INTRODUCTION AU LANGAGE (SUITE)

Moussa LO

UFR de Sciences Appliquées et de Technologie

Université Gaston Berger de Saint-Louis

COLLECTIONS

Limites des tableaux (1/2)

- Simples à programmer et peu gourmands en mémoire, les tableaux sont utilisés pour gérer des ensembles dont la taille est connue à l'avance.

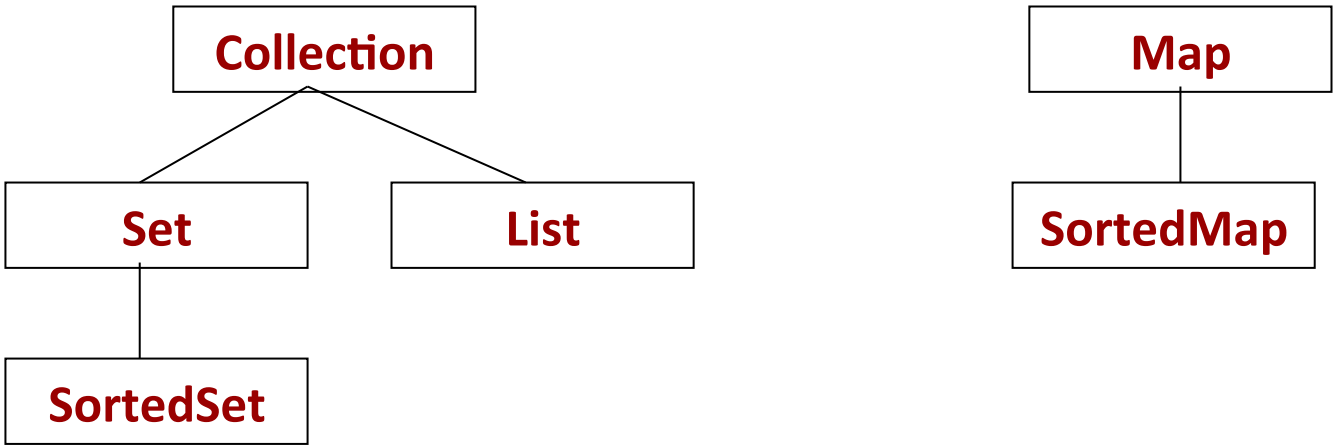
- Les tableaux sont inadéquats pour gérer une quantité importante d'informations du même type lorsque leur nombre n'est pas connu à l'avance.

Limites des tableaux (2/2)

- Les tableaux ne sont pas redimensionnables.
- L'insertion d'un élément au milieu d'un tableau n'est pas aisée.
- La recherche d'un élément dans un grand tableau est longue s'il n'est pas trié.
- Les indices pour accéder aux éléments d'un tableau doivent être entiers.

Collections Java

- Le package ***java.util*** fournit un ensemble d'interfaces et de classes facilitant la manipulation de collections d'objets.
- Une **collection** est un objet qui sert à stocker d'autres objets.
- Interfaces sur les collections organisées en deux catégories : Collection et Map.



Collections Java : interfaces

- **Collection** : groupe d'objets où la duplication peut être autorisée.

L'interface Collection spécifie des méthodes comme add, remove, addAll, removeAll, contains, containsAll, size, etc.

- **Set** : objets collections non ordonnées et sans doublons. SortedSet est un Set trié.

- **List** : collections ordonnées autorisant des doublons. Chaque objet possède une position dans la séquence et l'interface offre des méthodes permettant l'accès direct à un objet donné.

- **Map** : tableaux associatifs (dictionnaires) permettant de retrouver une information grâce à un index. SortedMap est un Map trié.

Collections Java : implémentations

	Classes d'implémentation				
		Table de Hachage	Tableau (Vecteur)	Arbre	Liste chaînée
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

Les classes du package java.util

- **ArrayList** : pour ajouter à la suite les uns des autres des éléments dans un ensemble ordonné.

- **LinkedList** : pour insérer de nombreux éléments au milieu d'un ensemble ordonné.

- **HashSet** : pour gérer un ensemble dont chaque élément doit être unique. Les performances de recherche sont améliorées.

- **TreeSet** : pour gérer un ensemble trié d'objets uniques.

- **HashMap** : pour accéder aux éléments d'un ensemble grâce à une clé.

- **TreeMap** : pour gérer un ensemble d'éléments trié dans l'ordre de leur clé d'accès.

Collections Java : Itérateurs

- Les collections peuvent être parcourues à l'aide d'itérateurs : interface Iterator.

