NFP121 — Programmation Avancée

—

Structures de données – Collections

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

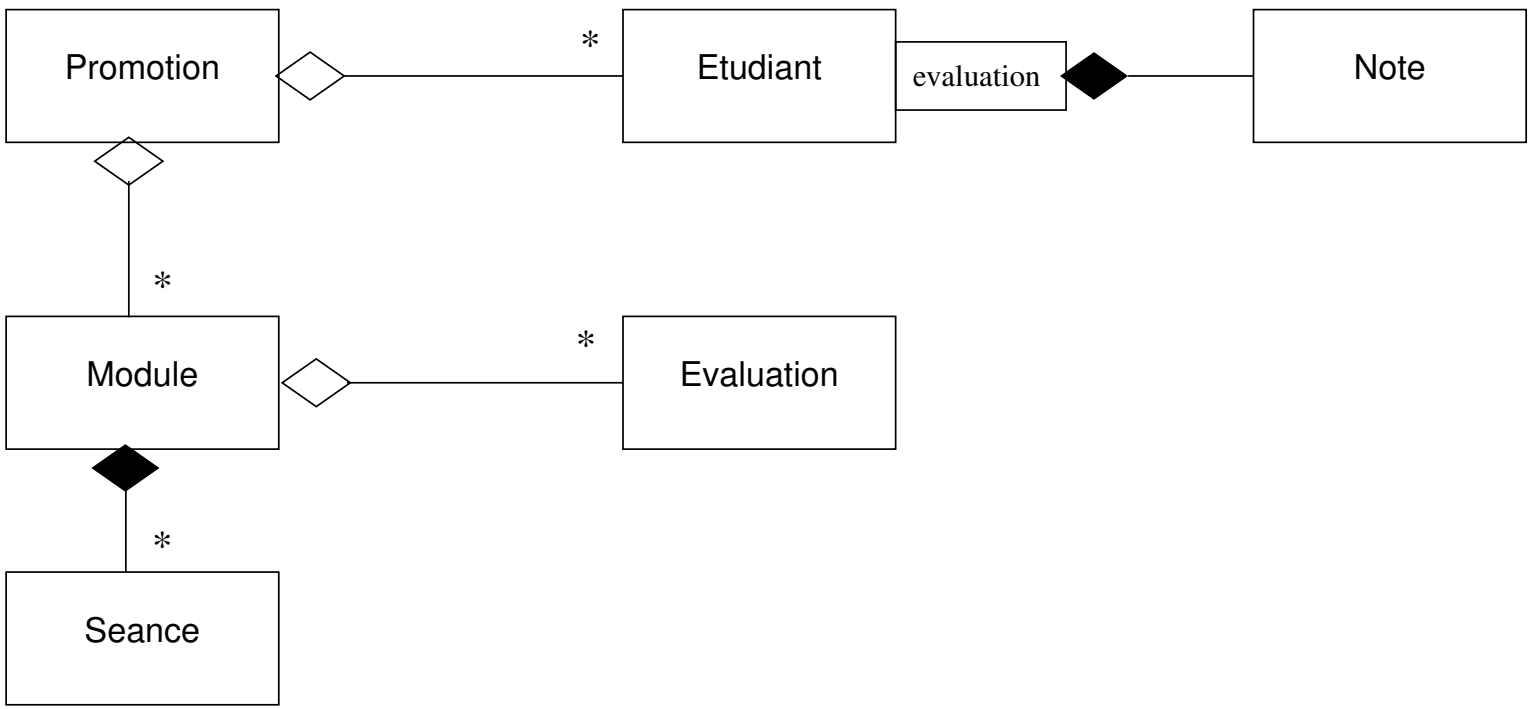
ENSEEIHT
Télécommunications & Réseaux

Sommaire

Sommaire

Motivation

- Comment traduire en Java les différentes relations de ce diagramme de classe ?
- Quelles informations supplémentaire faut-il ajouter sur le diagramme de classe ?



Objectifs

- Principales structures de données
- Techniques d'implantation
- Aspects avancés de la généricité
- Classes internes en Java
- UML et structures de données
- Quelques patrons de conception
- Un exemple pour comprendre les différents aspects

Sommaire

- Spécification d'un ensemble
- Utilisation d'un ensemble
- Quelques implantations
- Manipuler les éléments
- Classes internes
- Compléments sur les itérateurs
- Les itérateurs en Java
- Compléments sur la généricité

Ensemble

Les **opérations classiques** sur un ensemble sont :

- ajouter un élément,
- supprimer un élément,
- savoir si un élément est présent ou non,
- obtenir le cardinal de l'ensemble (nombre d'éléments de l'ensemble),
- ...mais aussi union, intersection...

Les principales **propriétés** sont :

- On ne peut pas avoir de double
- Les éléments ne sont pas repérés par une position

Utilisation possible :

- Compter le nombre de mots différents sur la ligne de commande.

L'interface Ensemble

Spécification syntaxique

```
1  public interface Ensemble<E> {
2      /** Obtenir le nombre d'éléments dans l'ensemble.
3       * @return nombre d'éléments dans l'ensemble.  */
4      int cardinal();
5
6      /** Savoir si un élément est présent dans l'ensemble.
7        * @param x l'élément cherché
8        * @return x est dans l'ensemble */
9      boolean contient(E x);
10
11     /** Ajouter un élément dans l'ensemble.
12      * @param x l'élément à ajouter */
13     void ajouter(E x);
14
15     /** Enlever un élément de l'ensemble.
16       * @param x l'élément à supprimer */
17     void supprimer(E x);
18
19 }
```

Programmation par contrat

Spécification du comportement

```
1   public interface Ensemble<E> {
2       //@ public invariant 0 <= cardinal();
3
4       /** Obtenir le nombre d'éléments dans l'ensemble.
5        * @return nombre d'éléments dans l'ensemble.  */
6       /*@ pure @*/ int cardinal();
7
8       /** Savoir si un élément est présent dans l'ensemble.
9         * @param x l'élément cherché
10        * @return x est dans l'ensemble */
11      /*@ pure @*/ boolean contient(E x);
12
13      /** Ajouter un élément dans l'ensemble.
14       * @param x l'élément à ajouter */
15      //@ ensures     // élément ajouté
16      //@     contient(x);
17      void ajouter(E x);
18
19      /** Enlever un élément de l'ensemble.
20        * @param x l'élément à supprimer */
21      //@ ensures     // élément supprimé
22      //@     ! contient(x);
23      void supprimer(E x);
24
25  }
```

Remarque : Les contrats donnés ici sont partiels.

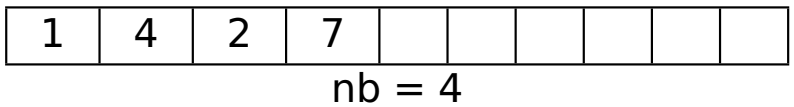
Compter le nombre de mots différents

```
1  public class CompteurMots {
2      public static void main(String[] args) {
3          Ensemble<String> mots = new EnsembleTab<String>(100);
4          for (String mot : args) {
5              mots.ajouter(mot);
6          }
7          System.out.println("Nombre de mots différents : " + mots.cardinal());
8      }
9
10 }
```

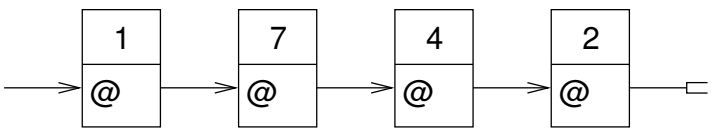
```
1  > java CompteurMots A B S A B BBB D D D D D D D A S BB
2  Nombre de mots différents : 6
```

Implantations possibles

- avec un tableau (trié ou non)
  - les éléments sont tassés en début du tableau
  - gestion d'une taille effective



- avec une liste chaînée
  - éléments chaînés entre eux

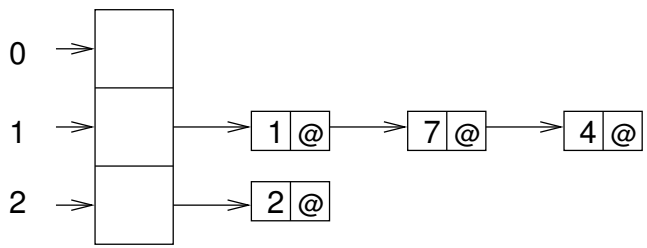


- avec un vecteur caractéristique
  - l'élément sert d'indice
  - une case du tableau indique si l'élément en indice est présent ou non

-	X	X	-	X	-	-	X	-	-
0	1	2	3	4	5	6	7	8	9

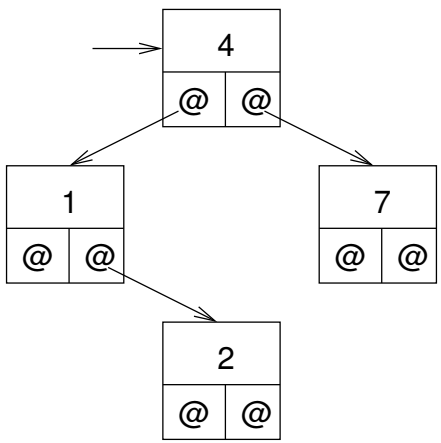
Implantations possibles (2)

- avec une table de hachage
  - généralisation des tableaux
  - fonction de hachage : Element $\rightarrow$ Entier (ici : mod 3)
  - gestion des conflits : par exemple, une liste



Implantations possibles (3)

- avec un ABR (Arbre Binaire de Recherche)
  - nécessite un ordre total sur les éléments
  - les éléments d'un sous-arbre gauche sont inférieurs à l'élément d'un nœud
  - les éléments d'un sous-arbre droit sont supérieurs à l'élément d'un nœud
  - recherche, ajout et suppression en log(n), n taille de la liste



