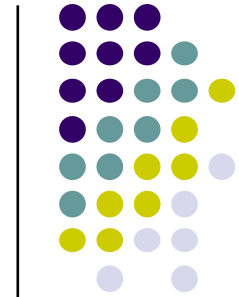
# Les collections

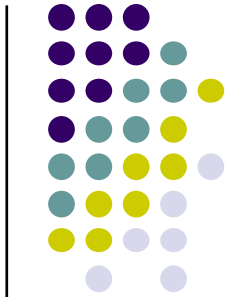
## Licence informatique 2ème Année

## POA – Prog. Objet Avancée

## Partie 2

*Françoise Greffier (+ Pierre-Alain Masson)*

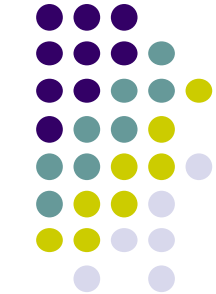
# Vue d'ensemble (1)



- Une ***structure de données collective*** est l'organisation *efficace* d'un ensemble d'éléments.
- Forme : tableaux, listes, piles, files d'attente…

  Exemples : promotion d'étudiants, sac de billes, pièces d'un jeu…
- ***Cette efficacité*** réside dans la quantité de mémoire utilisée pour stocker les éléments et le temps nécessaire pour réaliser les opérations (les plus fréquentes) sur ces données.

# Vue d'ensemble (2)



- Une ***collection*** regroupe plusieurs objets de même nature.

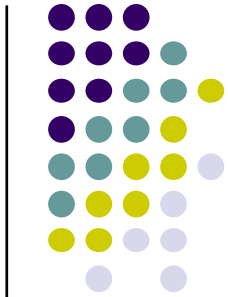
  Exemples : promotion d'étudiants, sac de billes, pièces d'un jeu ...

- Une ***structure de données collective*** (ou conteneur) implante une collection (ex: **ArrayList** d'étudiants)

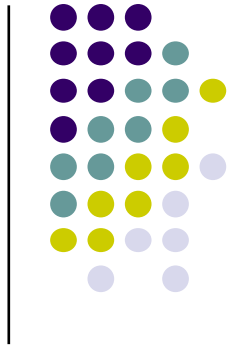
  Plusieurs implantations possibles :
  - ordonnées ou non, avec ou sans doublons, ...
  - accès, recherche, tris plus ou moins efficaces

- Objectifs : *réutilisabilité* + *standardisation*
  - Choisir dans les **API** (Application Programming Interface) une collection **adaptée** aux besoins de l'application.
  - Ne pas re-programmer les traitements répétitifs classiques (parcours, tri, recherche d'éléments, insertion, suppression, …)

# Exemples de collections

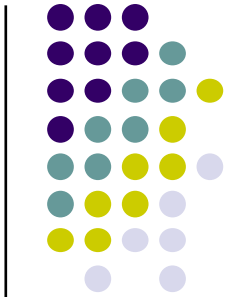


- Tableau — `type[]`
  - accès par index **efficace** : temps constant
  - recherche potentiellement efficace si le tableau est trié (dichotomie)
  - insertions et suppressions peu efficaces
  - Inconvénient : nombre d'éléments fixé à la création
- Liste doublement chaînée — `class LinkedList<E>`
  - accès séquentiel : premier, suivant (dernier, précédent)
  - insertions et suppressions efficaces
  - Recherche d'un élément : lente, non efficace
- Tableau dynamique — `class ArrayList<E>`
  - Accès par index + taille variable
  - Syntaxe et création plus lourde que pour les tableaux



# *Organisation des collections en Java*

# Collections en Java



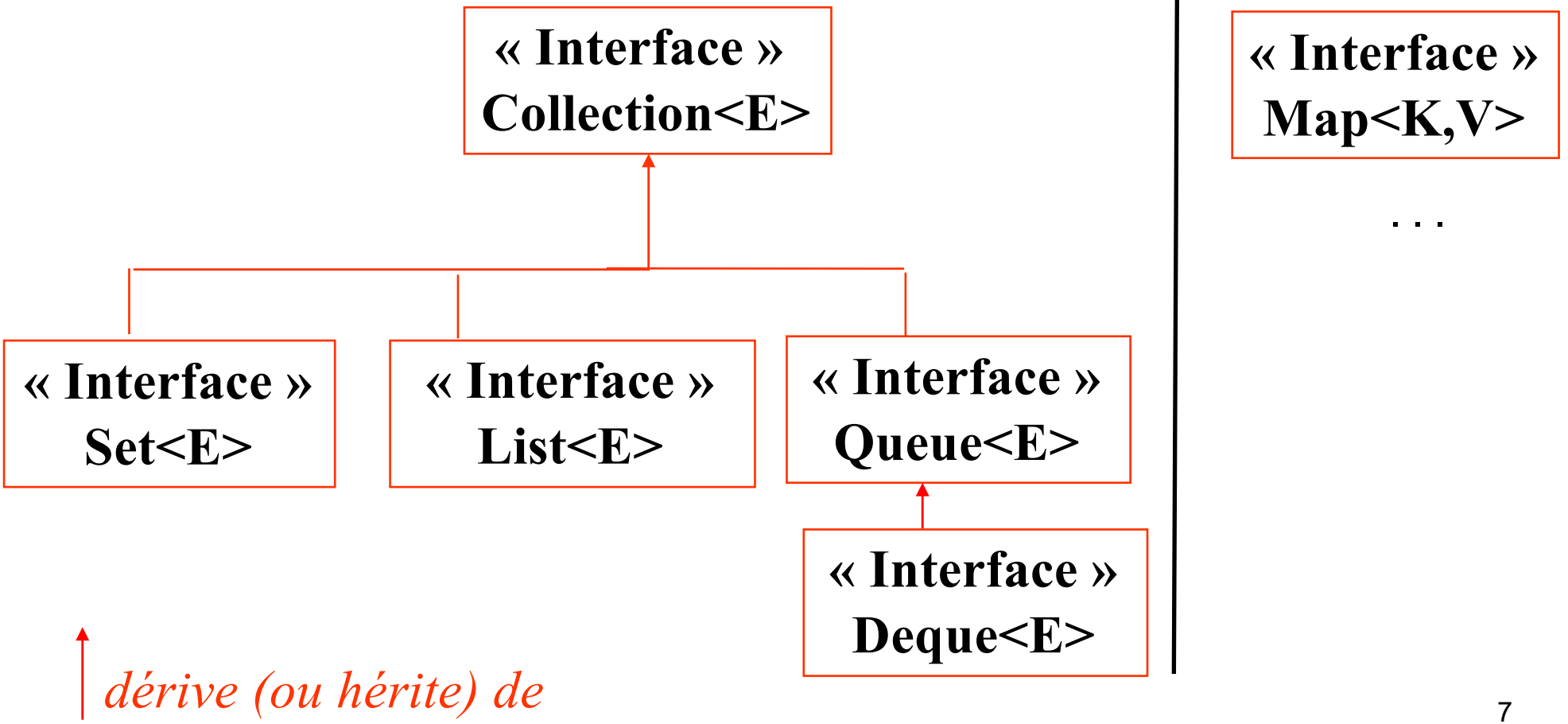
**Avant le JDK 1.5** (*Java Development Kit*), les collections n'étaient **pas génériques** en Java. On trouvait par exemple la collection **List**, **ArrayList** qui pouvait contenir des **Object** de n'importe quelle classe. Ainsi la signature de méthodes des API(s) n'utilisaient que la classe **Object**. Les traitements demandaient alors souvent de « transtyper» les valeurs manipulées => lourd !

**Depuis JDK 1.5**, Java propose une architecture d'interfaces, et de **classes génériques** permettant de stocker et de manipuler efficacement des collections.

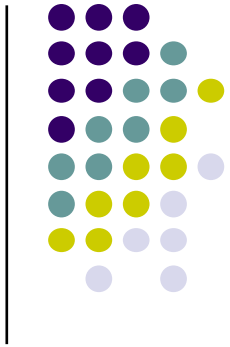
Paquetage : `java.util`
- Interfaces
- Classes abstraites et concrètes
- Algorithmes : tri, recherche…
- Vues

# Collections en Java

• Les collections en java sont réparties en deux groupes.
On le voit à travers les interfaces (schéma simplifié) qui permettent d'implémenter des collections.



# Paquetage `java.util` de Java

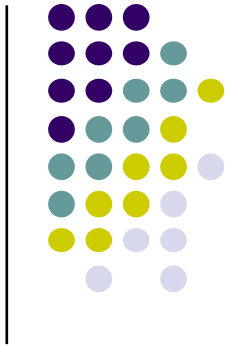


**Interface Collection<E>** **Interface MAP<K,V>**

• Les conteneurs qui implantent **l'interface Collection<E>** permettent de regrouper des objets « individuels » de type **E** ou d'un type compatible.

