Collections et itérateurs

Jonathan Lejeune

Sorbonne Université/LIP6-INRIA

Les collections

Définition d'une collection

Objet gérant un regroupement d'objets appelés éléments et se caractérise par

- une interface spécifiant ses opérations publiques
- une implantation spécifiant sa structure interne, ses contraintes et sa politique d'itération

Ce que fournit l'API Java

- Des interfaces

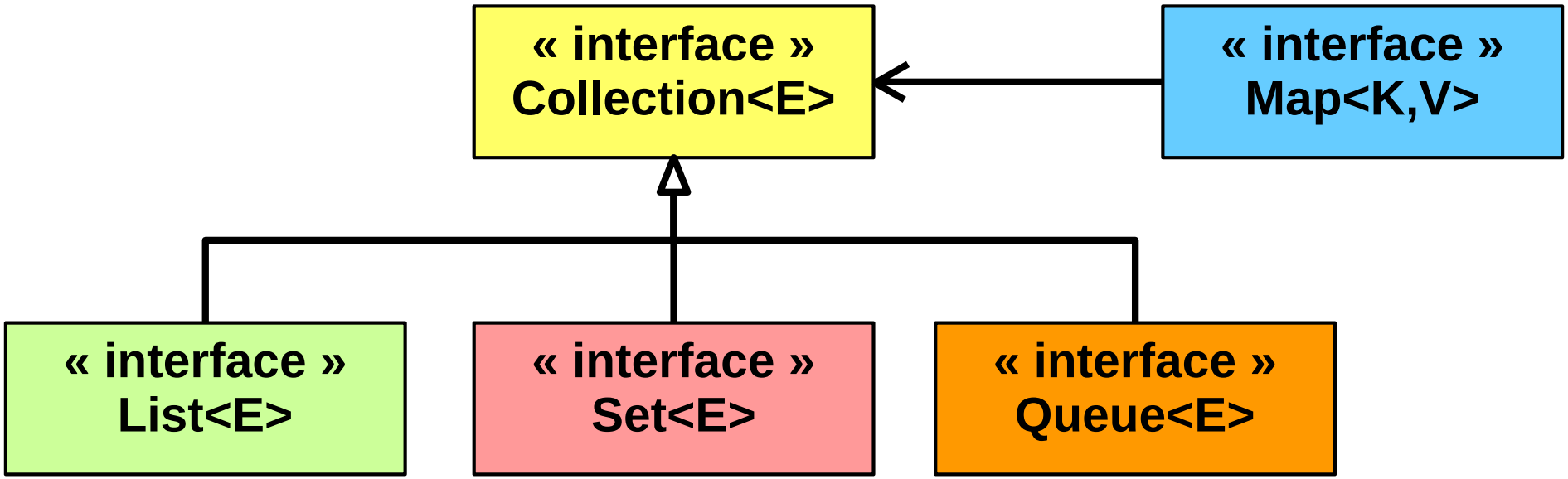
  ex : `Collection<E>`, `List<E>`, `Set<E>`, etc.
- Des classes d'implantations

  ex : `ArrayList<E>`, `HashSet<E>`, etc.
- Une classe utilitaire `Collections` fournissant des algorithmes

  ex : `binarySearch`, `copy`, `disjoin`, `frequency`, `sort`, etc.

Le tout regroupé dans le package `java.util`.

Les familles de collections



L'interface Collection<E>

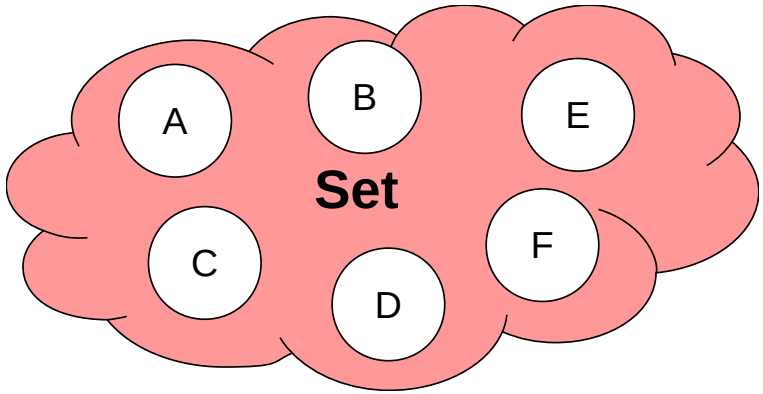
Principales méthodes

- `boolean add(E e)` : Ajouter un élément
- `boolean addAll(Collection<? extends E> c)` : Ajouter tous les éléments fournis en paramètre
- `void clear()` : Supprimer tous les éléments
- `boolean contains(Object o)` : Tester si un élément est présent
- `boolean equals(Object o)` : Tester l'égalité
- `boolean isEmpty()` : Tester si la collection est vide
- `boolean remove(Object o)` : Supprimer un élément s'il est présent
- `boolean retainAll(Collection<?> c)` : Ne laisser dans la collection que les éléments fournis en paramètres : les autres éléments sont supprimés
- `int size()` : nombre d'éléments
- `Object[] toArray()` : Retourner un tableau contenant tous les éléments
- `<T> T[] toArray(T[] a)` : Retourner un tableau typé de tous les éléments

Convention

Toute classe d'implantation doit fournir un constructeur qui prend en paramètre une collection.