Implantation avec un tableau

```
1  public class EnsembleTab<E> implements Ensemble<E> {
2      private E[] elements;   // stockage des éléments de l'ensemble
3      private int nb; // taille effective de elements
4
5      /** Construction d'un ensemble avec une capacité initiale.
6        * @param capaciteInitiale capacité initiale de l'ensemble */
7      public EnsembleTab(int capaciteInitiale) {
8          this.elements = (E []) new Object[capaciteInitiale];    // XXX
9          this.nb = 0;
10     }
11
12     public int cardinal() {
13         return nb;
14     }
15
16     /** Position de x dans le tableau elements. */
17     private int positionDe(E x) {
18         int i = 0;
19         while (i < nb && ! elements[i].equals(x)) {
20             i++;
21         }
22         return i;
23     }
24
25     public boolean contient(E x) {
26         return positionDe(x) < nb;
27     }
28
```

Implantation avec un tableau (2)

```
29      public void supprimer(E x) {
30          int p = positionDe(x);  // position de x dans elements
31          if (p < nb) {   // L'élément est présent à la position p
32              // remplacer l'élément d'indice p par le dernier élément
33              nb--;
34              elements[p] = elements[nb];
35              elements[nb] = null;        // utile ?
36          }
37      }
38
39      public void ajouter(E x) {
40          if (!contient(x)) {
41              garantirNonPlein();
42              this.elements[nb++] = x;
43          }
44      }
45
46      /** Agrandir le tableau s'il est plein.  */
47      private void garantirNonPlein() {
48          if (this.nb >= this.elements.length) {
49              // agrandir le tableau
50              this.elements = java.util.Arrays.copyOf(this.elements,
51                      this.elements.length + 3);  // 3 arbitraire !
52          }
53      }
54
55  }
```

Implantation avec des structures chaînées

```
1  /** Une cellule encapsule un élément et un accès
2   * à une autre cellule dite suivante.
3   */
4  class Cellule<E> {
5      E element;
6      Cellule<E> suivante;
7
8      Cellule(E element, Cellule<E> suivante) {
9          this.element = element;
10         this.suivante = suivante;
11     }
12
13     public String toString() {
14         // Attention, il ne faut pas que les cellules forment un cycle !
15         return "[" + this.element + "]--"
16                 + (this.suivante == null ? 'E' : ">" + this.suivante);
17     }
18
19 }
```

- classe locale au paquetage (non publique)
- droit d'accès paquetage $\Longrightarrow$ accessible que depuis le paquetage
- ni accesseurs, ni modifieurs $\Longrightarrow$ code un peu plus simple sur les transparents suivants

Implantation avec des structures chaînées (2)

```
1  public class EnsembleChaine<E> implements Ensemble<E> {
2     private Cellule<E> premiere;  // accès à la première cellule
3     private int nb;   // nombre d'éléments (évite de le calculer)
4
5     /** Construction d'un ensemble vide. */
6     public EnsembleChaine() {
7        this.premiere = null;
8        this.nb = 0;
9     }
10
11    public int cardinal() {
12       return nb;
13    }
14
15    public boolean contient(E x) {
16       Cellule<E> curseur = this.premiere;
17       while (curseur != null && ! curseur.element.equals(x)) {
18          curseur = curseur.suivante;
19       }
20       return curseur != null;
21    }
22
23    public void ajouter(E x) {
24       if (!contient(x)) {
25          this.premiere = new Cellule<E>(x, this.premiere);
26          this.nb++;
27       }
```

Implantation avec des structures chaînées (3)

```
28      }
29
30      public void supprimer(E x) {
31         if (this.premiere != null) {
32            if (this.premiere.element.equals(x)) {
33               this.premiere = this.premiere.suivante;
34               this.nb--;
35            } else {
36               // Chercher la cellule avant l'élément.
37               Cellule<E> curseur = this.premiere;
38               while (curseur.suivante != null && ! curseur.suivante.element.equals(x)) {
39                  curseur = curseur.suivante;
40               }
41
42               if (curseur.suivante != null) {
43                  curseur.suivante = curseur.suivante.suivante;
44                  this.nb--;
45               }
46            }
47         }
48      }
49
50   }
```

Autres utilisations

On peut vouloir :

- Afficher tous les éléments d'un ensemble ?
- Calculer la somme des éléments d'un ensemble de réels ?
- Vérifier qu'il n'y a pas de multiples dans un ensemble d'entiers ?
- Supprimer tous les éléments pairs d'un ensemble ?
- Déterminer l'intersection ou l'union de deux ensembles ?
- ...

et ceci doit fonctionner :

- quelque soit l'implantation de l'ensemble,
- et même quelque soit la structure de données

Comment faire ?

Afficher pour EnsembleTab et EnsembleChaine

```
1  public class Afficheur {
2
3      static public <E> void afficher(EnsembleChaine<E> ens) {
4          Cellule<E> curseur = ens.premiere;
5          while (curseur != null) {
6              System.out.println(curseur.element);
7              curseur = curseur.suivante;
8          }
9      }
10
11     static public <E> void afficher(EnsembleTab<E> ens) {
12         int i = 0;
13         while (i < ens.nb) {
14             System.out.println(ens.elements[i]);
15             i = i + 1;
16         }
17     }
18 }
```

- On considère ici que l'on a accès à l'état interne des classes
- Ce code ne peut donc pas compiler si on ne change pas les droits d'accès !
- Remarque : On aurait pu utiliser les types joker (<?>), voir T. 47

Utilisation des méthodes afficher

```
1  public class AfficheurMain {
2
3      /** Ajouter tous les éléments dans l'ensemble ens. */
4      static public <E> void addAll(Ensemble<E> ens, E...elements) {
5          for (E x : elements) {
6              ens.ajouter(x);
7          }
8      }
9
10     public static void main(String[] args) {
11         EnsembleTab<Integer> et = new EnsembleTab<Integer>(10);
12         addAll(et, 1, 4, 2, 7);
13         System.out.println("et␣contient␣:␣"); Afficheur.afficher(et);
14
15         EnsembleChaine<Integer> ec = new EnsembleChaine<Integer>();
16         addAll(ec, 1, 4, 2, 7);
17         System.out.println("ec␣contient␣:␣"); Afficheur.afficher(ec);
18     }
19
20 }
```

Résultat de AfficheurMain

```
et contient :
1
4
2
7
ec contient :
7
2
4
1
```

- On note que les éléments ne sont pas affichés dans le même ordre :
  - La position n'a pas d'importance dans un ensemble !
  - Une autre implantation aurait certainement donné un ordre différent.
- Peut-on afficher un ensemble de type Ensemble<E> ?
- Que se passe-t-il si on considère une nouvelle implantation de l'ensemble ?
  - Faut-il vraiment écrire une nouvelle méthode afficher ?

Comment unifier les deux solutions

- Les deux algorithmes ont la même structure.
- On obtient alors le même raffinage :

```
1  static public <E> void afficher(Ensemble<E> ens) {
2      // commencer
3      while (/* encore des éléments à parcourir */) {
4          System.out.println(/* l'élément courant */);
5          // passer à l'élément suivant
6      }
7  }
```

- **Comment le traduire en Java ?**

Définir de nouvelles opérations

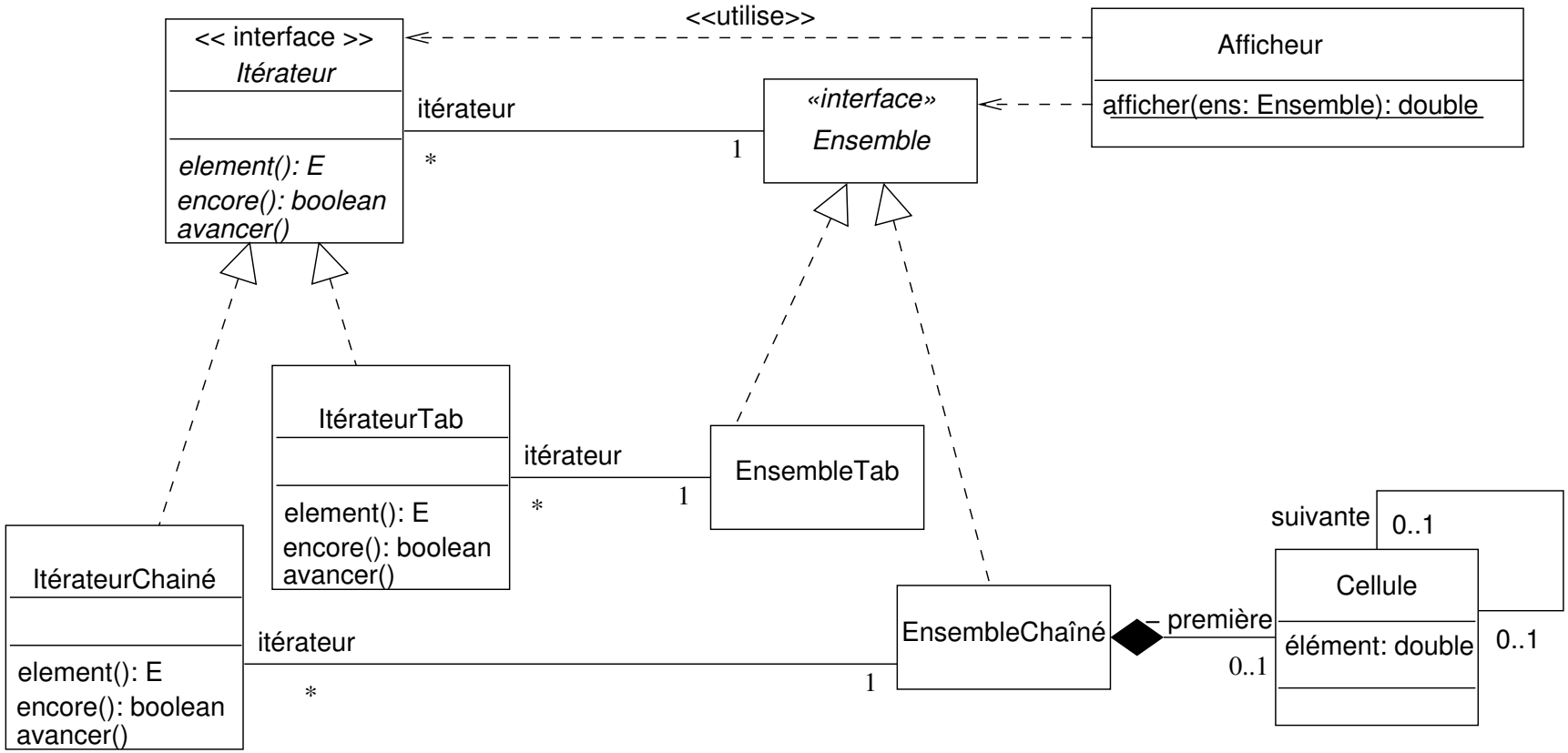
On identifie 4 nouvelles opérations :

- initialiser un parcours : `initialiser()`
- savoir s'il y a encore des éléments : `encore()`
- obtenir l'élément courant : `element()`
- passer à l'élément suivant : `avancer()`

Où définir ces opérations ?

