Tables des Matières

- Chapitre 1 : Généralités, Classe
- Chapitre 2 : Héritage, classe abstraite,interface
- Chapitre 3 : **Collection**
- Chapitre 4 : Exceptions
- Chapitre 5 : Interface graphique
- Chapitre 6 : Threads

Collection

- La programmation objet pose le problème suivant :

Collection

- La programmation objet pose le problème suivant :
  - comment stocker,

Collection

- La programmation objet pose le problème suivant :
  - comment stocker,
  - classer, manipuler les collections d’objets,

Collection

- La programmation objet pose le problème suivant :
  - comment stocker,
  - classer, manipuler les collections d’objets,
  - comment comparer des objets entre eux.

Collection

- La programmation objet pose le problème suivant :
  - comment stocker,
  - classer, manipuler les collections d'objets,
  - comment comparer des objets entre eux.

- Les tableaux peuvent contenir des références à des objets

Collection

- La programmation objet pose le problème suivant :
  - comment stocker,
  - classer, manipuler les collections d'objets,
  - comment comparer des objets entre eux.
- Les tableaux peuvent contenir des références à des objets
- Inconvénient des tableaux :
  - tous les éléments sont de même type

Collection

- La programmation objet pose le problème suivant :
  - comment stocker,
  - classer, manipuler les collections d'objets,
  - comment comparer des objets entre eux.
- Les tableaux peuvent contenir des références à des objets
- Inconvénient des tableaux :
  - tous les éléments sont de même type
  - le nombre d'éléments est décidé à sa création.

Collection : package java.util

- **Java propose les collections** : ce sont des objets qui contiennent des références d'un ensemble d'objets.

Collection : package java.util

- **Java propose les collections** : ce sont des objets qui contiennent des références d'un ensemble d'objets.

  1. les vecteurs dynamiques : **Vector**, **ArrayList**
  2. les listes chaînées :**LinkedList**,
  3. les ensembles : **HashSet** et **TreeSet**.

Collection : package java.util

- **Java propose les collections** : ce sont des objets qui contiennent des références d'un ensemble d'objets.
  1. les vecteurs dynamiques : **Vector**, **ArrayList**
  2. les listes chaînées :**LinkedList**,
  3. les ensembles : **HashSet** et **TreeSet**.
- **Exploiter les concepts communs :**
  1. généricité, itérateur,
  2. ordonnancement et relation d'ordre.

Collection : package java.util

- **Java propose les collections** : ce sont des objets qui contiennent des références d'un ensemble d'objets.
  1. les vecteurs dynamiques : **Vector**, **ArrayList**
  2. les listes chaînées :**LinkedList**,
  3. les ensembles : **HashSet** et **TreeSet**.
- **Exploiter les concepts communs :**
  1. généricité, itérateur,
  2. ordonnancement et relation d'ordre.
- **Opérations communes sur les objets :**
  1. ajout, suppression, tri
  2. construction à partir d'un élément et d'une collection.