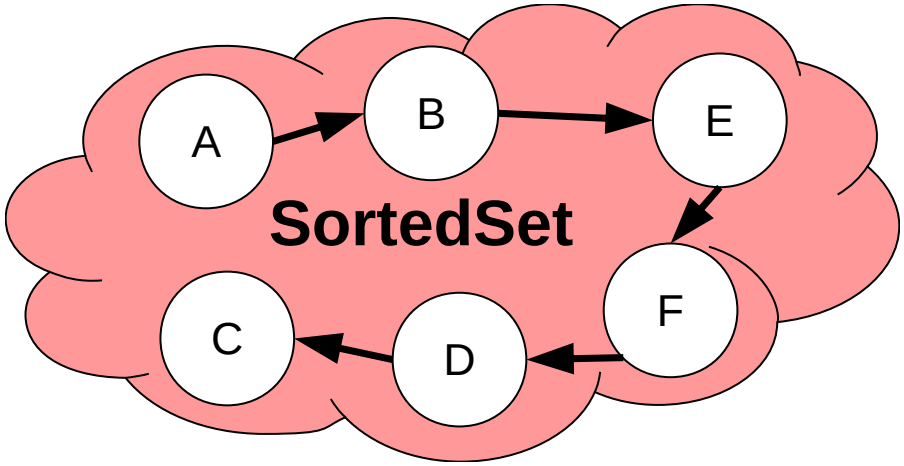
Les ensembles (interface `Set<E>`)



Spécifications

- N'offre pas d'autres méthodes que `Collection<E>`
- Les éléments sont identifiés par leur valeur
- Pas de doublon : $\nexists e_1, e2 \in Set$ tel que $e1.equals(e2)$
  `boolean equals(Object)` **éventuellement à redéfinir**
- Accès direct à un élément impossible (pas de `get`)
  $\Rightarrow$ l'élément est dans l'ensemble ou il n'y est pas (`contains`)
- Élément `null` interdit/autorisé en fonction de l'implémentation

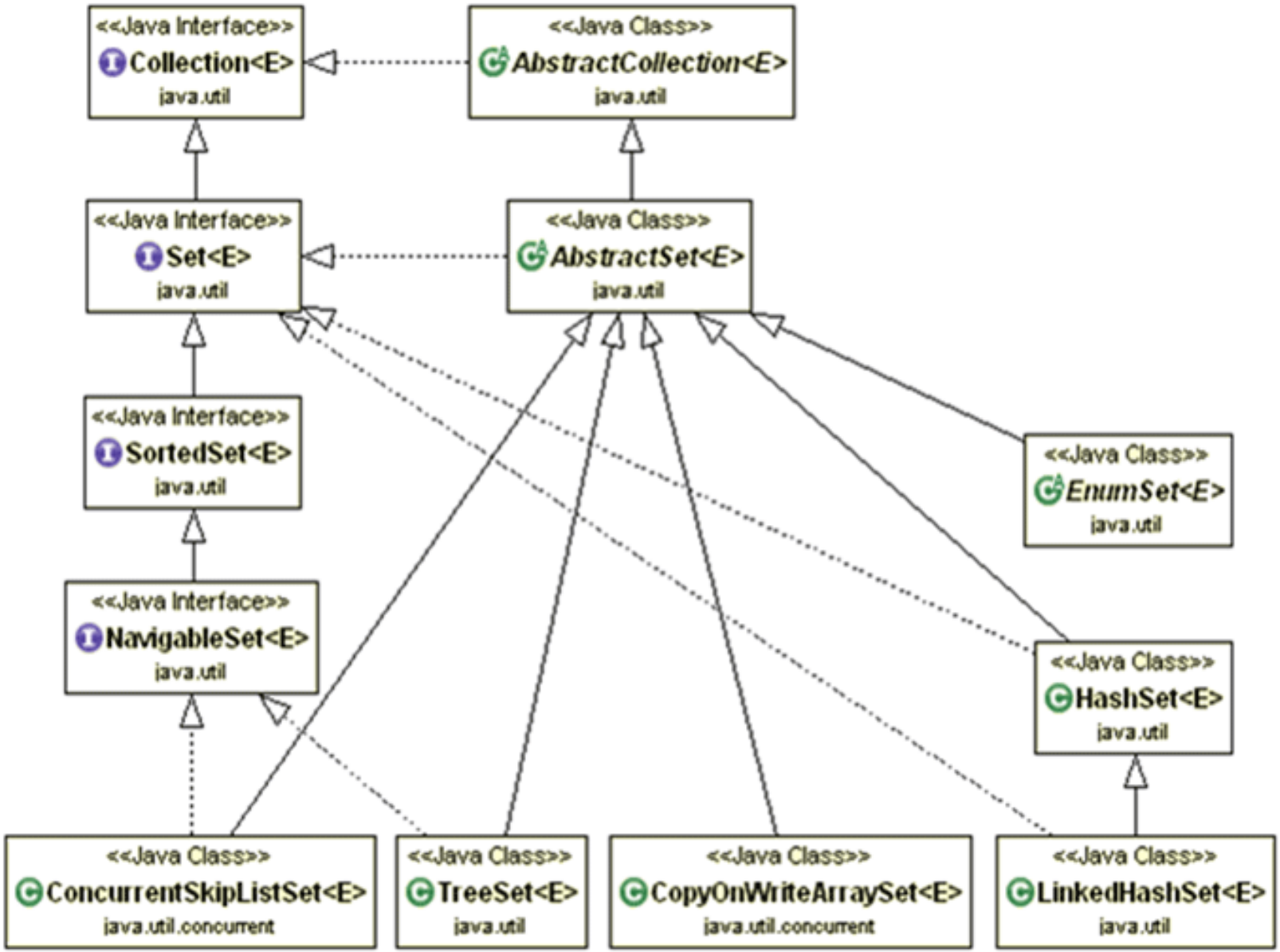
L'interface fille : SortedSet<E>



Spécifications

- Représente un ensemble ordonné
- Méthodes offertes :
  - `Comparator<? super E> comparator();`
  - `SortedSet<E> subSet(E fromElement, E toElement);`
  - `SortedSet<E> headSet(E toElement);`
  - `SortedSet<E> tailSet(E fromElement);`
  - `E first();`
  - `E last();`

