Généricité et l'infrastructure Collections de Java

Virginia Aponte

CNAM-Paris

14 mars 2017

1. Présentation de collections

L'infrastructure des collections de Java

C'est quoi ? bibliothèque Java (interfaces + classes) pour implanter des structures de données *abstraites* et *réutilisables*.

Comment ? une collection *est un objet* qui contient d'autres objets ;

Utilité ? pour stocker et manipuler ces collection d'objets via des *algorithmes connus et efficaces*,

Où ? dans le paquetage `java.util`

L'infrastructure

- Approche unifiée $\Rightarrow$ représenter + manipuler collections d'objets ;
- Composée de :
  - hiérachie d'interfaces génériques : ce sont leurs types ;
  - hiérarchie de classes génériques pour implanter ces interfaces ;
  - classes utilitaires :
    - (Implantation d'algorithmes) tris, recherche, etc,
    - conversion entre tableaux et collections, utilitaires tableaux,

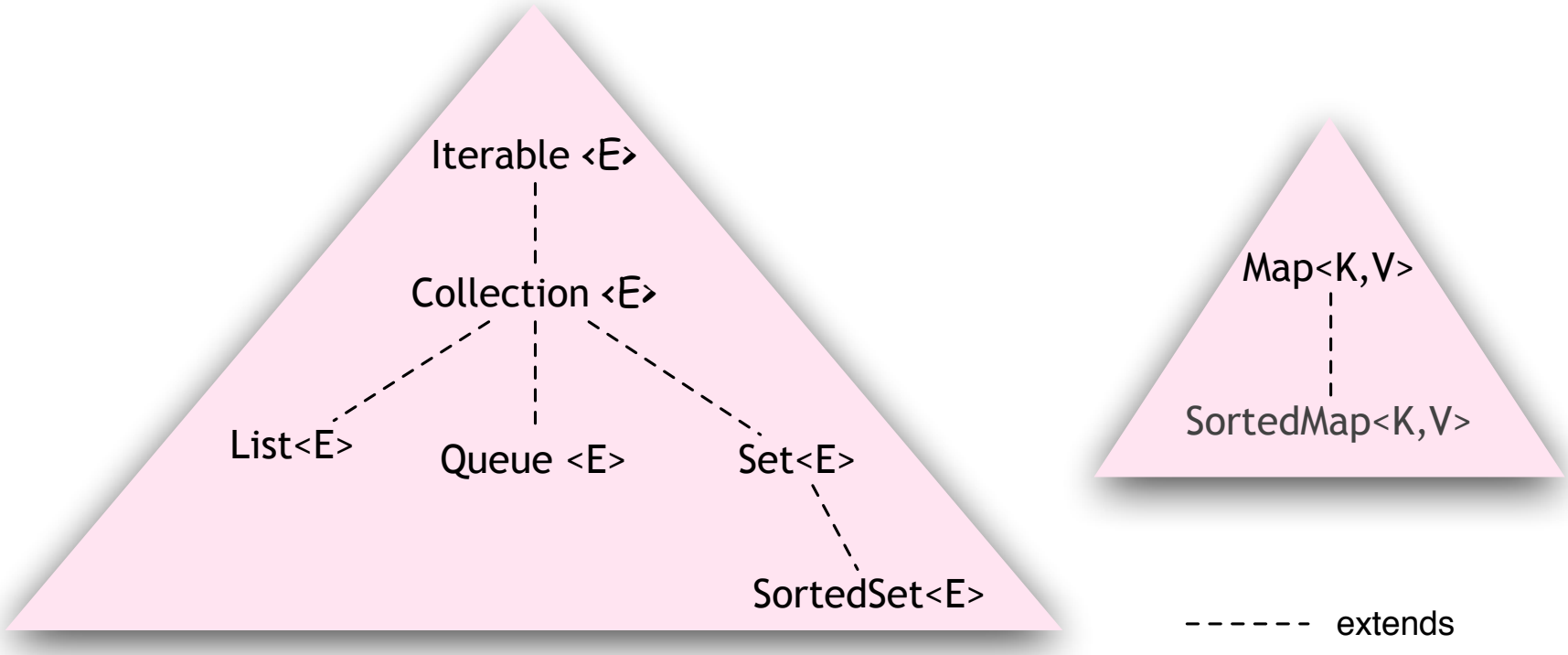
Généricité

Interfaces, classes, réutilisables sur toute sorte de types d'éléments/ collections concrètes.

