Collection c, Comparator comp) **`	Retourne le plus grand élément de la collection en se basant sur un comparateur
`Collection shuffle(Collection c)`	Permute aléatoirement les éléments de la collection spécifiée

* Pour garantir un fonctionnement correct, la collection doit être triée avant l'utilisation de la méthode binarySearch
** La notion des objets Comparator sera abordée dans la partie dédiée aux ArrayList

Aperçu du chapitre

technofuturTIC
© Wavenet 2015
WAVENET Intelligent IT Solutions

III . Interface List

Cette interface est implémentée par un certain nombre de collections, et garantit que ces classes implémenteront l'ensemble de ses méthodes. Elle dérive de l'interface Collection.

Cette interface suppose que la collection **utilise un index pour adresser les objets** qu'elle contient (collection ordonnée).

Les principales méthodes ajoutées sont :

boolean add (**int** index, Object o)	Ajouter un objet à l'index indiqué
boolean addAll (**int** index, Collection c)	Ajouter tous les objets d'une autre collection à l'index indiqué
Object get (**int** index)	Retourne l'objet à l'index indiqué
int indexOf (**Object** o)	Retourne le premier index de l'objet indiqué
int lastIndexOf (**Object** o)	Retourne le dernier index de l'objet indiqué
Object remove (**int** index)	Supprime l'objet à l'index indiqué
Object set (**int** index, Object o)	Remplace l'objet à l'index indiqué
List subList (**int** fromIndex, **int** toIndex)	Retourne une sous-liste de celle-ci

technofuturTIC

WAVENET Intelligent IT Solutions

Aperçu du chapitre

technofuturTIC

© Wavenet 2015

WAVENET Intelligent IT Solutions

IV . Classe ArrayList

Cette classe représente un **tableau dont la taille dont la taille peut varier dynamiquement**.

Cette classe **implémente** les méthodes des interfaces **List** et **Collection**.

Il existe 3 méthodes pour parcourir les éléments d'une ArrayList :

1. Via une boucle **for**

```
for (int i = 0 ; i < liste.size() ; i ++) {
        Object elt = liste.get(i);
        …
}
```


IV . Classe ArrayList

2. Via une boucle **for each**

```
for (Object o : liste) {
        Object elt = o;
        …
}
```

3. Via un **itérateur**

```
Iterator iterateur = liste.iterator();
while (iterateur.hasNext()) {
        Object elt = iterateur.next();
        …
}
```

IV . Classe ArrayList

Une ArrayList **non générique** peut contenir n'importe quoi. On parle de liste **hétérogène**.
Dans ce cas, toutes les méthodes retournent des objets de type **Object** (déconseillé).

```
ArrayList list = new ArrayList();
Iterator it = liste.iterator();
```

Dans une ArrayList **générique**, il faut spécifier le type des objets qu'on place dans la liste. On obtient ainsi une liste **homogène**.
Dans ce cas, toutes les méthodes retournent des objets du **type spécifié** (+ de contrôle).

```
ArrayList<Personne> listePersonnes = new ArrayList<Personne>();
Iterator<Personne> it = listePersonnes.iterator();
```


IV . Classe ArrayList – Comparable et Comparator

L'interface **Comparable<T>** est une interface contenant la déclaration de la méthode ***compareTo***.

On l'utilise seule quand on a besoin d'**un seul critère de comparaison**.

La méthode ***compareTo*** permet de définir l'**ordre naturel** des instances d'une classe.

Au cas où on besoin de **plusieurs critères** de comparaisons, on utilise des **comparateurs**.

Pour cela, il faut créer **une classe** par comparateur souhaité.

Une classe comparateur implémente l'interface **Comparator<T>** qui impose de donner une définition à la méthode ***compare***.

IV . Classe ArrayList – Comparable et Comparator

Les méthodes ***compareTo*** et ***compare*** ont un comportement semblable et renvoie un **nombre entier** comme résultat de la comparaison :

- Nombre négatif : le 1e élément est plus petit que le 2e
- 0 : les deux éléments sont identiques
- Nombre positif : le 1e élément est plus grand que le 2e

IV . Classe ArrayList – Comparable et Comparator

Exemple avec **Comparable<T>**

```
public class Personne implements Comparable<Personne> {
        …
        public int compareTo(Personne p) {
                return (
                        this.getNom().compareTo(p.getNom()) == 0) ?
                                this.getPrenom().compareTo(p.getPrenom()) :
                                this.getNom().compareTo(p.getNom())
                );
        }
}
```

IV . Classe ArrayList – Comparable et Comparator

Exemple avec **Comparator<T>**

```
public class ComparateurPersonne implements Comparator<Personne> {
        public int compare(Personne p1, Personne p2) {
                return (
                        p1.getNom().compareTo(p2.getNom()) == 0) ?
                                p1.getPrenom().compareTo(p2.getPrenom()) :
                                p1.getNom().compareTo(p2.getNom())
                );
}
```


IV . Classe ArrayList – Exercices

Créez une classe Cycliste qui possède les attributs suivants : classement (int), nom (String), prenom (String).
Cette classe implémente l'interface Comparable et l'ordre naturel est basé sur le classement.