Hiérarchie de type des Set



Implantations de Set à base de table de hachage

Caractéristiques

- Complexité de add, `contain` = $\mathcal{O}(1)$ si fonction hachage bien répartie
- **Fortement conseillé de bien redéfinir** `int hashCode();`

Deux implantations

- `HashSet<E>` : Table de hachage simple , Itération aléatoire sur les éléments
- `LinkedHashSet<E>` : table de hachage + liste chainée , Itération des éléments dans l'ordre d'insertion

Algorithme d'ajout d'un élément e

- $\nexists e' \in Set$ tel que $\mathcal{H}(e) = \mathcal{H}(e')$
  $\Rightarrow$ add e in Set and return true
- $\forall e' \in Set$ tel que $\mathcal{H}(e) = \mathcal{H}(e')$, $\nexists e'$ $e.equals(e')$
  $\Rightarrow$ add e in Set and return true
- Otherwise return false

Implantation de `Set` à base d'arbre

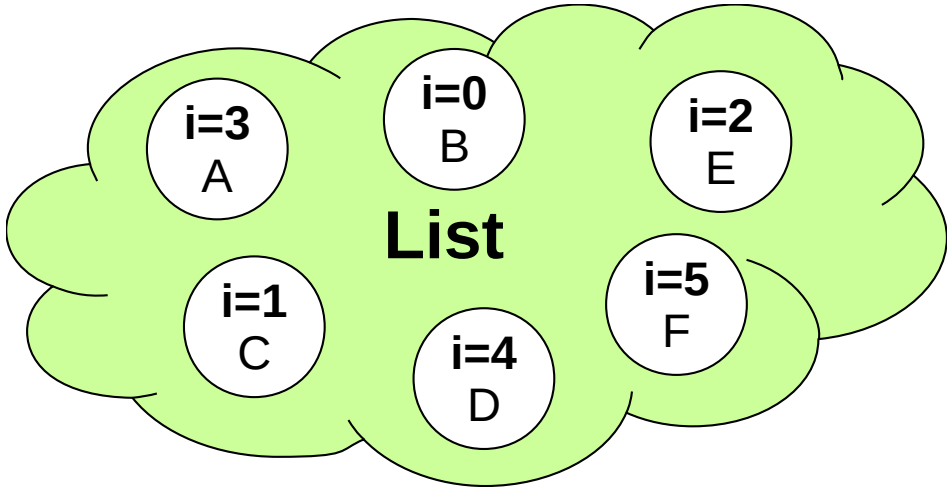
Caractéristiques

- Arbre rouge/noir (= Arbre binaire équilibré)
- Complexité de `add`, `contain` = $\mathcal{O}(log_2 N)$

Une implémentation héritant de `SortedSet<E>`

`TreeSet<E>` : Itération dans l'ordre naturel de `compareTo`

Les listes (interface `List<E>`)



Spécifications

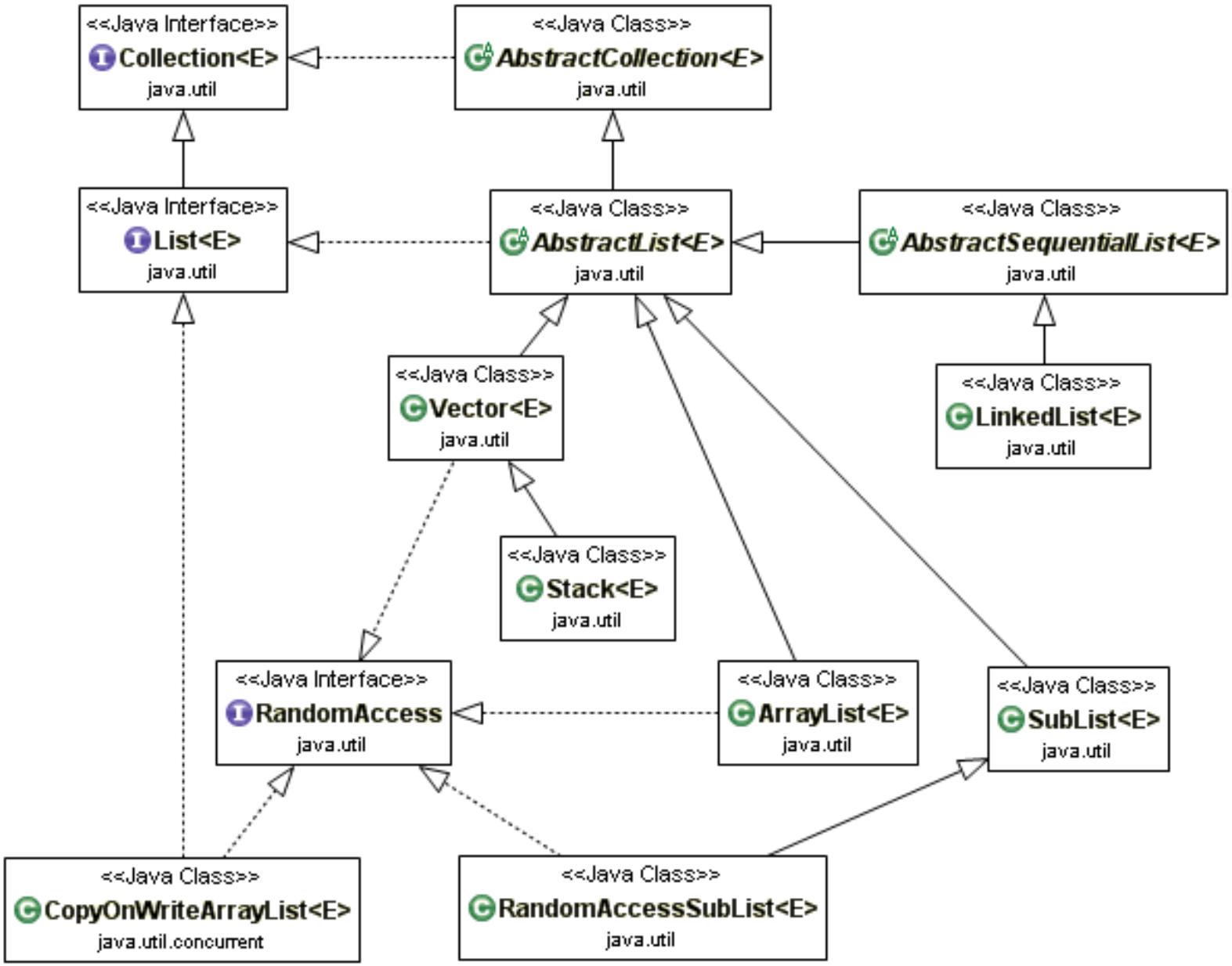
- Les éléments sont identifiés par un indice entier allant de 0 à size()-1
- doublons autorisés
- Accès direct à un élément se fait à partir de l'indice
- Élément `null` autorisé