• Les conteneurs qui implantent **l'interface Map<K,V>** permettent de regrouper des associations *clé/valeur* (clé de type **K**, valeur de type **V**).
Ex : dictionnaire, annuaire

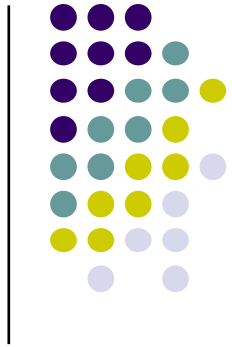
# Paquetage `java.util` de Java



**Interface Collection<E>** **Interface MAP<K,V>**

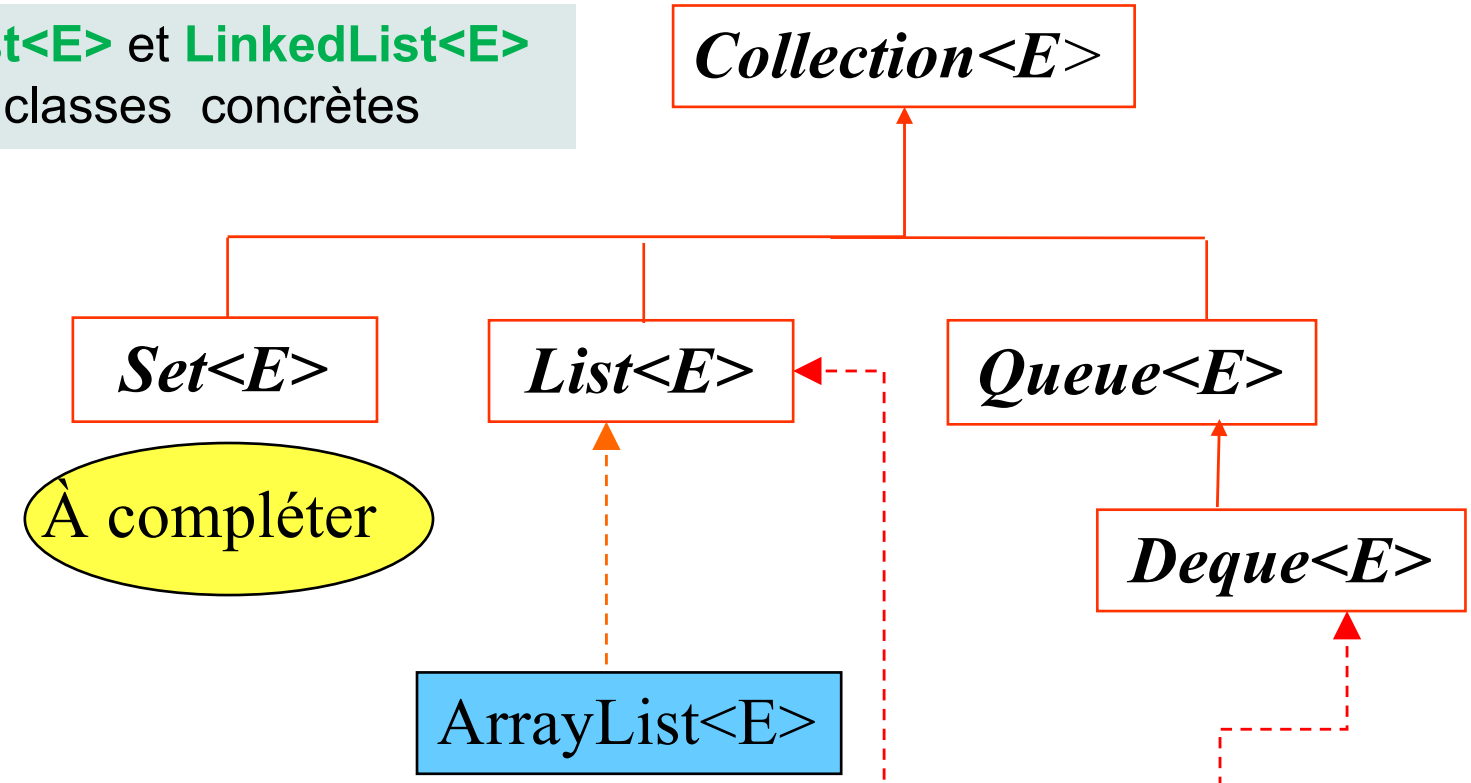
Nous nous intéressons nous d'abord aux conteneurs qui implémentent **l'interface Collection <E>**

# Interface Collection<E>



*implémente* *dérive de*

**ArrayList<E>** et **LinkedList<E>** sont des classes concrètes



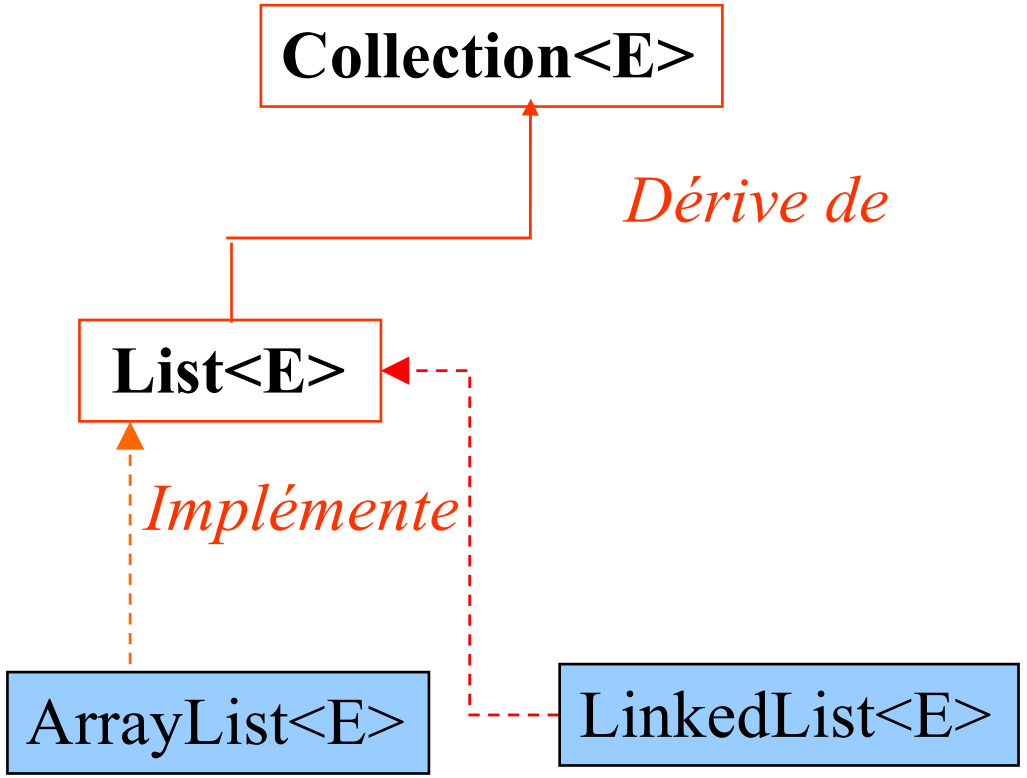

La ou les interface(s) implémentée(s) indique(nt) les **traitements** disponibles : ce sont les **méthodes** énumérées dans l'interface

# Interface List<E>

Deux classes concrètes et Génériques :
- **ArrayList<E>**
- **LinkedList<E>**

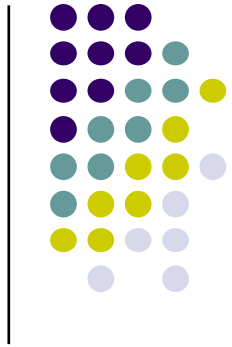
implémentent l'interface **List <E>** :

Ce sont deux structures de données collectives implémentant **List <E>** et héritant de **Collection<E>**



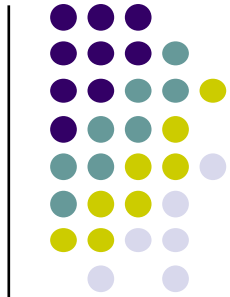

Apprendre à utiliser les colllections : vous devrez vous plonger dans les API pour connaître :

- Les opérations disponibles (**méhodes**)
- Comment les utiliser : syntaxe (**paramètres, type de retour**)+ spécification



# Interface Collection<E>

# Généralités : Collection<E>

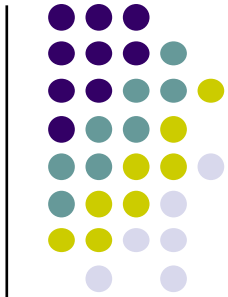


- **Méthodes courantes (ajout, suppression, etc.)**

- **Relation d'ordre entre les éléments**
  - **Ils doivent implanter l'interface Comparable<E>**
- **Égalité entre des éléments d'une collection**
  - **Ils doivent redéfinir la méthode equals() de Object**
- **Rangement des éléments de la collection**
  - **Ils doivent redéfinir la méthode hashCode() de Object**

- **Parcours d'une collection**
  - **Utilisation d'un itérateur (interfaces Iterable<E> ou Iterator<E> )**

# Egalité : equals() héritée de Object

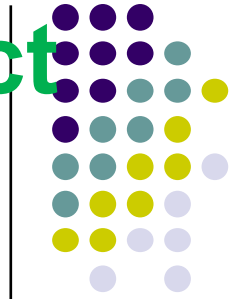


- **A faire redéfinir par les éléments**
- **`boolean equals(Object o)`**
  - Test d'égalité du **contenu** de **this** et **o**
  - Relation *réflexive*, *symétrique* et *transitive*
  - Contrat :
    - Égalité des références => égalité des objets référencés
    - **this** ne peut jamais être égal à **null**
    - Deux objets de classes différentes ne peuvent pas être égaux

```
@Override
public boolean equals(Object o) {
        if (this==o)
                return true;
        if (o==null || this.getClass() != o.getClass())
                return false;
// Puis : Transtypage descendant de O (commodité syntaxe)
// Enfin : test d'égalité du contenu de this et o
```