1. dans l'interface Ensemble et ses réalisations :
   - C'est possible
   - Mais on ne pourra faire qu'un seul parcours à la fois
   - Comment déterminer s'il y a des multiples dans l'ensemble ?
2. dans une autre classe qui gèrera le parcours :
   - Une interface `Iterateur` pour spécifier les opérations (sauf initialiser)
   - Ensemble spécifie une nouvelle méthode qui retourne un itérateur : `iterateur()`
   - Les réalisations de Ensemble définissent cette méthode et retourne l'itérateur adapté (une réalisation de `Iterateur`)
   - On peut alors faire plusieurs parcours simultanés.

Architecture de la solution



- Il s'agit du patron de conception Iterateur :
  - but : parcourir successivement tous les éléments d'un « conteneur » (concret ou abstrait)
  - synonyme : Curseur.
  - exemples de conteneur : ensemble, nombres premiers, etc.

L'interface Iterateur

```java
public interface Iterateur<E> {

    /** Existe-t-il des éléments non encore consultés ? */
    boolean encore();

    /** Obtenir l'élément courant.
     * @throws NoSuchElementException si tous les éléments ont été parcourus
     */
    E element();

    /** Avancer le curseur.
     * @throws NoSuchElementException si tous les éléments ont été parcourus
     */
    void avancer();

}
```

```java
public interface Ensemble<E> {
    ...

    /** Un itérateur sur l'ensemble. */
    Iterateur<E> iterateur();

}
```

Utilisation de l'itérateur

```
1  public class Afficheur {
2
3      static public <E> void afficher(Ensemble<E> ens) {
4          Iterateur<E> it = ens.iterateur();
5          while (it.encore()) {
6              System.out.println(it.element());
7              it.avancer();
8          }
9      }
10 }
```

qui correspond bien au raffinage :

```
1  static public <E> void afficher(Ensemble<E> ens) {
2      // commencer
3      while (/* encore des éléments à parcourir */) {
4          System.out.println(/* l'élément courant */);
5          // passer à l'élément suivant
6      }
7  }
```

Itérateur pour un ensemble chaîné

```java
/** Définition d'un itérateur sur une structure chaînée linéaire. */
class IterateurChaine<E> implements Iterateur<E> {
  private Cellule<E> curseur;

  public IterateurChaine(Cellule<E> debut) {
    this.curseur = debut;
  }

  @Override public boolean encore() {
    return curseur != null;
  }

  @Override public E element() {
    checkEncore();
    return curseur.element;
  }

  @Override public void avancer() {
    checkEncore();
    curseur = curseur.suivante;
  }

  private void checkEncore() {
    if (! this.encore()) {
      throw new java.util.NoSuchElementException("Parcours terminé");
    }
  }
}
```

```java
public class EnsembleChaine<E> implements Ensemble<E> {
  public Iterateur<E> iterateur() {
    return new IterateurChaine(this.premiere);
  }
}
```

- checkEncore() factorise la vérification de la fin de parcours.
- Nécessaire pour contrôler que les opérations sont appelées dans un ordre cohérent.

Vérifier s'il y a des multiples

- On peut utiliser simultanément plusieurs itérateurs !

```
1   public class Multiples {
2       /** Est-ce que e contient deux éléments non nuls x1 et x2 tels que x1 est
3        * multiple de x2 ? */
4       public static boolean contientMultiples(Ensemble<Integer> e) {
5           for (Iterateur<Integer> i1 = e.iterateur(); i1.encore(); i1.avancer()) {
6               int x1 = i1.element();
7               for (Iterateur<Integer> i2 = e.iterateur(); i2.encore(); i2.avancer()) {
8                   int x2 = i2.element();
9                   if (x1 != x2 && x1 != 0 && x2 != 0 && x1 % x2 == 0) {
10                      return true;
11                  }
12              }
13          }
14          return false;
15      }
16
17      public static void main(String[] args) {
18          Ensemble<Integer> ens = new EnsembleChaine<Integer>();
19          AfficheurMain.addAll(ens, 2, 3, 5, 7, 11);
20          assert ! contientMultiples(ens);
21          ens.ajouter(49);
22          assert contientMultiples(ens);
23      }
24  }
```

Fail-fast iterator

Problème : Que se passe-t-il si on modifie un ensemble pendant qu'un itérateur le parcourt (ajout ou suppression d'un élément) ?

- Ceci peut conduire à des incohérences et donc à des erreurs !
- Principe : le considérer comme une erreur de programmation
- Idée : le détecter au plus tôt (d'où *fail-fast*) et le signaler (ConcurrentModificationException)

Solution : compteur de modifications sur les ensembles

- chaque opération de modification d'un ensemble incrémente ce compteur
- lors de sa création, l'itérateur mémorise la valeur du compteur
- chaque opération de l'itérateur vérifie que le compteur n'a pas changé
- si changement, lever l'exception `ConcurrentModificationException`

Question : Comment accéder à ce compteur ?

- L'itérateur doit avoir accès à l'ensemble et à son attribut privé compteur
- Le mettre **`public`** ? Mais l'utilisateur lambda n'a pas à le connaître
- Le transmettre à l'itérateur ? type primitif, donc seulement copie de la valeur !

$\Longrightarrow$ Utiliser une classe interne (static, membre ou anonyme)

Itérateur pour ensemble chaîné

avec classe interne statique

```
1  public class EnsembleChaine<E> implements Ensemble<E> {
2      private long nbModifications;   // nb de modifications faites sur la liste
3
4      public Iterateur<E> iterateur() {
5          return new IterateurChaine<E>(this);
6      }
7
8      static private class IterateurChaine<E> implements Iterateur<E> {
9          private long nbModifsInitial;
10         private Cellule<E> curseur;
11         private EnsembleChaine<E> ens;
12
13         public IterateurChaine(EnsembleChaine<E> ens) {
14             this.ens = ens;
15             this.curseur = this.ens.premiere;
16             this.nbModifsInitial = this.ens.nbModifications;
17         }
18
19         @Override public boolean encore() {
20             this.checkModifications();
21             return this.curseur != null;
22         }
23
24         @Override public E element() {
25             this.checkModifications();
26             this.checkEncore();
```

Itérateur pour ensemble chaîné (2)

avec classe interne statique

```
27              return this.curseur.element;
28          }
29
30          @Override public void avancer() {
31              this.checkModifications();
32              this.checkEncore();
33              this.curseur = this.curseur.suivante;
34          }
35
36          private void checkEncore() {
37              if (! this.encore()) {
38                  throw new java.util.NoSuchElementException("Parcours terminé");
39              }
40          }
41
42          private void checkModifications() {
43              if (this.ens.nbModifications != this.nbModifsInitial) {
44                  throw new java.util.ConcurrentModificationException();
45              }
46          }
47      }
48
49  }
```

- Une classe interne a accès à ce qui est privé de la classe englobante
- Une classe interne a un droit d'accès
- Pourquoi transmettre explicitement l'objet EnsembleChaine à l'itérateur ?

Itérateur pour ensemble chaîné

avec classe interne membre

```
1  public class EnsembleChaine<E> implements Ensemble<E> {
2      private long nbModifications;   // nb de modifications faites sur la liste
3
4      public Iterateur<E> iterateur() {
5          return new IterateurChaine();
6      }
7
8      private class IterateurChaine implements Iterateur<E> {
9          private long nbModifsInitial;
10         private Cellule<E> curseur;
11
12         public IterateurChaine() {
13             this.curseur = premiere;
14             this.nbModifsInitial = nbModifications;
15         }
16
17         @Override public boolean encore() {
18             this.checkModifications();
19             return this.curseur != null;
20         }
21
22         @Override public E element() {
23             this.checkModifications();
24             this.checkEncore();
25             return this.curseur.element;
26         }
```

Itérateur pour ensemble chaîné (2)

avec classe interne membre

```
27
28          @Override public void avancer() {
29              this.checkModifications();
30              this.checkEncore();
31              this.curseur = this.curseur.suivante;
32          }
33
34          private void checkEncore() {
35              if (! this.encore()) {
36                  throw new java.util.NoSuchElementException("Parcours terminé");
37              }
38          }
39
40          private void checkModifications() {
41              if (nbModifications != this.nbModifsInitial) {
42                  throw new java.util.ConcurrentModificationException();
43              }
44          }
45      }
46
47  }
```

- Un objet de la classe interne a accès à l'objet de la classe englobante qui a permis sa création.
- Pourquoi nommer la classe ?

Itérateur pour ensemble chaîné

avec classe anonyme

```
1  public class EnsembleChaine<E> implements Ensemble<E> {
2      private long nbModifications;   // nb de modifications faites sur la liste
3
4      public Iterateur<E> iterateur() {
5          return new Iterateur<E>() { // classe anonyme
6              private long nbModifsInitial = nbModifications;
7              private Cellule<E> curseur = premiere;
8
9              @Override public boolean encore() {
10                 this.checkModifications();
11                 return this.curseur != null;
12             }
13
14             @Override public E element() {
15                 this.checkModifications();
16                 this.checkEncore();
17                 return this.curseur.element;
18             }
19
20             @Override public void avancer() {
21                 this.checkModifications();
22                 this.checkEncore();
23                 this.curseur = this.curseur.suivante;
24             }
25
26             private void checkEncore() {
```

Itérateur pour ensemble chaîné (2)

avec classe anonyme

```
27                  if (! this.encore()) {
28                      throw new java.util.NoSuchElementException("Parcours terminé");
29                  }
30              }
31
32              private void checkModifications() {
33                  if (nbModifications != this.nbModifsInitial) {
34                      throw new java.util.ConcurrentModificationException();
35                  }
36              }
37          };
38      }
39
40  }
```

- Une classe anonyme n'a pas de nom (impossible de la réutiliser).
- Elle a accès aux variables et paramètres de la méthode s'ils sont déclarés **final**

Remarque : Tout ceci marche car l'utilisateur n'a pas à connaître l'itérateur concret.