Les listes (interface `List<E>`)

Principales méthodes

- `void add(int index, E e)` Ajouter un élément à la position index
- `E get(int index)` : Retourner l'élément à la position index
- `int indexOf(Object o)` : Retourner la première position dans la liste du premier élément `equals` à o, -1 si l'élément n'est pas trouvé
- `int lastIndexOf(Object o)` : Retourner la dernière position dans la liste du premier élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé
- `ListIterator<E> listIterator()` : Renvoyer un Iterator positionné sur le premier élément de la liste
- `E remove(int index)` : Supprimer l'élément à la position index
- `E set(int index, E e)` : Remplacer l'élément à la position index
- `List<E> subList(int fromIndex, int toIndex)` : Obtenir une sous-liste des éléments compris entre `fromIndex` inclus et `toIndex` exclus

Hiérarchie de type des Listes



Principales implantations de `List<E>`

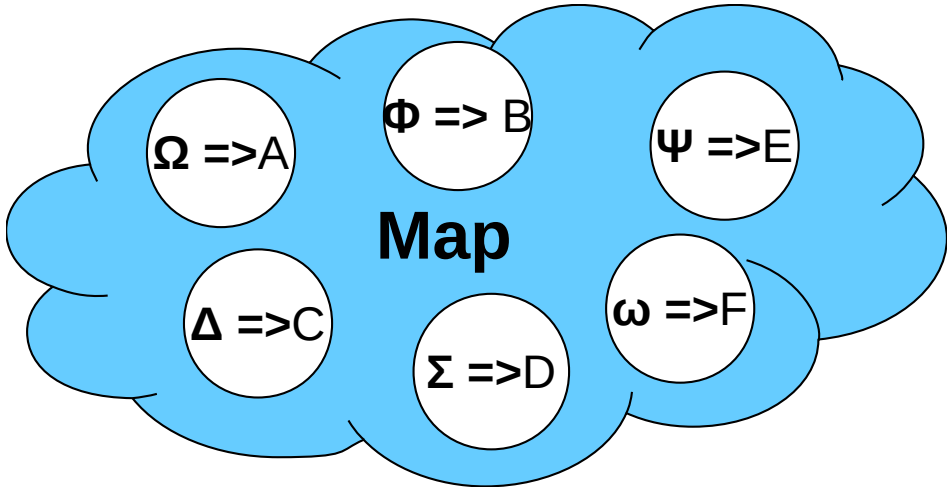
La classe `ArrayList<E>`

- Utilise un tableau pour stocker ses éléments
- La taille du tableau s'adapte automatiquement au nombre d'éléments
- Complexité ajout/suppression : $\mathcal{O}(N)$
- Complexité accès : $\mathcal{O}(1)$

La classe `LinkedList<E>`

- Utilise un double chaînage
- Il n'est pas redimensionnée quelque soit le nombre d'éléments
- Complexité ajout/suppression : $\mathcal{O}(1)$
- Complexité accès : $\mathcal{O}(N)$

Les map (interface `Map<K,V>`)