- *Exemple :*

```
import java.util.*;
public class test_iterateur {
    public static void main (String[] args) {
                Set ufrs = new TreeSet();
                ufrs.add("sat"); ufrs.add("seg"); ufrs.add("sjp");
                // iterer
                Iterator iter = ufrs.iterator();
                while (iter.hasNext()) {
                    String ufr = (String) iter.next();
                    System.out.println(ufr);
                }
        }
}
```

La boucle "for each" et les collections

La boucle « *for each* » permet le parcours d'une instance d'une classe implémentant l'interface *java.util.Iterable*.

```
// affichage des elements d'un ensemble
    Set ufrs = new TreeSet();
    ufrs.add("sat"); ufrs.add("lsh");
    ufrs.add("seg"); ufrs.add("2s");
    ufrs.add("sjp"); ufrs.add("s2ata");

    for (Object s : ufrs)
        System.out.print(s + "  ");
// s va prendre successivement les valeurs contenues dans ufrs
```

Collections Java : Map

- Les objets Map permettent d'établir une correspondance entre deux classes d'objets (*par exemple des termes anglais et français*).

- Ensemble de paires (clé, valeur) tq l'ensemble des clés est sans doublons.

- L'interface interne Entry permet de manipuler les éléments d'une paire au moyen des méthodes :

  ✓ getKey et getValue : retournent respectivement la clé et la valeur associée à cette clé.

  ✓ setValue : permet de modifier la valeur d'une paire.

Collections Java : Map

L'interface ***Map*** offre des méthodes permettant d'itérer sur les clés, les valeurs et sur les paires:

```
Map m = new HashMap();
// sur les clés
for (Iterator i = m.keySet().iterator(); i.hasNext();)
    System.out.println(i.next());
 // sur les valeurs
for (Iterator i = m.values().iterator(); i.hasNext();)
    System.out.println(i.next());
// sur les paires clé/valeur
for (Iterator i = m.EntrySet().iterator(); i.hasNext();) {
    Map.Entry e = (Map.Entry)i.next();
    System.out.println(e.getKey() + "/" + e.getValue());
}
```

Exemple : gérer un glossaire avec HashMap

```
import java.util.HashMap;
import javax.swing.JOptionPane;

public class Glossaire {
    public static void main (String arg[]){
            String definitionInstance = "Objet cree a partir d'une classe";
            String definitionCollection = "Instance d'une classe gerant un
ensemble d'elements";
            String definitionSousClasse = "Classe heritant d'une autre classe";

            HashMap glossaire = new HashMap();
            glossaire.put("instance",definitionInstance);
            glossaire.put("collection",definitionCollection);
            glossaire.put("sous classe",definitionSousClasse);
            glossaire.put("classe derivee",definitionSousClasse);

 ...
```

Exemple : gérer un glossaire avec HashMap

```
while (true){
   String motCherche = JOptionPane.showInputDialog("Que cherchez vous ?");
   if (motCherche == null) System.exit(0);
   String definition = (String)glossaire.get(motCherche);
   if (definition != null)
            JOptionPane.showMessageDialog(null,
                     motCherche + " : " + definition,
                     "Resultat de la recherche",
                     JOptionPane.INFORMATION_MESSAGE);
   else
            JOptionPane.showMessageDialog(null,
                     motCherche + " : " + "non defini",
                     "Resultat de la recherche",
                     JOptionPane.WARNING_MESSAGE);
  }
  }
}
```

Généricité

✓On peut définir et utiliser des classes et des méthodes génériques.

Classes génériques

Exemple : gérer un couple d'objets

```
public class Couple<A>{
        A elt1, elt2;
        Couple(A elt1, A elt2) {
                this.elt1 = elt1;
                this.elt2 = elt2;
        }
        public static void main(String args[]){
           Couple<String> c = new Couple<String>("Adja","20");
           String nom = c.elt1;
           Integer age1 = Integer.parseInt(c.elt2);
           Integer age2 = (Integer)c.elt2;
           // ne passe pas à la compilation
        }
}
```

Classes génériques

```
...
        public static void main(String args[]){
            Couple<String> c = new Couple<String>("Adja","20");
            String nom = c.elt1;
            Integer age1 = Integer.parseInt(c.elt2);
            Integer age2 = (Integer)c.elt2;
            // ne passe pas à la compilation
        }
}
```

Généricité et collections

☞ **Présente encore plus d'intérêt avec l'utilisation des collections.**

• Toutes les interfaces et les classes collections ont une forme paramétrée (*depuis la version Java 5*):

- Collection<E>,
- Vector<E>,
- Iterator<E>,
- etc.