Hiérarchie d'interfaces « collections » (1)

- 2 hiérarchies indépendantes d'interfaces :
  - « collections » d'objets (plusieurs objets dans une même structure) ;
  - « mappes », ou tables d'associations entre objets ;
- interface $\approx$ type des données générique (c.a.d., ne dépend pas du type de ses éléments) ;
- interface « fille » plus riche (contient + de méthodes) que son « ancêtre » dans l'arborescence.

Hiérarchie d'interfaces (extrait) collections



<E> : type générique E pour les éléments d'une collection

Interface `Collection<E>` : quelles opérations ?

Collection : objet contenant une quantité indeterminée d'objets.

Principales opérations :

- taille collection, tester si vide ;
- tester égalité, appartenance ;
- ajout/supression ;
- parcours (via itérateur) d'une collection ;
- conversion vers un tableau ;

Plus bas dans la hiérarchie :

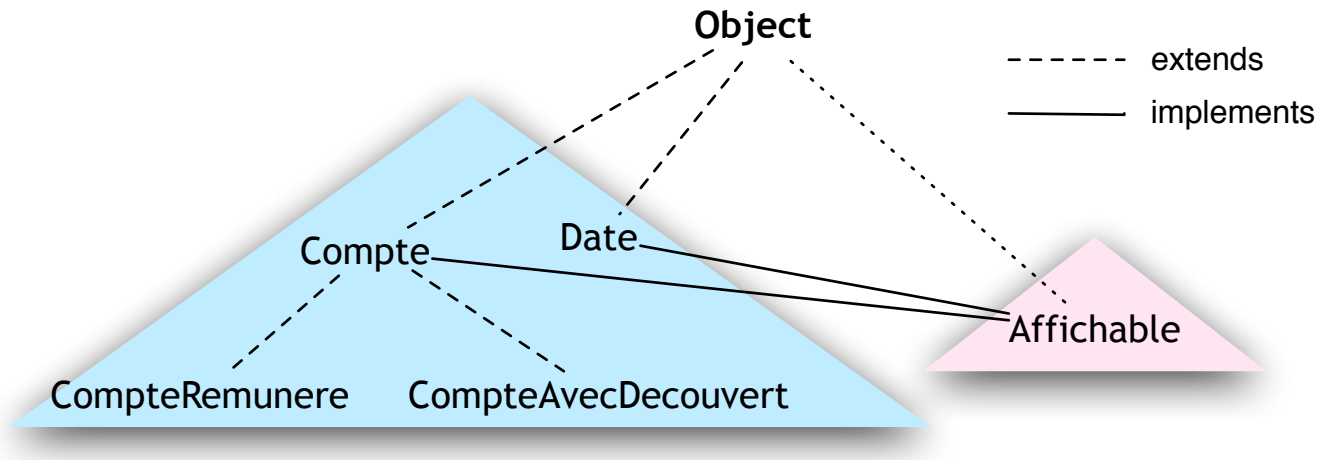
- Interfaces dérivées : List<E>, Set<E>, Queue<E> . . .
- Beaucoup de classes implantation : ArrayList<E>, LinkedList<E>, PriorytyQueue<E>, HashSet<E>, TreeSet<E> . . .

Interface `Collection<E>` (extrait)

```
public interface Collection<E> extends Iterable<E> {
 int size();
 boolean isEmpty();
 boolean add(E o);
 boolean remove(Object o);
 boolean equals(Object o);
 boolean contains(Object o);
 E[] toArray(E[] t);
 Iterator<E> iterator();
```

- les *classes* qui implantent cette interface
  $\Rightarrow$ *possèdent au moins* toutes ces méthodes ;
- `add, remove` retournent true si l'opération a pu s'effectuer.