Spécifications

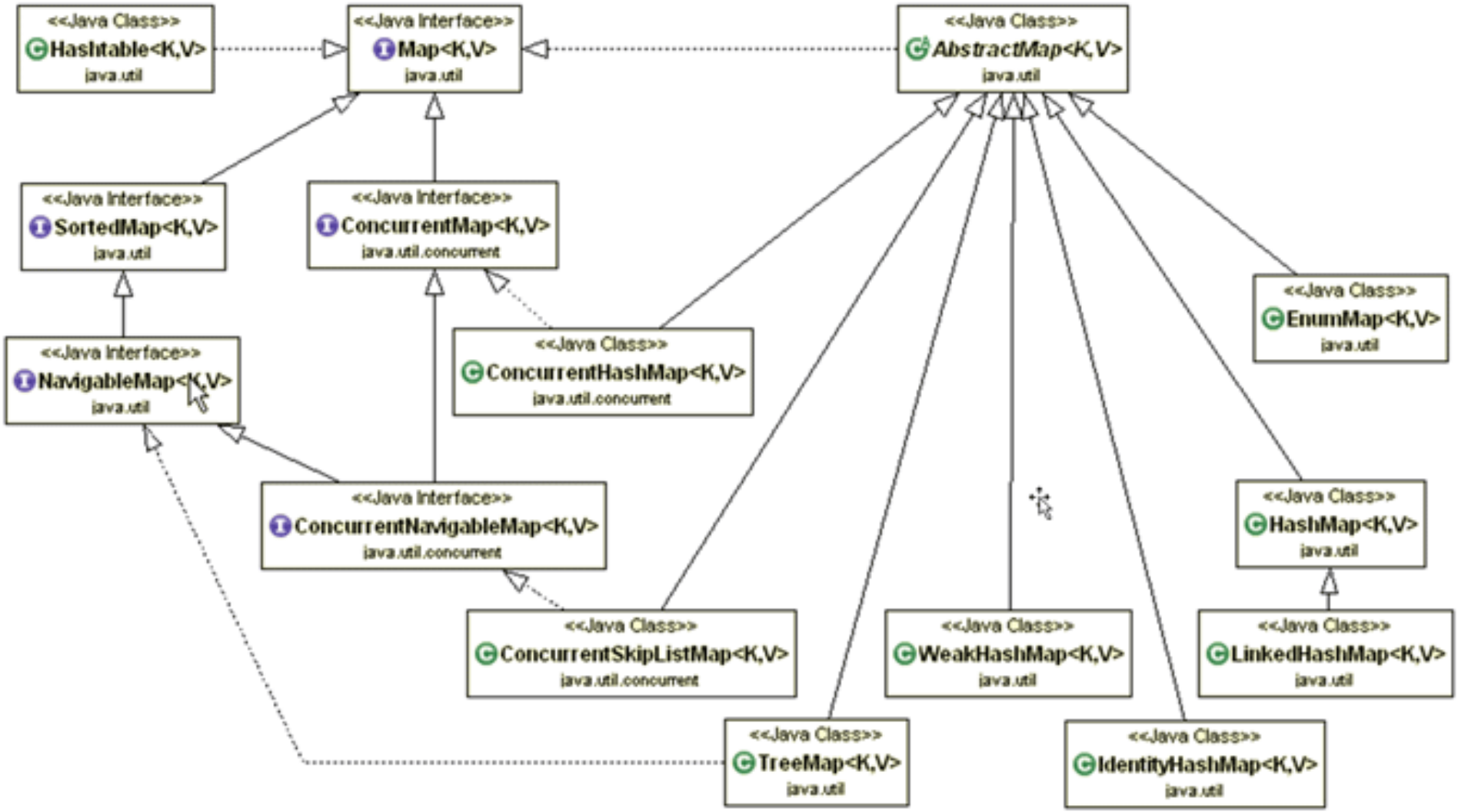
- Les éléments sont identifiés par un un objet clé
- Chaque clé est unique et n'est associée qu'à une seule valeur
- N'est pas une collection mais peut renvoyer :
  - Un `Set` de ses clés
  - Une `Collection` de ses valeurs
  - Un `Set` sur les couples (clé,valeur)

Les map (interface Map<K,V>)

Principales méthodes

- `void clear()` : Supprimer tous les éléments
- `boolean containsKey(Object k)` : Indiquer si la clé est dans la map
- `boolean containsValue(Object v)` : Indiquer si la valeur est dans la map
- `Set<Map.Entry<K,V>> entrySet()` : Renvoyer un ensemble contenant les paires clé/valeur de la map
- `V get(Object k)` : Renvoyer la valeur associée à la clé fournie en paramètre
- `boolean isEmpty()` : Indiquer si la map est vide
- `Set<K> keySet()` : Renvoyer un ensemble contenant les clés de la collection
- `V put(K key, V val)` : Insérer la clé et sa valeur associée fournies en paramètres
- `void putAll(Map<? extends K,? extends V>)` : Insérer plusieurs clés/valeurs
- `Collection<V> values()` : Renvoyer toutes les valeurs des éléments
- `V remove(Object k)` : Supprimer l'élément avec la clé k
- `int size()` : nombre de paires dans la collection

Hiérarchie de type des Map



Complément sur les Maps

Une généralisation de `Set` et `List`

- `Set<E>` = `Map<E,Void>`
  ⇒ les implem `HashMap`, `LinkedHashMap`, `TreeMap` ont les mêmes spécifications que leurs homologues `Set`
  ⇒ en pratique le plupart des `Set` se base sur des `Map`
- `List<E>` $\simeq$ `Map<Integer,E>`
  ⇒ utile si ensemble d'indice non contiguë, avec valeur négative

Exemple d'utilisation

- Ranger des objets et les retrouver rapidement
- Table d'unicité
- Cache d'opérations : clé =paramètre, valeur = résultat du calcul
- Compter un nombre d'occurrence pour chaque élément d'une liste
- Indexation de documents : clé= mot, valeur= liste de documents