Indique explicitement au compilateur qu'on redéfinit une méthode existante

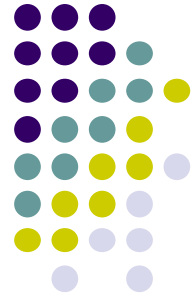
# Méthode hashCode() héritée de Object



- **Tout élément qui redéfinit equals() doit aussi redéfinir hashCode()**
- **`int hashCode()`**
  - Retourne un entier représentant l'instance courante
  - Contrat :
    - Le *hashCode* ne change que si l'une des propriétés testées dans **equals()** change
    - Si **c1.equals(c2)** alors **c1.hashCode()==c2.hashCode()**
    - Collisions à éviter mais possibles : des objets non-égaux pourraient avoir le même *hashCode*
  - Génération automatique de la méthode **hashCode()**
    - Les IDE proposent la génération automatique de **hashCode()** (ainsi que **equals()**)
    - Eclipse : menu Source / Generate hashCode() and equals()…

# Quelques méthodes de l'interface Collection <E>

https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html



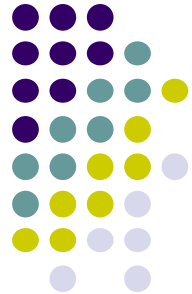
## Opérations basiques :

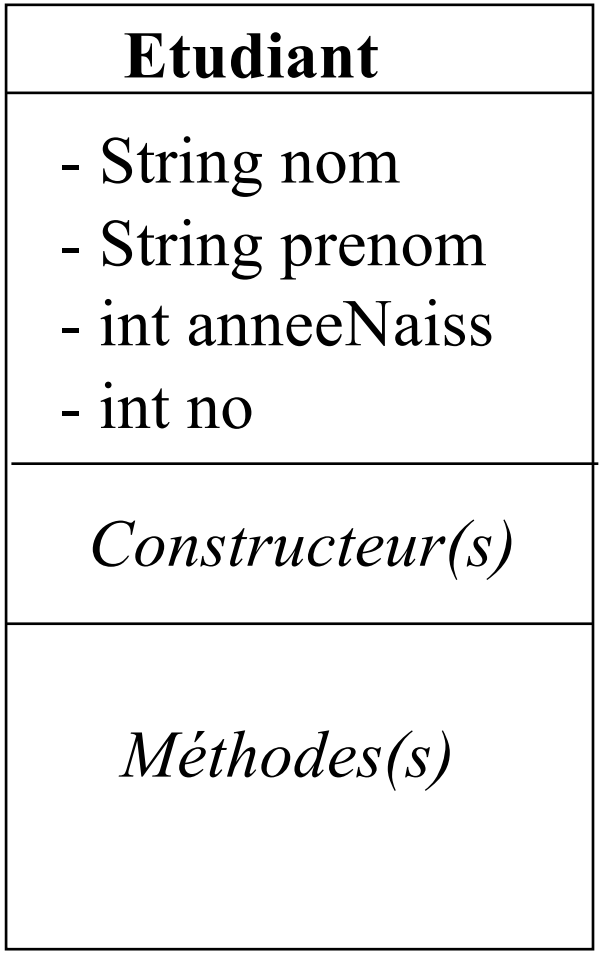
```
int size( );
boolean isEmpty( );
boolean contains(Object elt);
boolean add(E elt);
boolean remove(Object elt);
Iterator<E> iterator();
```

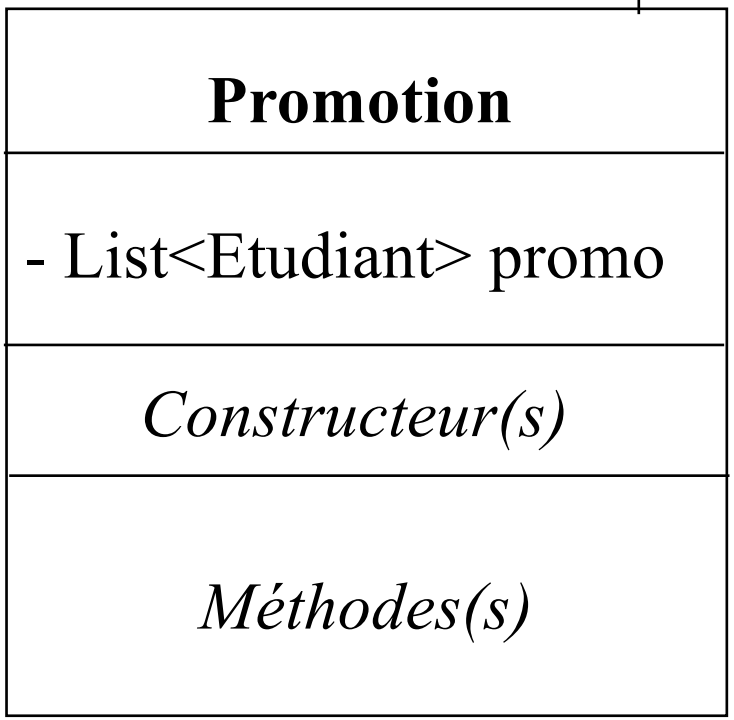
**boolean addAll (Collection <? extends E> c2)**

- ajoute les éléments de **c2** à la collection **this**
- Les éléments de **c2** doivent être de type compatible (identique ou dérivé) avec le type générique (noté **E**) de la collection **this**.

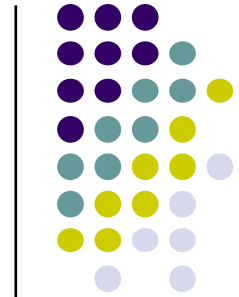
# Exemple : promotion d'étudiants dans une liste





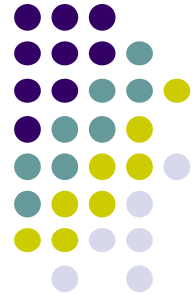



# Classe Etudiant



```
public class Etudiant{
   private String nom;
   private String prenom;
   private int anneeNaiss;
   private int no;

public Etudiant(String n,String p,int a,int no) {
        this.nom=n;
        this.prenom=p;
        this.anneeNaiss=a;
        this.no=no;
   }
...
}
```

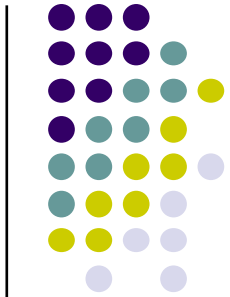
# Classe Promotion



```
import java.util.List;
import java.util.ArrayList;
public class Promotion {
   private   List<Etudiant> promo;

   public Promotion() {
      promo= new ArrayList<Etudiant>();
   }
   public void ajouter (Etudiant e) {
      promo.add(e);
      //add : méthode de Collection<E>
   }
...
}
```

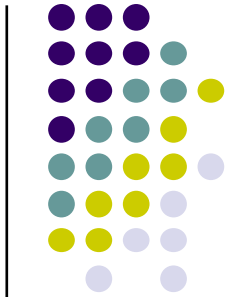
On type par l'interface

On instancie par une classe concrète



# *Classe Collections*

# Java.util.Collections



L'ensemble des algorithmes manipulant les collections se trouve dans la classe Collections. Les méthodes sont **statiques**.

- **TRIER**

```
static void sort(List<E>  l);
static void sort(List<E> l, Comparator<E> c);
```

- **MELANGER**

```
static void  shuffle(List<E> l);
```

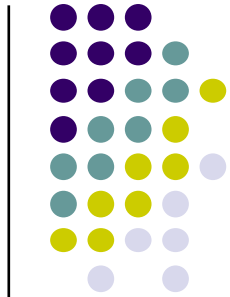
- **MANIPULER**

```
static void reverse (List<E>  l);
static void copy (List<E>  lDest, List<E> lSrc);
```

- **RECHERCHER**

min, max Etc.

# Trier les éléments d'une liste



*But : trier la liste d'étudiants par ordre alphabétique*

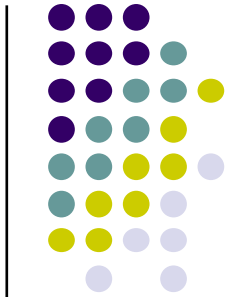
## Java.util.Collections

`static void sort(List<T> list)`

Sorts the specified list into ascending order, according to the natural ordering of its elements.

# Trier une promotion d'étudiants

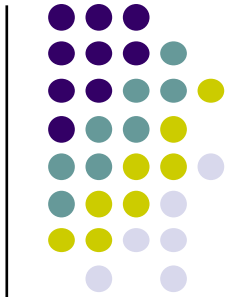
```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Promotion {
      private List<Etudiant> promo;

      public Promotion() {
          promo= new ArrayList<Etudiant>();
      }
public void trier () {
      Collections.sort(promo);
//Sorts the specified list into ascending order,
//according to the natural ordering of its elements.
}
}
```



Relation d'ordre *naturelle* sur les éléments d'une collection

*Exemple : trier la liste d'étudiants par ordre alphabétique (nom et prénom).*

# Relation d'ordre



**Ordonnancement des éléments d'une collection :**
Indépendamment d'un ordre naturel (ex : par rang), on a souvent besoin de classer les éléments d'une collection à partir de leur valeur (recherche de min, tri,…).

**Deux façons de définir une relation d'ordre sur E** :

1. Les éléments implémentent l'interface `Comparable<E>` (c'est **l'ordre naturel** (*natural ordering*))
2. On passe un objet comparateur aux méthodes de tri pour préciser la relation d'ordre.

# Interface Comparable<E>

Les éléments eux-même spécifient leur ordre naturel.