Généricité et collections

Exemple 1:

```
public class Ufrs{
        public static void main(String arg[]){
            ArrayList<String> ufrs = new ArrayList<String>();
            ufrs.add("sat"); ufrs.add("lsh");
            ufrs.add("seg"); ufrs.add("2S");
            ufrs.add("sjp"); ufrs.add("S2ATA");
            for (String ufr : ufrs)
                System.out.print(ufr + "  ");
            System.out.println();
        }
}
```

Généricité et collections

Exemple 2 :

```
public class Frequence {
    public static void main(String[] args) {
        Map<String, Integer> tab = new TreeMap<String, Integer>();
        for (String mot : args) {
                Integer freq = tab.get(mot);
                tab.put(mot, freq == null ? 1 : freq + 1);
        }
        System.out.println(tab);
    }
}
```

Méthodes génériques

✓Les méthodes peuvent elles aussi dépendre d'une liste de types paramètres.

qualifieurs <*types-param*> type-du-résultat nom-méthode (*args*)

Exemple 1:

```
import java.util.*;

public class MethodesGeneriques{
    static <A> void ecrire(A a){
            System.out.println(a);
    }

    …
```

Méthodes génériques

```
        …

    public static void main(String args[]){
            Scanner entree = new Scanner(System.in);
            System.out.println("Donner votre nom:");
            String nom = entree.nextLine();
            ecrire(nom);
            System.out.println("Donner votre age:");
            int age = entree.nextInt();
            ecrire(age);

    }
}
```

Méthodes génériques

Exemple 2:

```
static <T> void echangerGenerique(CoupleGen<T> c){
        T tmp;
        tmp = c.elt1;
        c.elt1 = c.elt2;
        c.elt2 = tmp;
}

public static void main(String args[]){
        CoupleGen<String> mots =
                new CoupleGen<String>("club","info");
        echangerGenerique(mots);
        ecrire(mots);
}
...
```

Méthodes génériques

```
…
class CoupleGen<A>{
        A elt1, elt2;

        CoupleGen(A elt1, A elt2){
                this.elt1 = elt1;
                this.elt2 = elt2;
        }

        public String toString(){
                return "(" + elt1 + "," + elt2 + ")";
        }
}
```

Méthodes avec nombre d'arguments variable

Méthodes avec nombre d'arguments variable

✓ Permet d'appeler une même méthode avec une liste d'arguments variable.

Exemple 1 :

```
public class MultiArgs {
        static void ecrireLesMots (String ... mots){
                for (String mot : mots)
                        System.out.print(mot + "  ");
                System.out.println();
        }
        public static void main(String arg[]){
                ecrireLesMots("UGB UFR SAT");
                ecrireLesMots("MIAGE INFO");
        }
}
```

Méthode avec nombre d'arguments variable

Exemple 2 :

```
public class MultiArgs {
    static double moyenne(String nom, Number... notes) {
        int nombre = notes.length;
        if (nombre > 0) {
            double somme = 0;
            for (Number x : notes)
                somme += x.doubleValue();
            return somme / nombre;
        }
        return -1;
        }
        public static void main(String arg[]){
            double moy = moyenne("Amy",10, 12.4, 18);
            System.out.println(moy);
        }
}
```

Importation de membres statiques

Importation de membres statiques

- ✓ Classiquement, on n'importe que des noms de classes.
- ✓ **On peut aussi importer des membres statiques.**

Exemple :

```
import static java.lang.System.out;
import static java.lang.Math.random;

public class TestImportStatic {
    public static void main(String arg[]){
            out.println("voici un nb aleatoire :"+random());
    }
}
```

ARCHIVES JAR

Fichiers d'archives JAR : définition

- Java Archive File
- Fichiers d'extension « **.jar** » qui regroupent un ensemble de ressources s'utilisant dans des programmes Java.
- Archives compressés au format « *rar* »
- Les archives java comportent essentiellement
  - ✓ des classes Java compilées « *.class* »,
  - ✓ et éventuellement toutes sortes de ressources (images, fichiers de configuration, etc.)

Fichiers d'archives JAR : utilité

- Permettent de déployer et de distribuer très facilement des **bibliothèques** de classes java.

- L'ajout à l'archive d'un fichier de configuration particulier, *appelé* ***manifeste***, permet de rendre l'archive exécutable (***java –jar archive***).

- Le manifeste doit alors contenir les informations concernant la classe à exécuter lors du lancement de l'archive.