Un outil de programmation très courant ⇒ à maîtriser

L'interface Queue<E>

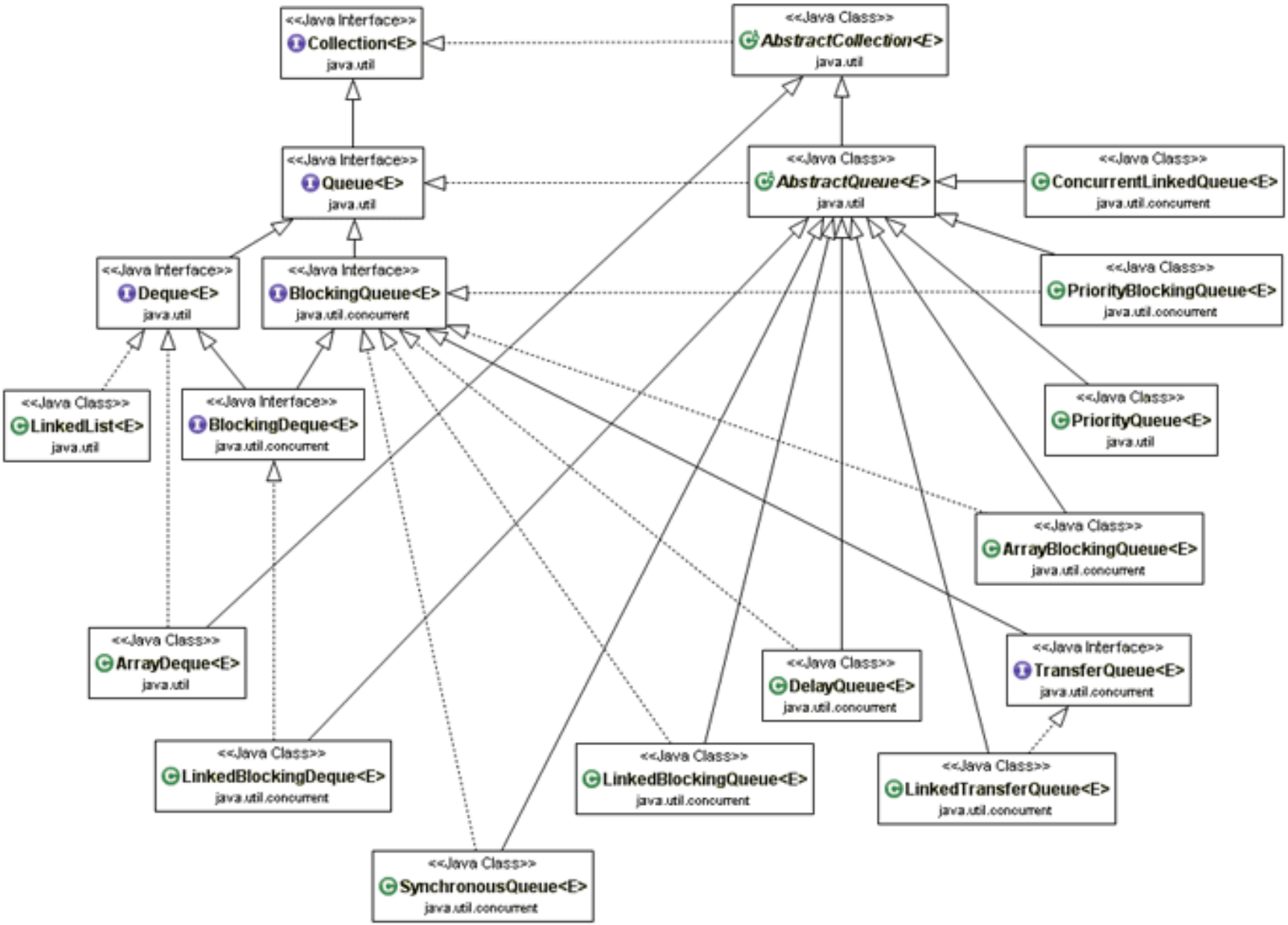
Spécifications

Définit les fonctionnalités pour une file d'objets (file d'attente)

Méthodes offertes

- `E element()` : Consulter le premier élément disponible sans le retirer de la collection. Cette méthode lève une exception si la collection est vide
- `boolean offer(E o)` : Ajouter l'élément dans la collection. Le booléen indique si l'ajout a réussi ou non
- `E peek()` : Consulter le premier élément disponible sans le retirer de la collection. Cette méthode renvoie null si la collection est vide
- `E poll()` Obtenir le premier élément et le retirer de la file. Cette méthode renvoie null si la collection est vide
- `E remove()` : Obtenir le premier élément et le retirer de la file. Cette méthode lève une exception si la collection est vide

Hiérarchie de type des Queue



Exemples d'implantations simplifiées de liste

Spécifications du problème

- Méthodes d'interface
  - `int size()` : taille de la liste
  - `boolean add(T e)` : ajouter à la fin de la liste
  - `T get(int i)` : lire l'élément à l'indice `i`
- Implantations
  - `MyArrayList<T>` qui se base sur un tableau
  - `MyLinkedList<T>` qui se base sur une liste simplement chaînée

La classe MyArrayList<T>

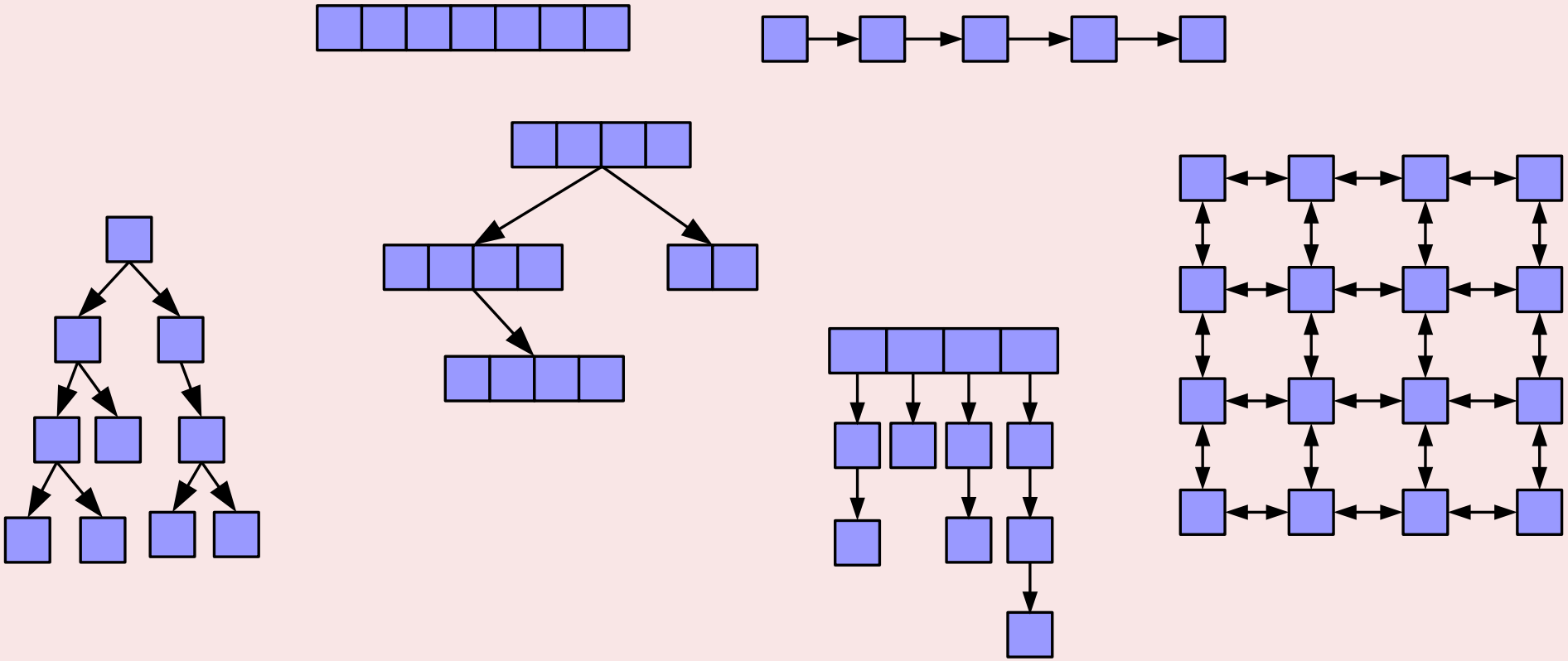
```
public class MyArrayList<T> {
  private T[] tab;
  private int size = 0;

  public MyArrayList(int max) {
    tab = (T[]) new Object[max];
  }

  public int size() {
    return size;
  }

  public boolean add(T element) {
    if (size == tab.length) {
      T[] tmp = (T[]) new Object[tab.length * 2];
      for (int i = 0; i < tab.length; i++) {
        tmp[i] = tab[i];
      }
      tab = tmp;
    }
    tab[size++] = element;
    return true;
  }

  public T get(int index) {
    if (index >= size)
      return null;
    return tab[index];
  }
}
```