Framework des Collections

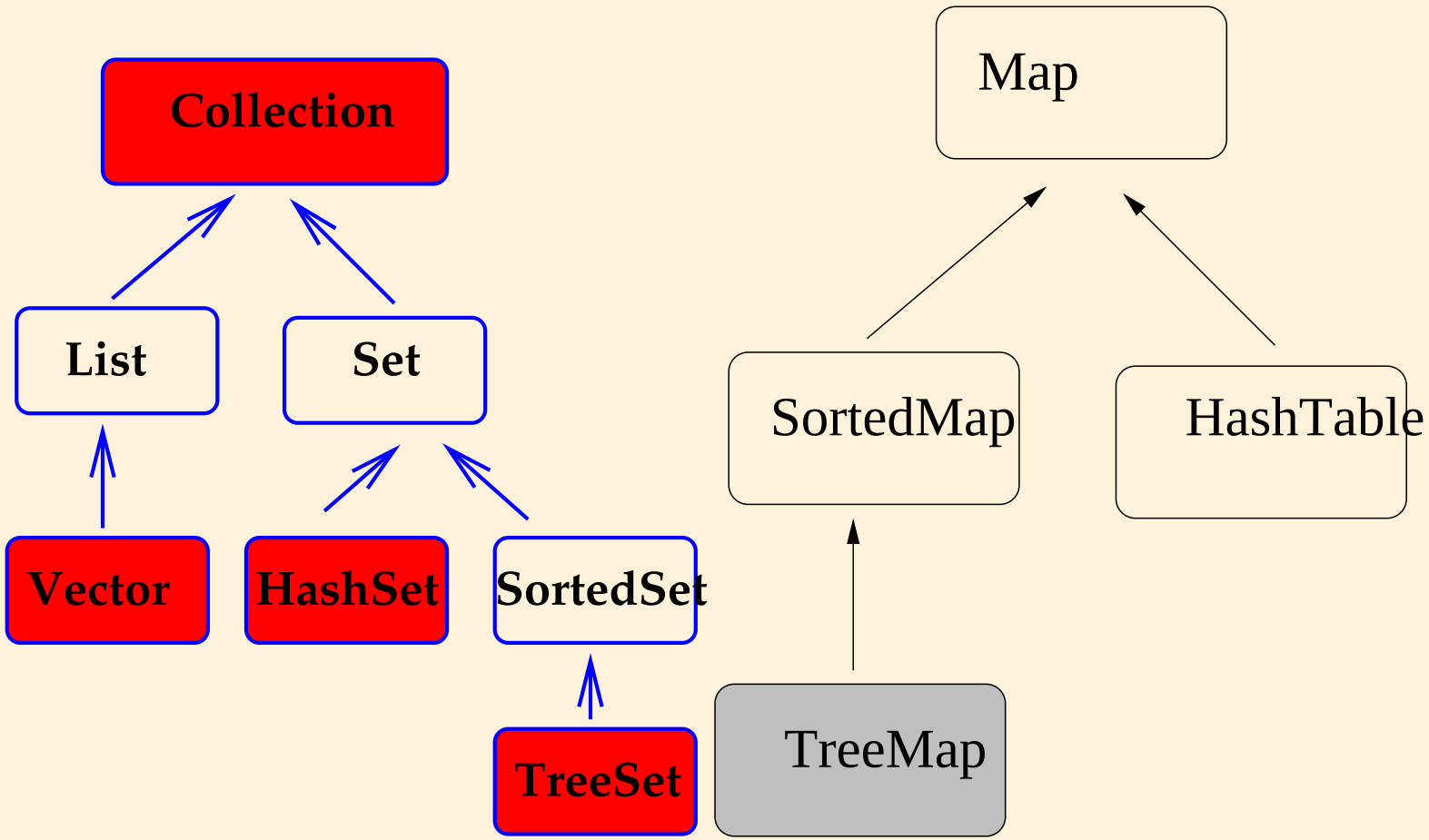

FIGURE: Framework de collections

Classe Vector

- **Elle permet de crèer des tableaux d'objets :**
  - la taille est dynamique,
  - tous les éléments sont de type Object.
    $\Longrightarrow$ la classe Vector peut avoir un contenu hétérogène.
- **Elle utilise de nombreuses méthodes :**
  - ajouter : add(Object o), add(int, Object o), addElement(Object o), insereElementAt(Object o, int i)
  - supprimer : remove(Object o), removeElement(Object o), removeElementAt(int i)
  - accéder : get(int i), elementAt(int i), indexOf(Object o), size().

Exemple avec Vector I

- Exemple : Jour de semaine avec Vector<String>.

```
1  Vector<String> semaine=new Vector<String>();
2  semaine.add("Lundi");
3  semaine.add("Mardi");
4  semaine.add("Mercredi");
5  semaine.add(2, "Samedi");//ajout d'un Element avec indice
6  System.out.println(semaine);
```

- Voici une trace d'exécution

```
1  [Lundi, Mardi, Samedi, Mercredi]
```

- Pour afficher sans Iterator

```
1  for(int i=0;i<semaine.size();i++)
2  System.out.println(semaine.get(i));
```

Iterator pour parcourir Vector I

- Pour afficher avec un Iterator

```
1  Iterator <String> it=semaine.iterator();
2  while(it.hasNext()){
3  System.out.println(it.next());}
```

- Pour supprimer : remove

```
1  Iterator <String> it2=semaine.iterator();
2  it2.next(); it2.next();
3  it2.remove();
4  System.out.println(semaine);
```

```
1  [Lundi, Samedi, Mercredi]
```

Vector avec le type Object I

- Pour gérer une collection hétérogène d'éléments

```
1  Vector <Object> vc=new Vector<Object>();
2  vc.add("Lundi"); vc.add("Mardi");
3  vc.add(54); // ajout d'un entier
4  vc.add("Mercredi");vc.add(2, "Samedi");
```

- instanceof pour déterminer le type de l'object

```
1  public static void afficher(Vector <Object> v){
2  Iterator <Object> it=v.iterator();
3  while(it.hasNext()){
4  Object o=(Object)it.next();
5  if (o instanceof String){
6  String s=(String)o; System.out.println("Element"+s);}
7  else if (o instanceof Integer){
8  Integer s=(Integer)o; System.out.println("Element"+s);}
9  }}
```