`Object` : racine de la hiérarchie de classes




```
public class Object {
   /* Teste l'egalite entre objet courant et o*/
   public boolean equals (Object o){ ...}
   /* Renvoi une representation textuelle des variables */
   public String toString() {...}
   ... // autres methodes
}
```

Méthodes dans `Object`

- méthodes « utilitaires » (`equals`, `toString`, etc)
- avec une implantation « par défaut ». Ex : `equals(o)` compare les adresses de `this` et de `o` ;
- présentes par défaut dans toute classe de la hiérarchie.
- les librairies Java s'en servent !
  - Ex : `contains` utilise `equals` lors de la recherche d'un objet ;
  - `System.out.print` utilise `toString` pour afficher un objet.

Bonne pratique

Redéfinir les méthodes de Object pour un fonctionnement adapté à la classe courante.

Les collections que nous étudions

Interface `List<E>`

« listes » d'objets de type E

- éléments avec une position, parcours séquentiel ;
- doublons admis ;
- exemple d'implantation : `ArrayList<E>`

Interface `Set<E>`

« ensembles » d'objets de type E :

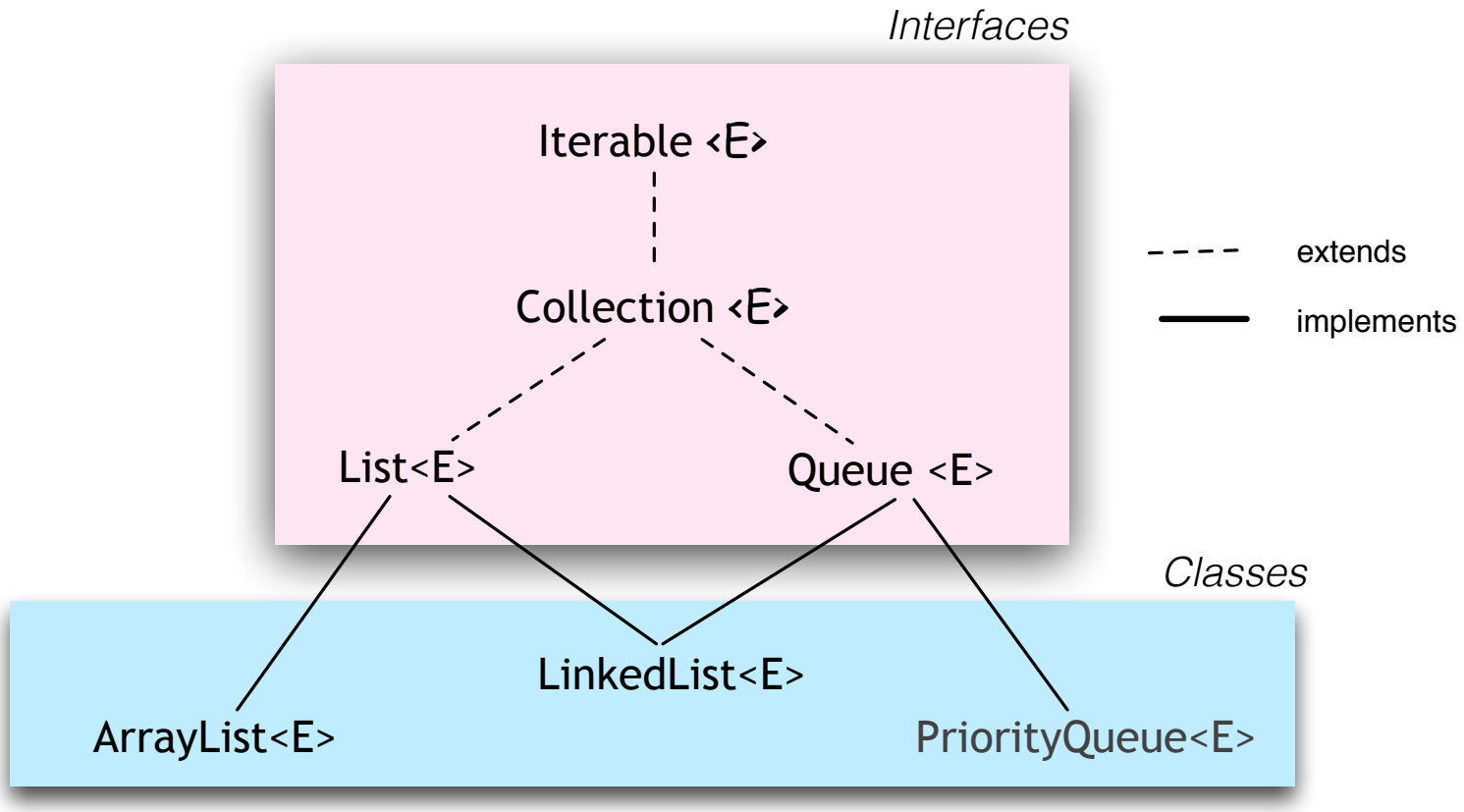
- pas de position pour les éléments ; parcours d'ordre indeterminé
- doublons interdits ;
- exemple implantations : `HashSet<E>, TreeSet<E>`