La classe MyLinkedList<T>

```
public class MyLinkedList<T>{
  private Chainon<T> tete = null;
  private Chainon<T> queue = null;
  private int size = 0;

  public boolean add(T element) {
   if (tete == null) {
     tete = new Chainon<T>(element, tete);
     queue=tete;
   } else {
      queue.setNext(new Chainon<T>(element, null));
      queue = queue.getNext();
    }
    size++;
   return true;
  }

  public int size() {  return size; }

  public T get (int index) {
    if (index < 0 || index >= size())
      return null;
    Chainon<T> cur = tete;
    for (int i = 0; i < index ; i++) {
      cur = cur.getNext();
    }
    return cur.getData();
  }
}
```

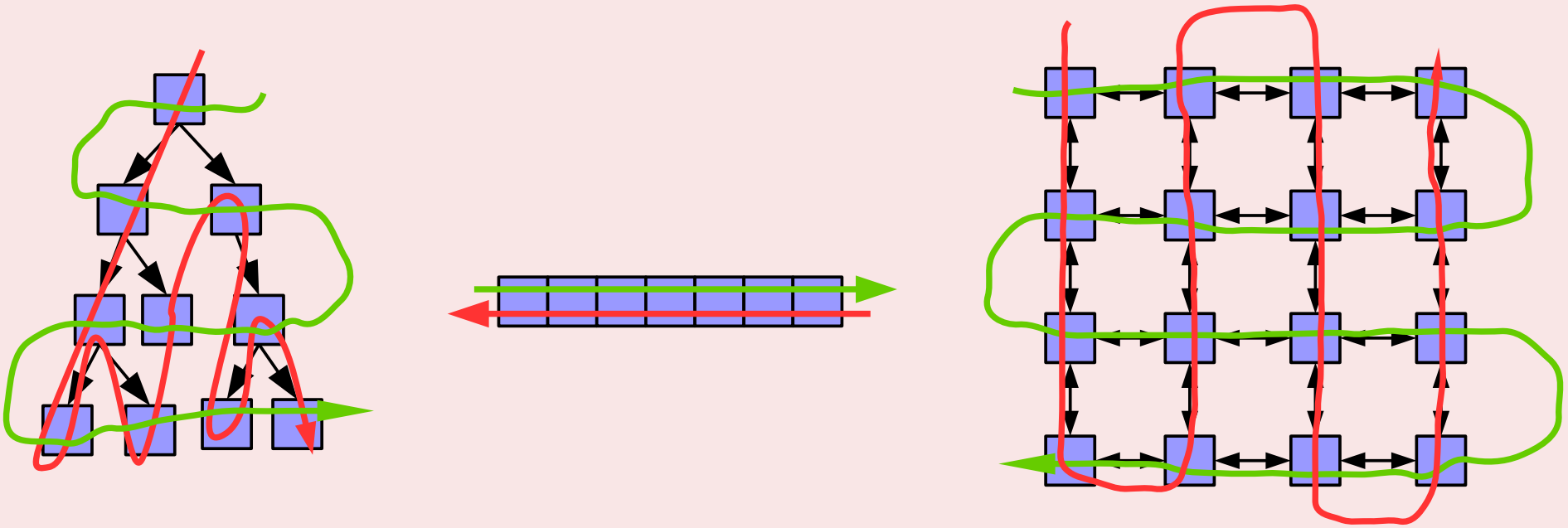
```
class Chainon<T> {
  private T data;
  private Chainon<T> next;

  public Chainon(T data,
                 Chainon<T> next)
                      {
    this.data = data;
    this.next = next;
  }

  public T getData() {
     return data;
  }

  public Chainon<T> getNext() {
     return next;
  }
  public void setNext(Chainon<T>
     next) {
     this.next = next;
  }
}
```

Problème 1 : Des structures très hétérogènes



Comment parcourir une collection indépendamment de sa structure ?

Problème 2 : Plusieurs parcours possibles



Comment s'abstraire de la politique de parcours ?
Quelle politique de parcours par défaut ?