Cette interface comprend une seule méthode

```
int compareTo(E elt);
```

*Retourne 0 si objet courant (this) égal à elt*

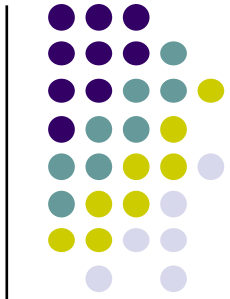
*Retourne une valeur négative si objet courant (this) est « inférieur » à elt*

*Retourne une valeur positive si objet courant (this) est « supérieur » à elt*

---

**ATTENTION** : le test d'égalité de la méthode **`equals()`** doit être cohérent avec l'effet de la méthode **`compareTo()`**

# Une relation d'ordre naturelle (alphabétique) classe étudiant



```
public class Etudiant implements Comparable<Etudiant>
{
        private String nom;
        private String prenom;
        private int annee_naiss;
        private int no;
```

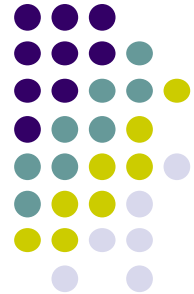
Sans cela, la méthode Collections.sort() ne peut pas fonctionner

```
public int compareTo (Etudiant e) {
   int r = this.getNom().compareTo(e.getNom());
   if (r==0)
     r =this.getPrenom().compareTo(e.getPrenom());
   return r;
}
```

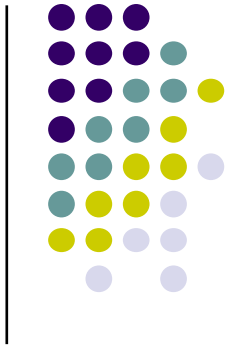


```
}
```

# Trier une promotion d'étudiants



```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Promotion {
      private List<Etudiant> promo;

      public Promotion() {
          promo= new ArrayList<Etudiant>();
      }

      public void trier () {
            Collections.sort(promo);
      }
```

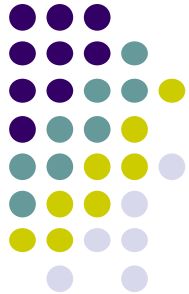


Relation d'ordre *alternative* sur les éléments d'une collection

*Exemple : trier la liste d'étudiants par années de naissance croissantes*

On fournit à la méthode **sort()** une instance qui implémente l'interface **Comparator<E>**

# Trier les éléments d'une liste



## Java.util.Collections

```
static void sort (List <T> list,
        Comparator <? super T> c)
```

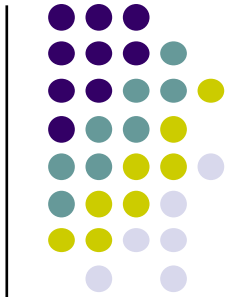
Sorts the specified list according to the order induced by the specified comparator.

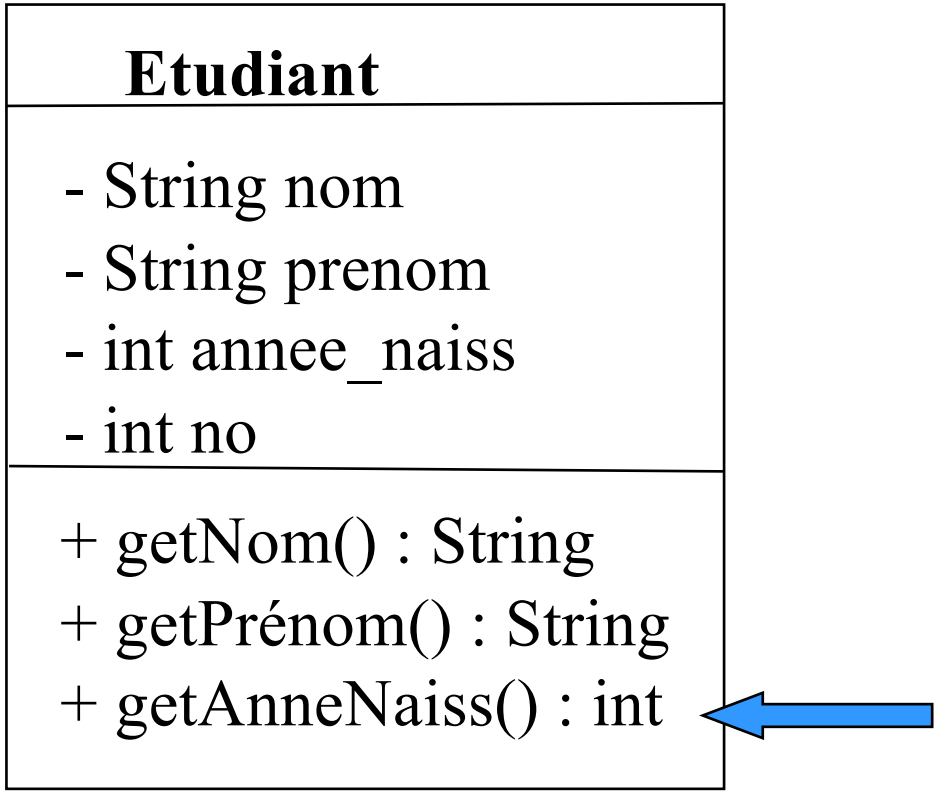
# Interface Comparator<E>

Cette interface comprend une seule méthode :

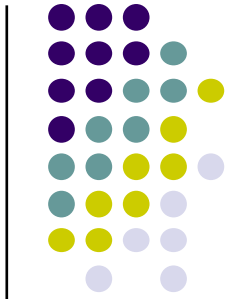
```
int compare(E e1, E e2)
//Compares its two arguments for order.
// Returns a negative integer, zero,
//or a positive integer
// as the first argument is less than, equal to,
// or greater than the second.
```

# Exemple de comparateur (relation d'ordre sur les années de naissance)



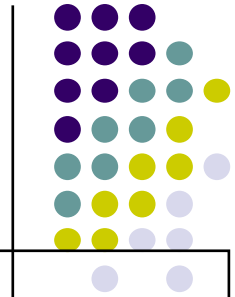


# Exemple de comparateur (relation d'ordre sur les années de naissance)



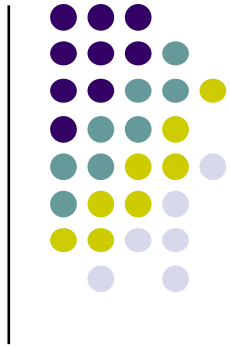
```
import java.util.Comparator;

public class CompAnnee implements Comparator<Etudiant> {
  public int compare(Etudiant e1, Etudiant e2){
    return (new Integer(e1.getAnneeNaiss())).compareTo
                          (new Integer(e2.getAnneeNaiss()));
  }
}
```

# Trier une promotion d'étudiants par années de naissances croissantes



```
import java.util.List;
import java.util.ArrayList;
import java.Collections;

public class Promotion {
     private List<Etudiant> promo;

     public Promotion(  ) {
         promo= new ArrayList<Etudiant> ();
     }
 public void trierAnneeNaiss ( ) {
   Collections.sort (promo, new CompAnnee());
 }
}
```

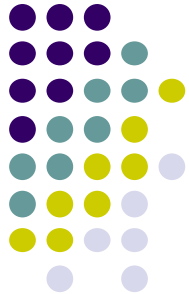
Création du comparateur



Relation d’ordre sur les éléments d’une collection

*Egalité de deux éléments*

# Égalité de deux éléments d'une collection



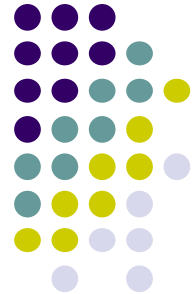
***Une collection ne contient que des références sur des instances.***

**Égalité des éléments d'une collection :**

Dans la classe `Object` : la méthode `equals` teste l'égalité des références. Par conséquent, il est nécessaire de redéfinir la méthode `equals` dans la classe E, si l'on veut que le test d'égalité porte sur la valeur (ou une partie de la valeur) des instances issues de la classe E.

Cette méthode `equals` peut-être déclenchée par des méthodes des interfaces des collections.

# Test d'égalité de deux objets



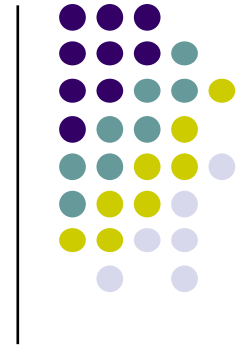
```
public boolean equals (Object o)
```

REMARQUE : la méthode `equals` est redéfinie dans la classe `String` et dans les classes enveloppes (ex : `Integer`)

---

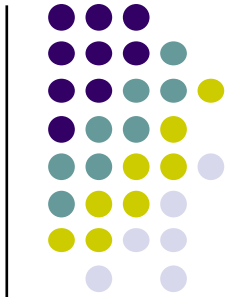
**ATTENTION** : le test d'égalité de la méthode **`equals`** doit être cohérent avec l'effet de la méthode **`compareTo`** dans la classe élément (exemple : classe `Etudiant`).

# Égalité : méthode `equals`



```
public class Etudiant {
…
    public boolean equals (Object o) {
    // est retourné vrai si this et o ont un prénom et un nom identiques;faux sinon

    if (! (o instanceof Etudiant)) return false;
    Etudiant e=(Etudiant)o;
    return (this.getNom().equals(e.getNom()) &&
                  this.getPrenom().equals(e.getPrenom()))
    }

}
```

L'opérateur `instanceof` retourne `false` si l'objet o est égal à `null`.