ArrayList I

- Implémente l'interface **List**. Sa taille est variable.

```
1     ArrayList v1=new ArrayList(); // vecteur vide
2     ArrayList v2=new ArrayList(c) // a partir d'une
          collection c.
```

- Création d'un **ArrayList** qui comporte 10 Integer .

```
1     ArrayList v=new ArrayList();
2     for(int i=0; i<10;i++) v.add(new Integer(i));
3     System.out.println("B : taille de v="+v.size());
4   --> B : taille de v=10
```

- On affiche les éléments avec la méthode *get()*

```
1     for(int i=0; i<v.size(); i++)
2     System.out.print(v.get(i)+ " ");
3   --> B : contenu de v [0 1 2 3 4 5 6 7 8 9]
```

ArrayList II

- On supprime un élément de position *i* avec *remove*

```
1     v.remove(3);
2     System.out.println("C : contenu de v"+v);
3   --> C : contenu de v[0, 1, 2, 4, 5, 6, 7, 8, 9]
```

- On ajoute et modifie un élément à 1 position donnée

```
1     v.add(2,new Integer(100));
2     System.out.println("D : contenu de v"+v);
3   --> D : contenu de v[0, 1, 100, 2, 4, 5, 6, 7, 8, 9]
4     v.set(5,new Integer(33));
5     System.out.println("E : contenu de v"+v);
6   --> E : contenu de v[0, 1, 100, 2, 4, 33, 6, 7, 8, 9]
```

Les Listes Chainées :LinkedList I

- Manipuler les listes doublement chainées : add , remove . . . , l'itérateur est : ListIterator
- Exemple de programme sur les listes chainées de chaines (String) qui illustre les fonctions de la classe ListIterator.

```
1  import java.util.*;
2  public class Liste1 {
```

```
1  public static void afficher(LinkedList l){
2  ListIterator iter=l.listIterator();
3  while(iter.hasNext())
4  System.out.print(iter.next()+ " ");
5  System.out.println();
6  }}
```

Les Listes Chainées :LinkedList I

```
1   public static void main(String arg[]){
2     LinkedList l=new LinkedList();
3     System.out.print("Liste A : "); afficher(l);
4   --> Liste A:
```

```
1     l.add("Langage C"); l.add("C++");
2     System.out.print("Liste B : "); afficher(l);
3   -->Liste B : Langage C C++
```

```
1     ListIterator it=l.listIterator();
2     it.next(); // On se place sur le premier
3     it.add("UML");it.add("Algorithme");
4     System.out.print("Liste C : ");afficher(l);
5   -->Liste C : Langage C UML Algorithme C++
```

Les Listes Chainées :LinkedList II

```
1    it=l.listIterator();
2    it.next(); // On progresse d'un element
3    it.add("Complexite"); it.add("Algebre");
4    System.out.print("Liste D : "); afficher(l);
5  -->Liste D:Langage C Complexite Algebre UML Algorithme C++
```

```
1    it=l.listIterator(l.size());
2    while(it.hasPrevious()){
3     String ch=(String) it.previous();
4     if (ch.equals("UML")) {it.remove();break;}}
5    System.out.print("Liste E : "); afficher(l);
6  -->Liste E : Langage C Complexite Algebre Algorithme C++
```

```
1    it=l.listIterator();
2    it.next();it.next(); // On se place sur le 2eme
3    it.set("Comptabilite"); // on remplace par "Comptabilite"
4    System.out.print("Liste F : "); afficher(l);
5  -->Liste F:Langage C Comptabilite Algebre Algorithme C++}
```

Hashtable I

- Permet de gérer des paires **clé/valeur**. La clé est généralement de type (String)
- On récupère l'elément par la valeur de la clé ou par la valeur de l'objet
- Les méthodes sont synchronized. Un et un seul thread accéde à lobjet à uinstant donnée

```
1  Hashtable<Integer,String> h= new Hashtable<Integer,String
       >()
2  h.put(1,"Metz");
3  h.put(2, "Thionville");
4  h.put(3,null);//NullPointerEcxeption
5  System.out.println("Villes :"+h.get(1)+","+ h.get(2));
```

HashMap I

- Comme Hashtable, il gère des paires **clé/valeur**.
- la clé et la valeur peuvent être **null**
- les méthodes ne sont pas synchronized : meilleure performance

```
1     HashMap<String,Integer> repertoire =new HashMap<String,
          Integer>();
2     repertoire.put("Mr Dupont", 87657898);
3     repertoire.put("Pierre", 87567890);
4     repertoire.put("Miro", 87567890);
5     System.out.println(repertoire);
6     System.out.println(repertoire.get("Miro"));
7     repertoire.remove("Pierre");
8     System.out.println(repertoire);
```