Le design-pattern Iterator

Définition

- Patron de conception comportemental
- Permet de parcourir l'ensemble des éléments d'une collection sans exposer sans structure interne.

Étapes de mise en place

1) Abstraire le parcours $\Rightarrow$ Définition de interface itérateur
2) Permettre à la collection de produire un itérateur
3) Implanter la classe concrète de l'itérateur pour la collection voulue
4) Mettre à jour éventuellement le code client

Étape 1 : Abstraire le parcours

Quels sont les problèmes communs à tout parcours confondu ?

Étape 1 : Abstraire le parcours

Quelques problèmes communs

- **Y a-t-il encore des éléments à parcourir ?**
- **Quel est l'élément courant du parcours ?**
- **Aller au prochain élément à parcourir ?**
- Aller à l'élément précédent
- Effacer l'élément courant

Avec l'API Java : l'interface `Iterator<E>`

```
public interface Iterator<E> {
  boolean hasNext();// Est-ce la fin du parcours ?
  E next();//retourne l'élément courant et pointe sur
           //l'élément suivant
  default void remove() {
     throw new UnsupportedOperationException("remove");
  }
}
```

Étape 2 : Produire un itérateur depuis une collection

Comment faire pour que toute collection puisse produire un itérateur ?

Étape 2 : Produire un itérateur depuis une collection

Solution

Ajouter à l'interface des collections une (ou plusieurs) méthode renvoyant un nouvel objet itérateur

Avec l'API Java : l'interface `Iterable<E>`

```
public interface Iterable<E> {
    Iterator<E> iterator(); //renvoie un itérateur sur la
        collection
}
```

L'interface `Collection<E>` est une sous interface de `Iterable<E>`.

$\Rightarrow$ Toute collection doit implanter la méthode `iterator()`

$\Rightarrow$ Un seul parcours possible

Étape 3 : Implanter un itérateur

Comment coder une classe qui accède à l'état interne d'une collection sans déroger au principe d'encapsulation ?

Étape 3 : Implanter un itérateur

Solution 1

- Programmer une classe publique itérateur
  - ✗ Découplage de l'itérateur de sa collection
  - ✗ Accès difficile aux attributs de structure de la collection
  - ✗ On expose une classe qui n'a pas de sens d'exister sans sa collection
- Ajouter des getters sur les attributs de structures de la collection
  - ✗ L'exposition de la structure interne même via des méthodes peut être source de bug

Solution 2

La classes itérateur est une classe interne de la classe de collection :

- ✓ L'itérateur contient une référence implicite sur la collection
- ✓ L'itérateur peut accéder à l'état interne de la collection
- ✓ La classe d'itérateur n'est visible que par la collection

Étape 3 : Exemple MyArrayList (classe nommée)

```
public class MyArrayList<T> implements Iterable<T>{
  .....
  private class MyArrayListIterator implements Iterator<T> {
     private int index = 0;

     public boolean hasNext() {
       return index < size;
     }
     public T next() {
       if (!hasNext()) {
          throw new NoSuchElementException();
       }
       return tab[index++];
     }
  }

  public Iterator<T> iterator() {
    return new MyArrayListIterator();
  }
}
```

Étape 3 : Exemple MyLinkedList (classe anonyme)

```
public class MyLinkedList<T> implements Iterable<T>{
  .....
  public Iterator<T> iterator() {
    return new Iterator<T>() {//début classe anonyme
      private Chainon<T> cur = tete;

      public boolean hasNext() {
        return cur != null;
      }

      public T next() {
        if (!hasNext()) {
          throw new NoSuchElementException();
        }
        T data = cur.getData();
        cur = cur.getNext();
        return data;
      }
    };//fin classe anonyme
  }
}
```

Étape 4 : Mise à jour du code client

Comment utiliser l'interface de l'itérateur coté client ?

Étape 4 : Mise à jour du code client

Solution

- Récupérer une instance d'itérateur (cf. étape 2)
- Utiliser les méthodes de l'interface

En java

```
Collection<T> c= ??????;
Iterator<T> it = c.iterator();
while(it.hasNext()){
    T e = it.next();
    //traitement pour e
}
```

En java avec boucle foreach

```
Collection<T> c= ??????;
for(T e : c){
  //traitement pour e
}
```

.

Algorithmes de la classe Collections

Tri	`<T extends Comparable<? super T>> void sort(List<T> list)` `<T> void sort(List<T> list, Comparator<? super T> c)`
Recherche	`<T> int binarySearch(List<? extends Comparable<? super T>> list, T key)` `<T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)`
Nombre occurrences	`int frequency(Collection<?> c, Object o)`
Disjointure	`boolean disjoint(Collection<?> c1, Collection<?> c2)`
Mélange aléatoire	`void shuffle(List<?> list)` `void shuffle(List<?> list, Random rnd)`
Remplissage	`<T> void fill(List<? super T> list, T obj)` `<T> boolean addAll(Collection<? super T> c, T... elements)`
Copie	`<T> void copy(List<? super T> dest, List<? extends T> src)`

Collections vides (Singleton)	`<T> Iterator<T> emptyIterator()`
	`<T> Set<T> emptySet()`
	`<T> List<T> emptyList()`
	`<K,V> Map<K,V> emptyMap()`

Apport de Java 8

Apport du paradigme fonctionnel

- Certaines interfaces implantables via des lambda expression
- Un nouvel outil de programmation puissant : `Stream`
- Ajout de l'interface `Stream<E>` dans l'interface `Collection<E>`
  $\Rightarrow$ la collection est vue comme une source dans un pipeline de calcul
- Ajout de `void forEachRemaining(Consumer<? super E> action)` dans `Iterator<E>`
- ajout de `void forEach(Consumer<? super T> action)` et `Spliterator<T> spliterator()` dans `Iterable<T>`