# *Parcourir une collection*

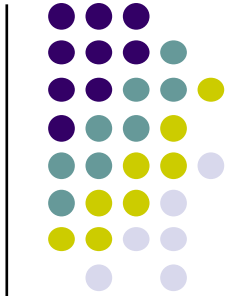
# Parcours d'une collection

Pour « parcourir » successivement, un à un, les éléments d'une collection, on utilise un **itérateur** associé à cette collection .

Chaque classe qui est une collection dispose d'une méthode nommée `iterator` fournissant un itérateur monodirectionnel associé à la collection. Cet itérateur permet (en s'affranchissant de son implémentation) de parcourir une collection

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
public class Promotion {
   private  List<Etudiant> promo;
. . .
public void parcourir (  ) {
      Iterator<Etudiant> iter = promo.iterator( );
      …
}
```

# Interface Iterator<E>

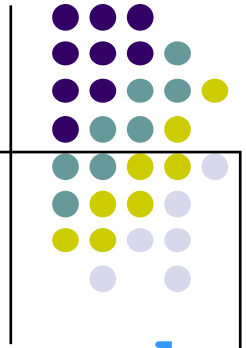


Pour « parcourir » successivement, un à un, les éléments d'une collection, on utilise un **itérateur.**
L'interface Iterator<E> est spécialisée dans le parcours séquentiel des éléments d'une collection.
Elle propose les méthodes suivantes :

```
public interface Iterator<E> {
    boolean   hasNext();
    E next( );
    void remove(); //optionnel
}
```

Un itérateur implémente l'interface Iterator<E>

# Interface Iterable<E>



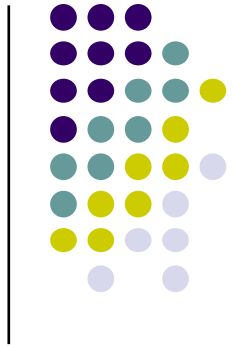
```
public interface Iterable<E> {
    Iterator<E> iterator()
// est retourné : un itérateur monodirectionnel
}
```

Chaque classe qui est une collection dispose d'une méthode nommée `iterator` car `Collection<E>` implémente `Iterable<E>`

EXEMPLE

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
public class Promotion {
   private  List<Etudiant> promo;

public void parcourir ( ) {
       Iterator<Etudiant> iter = promo.iterator( );

       . . . // à terminer
}
```

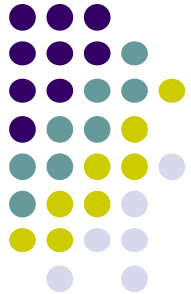
# Collections en Java






Chaque classe qui est une collection dispose d'une méthode nommée `iterator` car `Collection<E>` implémente `Iterable<E>`

# Propriétés d'un itérateur monodirectionnel

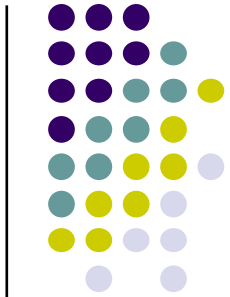


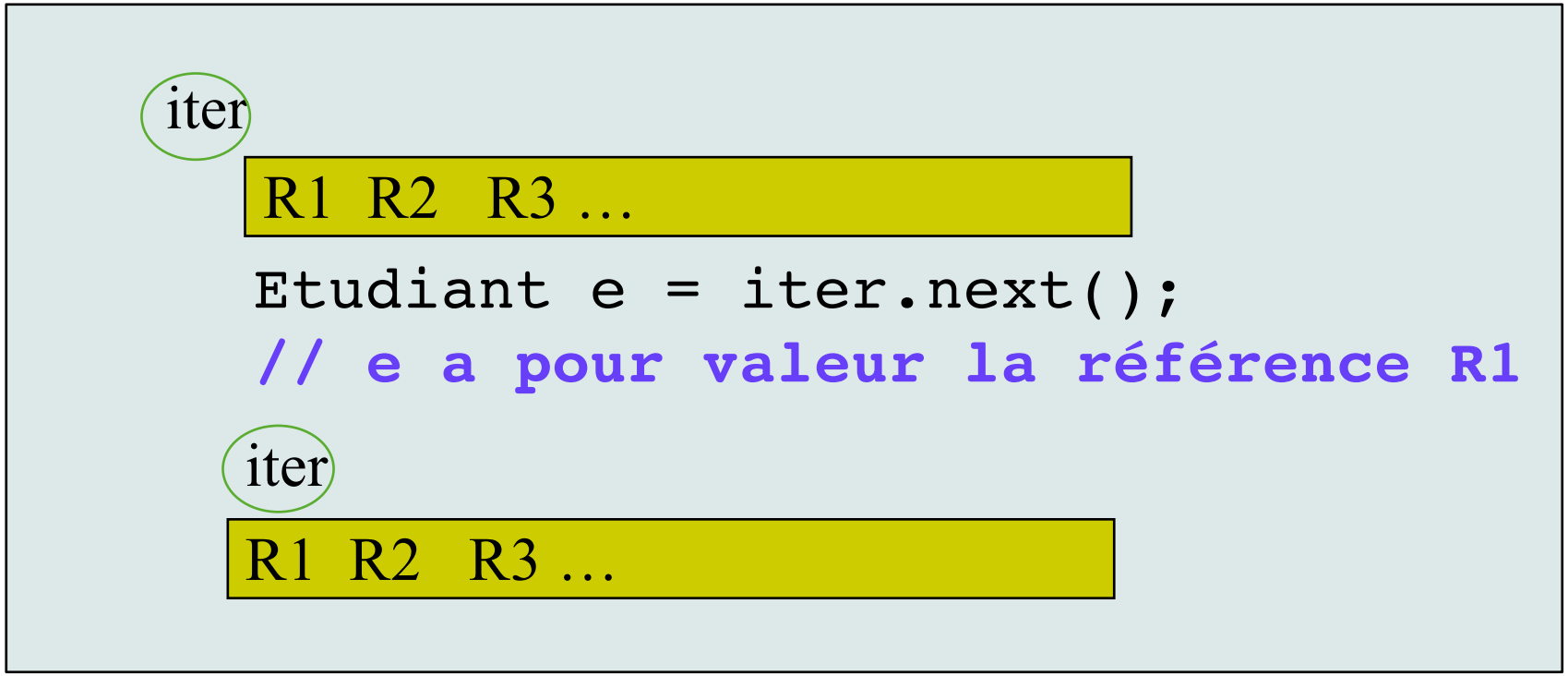
A tout instant l'itérateur indique une position courante.
Cette position courante positionne l'accès au premier ou au $n^{ième}$ élément de la collection associée. Il peut aussi indiquer une position située immédiatement après le dernier élément.

C'est via la position qu'indique l'itérateur qu'on pourra accéder au premier élément, au suivant… etc., jusqu'au dernier.

Lorsqu'on construit un itérateur associé à une collection, cet itérateur indique la position qui précède le premier élément de la collection s'il existe.

# Propriétés d'un itérateur monodirectionnel

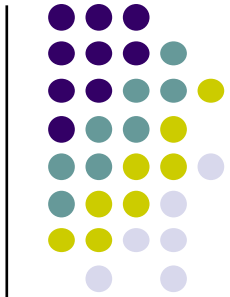




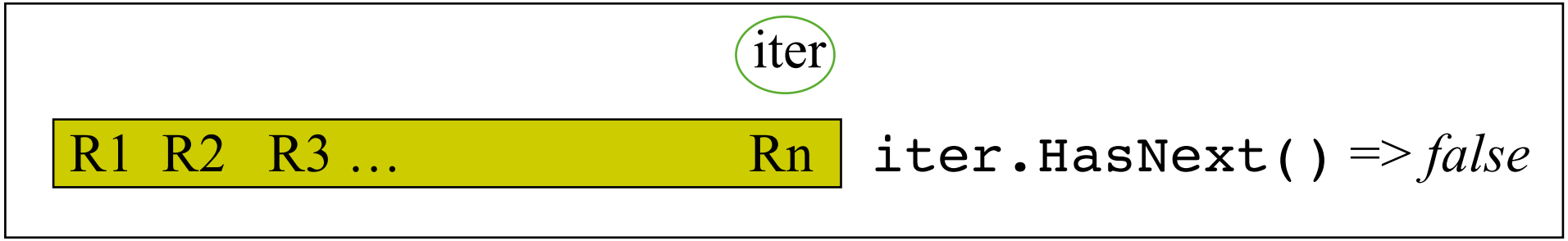

L'exécution de la méthode `next()` permet :
d'abord de faire progresser l'itérateur d'une position en avant.
PUIS de retourner la référence de l'élément courant (ici R1)

En cas de problème (ex : pas d'élément suivant) l'exception `NoSuchElementException` est déclenchée.

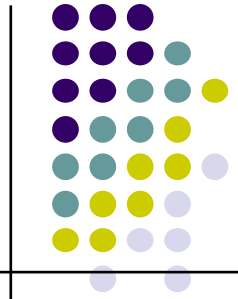
# Propriétés d'un itérateur monodirectionnel



La méthode has`Next:` retourne `false` si l'itérateur n'a pas d'élement suivant.