Créez ensuite une classe Course qui possède les attributs suivants : nom (String), classement (ArrayList).
Cette classe Course possède les méthodes suivantes :

- Course(String)
- ajouterCycliste(Cycliste) throws DoublonException
- suppressionCycliste(Cycliste)
- remplacerCycliste(int, String, String)
- getPremier()
- getDernier()
- trierParNom()
- trierParPrenom()

Aperçu du chapitre

V . Interface Set

Cette interface est implémentée par un certain nombre de collections, et garantit que ces classes implémenteront l'ensemble de ses méthodes.

Elle dérive de l'interface Collection, **sans ajouter de nouvelles méthodes**.

Elle sert seulement à indiquer informellement que la collection implémentant cette interface ne contient **aucun doublon** d'objet (objets comparés par la méthode ***equals***).

De plus, elle ne conserve pas l'ordre d'ajout des éléments.

Cette interface correspond donc aux ensembles mathématiques

Aperçu du chapitre

VI . Classe HashSet

La classe **HashSet** implémente l'interface Set en utilisant une **table de hachage** sans ajouter de nouvelles méthodes.

Pour **parcourir** une HashSet, on utilise un **for each** ou un **itérateur**.

La fonction ***hashCode*** est utilisée pour retourner le **code de hachage**, c'est-à-dire un entier dérivé de l'objet.

La HashSet répartit ses objets en utilisant la formule suivante :

codeDeHachage % nombreElements

VI . Classe HashSet

L'avantage du code de hachage est de permettre un **accès direct** au élément et non un accès séquentiel comme pour les listes.

Le contrat de la méthode ***hashCode*** est le suivant :

- Cohérence : tant que l'état de l'objet ne change pas, la méthode renvoie toujours la même valeur
- Si x.equals(y) retourne true alors x.hashCode() == y.hashCode()

Remarque : il n'est pas impossible pour deux objets différents de retourner le même code.

Par défaut, le code de hachage renvoyé est déduit de l'adresse mémoire de l'objet.

Dans le cas où la méthode ***equals*** a été redéfinie, il est obligatoire de redéfinir la méthode ***hashCode***!

VI . Classe HashSet

Implémentation de ***hashCode***:

1. Initialiser le code de hachage (int code = …;)

2. Pour chaque champ utilisé dans la méthode equals, attribuer un code :

boolean :	champ ? 1 : 0
byte, short, char, int :	(int) champ
long :	champ^(champ >>> 32)
float :	Float.floatToIntBits(champ)
double :	Double.toLongBits(champ)
référence :	champ.hashCode()
tableau :	traiter chaque élément comme un champ

3. Combiner tous les codes obtenus : result = 37 * result + code

VI . Classe HashSet

```
public class Address {

   private String street;
   private String zipcode;
   private String city;

   public int hashCode() {
            int result = 17;

            result = 37 * result +
                     (street == null ? 0 : street.hashCode());
            result = 37 * result +
                     (zipcode == null ? 0 : zipcode.hashCode());
            result = 37 * result +
                     (city == null ? 0 : city.hashCode());

            return result;
   }
}
```


VI . Classe HashSet – Exercices

Reprenez l'exercice sur les cyclistes et adaptez-le pour remplacer les ArrayList par des HashSet

Aperçu du chapitre

VII . Classe TreeSet

La classe **TreeSet** garantit un **ordonnancement des éléments** selon un ordre croissant, en accord avec l'ordre naturel des éléments ou à l'aide d'un comparateur fourni au moment de la création d'une instance.

La classe TreeSet implémente l'interface Set et ajoute les méthodes suivantes :

T first()	Retourne le premier élément
T last()	Retourne le dernier élément
TreeSet<T>descendingSet ()	Retourne un TreeSet en inversant l'ordre des éléments
TreeSet<T> subSet(T t1, T t2)	Retourne un TreeSet de t1 compris jusqu'à t2 non compris
Object higher(T t)	Retourne l'élément suivant t
Object lower(T t)	Retourne l'élément précédent t

VII . Classe TreeSet – Exercices

Reprenez l'exercice sur les cyclistes et adaptez-le pour remplacer les ArrayList par des TreeSet

Aperçu du chapitre

© Wavenet 2015

VIII . Interface Map

Cette interface est implémentée par les collections qui **associent une clé à un objet**. L'accès aux objets est donc effectué par une clé unique.