Classes internes : les principes sur un exemple

```
1   class E {   // classe engobante
2       private int x = 4;
3       private int y = 12;
4
5       static class S {    // classe interne statique
6           void m() {
7               // System.out.println(x);   // Erreur !
8               System.out.println(new E().x);  // OK
9           }
10      }
11
12      class M {   // classe interne membre
13          private int y = 7;
14          void m() {
15              System.out.println(x);  // 4
16              System.out.println(y);  // 7
17              System.out.println(E.this.y);   // 12
18          }
19      }
20  }
21
22  public class ExempleClasseInternes {
23      public static void main(String[] args) {
24          E.S s = new E.S();  // OK
25          // E.M m1 = new E.M();  // Erreur !
26          E e = new E();
27          E.M m2 = e.new M(); // OK
28          m2.m();
29      }
30  }
```

Comment faire pour modifier l'ensemble pendant un parcours ?

Exercice : Comment faire pour supprimer les éléments pairs d'un ensemble d'entier ?

Solution naïve :

```
1  public class SupprimerPairsErreur {
2      public static void main(String[] args) {
3          Ensemble<Integer> ens = new EnsembleChaine<Integer>();
4          AfficheurMain.addAll(ens, 1, 4, 2, 7);
5          Iterateur<Integer> it = ens.iterateur();
6          while (it.encore()) {
7              int x = it.element();
8              if (x % 2 == 0) { // x est pair
9                  ens.supprimer(x);
10             }
11             it.avancer();
12 }   }   }
```

Constat : on ne peut pas utiliser l'opération `supprimer` de l'ensemble

```
1  Exception in thread "main" java.util.ConcurrentModificationException
2      at EnsembleChaine$IterateurChaine.checkModifications(EnsembleChaine.java:100)
3      at EnsembleChaine$IterateurChaine.avancer(EnsembleChaine.java:87)
4      at SupprimerPairsErreur.main(SupprimerPairsErreur.java:11)
```

Comment faire pour modifier l'ensemble pendant un parcours ? (2)

- La modification doit passer par l'itérateur...
  - ...pour qu'il puisse mettre à jour son compteur de modification).
- Conséquence : on ajoute une opération `supprimer` sur l'itérateur :
  - elle supprime de l'ensemble l'élément retourné par `element`

```
1   public class SupprimerPairs {
2       public static void main(String[] args) {
3           Ensemble<Integer> ens = new EnsembleChaine<Integer>();
4           AfficheurMain.addAll(ens, 1, 4, 2, 7);
5           Iterateur<Integer> it = ens.iterateur();
6           while (it.encore()) {
7               int x = it.element();
8               if (x % 2 == 0) { // x est pair
9                   it.supprimer();
10              }
11              it.avancer();
12  }   }   }
```

Parcourir différents types de structures de données

Problème : L'itérateur a été défini dans le contexte des ensembles. Comment faire si on veut parcourir d'autres types de structures de données : une liste, un arbre, les entiers pairs, les nombres premiers, les décimales de π...

Solution :

- Le mécanisme des itérateurs est général (c'est un patron de conception) !
- Il faut juste obtenir un itérateur.
- On définit donc une interface `Iterable` qui donne accès à un itérateur.

```
1  /** Un élément est Iterable, s'il fournit un itérateur. */
2  public interface Iterable<E> {
3
4      /** Obtenir un itérateur. */
5      Iterateur<E> iterateur();
6
7  }
```

- L'interface `Ensemble` hérite alors de `Iterable`.
- Plus généralement, les structures de données réalisent une telle interface.
- On dispose donc d'un moyen de récupérer un itérateur.

Les itérateurs en Java

```
1  public interface Iterator<E> {  // extrait des API Java (avec coupes)
2      /** Returns <tt>true</tt> if the iteration has more elements.
3       * @return <tt>true</tt> if the iterator has more elements.
4       */
5      boolean hasNext();
6
7      /** Returns the next element in the iteration.
8       * @return the next element in the iteration.
9       * @exception NoSuchElementException iteration has no more elements.
10      */
11     E next();
12
13     /** Removes from the underlying collection the last element returned by the
14      * iterator (optional operation).  This method can be called only once per
15      * call to <tt>next</tt>.  The behavior of an iterator is unspecified if
16      * the underlying collection is modified while the iteration is in
17      * progress in any way other than by calling this method.
18      *
19      * @exception UnsupportedOperationException if the <tt>remove</tt>
20      *        operation is not supported by this Iterator.
21
22      * @exception IllegalStateException if the <tt>next</tt> method has not
23      *        yet been called, or the <tt>remove</tt> method has already
24      *        been called after the last call to the <tt>next</tt>
25      *        method.
26      */
27     void remove();
28 }
```

Iterable et foreach

- L'interface `java.lang.Iterable<T>` spécifie l'accès à un itérateur.

```
1  public interface Iterable<T> {  // Extrait des API Java
2      /** Returns an iterator over a set of elements of type T.
3       * @return an Iterator.  */
4      Iterator<T> iterator();
5  }
```

- En Java, on peut utiliser un `foreach` sur tout « Iterable ».

Avec un `foreach` :

```
1  Iterable<E> collection = ...
2  for (E o : collection) {
3      faire(o);
4  }
```

Il est équivalent à (facilité syntaxique) :

```
1  Iterable<E> collection = ...
2  Iterator<E> it = collection.iterator();
3  while (it.hasNext()) {
4      E o = it.next();
5      faire(o);
6  }
```

- **Remarque :** On ne peut pas utiliser `remove()` avec un **foreach**.
- En Java, `next()` correspond à la fois à `element()` et `avancer()`.

Exemple : supprimer les entiers pairs

```
1  import java.util.*;
2  public class SupprimerPairs {
3      public static void main(String[] args) {
4          Set<Integer> ens = new HashSet<Integer>();
5          Collections.addAll(ens, 2, 3, 6, 9, 1, 4, 7);
6          System.out.println("ens = " + ens);
7
8          // supprimer les entiers pairs
9          Iterator<Integer> it = ens.iterator();
10         while (it.hasNext()) {
11             Integer entier = it.next();
12             if (entier % 2 == 0) {
13                 it.remove();
14             }
15         }
16
17         System.out.println("ens = " + ens);
18     }
19 }
```

```
1  ens = [1, 2, 3, 4, 6, 7, 9]
2  ens = [1, 3, 7, 9]
```

Généricité et sous-typage

Exercice 1 Y a-t-il une relation de sous-typage entre `EnsembleTab<Object>` et `EnsembleTab<String>` ? Dans quel sens ?

Généricité et sous-typage

```
1  public class TestErreurGeneriticiteSousTypage {
2      public static void main(String[] args) {
3          EnsembleTab<String> ls = new EnsembleTab<String>(10);
4          EnsembleTab<Object> lo = ls;
5          lo.ajouter("texte");
6          lo.ajouter(15.5);   // en fait new Double(15.5);
7          String s = ls.iterateur().element();
8          System.out.println(s);
9      }
10 }
```

```
1  TestErreurGeneriticiteSousTypage.java:4: error: incompatible types: EnsembleTab<String> ca
2          EnsembleTab<Object> lo = ls;
3                                   ^
4  1 error
```

Remarque : Si EnsembleTab était immuable, envisager une relation de sous-typage serait possible.

Exercice 2 On considère la méthode `afficher` et le programme de test ci-dessous.

```
1       /** Afficher tous les éléments de ens. */
2       static public  void afficher(Ensemble<Object> ens) {
3           Iterateur<Object> it = ens.iterateur();
4           while (it.encore()) {
5               System.out.println(it.element());
6               it.avancer();
7           }
8       }
```

```
1       public static void main(String[] args) {
2           Ensemble<Object> lo = new EnsembleTab<Object>(5);
3           lo.ajouter("deux");
4           lo.ajouter("un");
5           afficher(lo);
6
7           Ensemble<String> ls = new EnsembleTab<String>(5);
8           ls.ajouter("deux");
9           ls.ajouter("un");
10          afficher(ls);
11      }
```

2.1 Que donnent la compilation et l'exécution de ce programme ?

2.2 Proposer une nouvelle version de la méthode `afficher`.

Utilisation du type *joker*

- Erreur de compilation :
  - `Ensemble<String>` n'est pas un sous-type de `Ensemble<Object>` ! Voir T. 44.
- Une solution consisterait à utiliser une méthode générique :

```
1       static public <T> void afficher(Ensemble<T> ens) {
2           Iterateur<T> it = ens.iterateur();
3           while (it.encore()) {
4               System.out.println(it.element());
5               it.avancer();
6           }
7       }
```

- Une meilleure solution consiste à utiliser un type *joker* (*wildcard*) <?> :

```
1       static void afficher(Ensemble<?> ens) {
2           Iterateur<?> it = ens.iterateur();
3           while (it.encore()) {
4               System.out.println(it.element());
5               it.avancer();
6           }
7       }
```

  - `Ensemble<?>` est appelée « ensemble d'inconnus ». Le type n'est pas connu !
  - Cette solution est meilleure car T n'était jamais utilisé (ne servait à rien).
  - **Rq :** On peut utiliser des types joker contraints (<? **extends** Type>).

Limite des types joker

```
1       /** Copier les éléments de source dans la liste destination. */
2       public static void copier(Ensemble<?> destination, Object[] source) {
3           for (Object o : source) {
4               destination.ajouter(o);
5           }
6       }
```

- Le compilateur interdit d'ajouter un objet dans une liste d'inconnus !
- La solution consiste à prendre un paramètre de généricité :

```
1       public static <T> void copier(Ensemble<T> destination, T[] source) {
2           for (T o : source) {
3               destination.ajouter(o);
4           }
5       }
```

- Que donnent alors les instructions suivantes ?

```
1           String[] tab = { "un", "deux" };
2           copier(new EnsembleTab<Object>(2), tab);
```

Exercice 3 Écrire une méthode de classe qui ajoute dans un ensemble les éléments d'un autre.