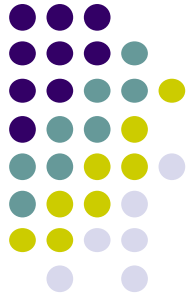


# Parcours d'une collection



```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class Promotion {
   private  List<Etudiant> promo;
. . .
public void parcourir ( ) {
      Iterator<Etudiant> iter = promo.iterator( );
      while (iter.hasNext()) {
         Etudiant e=iter.next();
          // traiter élément courant e
        }
}
```

# Parcours d'une collection–version 2 for … each



C'est `Iterable <E>` qui permet d'utiliser la boucle "foreach" apparue avec Java 5

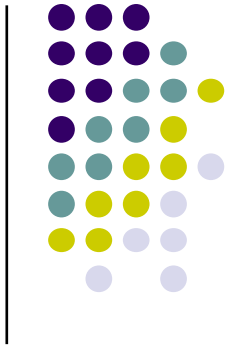
```
public class Promotion {
   private  List<Etudiant> promo;
public void parcourir ( ) {
          for (Etudiant e : promo) {
              // traiter élément courant e
            }
}
```

## Boucle *for..each*

La variable e prend successivement la valeur de chacune des références incluses dans la collection nommée promo.

**ATTENTION** : cette façon de programmer le parcours n'est pas exploitable si l'on doit modifier la collection, par exemple en utilisant des méthodes telles que *remove* ou *add* qui se fondent sur la position courante d'un itérateur.

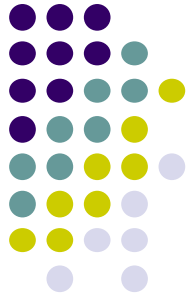
# `remove` de l'interface Iterator<E>



La méthode `remove()` de l'interface **Iterator<E>**
permet de supprimer l'élément « pointé » par le dernier `next()`,
et seulement le dernier.

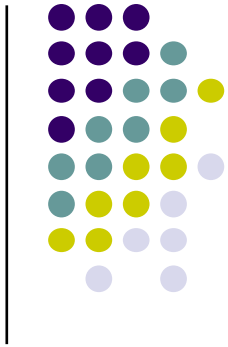
Cette méthode est optionnelle : elle n'est pas obligatoirement disponible.
Une méthode optionnelle n'est pas forcément implantée dans toutes les classes qui implémentent l'interface.
Quand elle n'est pas implémentée elle peut renvoyer une exception de type `UnsupportedOperationException`.

# Parcours unidirectionnel d'une collection



```
import java.util.Collection;
import java.util.ArrayList;
import java.util.Iterator;
public class PromotionV2{
   private Collection<Etudiant> promo;

        public PromotionV2() {
                promo= new ArrayList<Etudiant>();
        }
        public boolean ajouter (E elt) {
                return promo.add(elt);
        }

        public void parcourir () {
                for (Etudiant e: promo)
                        System.out.println (e);
        }
        public void parcourir2 () {
                Iterator<Etudiant> iter = promo.iterator( );
                while (iter.hasNext())
                        System.out.println (iter.next());
        }
}
```

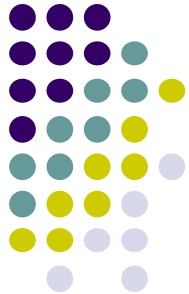
# Les itérateurs bidirectionnels



Certaines collections qui implémentent l'interface List<E> (listes chaînées et tableaux dynamiques) peuvent être parcourues dans les deux sens.
On a alors recours à un **itérateur bidirectionnel** qui est un objet implémentant l'interface ListIterator<E> dérivée de Iterator<E>.

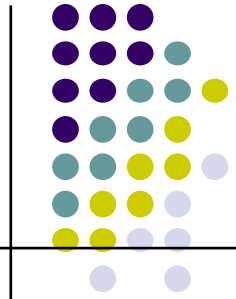
Un objet de type ListIterator<E> possède donc toutes les méthodes de Iterator<E>.
Il possède des méthodes supplémentaires, avec notamment :
`previous( )` et `hasPrevious( )`

# Les listes doublement chaînées (LinkedList<E>)

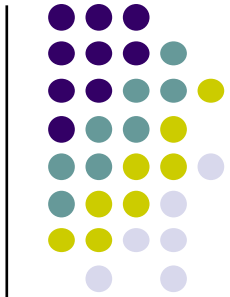


Les listes chaînées sont doublement chaînées :
On associe à chaque élément de la liste chaînées, une référence vers son prédécesseur et une autre référence vers son successeur.
=> itérateurs bidirectionnels très adaptés pour cette structure collective.

# Parcours de la fin vers le début d'une liste doublement chaînée

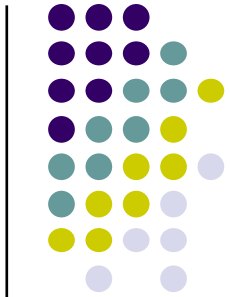


```
import java.util.List;
import java.util.ArrayList;
import java.util.ListIterator;

public class Promotion {
   private  List<Etudiant> promo;

public void parcoursInverse ( ) {
       ListIterator<Etudiant> iter;
       iter= promo.listIterator(promo.size());
       // la position courante de iter est en fin de liste
       while (iter.hasPrevious()) {
          Etudiant e=iter.previous();
           // traiter élément courant e
         }
}
...
}
```



# *List<E> en détails*

# Polymorphisme d'Interfaces



```
public class Promotion {
private  Collection<Etudiant> promo;      1  ?
private  List<Etudiant> promo;            2  ?
        public void parcoursInverse ( ) {…}
```

**Pb avec 1** :  **private** Collection <Etudiant> promo;

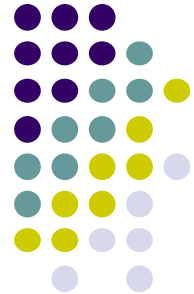
*Promo est de type déclaré  Collection<Etudiant>*

… promo=**new** ArrayList<Etudiant>;

*promo est de type réel  ArrayList<Etudiant>;*

**On ne peut appliquer à promo que les méthodes prévues dans l'interface du type déclaré Collection<E> :**
**OK pour les itérateurs monodirectionnels. MAIS on ne peut pas utiliser les itérateurs bidirectionnels.**

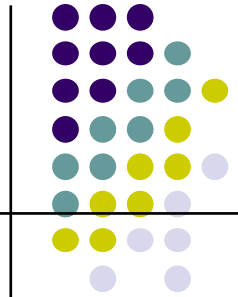
# Quelques méthodes de l'interface List <E>



## Opérations basiques :

```
boolean add(E elt);

int indexOf(Object o);
// retourne la position de la première occurrence de o

E get(int i);
// retourne l'élément de position i
//si i incorrect =>java.lang.IndexOutOfBoundsException

E set(int i, E elt);
// Remplace l'élément à la position fournie en paramètre
// Retourne l'ancien élément à la position fournie

void add(int i, E elt);
// Ajoute un élément à la position fournie en paramètre

. . .
```

# Ajout d'un élément : add



L'interface Collection<E> possède la méthode

**`boolean add (E elt)`** ;

Effet : ajouter **`elt`** en **fin** de collection

Par exemple : dans une liste chaînée ou dans un tableau dynamique.

**Elle est indépendante d'un quelconque itérateur.**

Retour : retourne *true* si l'ajout a été réalisé, *false* sinon.

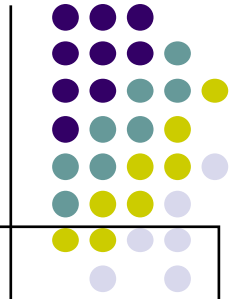
Exemple : retourne *false* si l'on tente d'ajouter dans un **ensemble** (Set<E>) un élément déjà « présent ».

L'interface ListIterator<E> possède une méthode d'ajout :

**`void add (E elt) ;`** Appel : `iter.add(e);`

Effet : ajouter **elt** à la position courante dans une liste chaînée ou dans un tableau dynamique. Si la position courante est en fin de collection alors l'ajout se fait à la fin. La position courante est ensuite déplacée après l'élément ajouté.

# Suppression d'un élément : remove



L'interface Collection<E> possède la méthode

**`void remove (Object e )`**;

Effet : supprime la première occurrence de **`elt`** si elle existe.

**Elle est indépendante d'un quelconque itérateur.**

Retour : retourne *true* si la suppression a été réalisée, *false* sinon.

Retourne *false* si l'on tente de supprimer un élément non « présent ».

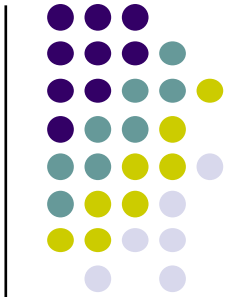
L'interface Iterator<E> possède une méthodes de suppression :

**`void remove ( ) ;`** Appel : `iter.remove();`

Effet : supprime l'élément situé à la position courante dans une liste chaînée ou dans un tableau dynamique.

Exemple d'utilisation : suppression sous une condition.

# Remplacer un élément : set

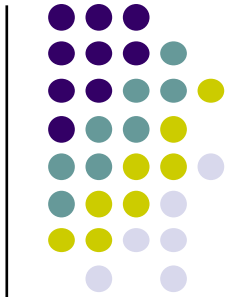


La méthode **`set`** **n'existe pas** dans l'interface Collection<E>.
La méthode : **`E set (int index, E elt)`** existe dans l'interface List<E>.
Effet : remplace l'élément situé à l'indice index par **`elt`**
**Elle est indépendante d'un quelconque itérateur.**
Retour : retourne *l'élément* qui a été remplacé.

L'interface ListIterator<E> possède une méthode de remplacement :

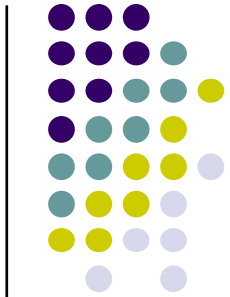
**`void set (E elt ) ;`** Appel : `iter.set(e);`

Effet : remplace l'élément situé à la position courante par e dans une liste chaînée ou dans un tableau dynamique.
Exemple d'utilisation : remplacement sous une condition.