Les classes de la bibliothèque (*library*)

- *implantent* une ou plusieurs interfaces ;
- certaines classes sont *dérivées* (héritage) d'une autre dite « parente »
  - classe dérivée : possède *au moins toutes les méthodes* de la classe parente, plus d'autres qui lui sont propres ;
- peuvent appartenir à *plusieurs* hiérarchies.

Interfaces + classes ⇒ « les séquences »

Extrait hiérarchies pour les structures « séquences » :



<E> : type générique E pour les éléments

2. Les stuctures de données génériques

1. Structures des données « génériques » (1)

Concept : Entité qui encapsule plusieurs éléments de type (*générique*) (de type commun mais indéterminé) :

- décrit l'organisation des éléments dans la structure, sans s'occuper de leur nature interne (type)
- décrit les opérations pour agir sur la structure (chercher, ajouter, etc),

Exemples : tableaux, listes, arbres binaires, piles, etc.

Intérêt : selon organisation, opérations plus ou mois efficaces $\Rightarrow$ choix adapté à une application.

Abstraction : algorithmes (en général) indépendants du type des éléments.

1. Structures des données « génériques »(2)

Syntaxe en Java : `NomStructure<T>`

Classe ou interface de nom `NomStructure` dont les éléments son de type `T`.

- **<T>** : signifie « n'importe quel type objet »
- le code (éventuel) est indépendant de la nature de `T`

Polymorphisme (une forme de) :

utilisables pour n'importe quel type `T` : T= String, T= Compte, T= List(Compte), . . .

Exemples : classe ArrayList<T> de la bibliothèque Java

- dans ArrayList<String> $\Rightarrow$ T = String
- dans ArrayList<Compte> $\Rightarrow$ T = Compte

Exemple de classe générique : ArrayList<E>

```
public class ArrayList<E>
          implements List<E>, Collection<E>, Iterable<E> .
    public boolean add(E el) {...}
    public E get(int i){...}
 ....
}
```

- `E` : type (quelconque) des éléments de la liste ;
- implante plusieurs interfaces génériques ;
- les méthodes ont des types génériques :
  - `boolean add(E el)` : ajout d'un élément de type `E` ;
  - `E get(int i)` : retourne l'élément (type `E`) de position i
  - . . . etc.

3. Rappel : une collection,

`ArrayList<E>` (implante `List<E>`)

Collections : la classe ArrayList<E>

ArrayList<E>

- classe générique de Java :
  - éléments de type E ;
  - ce type peut être quelconque (objet) ;
  - on doit juste le préciser à la déclaration.
- fonctionne comme un tableau de taille *automatiquement* variable
  - chaque élément à une position, le premier à 0 ;
  - on peut y ajouter autant d'éléments qu'on veut,
  - on peut les accéder/modifier/supprimer selon leur position.