Tableau et sous-typage

Exercice 4 Que donnent la compilation et l'exécution de ce programme ?
Comment l'adapter pour fonctionner sur des listes ?

```
1  public class TableauxSousTypage {
2      public static void copier(Object[] destination, Object[] source) {
3          assert destination.length >= source.length
4                  : "Capacité␣insuffisante␣:␣" + destination.length
5                  + "␣<␣" + source.length;
6          for (int i = 0; i < source.length; i++) {
7              destination[i] = source[i];
8          }
9      }
10
11     public static void main(String[] args) {
12         String[] tab1 = { "un", "deux", "trois" };
13         String[] tab2 = new String[tab1.length];
14         copier(tab2, tab1);
15
16         Object[] tab3 = { "un", "deux", new Integer(3) };
17         copier(tab2, tab3);
18     }
19 }
```

Tableau et sous-typage : héritage du passé !

Compilation : OK

- C'est surprenant et peu logique d'après T. 44.

Résultats de l'exécution :

```
1  Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer
2      at TableauxSousTypage.copier(TableauxSousTypage.java:7)
3      at TableauxSousTypage.main(TableauxSousTypage.java:17)
```

Conséquences :

- Si B est un sous-type de A, alors B[] est un sous-type de A[].
  - *Justification :* compatibilité ascendante !
- Typage incomplet $\Longrightarrow$ erreur détectée à l'exécution (`ArrayStoreException`)

Et avec les ensembles ?

Première solution ?

```
1       public static <T> void copier(Ensemble<T> destination, Ensemble<T> source) {
2           Iterateur<T> it = source.iterateur();
3           while (it.encore()) {
4               destination.ajouter(it.element());
5               it.avancer();
6           }
7       }
```

- **Problème :** Peut-on ajouter un ensemble de PointNommé à un ensemble de Point ?

Deuxième solution ?

```
1       public static <T1, T2>
2       void copier(Ensemble<T1> destination, Ensemble<T2> source) {
3           Iterateur<T2> it = source.iterateur();
4           while (it.encore()) {
5               destination.ajouter(it.element());
6               it.avancer();
7           }
8       }
```

Et non !

Et avec les listes ? (suite)

Solution : Dire qu'il y a une relation de sous-typage entre les deux types

```
1       public static <T1, T2 extends T1>
2       void copier(Ensemble<T1> destination, Ensemble<T2> source) {
3           Iterateur<T2> it = source.iterateur();
4           while (it.encore()) {
5               destination.ajouter(it.element());
6               it.avancer();
7           }
8       }
```

Pourquoi deux types ?

```
1       public static <T>
2       void copier(Ensemble<? super T> destination, Ensemble<? extends T> source) {
3           Iterateur<T> it = source.iterateur();
4           while (it.encore()) {
5               destination.ajouter(it.element());
6               it.avancer();
7           }
8       }
```

- **Plus général :** Le premier joker est tout super type de T.
- Ici on pourrait supprimer extends ou super, sauf si un autre paramètre imposait T

Sommaire

- Pile
- File
- Liste
- Tableau associatif

Le TAD Pile

```
1   TYPES
2       PILE[X]
3
4   FONCTIONS
5       nouvelle : → PILE[X]
6       empiler : X * PILE[X] → PILE[X]
7       dépiler : PILE[X] ↛ PILE[X]
8       sommet : PILE[X] ↛ X
9       est_vide : PILE[X] → BOOLÉEN
10
11  PRÉCONDITIONS
12      pré  dépiler(s) = non est_vide(s)
13      pré  sommet(s) = non est_vide(s)
14
15  AXIOMES
16      Pour tout x: X; p: PILE[X]
17          est_vide(nouvelle)
18          non est_vide(empiler(x, p))
19          sommet(empiler(x, p)) = x
20          dépiler(empiler(x, p)) = p
```

Caractéristiques d'une pile

Opérations : Les opérations minimales sur une pile sont :

- `initialiser` : créer une pile vide ;
- `empiler` : ajouter un élément en sommet de la pile ;
- `dépiler` : supprimer l'élément en sommet de pile ;
- `sommet` : obtenir l'élément en sommet de pile ;
- `estVide` : savoir si la pile est vide.

Réalisations : Deux réalisations classiques possibles :

- en utilisant un tableau ;
- en utilisant des structures chaînées et l'allocation dynamique de mémoire.

Le TAD File

But : Modéliser une file d'attente de type FIFO (First In, First Out).

```
1   TYPES
2       FILE[X]
3   FONCTIONS
4       nouvelle : → FILE[X]
5       ajouter : X ∗ FILE[X] → FILE[X]
6       extraire : FILE[X] ↛ FILE[X]
7       tête : FILE[X] ↛ X
8       est_vide : FILE[X] → BOOLÉEN
9   PRÉCONDITIONS
10      pré  ajouter(s) = non est_vide(s)
11      pré  tête(s) = non est_vide(s)
12  AXIOMES Pour tout x: X; f: FILE[X]
13      est_vide(nouvelle)
14      non est_vide(ajouter(x, f))
15      tete(ajouter(x, nouvelle)) = x
16      tete(ajouter(x, f)) = tete(f) si f ≠ nouvelle
17      extraire(ajouter(x, nouvelle)) = nouvelle
18      extraire(ajouter(x, f)) = ajouter(x, extraire(f))) si n ≠ nouvelle
```

Les opérations `ajouter` et `extraire` sont parfois appelées `enfiler` et `défiler`.

Les files

Opérations : Les opérations minimales sur une file sont :

- `initialiser` : créer une file vide ;
- `ajouter` : ajouter un élément en fin de file ;
- `extraire` : supprimer l'élément en début de file. En général, cette opération retourne l'élément extrait ;
- `tête` : obtenir l'élément en début de file ;
- `estVide` : savoir si la file est vide.

Exercice 5 Proposer une réalisation de File en utilisant un tableau pour stocker les éléments. Estimer le temps d'exécution de chaque opération.

Exercice 6 Proposer une réalisation de file en utilisant des structures chaînées. Estimer le temps d'exécution de chaque opération.

Exercice 7 Une file avec priorité permet d'ajouter de nouveaux éléments dans la file en précisant une priorité (entier positif). Si la priorité n'est pas précisée, c'est la priorité la plus faible (0) qui est utilisée.
Proposer des implantations possibles.

Liste

Principe : Une liste est une structure de données dans laquelle on repère un élément par son rang (sa position, son index, son indice).
Il n'y a pas consensus sur les opérations d'une liste :

- insérer un élément à une position
- supprimer un élément qui est à une certaine position
- remplacer l'élément qui est à une certaine position
- obtenir l'élément qui est à une certaine position
- obtenir la taille de la liste (son nombre d'élément)
- obtenir la position d'un élément dans la liste
- ...

Réalisations : Tableau ou structures chaînées.

Tableau associatif

Principe : Un tableau associatif est une structure de données dans laquelle on peut stocker une donnée grâce à une clé.
Les **opérations classiques** sont :

- ajouter un élément en précisant sa clé ;
- savoir si une clé est utilisée ;
- récupérer un élément à partir de sa clé ;
- supprimer l'élément associé à une clé.

Exemples :

- Le dictionnaire : la clé est un mot, la donnée, un texte.
- Répertoire téléphonique : la clé est un nom, la donnée, le téléphone.

Remarque :

- Équivalent à un tableau classique en généralisant le type des indices.

Réalisations :

- Les mêmes que pour l'ensemble.

Sommaire

Possibilités de représentation

Données contiguës en mémoire (*exemple :* L'ensemble tableau, T. 14)

- On accède à un élément grâce à sa position (indice ou index).
- L'accès est donc en temps constant (@donnée = index * taille(donnée)).
- L'insertion d'un élément est coûteuse (décaler).
- La suppression peut être coûteuse si l'ordre des éléments est important.

Données liées par un chaînage (*Exemple :* L'ensemble chaîné, T. 16)

- Chaque élément est encapsulé dans une cellule.
- Une cellule a un ou plusieurs accès (chaînages) sur d'autres cellules.
- L'accès nécessite de parcourir les cellules (temps *linéaire*).
- L'insertion et la suppression sont réalisées en temps constant.

Structures arborescentes :

- Diminuer les temps de traitement (log)

Mémoire automatique et dynamique

- l'organisation (contiguë ou chaînage) est indépendante de la mémoire.
- pour une même structure de données, il est préférable de ne pas mélanger mémoire automatique et mémoire dynamique

Sommaire

- Les interfaces
- Réalisations
- Algorithmes

Les collections

Définition : Une *collection* est un objet qui représente un groupe d'*éléments* de type E (généricité depuis java 1.5).