# *Efficacité des algorithmes*

# Efficacité des algorithmes

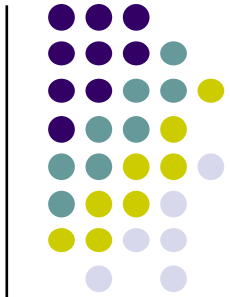


**L'efficacité d'une méthode est mesurée par son temps d'exécution.**

**Ce temps t est noté O(*x*), ce qui veut dire que t est une fonction de *x*. O(*x*) est appelé complexité.**

Par exemple :
Pour savoir si un élément est inclus dans une liste chaînée (`contains`), on doit accéder au K-ième élément de la liste chaînée.
Cette opération nécessite K étapes. Les éléments étant aléatoirement répartis dans une liste chaînée, on pourra dire qu'en moyenne cette opération se réalise en N/2 étapes.
Autrement dit la complexité est proportionnelle à la taille de la liste.
Ce qui se note 0(N). N étant la taille de la liste.

# Efficacité des algorithmes



**`LinkedList<E> (liste chaînée) :`**

• **boolean** contains(object elt) : O(N) (=linéaire) car on recherche l'élément en parcourant potentiellement tous les éléments de la liste.

• **Ajout ou suppression à la position courante** : O(1) (= temps constant) car on modifie les références à prédécesseur et à successeur.

• **boolean** remove(Object elt) : O(N) car on cherche l'élément à supprimer en parcourant potentiellement tous les éléments de la liste.

# Efficacité des algorithmes

**`ArrayList<E> (tableau dynamique) :`**

- **boolean** contains(Object elt) : O(N)

- **Ajout ou suppression à la position courante** : O(N)
  car on décale les éléments.

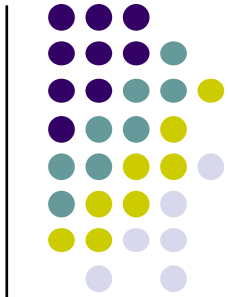
- **boolean** remove(Object elt) : O(N) car on cherche l'élément à supprimer en consultant potentiellement tous les éléments de la liste.

- **Accès à un élément par son indice**

List<E> : `E get(int indice)`

est très efficace dans un tableau dynamique : O(1).
alors que dans une liste chaînée : O(N)

# Efficacité des algorithmes



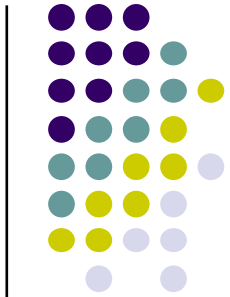
**`ArrayList<E> (tableau dynamique):`**

- **coûteux** d'ajouter et de supprimer un élément
- **Accès par indice peu couteux**

**`LinkedList<E> (liste doublement chaînée)`**

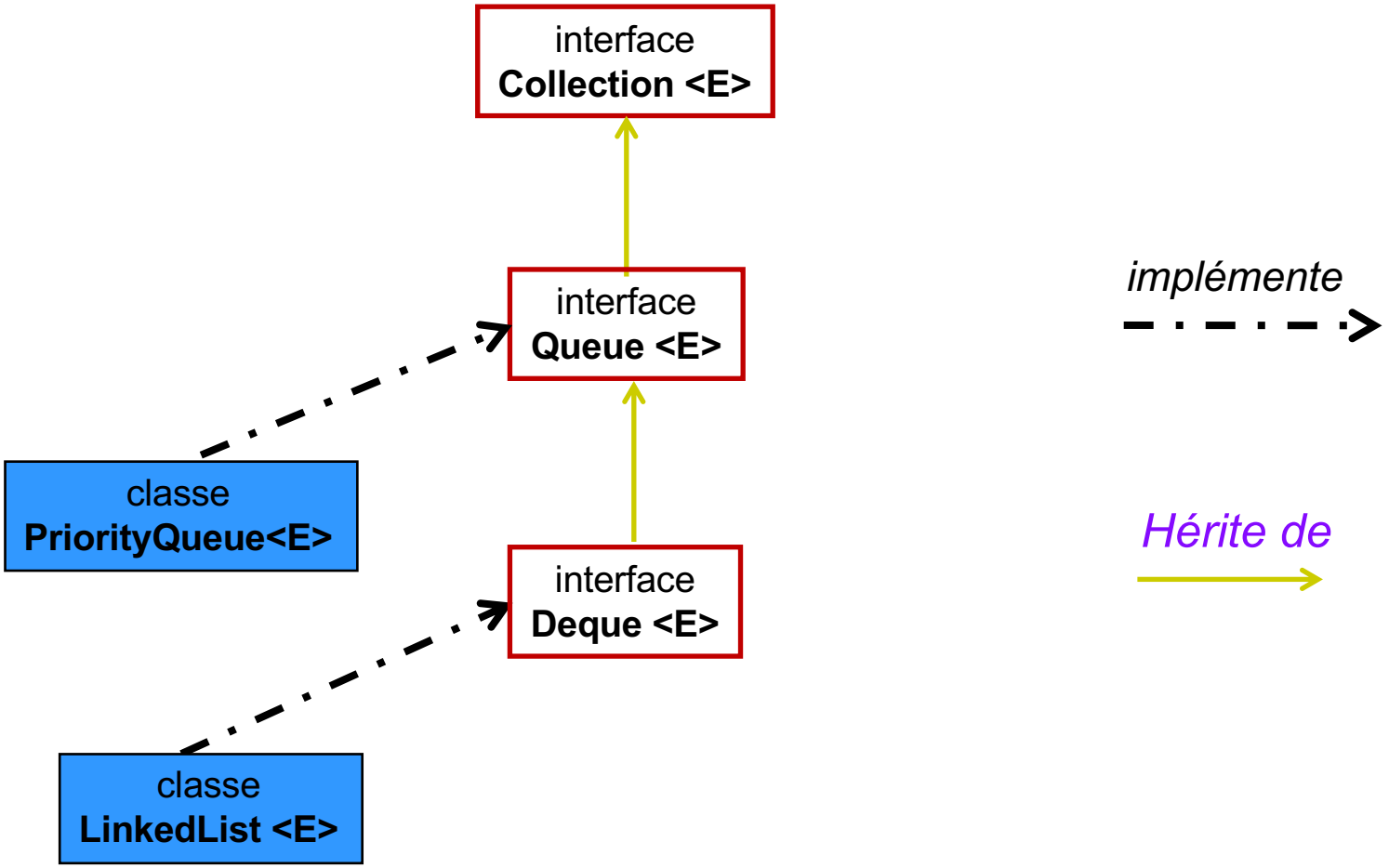
- accès uniquement séquentiel **coûteux** (temps linéaire)
- **insertion/suppression d'un élément via un itérateur peu couteux**

# Files d’attente

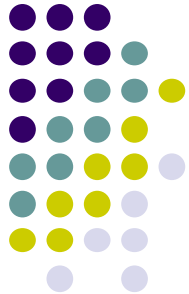


*Queue<E>, Deque<E>, PriorityQueue<E>*

# Les files d'attente en Java
# Schéma simplifié des API



# Interface Queue<E> et Deque<E>

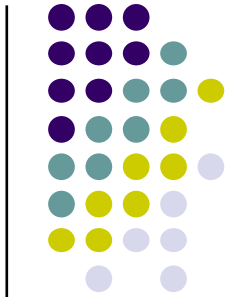


**Queue<E>** et **Deque<E>** sont deux interfaces qui permettent d'implanter des files d'attente.
L'interface **Queue<E>** étend l'interface **Collection<E>**
L'interface **Deque<E>** étend l'interface **Queue<E>**

**Caractéristiques :**
Seules deux positions autorisent les accès, les ajouts et suppressions : le début et la fin de la suite d'éléments.

L'interface **Queue<E>** définit typiquement une file (FIFO) selon cette règle : FIFO (first-in-first-out).
Premier-arrivé, premier-servi.

# Interface Queue<E>



**Queue<E>**

**Opérations de base : Ajouter supprimer consulter**

- Soit avec les méthodes issues de Collection<E> (`add, remove`), qui lancent une exception en cas de problème.