Création d'une archive JAR

- Exécuter la commande

jar *cvf* nom_archive liste_des_fichiers

- Les différents fichiers de la liste sont séparés par des espaces.
- L'option « **c** » crée une nouvelle archive.
- L'option « **f** » permet de spécifier le nom de l'archive.
- L'option « **v** », génère un affichage sur la sortie standard.
- *Exple* : jar cvf essai.jar f1.class f2.class

Extraction et Exécution d'une archive JAR

- Extraction du contenu d'une archive JAR :

jar xf nom_archive [liste_des_fichiers_à_extraire]

- Exécution d'une archive JAR :

java [options] –jar archive_java.jar [classe_principale]

Ajout d'un manifeste à une archive JAR

- **Manifeste** : fichier texte qui permet, entre autres, de spécifier la classe qui sera exécutée lors du lancement d'une archive Java.

- Définition du manifeste :

  ✓écrire un fichier texte

  ✓et le passer en paramètre lors de la création de l'archive en utilisant l'option « m ».

jar cvmf nom_manifeste nom_archive liste_des_fichiers

- Créé sous le nom *MANIFEST.MF* dans le répertoire *META-INF* situé à la racine de l'archive.

Ajout d'un manifeste à une archive JAR

- Le manifeste comporte un renseignement par ligne.

- Ligne = type de renseignement : valeur

- *Rq :* Chaque ligne, pour être valide, doit obligatoirement se terminer par un caractère de retour à la ligne.

Ajout d'un manifeste à une archive JAR

-**Main-Class**: "nom de la classe"

-**Implementation-Title** : "titre du package"

-**Implementation-Version** : "n° de version "

-**Implementation-Vendor**: "organisation vendant le produit"

-**Specification-Title** : "titre de la spécification"

-**Specification-Version** : "n° de version "

-**Specification-Vendor** : "organisation vendant le produit"

Annotations

Annotations

- ✓ Métadonnées permettant :
  - d'ajouter des données sémantiques au code.
  - de préciser la façon dont ces données doivent être traitées.
  - à certains outils de générer des constructions additionnelles à la compilation ou à l'exécution ou de renforcer un comportement voulu au moment de l'exécution.

Annotations

- ✓ Exemple :

```
@Entity
public class Personne {
        @Id
        @GeneratedValue
        private int id;
        ...
}
```

Annotations

✓ Contrairement aux commentaires JavaDoc, les annotations ne disparaissent pas lors de la compilation.

✓ Sont reconnues et traitées par le compilateur et sont généralement conservées avec les classes produites.

Déclaration d'une annotation

Syntaxe :

```
public @interface nomTypeAnnotation{

        // déclaration des éléments de l'annotation
        type nomElement(); // ou type nomElement
default valeur;
}
```

Exemple :

```
public @interface Auteur{
    String nom();
    String prenom();
}
```

Utilisation d'une annotation

```
@Auteur(nom="Lo", prenom="Moussa")

public void methodeAnnotee(){

                    // corps de la methode

}
```

Exploitation d'une annotation

```
public static void main(String arg[]){

   try{
       Auteur auteur = Class.forName("CodeAnnote").
                                   getMethod("methodeAnnotee").
                                   getAnnotation(Auteur.class);

       System.out.println(auteur.prenom()+"  "+auteur.nom());
    }

    catch(Exception e){
         System.out.println(e.getMessage());
    }
}
```

Quelques annotations prédéfinies

✓**@Deprecated** : le compilateur lancera un avertissement lors de tout emploi d'un membre ainsi annoté.

✓**@Overrides** : indique que la méthode ainsi annotée doit redéfinir une méthode héritée; sinon le compilateur lance un message d'erreur.

✓**@Retention** : définit la durée de vie de l'annotation : compilation (valeur SOURCE), bytecode (CLASS) ou exécution (RUNTIME).

✓**@Target** : déclare les possibilités d'utilisation de l'annotation : type (valeur TYPE), champ (FIELD), méthode (METHOD), paramètre (PARAMETER), etc.

Annotations :exemple

- ✓ **Types d'Annotation**

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)

public @interface Auteur{
        String nom();
        String prenom();
}
```

Annotations :exemple

- ✓ **Types d'Annotation**

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)

public @interface Copyright{
     String value();
}
```

Annotations :exemple

✓Utilisation d'Annotations

```
public class CodeAnnote{
  @Auteur(nom="Lo", prenom="Moussa")
  @Copyright("@Copyright UFR SAT 2008")
   public void methodeAnnotee(){            // corps de la methode    }
    public static void main(String arg[]){
       try{
            Auteur auteur =

Class.forName("CodeAnnote").getMethod("methodeAnnotee").getAnnot
ation(Auteur.class);
            Copyright cr =
Class.forName("CodeAnnote").getMethod("methodeAnnotee").getAnnot
ation(Copyright.class);
            System.out.print(auteur.prenom()+"  "+auteur.nom()+"
"+cr.value());
          }
          catch(Exception e){ System.out.println(e.getMessage()); }
        }
```