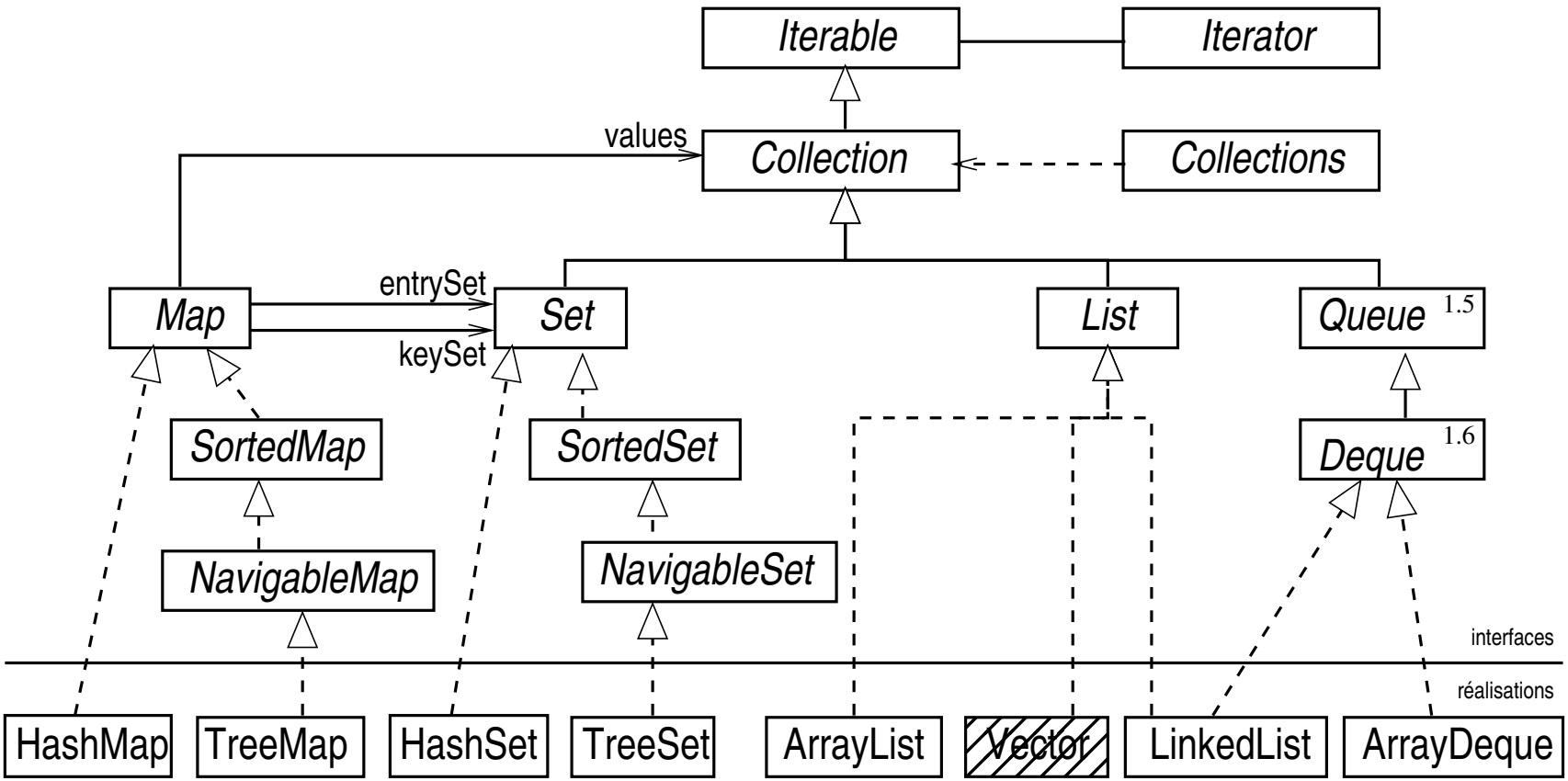
Principaux constituants :

- **Interfaces :** types abstraits de données spécifiant les collections :
  - un type (liste, ensemble, file, tableau associatif, etc.)
  - les opérations disponibles sur ce type
  - la sémantique (informelle) des opérations
- **Réalisations (implantations) :** réalisations concrètes des interfaces...
  - en s'appuyant sur différentes solutions pour stocker les éléments (tableau, structures chaînées, table de hachage, arbre, etc.).
- **Algorithmes :**
  - algorithmes classiques sur une collection (chercher, trier, etc.)
  - polymorphes : fonctionnent avec plusieurs collections

Intérêt des collections

- **Réduire les efforts de programmation**
  - réutiliser les collections de l'API
- **Augmenter la vitesse et la qualité des programmes**
  - efficaces car réalisées par des experts
  - les collections fournissent des opérations de haut niveau (testées !)
  - possibilité de substituer une réalisation par une autre
- **Permettre l'interopérabilité entre différentes API**
  - les données échangées le sont sous forme de collections.
- **Faciliter l'apprentissage de nouvelles API**
  - pas de partie spécifique traitant les collections
- **Favoriser la réutilisation logicielle :**
  - les anciens algorithmes fonctionneront avec les nouvelles collections
  - et les anciennes collections avec les nouveaux algorithmes

Hiérarchie des *collections* en Java (extrait)



Principales *collections*

- **Collection :** le type le plus général des collections.
- **List :** les éléments ont une position (numéro d'ordre, rang, index...)
  - opérations relatives à la position
- **Set :** ensemble au sens mathématique :
  - pas de double (ensemble mathématique), pas de position
  - **SortedSet** : éléments munis d'une relation d'ordre (éléments ordonnés)
    - Ne pas modifier un objet de la collection si incidence sur la relation d'ordre !
  - **NavigableSet** : *SortedSet* avec des opérations pour trouver un élément proche
    - lower, floor, ceiling, and higher
- **Queue** : file (d'attente), avec politique FIFO ou autre.
- **Deque** : (Double Ended Queue) :
  - opérations sur les deux extrémités de la file (File, Pile)
- **Map** : tableau associatif (accès aux éléments par une clé)
  - Attention, ce n'est pas un sous-type de collection !
  - sous-types : **SortedMap** et **NavigableMap** (relation d'ordre sur les clés)

L'interface *java.util.Collection*

Choix de conception

- *Collection* : super-type des structures de données de Java (sauf Map)
- Avant Java 1.5, les éléments d'une collection étaient du **type Object**
  Depuis 1.5, les collections sont **paramétrées par E**, type des éléments
- Une collection peut autoriser ou non plusieurs **occurrences** d'un même élément (List vs Set)
- Les éléments peuvent être **ordonnés** (position) ou non (List vs Set)
- Les éléments peuvent être **triés** ou non (Set vs SortedSet)
- Pour **limiter le nombre d'interfaces**, certaines méthodes peuvent :
  - ne pas être définies sur un sous-type (`UnsupportedOperationException`)
  - lever `ClassCastException` (cf `checkedList, etc.`)
- Toute réalisation d'une *Collection* devrait définir :
  - un constructeur par défaut (qui crée une collection vide) et
  - un constructeur qui prend en paramètre une collection (conversion)

L'interface *java.util.Collection<E>*

C'est la racine de la hiérarchie définissant les collections (groupe d'éléments).

```
boolean add(E o)            // ajouter l'élément (false si pas de modification)
boolean addAll(Collection<? extends E> c)  // ajouter les éléments de c
void clear()                // supprimer tous les éléments de la collection
boolean contains(Object o)                  // est-ce que la collection contient o ?
boolean containsAll(Collection<?> c)        //  ''  ''  tous les éléments de c ?
boolean isEmpty()                  // vide ?
Iterator<E> iterator()             // un itérateur sur la collection
boolean remove(Object o)           // supprimer l'objet o (collection changée ?)
boolean removeAll(Collection<?> c)    // supprimer tous les éléments de c
boolean retainAll(Collection<?> c)    // conserver les éléments de c
int size()                 // le nombre d'éléments dans la collection
Object[] toArray()         // un tableau contenant les éléments de la collection
<T> T[] toArray(T[] a)     // un tab. (a si assez grand) avec les éléments
```

Remarque : Certaines opérations sont optionnelles ou imposent des restrictions. Dans ce cas, elles doivent lever une exception !

L'interface *java.util.List<E>*

Structure de données où chaque élément peut être identifié par sa position. En plus des opérations de `Collection`, sont (re)spécifiées :

```
boolean add(E e)                  // ajouter e à la fin de la liste
E set(int i, E o)                 // remplacer l'élément à l'indice i par e
void add(int i, E e)              // insérer e à l'index i
boolean addAll(int, Collection<? extend E> c) // insérer les éléments de c à l'index i

E get(int i)                      // élément à l'index i
int indexOf(Object o)             // index de l'objet o dans la liste ou -1
int lastIndexOf(Object o)         // dernier index de o dans la liste
List<E> subList(int from, int to) // liste des éléments [from..to[

E remove(int i)                   // supprimer l'élément à l'index i

ListIterator<E> listIterator()    // itérateur double sens
ListIterator<E> listIterator(int i)   // ... initialisé à l'index i
```

L'interface `java.util.Queue<E>`

- a été ajoutée par Java 1.5
- implante les opérations spécifiées sur Collection
- fournit des opérations supplémentaires pour :

	lève une exception	retourne une	valeur spécifique
ajouter	add(E)	offer(E)	false
supprimer	remove	poll	null
examiner	element	peek	null

- l'élément **`null`** est généralement interdit dans une Queue.
- principalement une politique type FIFO (First In, First Out) mais d'autres politiques sont possibles (file avec priorité).

java.util.Map<K, V> : tableau associatif

- K : type des clés
- V : type des valeurs

```
V put(K k, V v)              // ajouter v avec la clé k (ou remplacer)
V get(Object k)              // la valeur associée à la clé (ou null)
V remove(Object k)           // supprimer l'entrée associée à k
void putAll(Map<? extends K,? extends V> m)  // ajouter les entrées de m

boolean containsKey(Object k)       // k est-elle une clé utilisée ?
boolean containsValue(Object v)     // v est-elle une valeur de la table ?

Set<Map.Entry<K, V>> entrySet()     // toutes les entrées de la table
Set<K>               keySet()       // l'ensemble des clés
Collection<V>        values()       // la collection des valeurs

int size()                 // nombre d'entrées dans la table
boolean isEmpty()          // la table est-elle vide ?
void clear()               // vider la table
```

Exemple d'utilisation des Map

```
1  import java.util.*;
2  public class CompteNbOccurrences {
3      /** Compter le nombre d'occurrences des chaînes de args... */
4      public static void main(String[] args) {
5          Map<String, Integer> occ = new HashMap<String, Integer>();
6          for (String s : args) {
7              int ancien = occ.containsKey(s) ? occ.get(s) : 0;
8              occ.put(s, ancien + 1);
9          }
10         System.out.println("occ = " + occ);
11         System.out.println("clés = " + occ.keySet());
12         System.out.println("valeurs = " + occ.values());
13         System.out.println("entrées = " + occ.entrySet());
14
15         // afficher chaque entrée
16         for (Map.Entry<String, Integer> e : occ.entrySet()) {
17             System.out.println(e.getKey() + " --> " + e.getValue());
18         }
19     }
20 }
```

Exemple d'utilisation des Map (exécution)

```
1  > java CompteNbOccurrences A B C A C D E A A D E
```

```
1  occ = {A=4, B=1, C=2, D=2, E=2}
2  clés = [A, B, C, D, E]
3  valeurs = [4, 1, 2, 2, 2]
4  entrées = [A=4, B=1, C=2, D=2, E=2]
5  A --> 4
6  B --> 1
7  C --> 2
8  D --> 2
9  E --> 2
```

Question : Comment afficher les chaînes dans l'ordre ?