HashMap :Gestion d'un répertoire d'amis I

- La clé est de type String. La valeur est de type HashSet<String>

```
1  public class TestHashMap {
2  public static void main(String[] args) {
3  HashMap<String,HashSet<String>> amies =new HashMap<String,
       HashSet<String>>();
4  HashSet<String> amiPierre=new HashSet<String>();
5  amiPierre.add("Sami");
6  amiPierre.add("Paula");
7  amies.put("Pierre", amiPierre);
8  HashSet<String> amiAlice=new HashSet<String>();
9  amiAlice.add("Remi");
10 amiAlice.add("Paula");
11 amies.put("Alice", amiAlice);
12 System.out.println(amies);
13 }
14 }
```

HashSet I

- HashSet ne tolère pas la duplicaion des valeurs.
- **add** au lieu de **put**.
- **contains** pour l'appartenance.

```
1  HashSet<String> E = new HashSet<String>();
2  E.add ("CANADA");
3  E.add ("NIGERIA");
4  E.add ("INDONESIE");
5  if (E.contains("CANADA"))System.out.println("Existe deja
       dans E");
6  else E.add("PB");
```

Gestion de Collection Personne I

- Soit la classe Personne

```
1  public class Personne{
2  private String nom;
3  private int age;
4  public Personne(String nom, int age) {
5     this.nom = nom;
6     this.age = age;
7  }
8  public String toString() {
9     return "Personne [nom=" + nom + ", age=" + age + "]";
10 }
11 }
```

Gestion de Collection Personne I

- Voici une classe qui gère une collection de personnes

```
1  public class Groupe {
2     private ArrayList<Personne> g;
```

- Voici le constructeur

```
1  public Groupe() {
2  this.g = new ArrayList<Personne>();
3  }
```

- Pour ajouter une personne à la collection

```
1  public void add(Personne p){
2  g.add(p);
3  }
```

Gestion de Collection Personne I

- Pour afficher la collection

```
1  public void afficher(){
2  Iterator<Personne> it=g.iterator();
3  while(it.hasNext()){
4  System.out.println(it.next());
5  }}
```

Gestion de Collection Personne I

- Voici la classe TestGroupe

```
1  public static void main(String [] args){
2     Groupe g = new Groupe();
3     Personne p1= new Personne("Albert", 56);
4     Personne p2= new Personne("Fantine", 30);
5     Personne p3= new Personne("Sandrine", 18);
6     g.add(p1); g.add(p2); g.add(p3);
7     g.afficher();}
```

- Voici une trace d'éxécustion

```
1  Personne [nom=Albert, age=56]
2  Personne [nom=Fantine, age=30]
3  Personne [nom=Sandrine, age=18]
```

La sérialisation I

- C’est le mécanisme qui permet d’envoyer un objet dans un **flux binaire**. On parle d’objet **persistant**.
- Stocker dans un fichier ou dans une base de données ou de le transmettre vers une autre machine (flux réseaux)
- La sérialisation s’appuie sur les streams binaires : **InputStream** et **OutputStream**.
- Une spécialisation des ces deux classes permet de manipuler les objets en lecture et ecriture : **ObjectInputStream** et **ObjectOutputStream**