ArrayList<E> : déclaration et création

- type E (éléments) necéssairement de objet :
  - à préciser lors d'une déclaration, ex : `ArrayList<String>`
- création *avant utilisation* via `new`,

```
ArrayList<String> maListe;
maListe= new ArrayList<String>();// liste vide
```

- autres constructeurs pour initialiser avec paramètre liste ou tableau ;
- Important : déclarer en début de fichier,

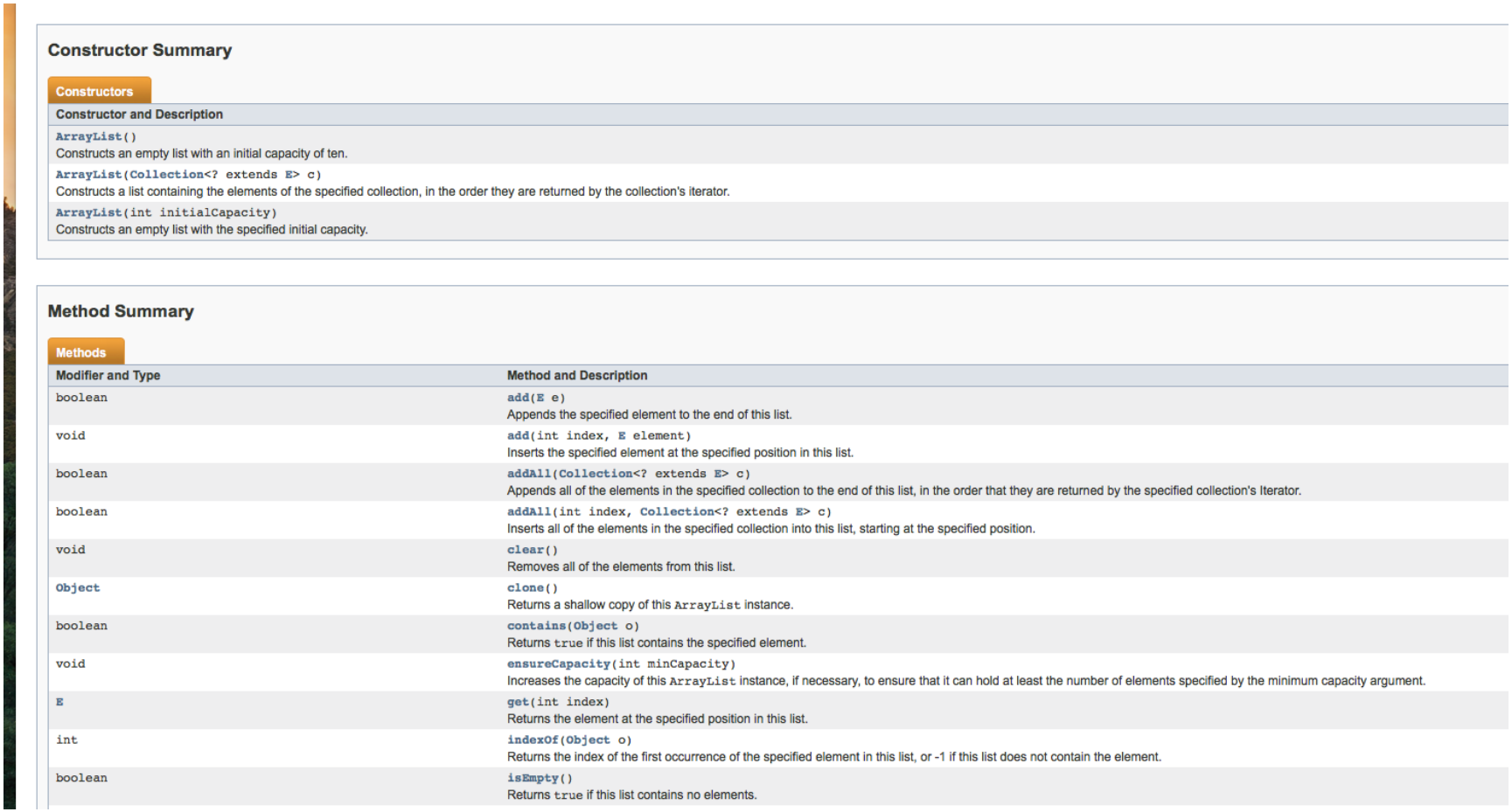
```
import java.util.ArrayList;
public class ...
```