Les principales méthodes de cette interface sont :

void clear()	Vider la collection
boolean containsKey (**Object** key)	Teste si la clé existe
boolean containsValue (**Object** value)	Teste si la valeur existe
Set entrySet()	Retourne l'ensemble des associations clés-valeurs
Set keySet()	Retourne l'ensemble des clés
Collection values()	Retourne la collection de valeurs
Object put (**Object** key, **Object** value)	Associe la clé à la valeur spécifiée, et retourne la valeur précédemment associée
boolean putAll (**Map** m)	Ajouter tous les objets d'une autre map à celle-ci
Object get (**Object** key)	Retourne la valeur associée à la clé spécifiée
Object remove (**Object** key)	Supprime l'objet associé à la clé, et retourne cet objet

Aperçu du chapitre

technofuturTIC
© Wavenet 2015
WAVENET Intelligent IT Solutions

IX . Classes HashMap

La classe **HashMap** implémente l'interface **Map**.

Comme pour la classe **HashSet**, elle se base sur un **code de hachage** pour stocker ses éléments.

Les métodes ***equals*** et ***hashCode*** doivent porter sur les **clés** et non sur les valeurs.

Parcours d'une hashMap :

```
Set<Map.Entry<K, V>> entrees = hashMap.entrySet();
Iterator<Map.Entry<K, V>> it = entrees.iterator();
while (it.hasNext()) {
        Map.Entry<K, V> entree = it.next();
        K cle = entree.getKey();
        V valeur = entree.getValue();
        …
}
```


IX . Classe HashMap – Exercices

Reprenez l'exercice sur les cyclistes et adaptez-le pour remplacer les ArrayList par des HashMap

Aperçu du chapitre

X . Classes TreeMap

La classe **TreeMap** implémente l'interface **Map**.

Cette classe a un comportement semblable à celui des **TreeSet**.

La méthode ***equals*** doit porter sur les **clés** et non sur les valeurs.

Le **tri** se base également sur les **clés**.

Le parcours d'une **TreeMap** est identique à celui d'une **HashMap**.

X . Classe TreeMap – Exercices

Reprenez l'exercice sur les cyclistes et adaptez-le pour remplacer les ArrayList par des TreeMap

Aperçu du chapitre

© Wavenet 2015

XI . Les énumérations

Cette **structure** permet de contenir une **série de données constantes** ayant un type sûr, ce qui veut dire que **ni le type**, **ni la valeur réelle** de chaque constante **n'est précisé**.

Il est possible de tester si une valeur ayant le type de l'énumération est égal à une des valeurs de l'énumération à l'aide d'une commande **switch**.

```
enum Animal {KANGOUROU, TIGRE, CHIEN, SERPENT};

public class Test {
        public static void main(String[] args) {
                String aniMsg;
                Animal bebete = Animal.TIGRE;
                switch(bebete) {
                        case KANGOUROU : aniMsg = "kangourou"; break;
                        case TIGRE : aniMsg = "tigre"; break;
                        case CHIEN : aniMsg = "chien"; break;
                        case SERPENT : aniMsg = "serpent"; break;
                }
                System.out.println("L'animal est un " + aniMsg);
        }
}
```

technofuturTIC

WAVENET Intelligent IT Solutions

Exercices

1. Créer classe Personne avec nom, prénom, adresse, téléphone.

Créer une classe Agenda qui peut contenir un ArrayList de Personne.

Créer un programme de gestion de l'agenda (classe Main) avec des opérations CRUD (le nom et le prénom restent fixes).

Attention, les doubles sont interdits!

Il doit être également possible d'afficher l'ensemble des Personnes triées par nom (Comparable) ou par prénom (Comparator). L'affichage sera fera à l'aide d'un Iterator.

Attention aux exceptions!

technofuturTIC

WAVENET Intelligent IT Solutions

Exercices

2. Modifier l'exercice précédent en remplaçant l'attribut de type ArrayList<Personne> par un attribut de type HashMap<Personne, ArrayList<RendezVous>>.

Un RendezVous est composé d'un lieu, d'une date et d'un motif.

Les RendezVous d'une Personne doivent pouvoir être gérés avec des opérations CRD (pas d'update!).

Attention, une Personne ne peut pas avoir deux RendezVous identiques!

Il doit être possible d'afficher la liste des RendezVous d'une personne recherchée dans l'Agenda.

Les RendezVous seront triés par ordre croissant selon le temps séparant le RendezVous d'aujourd'hui. Les RendezVous passés seront affichés en fin de liste.

technofuturTIC

WAVENET Intelligent IT Solutions