Exemple : `add(E e)` : insère e en fin de file. En cas de poblème, déclencheune exception de type **NoSuchElementException.**

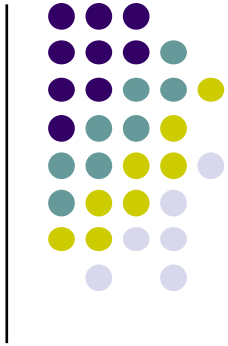
- Soit avec des méthodes spécifiques: `offer, poll, peek`

- **`boolean`** `offer(E e)` //ajouter e (en tête)

- `E poll()` // supprime l'élément de tête
  // Retourne une référence sur l'élément tête ou null

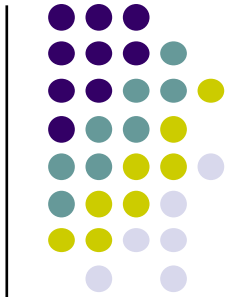
- `E peek()` // retourne une référence sur l'élément tête ou null

# Interface Queue<E>



| | *Throws exception* | *Returns special value* |
|---|---|---|
| **Insertion** | add(e) | offer(e) |
| **Suppression** | remove() | poll() |
| **Sélection** | element() | peek() |

# Implémentations de Queue<E>

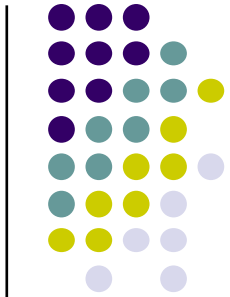


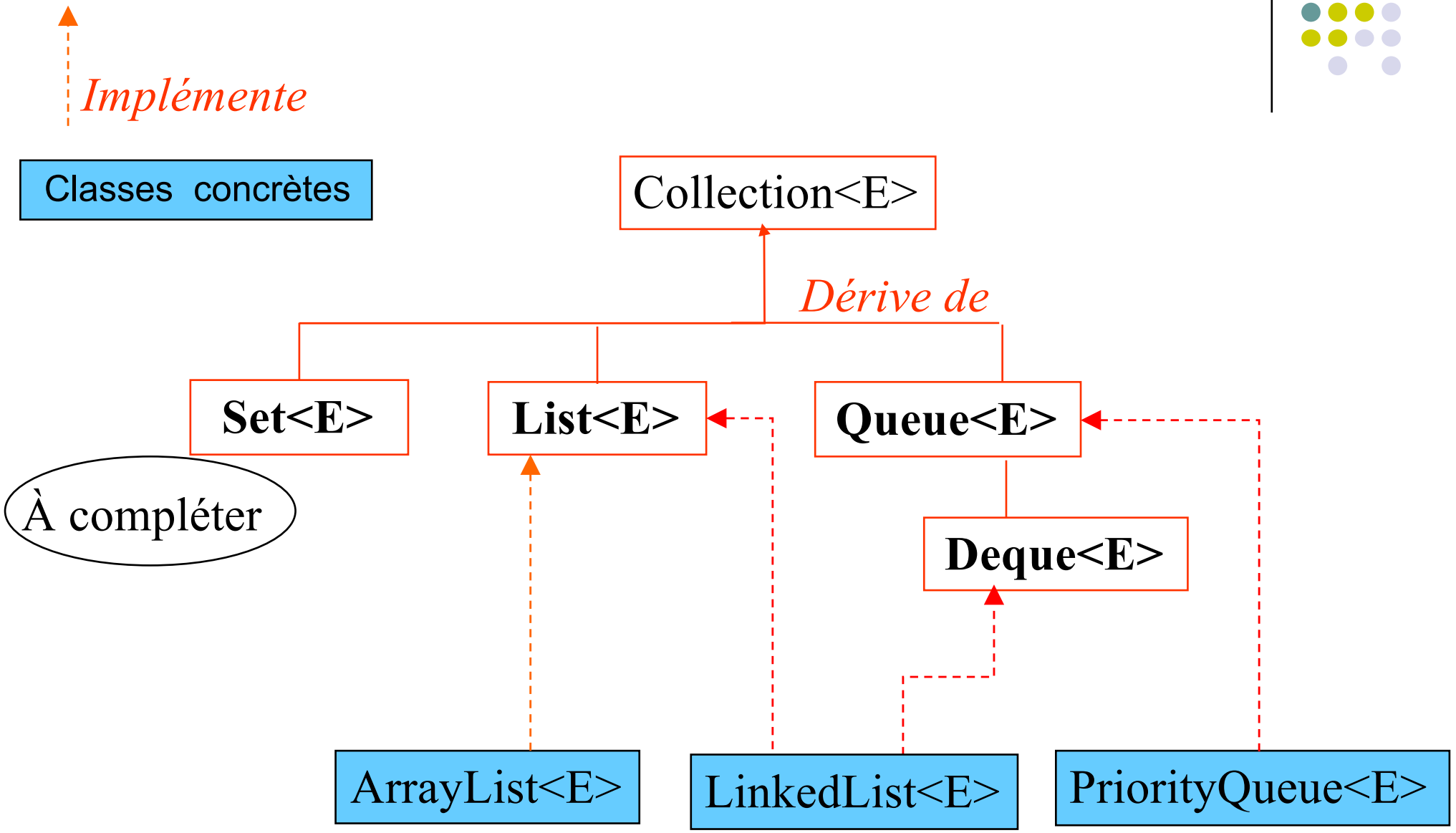
Les classes suivantes implémentent l'interface Queue<E> :

- `LinkedList<E>`

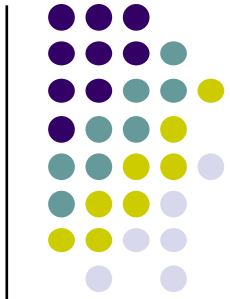
- `PriorityQueue<E>`

# Organisation





## Exemple : queue ( fileAt.java)



```
import java.util.Queue;
import java.util.LinkedList;

public class file{
        public static void main(String[] args) {
        Queue<Integer> fileAt=new LinkedList<Integer>();
        // Integer est une classe enveloppe (ou wrapper)

        fileAt.offer(1);
        fileAt.offer(9);

System.out.println(fileAt.poll()); //supprime elt de tete -> 1

System.out.println(fileAt.peek());//consultation elt tete -> 9

System.out.println(fileAt.poll());// supprime elt de tete -> 9

// la file est vide
System.out.println(fileAt.peek());
//consultation elt de tete -> null
        }
}
```

# Interface Deque<E>
## (*double-ending queue*)

**Deque<E> est une « sous-interface » de Queue<E>**

Cette interface hérite de l'interface Queue<E>.
Elle représente une file d'attente (Queue<E>) dont les éléments peuvent être ajoutés en début et en fin de la suite d'éléments.
Elle définit la notion de **file d'attente à double extrémité**.

**Méthodes spécifiques:**

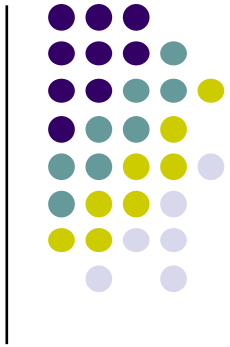
- `boolean offerLast(E e)` // ajoute e en fin
- `boolean offerFirst(E e)` // ajoute e en tête
- `E pollFirst()` // supprime l'élément de tête
- `E pollLast()` // supprime l'élément de fin
- `E peekFirst ()`

// retourne une référence sur l'élément de tête ou null

- `E peekLast()`

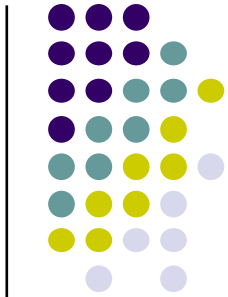
// retourne une référence sur l'élément de fin ou null

# Interface Deque<E>
## (*double-ending queue*)



Une collection de type `Deque` peut être utilisée dans le mode LIFO (Last In First Out) pour agir comme une pile : dans ce cas, les éléments sont insérés et retirés uniquement en tête de la collection.
Il est recommandé d'utiliser une instance de type `Deque` plutôt qu'une collection de type `Stack` pour implémenter une pile.

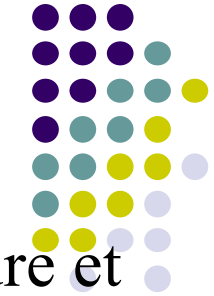
# Classe PriorityQueue<E>



La classe PriorityQueue<E> est une **file à priorité** :
l'ordre de sortie des éléments ne dépend plus de leur date d'insertion, comme une FIFO, mais d'une priorité entre éléments.
En pratique les éléments doivent être munis d'une **relation d'ordre** et l'élément le plus prioritaire (le prochain à sortir) est le plus petit selon cet ordre.

**Méthodes spécifiques :**
La classe PriorityQueue<E> implémente l'interface Queue<E>.

# Classe PriorityQueue<E>



La classe PriorityQueue<E> est une **file à priorité** :
En pratique les éléments doivent être munis d'une relation d'ordre et l'élément le plus prioritaire (le prochain à sortir) est le plus petit selon cet ordre.

**Constructeurs**

•**PriorityQueue** ()
Les éléments sont ordonnés automatiquement selon la relation d'ordre de E : la classe E des éléments implémente l'interface Comparable<E>

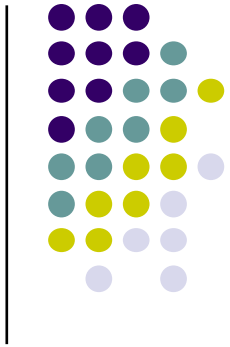
•**PriorityQueue**(**int** initialCapacity)
Les éléments sont ordonnés automatiquement selon la relation d'ordre de E : la classe E des éléments implémente l'interface Comparable<E>

•OU selon un comparateur (Comparator).
**PriorityQueue** (**int** initialCapacity, Comparator<? Super E>comparator)
Le comparateur est passé en argument au constructeur.

Exemple : filePrio.java

```
import java.util.PriorityQueue;

public class filePrio{
        public static void main(String[] args) {
        PriorityQueue<Integer> file=new PriorityQueue<Integer>();
        // Integer est une classe enveloppe (ou wrapper)
        file.offer(9);
        file.offer(1);
        file.offer(7);

        System.out.println(file.poll());
        // retrait elt proritaire -> 1
        System.out.println(file.peek());
        //consulter elt tete -> 7
        System.out.println(file.poll());
        //retrait elt tete (proritaire)  -> 7
        System.out.println(file.poll()); // retrait elt tete  -> 9
        // la file est vide
        System.out.println(file.peek());
         }
}
```

# Test `PriorityQueue`



Exemple : TestPriorityQueue.java