Relation d'ordre

- Deux manières de définir la relation d'ordre :
  - ordre explicite défini sous la forme d'un `java.util.Comparator`
  - sinon ordre naturel : le type de l'élément doit réaliser l'interface `java.lang.Comparable`
- Les classes qui ont besoin d'une relation d'ordre :
  - soit prennent un comparateur en paramètre (constructeur)
  - soit, à défaut, utilisent l'ordre naturel
  - soit signalent un problème !

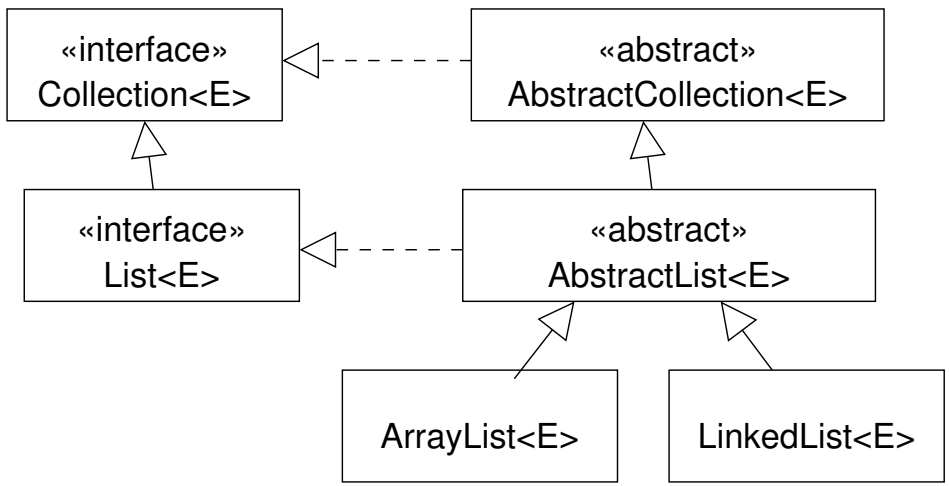
Principales réalisations

Interface	**Réalisation**				
	table de hachage	tableau	arbre	liste chaînée	liste chaînée + table hachage
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue		ArrayDeque		LinkedList	
Deque		ArrayDeque			
Map	HashMap		TreeMap		LinkedHashMap

- Il y en a d'autres ! En particulier dans le paquetage `java.util.concurrent`.
- Dans `LinkedHashMap` et `LinkedHashSet`, la liste conserve l'ordre d'insertion des éléments (utilisée lors d'un parcours)

Les classes abstraites

- **Objectif :** Factoriser le code commun à plusieurs réalisation.
  $\Longrightarrow$ écrire plus facilement de nouvelles réalisations.



- AbstractCollection définit toutes les opérations de Collection sauf size et iterator
  - Collection concrète non modifiable : définir seulement `size` et `iterator`
  - Collection concrète modifiable : définir aussi add (et `remove` sur l'iterator).
- En réalité, `LinkedList` n'hérite pas directement de `AbstractList`.

Les algorithmes : la classe Collections

Classe *utilitaire* qui contient des méthodes pour :

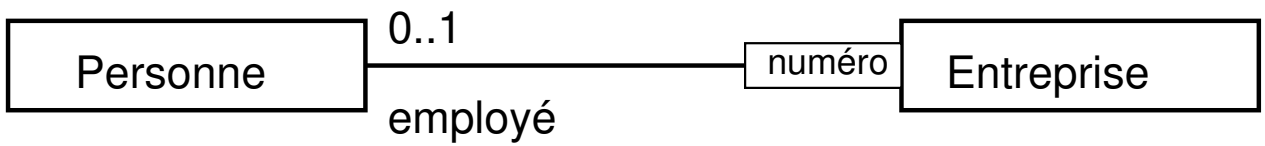
- trier les éléments d'une collection (*sort*)
  - rapide (en $n \log(n)$)
  - stable (l'ordre est conservé entre les éléments égaux)
- mélanger les éléments d'une collection (*shuffle*)
- manipulation des données :
  - *reverse :* inverser l'ordre des éléments d'une List
  - *fill :* remplacer tous les éléments d'une List par une valeur
  - *copy :* les éléments d'une liste source vers une liste destination
  - *swap :* permuter les éléments de deux listes
  - *addAll :* ajouter des éléments à une collection
- Chercher des éléments (*binarySearch*), nécessite une relation d'ordre
- Compter le nombre d'occurrences d'un élément (*frequency*) et vérifier si deux collections sont disjointes (*disjoint*)
- Trouver le min et le max (nécessite une relation d'ordre).

Remarque : Voir `Comparable` et `Comparator` pour les relations d'ordre.

Sommaire

UML : Choix de la structure de données adaptée

- Le diagramme UML peut être complété pour préciser la structure de données adaptée grâce aux contraintes :
  - unique : donc Set
  - ordered : List (ou OrderedSet)
    - dépend si l'ordre signifie rang
    - ou relation d'ordre sur les éléments
  - non-unique, non-ordered : Multi-ensemble (Bag) : n'est pas dans les collections de Java
- Un qualificatif UML incite à prendre un tableau associatif (Map)



Sommaire

- Interface de marquage
- Adaptateur
- Mandataire (Proxy)

Efficacité de List.get(int)

Exercice 8 L'interface `java.util.List` de l'API Java propose deux réalisations : ArrayList et LinkedList. On constate que les méthodes search et sort de Collections doivent faire attention car l'opération get(int) n'a pas la même complexité pour les deux réalisations.
Ce problème est plus général : dès qu'on manipule une List, on doit s'interroger sur l'efficacité de get(int).
Comment faire pour savoir si get(int) est efficace ?

Interface de marquage

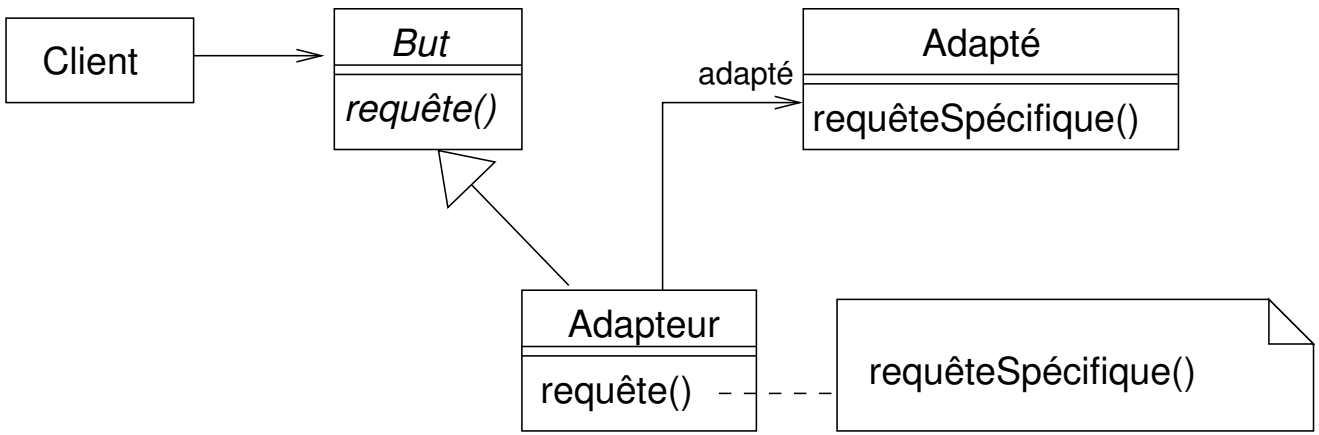
- Utiliser une interface de marquage.
- **Principe :** Définir une propriété sémantique booléenne sur une classe
- Application
  - L'interface `RandomAccess` définit cette propriété.
  - `ArrayList` la réalise.
  - `LinkedList` ne la réalise pas.
  - Remarque : `RandomAccess` ne contient aucune méthode !
- **Intérêt :** tester la propriété sans connaître la classe réelle
  - utilisation de **`instanceof`**
  - exemple : `if (l instanceof RandomAccess) {`
- ATTENTION La propriété ne peut jamais être supprimée/désactivée
  - obligatoirement héritée par les sous-types

Utiliser un tableau là où une liste est attendue

Exercice 9 : Trier un tableau avec Collections.sort
On veut pouvoir utiliser un tableau comme paramètre d'une méthode (sort de Collections par exemple) qui prend en paramètre une liste (List).
Comment faire ?