Quelques méthodes de la classe ArrayList<E>

- `int size()` : renvoie la longueur de la liste ;
- `boolean isEmpty()` : teste si la liste est vide ;
- `E get(int i)` : renvoie le contenu de la case numéro `i`. Le type de l'objet retourné est E.
- `add(T el)` : ajoute l'élément `el` à la *fin* de la liste.
- `set(int i, T el)` : remplace la valeur dans la case i par `el`. `i` doit être inférieur à la taille de la liste.
- `remove(int i)` supprime l'élément dans la case `i` ;
- `remove(Object el)` supprime la première occurrence de l'élément de valeur `el`. S'il est présent plusieurs fois, n'est enlevé qu'une seule fois. Le contenu des cases est décalé, et la longueur de la liste diminue de 1. Si non présent, la liste n'est pas modifiée.

Beaucoup d'autres méthodes

Il faut aller voir la documentation en ligne !



Constructor Summary

Constructors

Constructor and Description
`ArrayList()` Constructs an empty list with an initial capacity of ten.
`ArrayList(Collection<? extends E> c)` Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
`ArrayList(int initialCapacity)` Constructs an empty list with the specified initial capacity.

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`add(E e)` Appends the specified element to the end of this list.
`void`	`add(int index, E element)` Inserts the specified element at the specified position in this list.
`boolean`	`addAll(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
`boolean`	`addAll(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list, starting at the specified position.
`void`	`clear()` Removes all of the elements from this list.
`Object`	`clone()` Returns a shallow copy of this `ArrayList` instance.
`boolean`	`contains(Object o)` Returns `true` if this list contains the specified element.
`void`	`ensureCapacity(int minCapacity)` Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
`E`	`get(int index)` Returns the element at the specified position in this list.
`int`	`indexOf(Object o)` Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`boolean`	`isEmpty()` Returns `true` if this list contains no elements.

Exemple : ArrayList de comptes

```
ArrayList<Compte> lc = new ArrayList<Compte>();
Compte c1 = new Compte();
Compte c2 = new Compte();

lc.add(c1); // ajout en position 0
lc.add(c2); // ajout en position 1
double s = lc.get(1).getSolde(); // solde de c2
```

Exemple (2) : somme des soldes

- `lc` est un ArrayList de comptes ;
- pour parcourir la liste : `i< lc.size()` ;
- pour obtenir le compte de position i : `lc.get(i)` ;
- pour obtenir le solde de ce compte : `lc.get(i).donneSolde()`

```
public static double sommeSoldes(ArrayList<Compte> lc){
    double s=0;
    for (int i= 0; i < lc.size(); i++) {
        s= s + lc.get(i).getSolde();
    }
    return s;
}
```

4. Trier une collection

Pour trier : établir un ordre sur les éléments

Algorithmes de tri ⇒ comparent objets à trier

1 objet est-il plus petit, plus grand ou égal qu'un autre ?

La classe `Collections` (avec un « s » ! ! !) :

- contient méthodes utilitaires **statiques** sur collections (tri, recherche, conversion, inversion etc.)
- utilisent comparaison « standardisée » d'objets E, soit :
  1. utilisation d'un objet `Comparator<E>` (solution la + simple)
  2. **ou** objets à comparer *implantent* `Comparable<E>`
- appel à `Collections.sort(....)` ⇒ après définition d'un parmi ces modes de comparaison !