La sérialisation

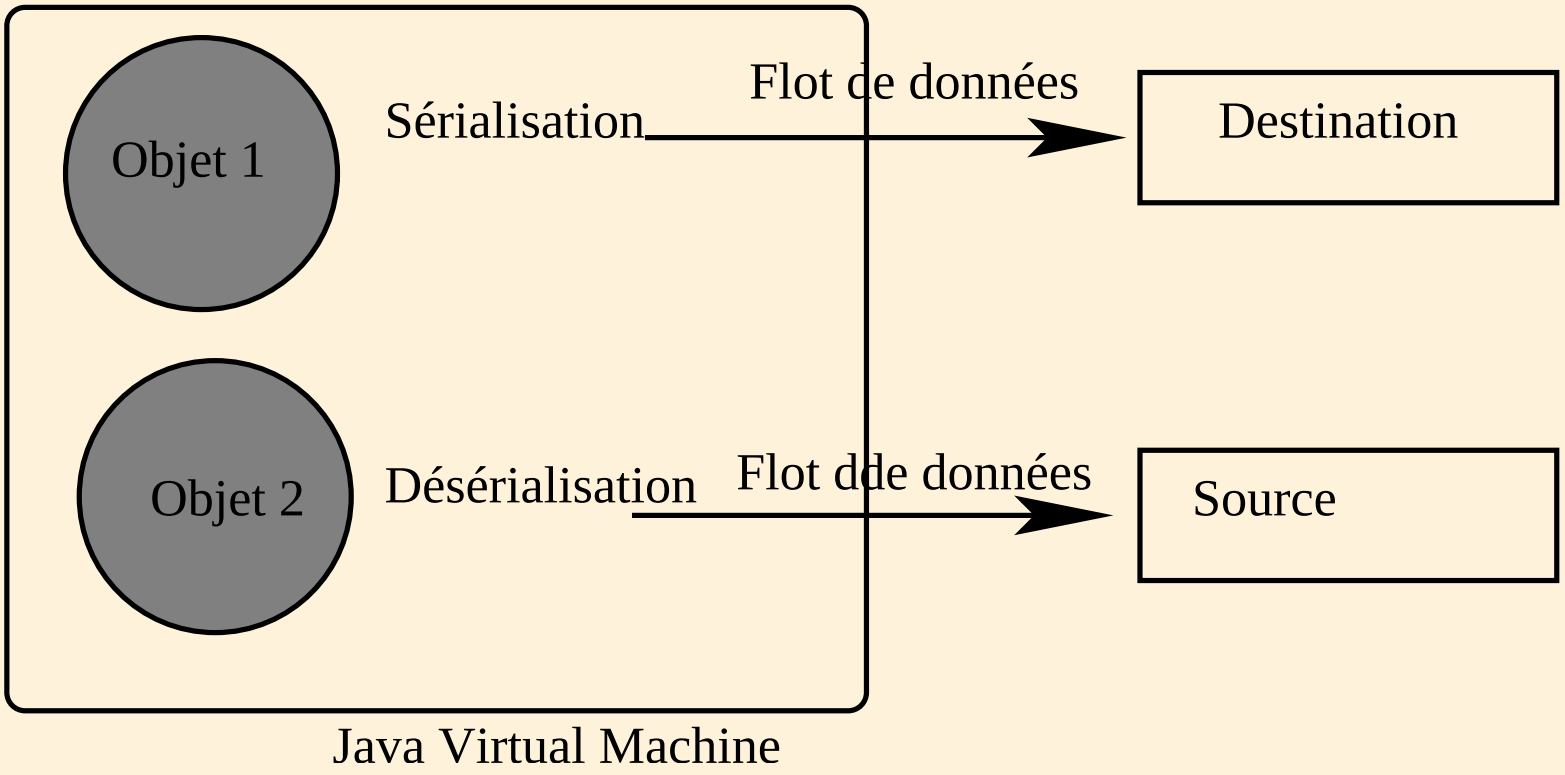

FIGURE: Sérialisation

La sérialisation des objets I

- La plate-forme Java permet de sauvegarder les objets en tant que tel au sein de fichier. Elle permet aussi de les restaurer.
- la sérialisation des objets assure la sauvegarde des informations de la façon suivante :
  - sauvegarde tous les types mis en jeu dans les declarations des objets.
  - sauvegarde les valeurs des données (type simple) de chaque objet et conserve les liens vers les autres objets qui compose l'objet.
  - sauvegarde les signatures des méthodes mais pas leurs codes.

La sérialisation des objets I

- Voici comment sauvegarder la collection de personnes :
- Il faut que la classe **Personne** implémente l'interface **Serializable**

```
1  class Personne implements Serializable{...}
```

- Il faut que la classe **Groupe** implémente l'interface **Serializable**

```
1  class Groupe implements Serializable{...}
```

La sérialisation des objets I

- Pour sauvegarder la collection de personnes *g* dans Fichier.obj

```
1  try{
2    FileOutputStream f=new FileOutputStream("Fichier.obj");
3    ObjectOutputStream s=new ObjectOutputStream(f);
4    s.writeObject(g);
5    s.flush();
6   }
7  catch(IOException e)
8   {
9    System.out.println(" Probleme IO");
10  }
```

La restauration des objets I

- Il faut un constructeur sans paramétre dans la classe Groupe

```
1  Groupe g1=new Groupe();
2  try{
3    FileInputStream f=new FileInputStream("Fichier.obj");
4    ObjectInputStream s =new ObjectInputStream(f);
5    g1=(Groupe)s.readObject();
6  }
7  catch(IOException e){
8    System.out.println("Nouveau Fichier");
9  }
10 catch(ClassNotFoundException e){
11   System.out.println ("probleme");
12 }
```