4.1 – Trier une collection avec Comparable<T>

Interface `Comparable<T>`

- Fait partie du framework Collections.
- Contient une unique méthode :

```
interface Comparable<T> {
    public int compareTo(T o);
}
```

  - T $\rightarrow$ type des objets à comparer
  - compare l'objet courant `this` avec l'objet `o` et renvoie un entier :
    - négatif si `this` est plus petit que `o` ;
    - positif si si `this` est plus grand que `o` ;
    - 0 s'ils sont égaux

String, Integer, etc. implantent comparable

- Beaucoup de types prédéfinis implantent Comparable,
  - Ex : `String` implante Comparable selon l'ordre *lexicographique* (selon l'alphabet).

```
public class String implements Comparable<String> {
   public int compareTo(T o){...
     // comparaison des codes Unicode sur caracteres
     // de this et de o...
}
```

Exemple : String implante comparable

```
String s1 = "abc";
String s2 = "abb";
String s3 = "abc"; // contenu identique à s1
System.out.println("s1 compare s2? "+s1.compareTo(s2));
System.out.println("s2 compare s1? "+s2.compareTo(s1));
System.out.println("s1 compare s3? "+s1.compareTo(s3));
```

Affiche :

```
s1 compare s2? 1
s2 compare s1? -1
s1 compare s3? 0
```

Pourquoi ?

Trier avec `Comparable<T>`

Supposons que `List<T>lt` est la liste des objets (de type T) à trier :

1. modifier la classe T :
   - ajouter dans son entête → `implements Comparable<T>;`
   - ajouter dedans → 1 méthode `public int compareTo(T o).`
   - ajouter dedans → 2 méthodes :
     - `public boolean equals(Object)`
     - `public int hashCode()`
2. pour réaliser le tri → appeler la méthode

   `void Collections.sort(lt)`

   sur la liste à trier
   - elle utilisera la méthode `compareTo` ajoutée dans la classe T pour les comparaisons pendant le tri

Implantation de `Comparable<Compte>` (ordre de numéros)

- Implantons `Comparable` sur `Compte` selon l'ordre de numéros de comptes ;
- nous ajoutons `compareTo` qui compare ces numéros.

```
public class Compte implements Comparable <Compte> {
    private int numero;
    private double solde;

   @Override
    public int compareTo(Compte o) {
        if (this.getNumero()< o.getNumero()) return -1;
        else if (this.getNumero()> o.getNumero()) return 1;
        else return 0;
    }
```

Implanter `Comparable<Compte>` (2)

Ajout de la méthode `equals` dans `Compte` :

```
public class Compte implements Comparable <Compte> {
      ......
       @Override
    public boolean equals(Object obj) {
        if (this == obj)
             return true;
        if (obj == null)
             return false;
        if (getClass() != obj.getClass())
             return false;
        Compte other = (Compte) obj;
        if (numero != other.numero)
             return false;
        return true;
   }
```

Exercice : expliquez tous les cas de ce code.

Implanter `Comparable<Compte>` (3)

Ajout de la méthode `hashCode()` dans `Compte` :

```
public class Compte implements Comparable <Compte> {
      ......
      @Override
      public int hashCode() {
          final int prime = 31;
          int result = 1;
          result = prime * result + numero;
          return result;
      }
```

Cette méthode a été générée via *Eclipse* !
Nous l'étudierons plus tard . . .

Client : affichage par ordre des numéros

```
public class Client {
    private String nom;        // Nom titulaire
    private ArrayList<Compte> cmptes;  // ses comptes

    public void affiche() {
        if (cmptes.isEmpty()) {
            System.out.println("Aucun compte");
            return;
        }
        System.out.println("Comptes de "+nom+ " :");
        ArrayList<Compte> t= new ArrayList<Compte>(cmptes);
        Collections.sort(t);
        for (Compte c: t) {
            c.affiche();
        }
        System.out.println();
    }
```

Programme exemple

```
public static void main(String[] args) {
        Compte c1 = new Compte (1,2500);
        Compte c2= new Compte (7, 200);
        Compte c3 = new Compte (3, 45);
        Client lucie = new Client("Lucie");
        lucie.ajout(c2);
        lucie.ajout(c1);
        lucie.ajout(c3);
        lucie.ajout(c2); // doublon
        lucie.affiche();  // affiche ordre numéros
```

Affichages

```
Comptes de Lucie :
Numero: 1, solde: 2500.0

Numero: 3, solde: 45.0

Numero: 7, solde: 200.0

Numero: 7, solde: 200.0
```

4.2 – Trier avec un objet Comparator<E>

Interface `Comparator<E>`

Contient une unique méthode :

```
interface Comparator<E> {
   public static int compare(E o1, E o2);
}
```

- E $\rightarrow$ type des objets à comparer
- compare les objets `o1, o2` et renvoie un entier :
  - négatif si `o1` est plus petit que `o2` ;
  - positif si si `o1` est plus grand que `o2` ;
  - 0 s'ils sont égaux.

`Collections.sort(List<E>, Comparator<E>)`

Méthode qui trie `List<E>` selon l'ordre donné par `Comparator<E>`.

Pour utiliser un `Comparator<E>`

Supposons que `la` soit une liste (`List<A>`) d'objets de type A à trier :

1. déclarer nouvelle classe `CompA` qui implante `Comparator<A>` ;
2. créer un objet instance de cette classe :
   ```
   Comparator<A> comp = new CompA();;
   ```
3. appeler la méthode
   ```
   void Collections.sort(la, comp)
   ```
   - sur la liste `la` et
   - sur l'objet Comparator `comp`

$\Rightarrow$ `Collections.sort(la, comp)`

Exemple : odre de soldes pour les comptes

Une classe qui implante un Comparator de Comptes selon l'ordre de leurs soldes.

```
class ordreParSolde implements Comparator<Compte>{
  public  int compare(Compte o1, Compte o2){
     if (o1.getSolde() < o2.getSolde()) {
          return -1;
     } else if (o1.getSolde() > o2.getSolde()){
          return 1;
     } else { return 0;  }
 }
```

- `Compte` → type des objets à comparer ;
- pour fabriquer un Comparator<Compte> ⇒ `new ordreParSolde();`
- il suffira de passer cet objet à la méthode `Collections.sort`.

La classe Client

Supposons que la classe Client possède une variable d'instance avec les comptes d'un client :

```
private ArrayList<Compte> cmptes;
```

- nous voulons ajouter une méthode qui affiche ces comptes triés par leurs soldes ;
- nous allons utliser le Comparator `ordreParSolde`.

```
public class Client {
  private String nom;
  private ArrayList<Compte> cmptes;
  ...
}
```

---

afficheParSolde() de la classe Client

```
public void afficheParSolde() {
   if (cmptes.isEmpty()) {
       System.out.println("Aucun compte");
       return;
    }
    System.out.println("Comptes de "+nom+ " par solde:");
    // Copie sur une liste nouvelle
    ArrayList<Compte> t = new ArrayList<Compte>(cmptes);
    // Appel methode de tri
    Collections.sort(t,new ordreParSolde());
    for (Compte c: t) {
       c.affiche();
    }
}
```

Afficher par ordre des soldes

```
Compte c1 = new Compte (1,2500);
Compte c2= new Compte (7, 200);
Compte c3 = new Compte (3, 45);
Client lucie = new Client("Lucie");
lucie.ajout(c2);
lucie.ajout(c1);
lucie.ajout(c3);
lucie.ajout(c2); // doublon
lucie.afficheParSolde();
```

Affiche :

```
Comptes de Lucie tries par solde:
Numero: 3, solde: 45.0,
Numero: 7, solde: 200.0
Numero: 7, solde: 200.0
Numero: 1, solde: 2500.0
```