Le patron Adaptateur

- Définir une classe jouant le rôle d'adaptateur entre la liste et le tableau :
  - elle définit les opérations de la liste
  - et les traduit en terme d'opérations sur le tableau



Client	But	Adapté	Adaptateur	requête	requêteSpécifique
sort	List	array	Arrays$ArrayList	opération de List	opération de tableau

Exemple

```java
import java.util.*;
public class SortTableau {
    public static void main(String[] args) {
        List<Integer> l = new ArrayList<Integer>();
        Collections.addAll(l, 2, 4, 1, 8, 9, 3, 1);
        System.out.println("l = " + l);
        Collections.sort(l);
        System.out.println("l = " + l);

        Integer tab[] = { 2, 4, 1, 8, 9, 3, 1 };
        List<Integer> lt = Arrays.asList(tab);
        System.out.println("tab = " + Arrays.toString(tab));
        Collections.sort(lt);
        System.out.println("tab = " + Arrays.toString(tab));
        System.out.println("La classe de lt = " + lt.getClass());
    }
}
```

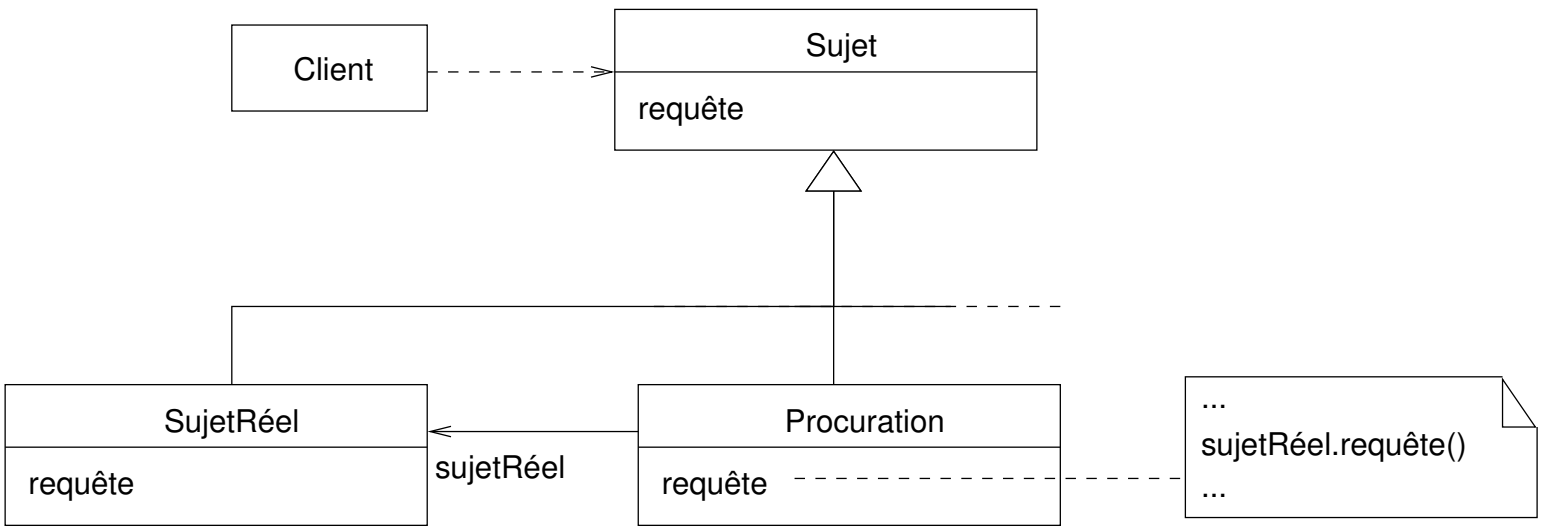
```
l = [2, 4, 1, 8, 9, 3, 1]
l = [1, 1, 2, 3, 4, 8, 9]
tab = [2, 4, 1, 8, 9, 3, 1]
tab = [1, 1, 2, 3, 4, 8, 9]
La classe de lt = class java.util.Arrays$ArrayList
```

Remarque : Il existe des méthodes `sort` dans la classe `java.util.Arrays`.

Être sûr qu'une liste ne sera pas modifiée par une méthode

Exercice 10 : Ne pas pouvoir modifier une liste
Si l'on fournit une liste comme paramètre effectif d'une méthode, on n'a aucune garantie en Java que cette liste ne sera pas modifiée par la méthode.
Comment faire pour être sûr que notre liste ne sera pas modifiée ?

- Solution 1 : Faire une copie de la liste
  - Inconvénient : coûteux (si la méthode ne modifie pas la liste !)
- Solution 2 : Utiliser un mandataire (intermédiaire ou proxy) qui vérifie si l'opération peut être appelée et lève une exception dans le cas contraire.
  C'est le patron Proxy ou Mandataire.



Exemple

```
1  import java.util.*;
2  public class ProxyList {
3      private static void m1(List<Integer> l) {
4          System.out.println("taille = " + l.size());
5          System.out.println("premier = " + l.get(0));
6          l.remove(0);
7      }
8      public static void main(String[] args) {
9          List<Integer> l = new ArrayList<Integer>();
10         Collections.addAll(l, 2, 4, 1, 8, 9, 3, 1);
11         System.out.println("l = " + l);
12         m1(l);
13         List<Integer> rol = Collections.unmodifiableList(l);
14         System.out.println("rol = " + rol);
15         System.out.println("Classe de rol : " + rol.getClass());
16         m1(rol);
17     }
18 }
```

Exécution

```
1  l = [2, 4, 1, 8, 9, 3, 1]
2  taille = 7
3  premier = 2
4  rol = [4, 1, 8, 9, 3, 1]
5  Classe de rol : class java.util.Collections$UnmodifiableRandomAccessList
6  taille = 6
7  premier = 4
8  Exception in thread "main" java.lang.UnsupportedOperationException
9      at java.util.Collections$UnmodifiableList.remove(Collections.java:1317)
10     at ProxyList.m1(ProxyList.java:6)
11     at ProxyList.main(ProxyList.java:16)
```

Utilisation du patron Proxy/Mandataire

- procuration à distance : représentant local d'un objet distant
- procuration virtuelle : création d'objet à la demande
- procuration de protection : contrôle les accès
- référence intelligente (*smart pointer*) : remplaçant d'un pointeur brut qui ajoute compteur de références, chargement en mémoire d'un objet persistant...

Sommaire

- Type Abstrait de Données (TAD)

Le Type Abstrait de Données (TAD) Ensemble

```
1  TYPES
2      Ensemble[X]
3  FONCTIONS
4      ensemble_vide : -> Ensemble
5      ajouter : Ensemble * X : -> Ensemble
6      card : Ensemble -> Cardinal
7      dans : Ensemble * X -> Booléen
8      supprimer : Ensemble * X -> Ensemble
9      union : Ensemble * Ensemble -> Ensemble
10     intersection : Ensemble * Ensemble -> Ensemble
11 AXIOMES
12     SOIT e, e1, e2: Ensemble; x,y: X
13
14     supprimer (ensemble_vide, x) = ensemble_vide
15     supprimer (ajouter (e, x), y) =
16         SI x = y ALORS supprimer (e, y) SINON ajouter (supprimer (e, y), x)
17     dans (ensemble_vide, x) = faux
18     dans (ajouter (e, x), y) =
19         SI x = y ALORS vrai SINON dans (e y)
20     card (ensemble_vide) = 0
21     card (ajouter (e, x)) = 1 + card (supprimer (e, x))
22     union (ensemble_vide, e) = e
23     union (ajouter (e1, x), e2) = ajouter (union (e1, e2), x)
24     intersection (ensemble_vide, e) = ensemble_vide
25     intersection (ajouter (e1, x), e2) =
26         SI dans (e2, x)
27         ALORS ajouter (intersection (e1, e2), x)
28         SINON intersection (e1, e2)
29 FIN Ensemble.
```

Définition d'un TAD

Définition : Un TAD formalise la syntaxe et la sémantique des opérations d'un type indépendamment d'une réalisation (implantation) particulière.
Un type abstrait de données est un **contrat** entre :

- **l'utilisateur du type** qui sait comment utiliser les opérations ;
- l'**implémenteur du type** : La représentation du type et l'implémentation des opérations ne sont pas imposées par le TAD. L'implémenteur est libre de faire les choix qu'il considère judicieux.

Un type abstrait de données permet :

- l'**encapsulation** : regrouper au même endroit les données (les types) et les opérations qui les manipulent ;
- l'**abstraction de données** (ou **masquage d'information**, « *information hiding* ») : ne pas révéler les détails de réalisation.

Structure d'un TAD

But : Décrire précisément, sans ambiguïté, complètement et sans sur-spécification une structure de données abstraite (sans faire de supposition sur son implémentation).
Description : La description d'un TAD est structurée en 4 parties :

- **types :** nom des types en cours de spécifications (ex : Pile, Liste...) ;
- **fonctions :** nom et signature des opérations applicables sur les types ;
  Remarque : Le terme « fonction » est à prendre au sens algorithmique et mathématique : fonctions partielles dont tous les paramètres sont en `in`. Elles n'ont pas d'effet de bord !
- **axiomes :** définition **implicite** des fonctions.
  Remarque : Ils sont exprimés en utilisant la réécriture ($\simeq$ récursivité).
- **préconditions :** préciser le domaine des fonctions partielles.

Propriétés d'un TAD

Propriétés : Voici les principales propriétés (à respecter) :

- Dire ce qu'il faut faire (spécifier), avec précision et rigueur ;
- Ne pas dire comment le faire (seulement spécifier) ;
- Exprimer les propriétés intrinsèques aux opérations ;
- Éviter d'imposer des contraintes d'implémentation.

Intérêts :

- Ne pas être encombré par les détails de réalisation/implémentation
- Ne pas être influencé par des contraintes supplémentaires
- Repousser les choix de représentation aux étapes ultérieures

Inconvénients :

- Les opérations sont connues mais pas leur coût d'utilisation !
- Le TAD doit être stable (pérennité des applications l'utilisant)

Classification des fonctions d'un TAD

La signature d'une fonction permet de préciser sa nature :

- les **constructeurs** : un constructeur est une opération qui produit un élément du type à partir d'éléments d'autres types.
  Propriété : Le type n'apparaît que dans les résultats
  $\Longrightarrow$ Ils deviennent des constructeurs ou des constantes.
- les **accesseurs** (ou **observateurs**) : ce sont des opérations qui fournissent une information sur un élément du type.
  Propriété : Le type n'apparaît pas dans les résultats
  $\Longrightarrow$ Ils deviennent des requêtes (fonctions).
- les **modifieurs** ou **transformateurs** : ce sont des opérations qui modifient (transforment, altèrent) un élément du type.
  Propriété : Le type apparaît dans les données et les résultats.
  $\Longrightarrow$ Ils deviennent des commandes (procédures).

Remarque : Un modifieur peut aussi être défini comme une requête (approche fonctionnelle).

Réalisation d'un TAD

Un type abstrait de données ne précise pas les détails de réalisation.

Le programmeur (implémenteur) d'un TAD doit alors :

- choisir une représentation pour le ou les types ;
- implanter les fonctions du TAD ;

En pratique, on utilise :

- une interface pour spécifier le TAD (fonctions du TAD)
- la généricité si le TAD est paramétré par un ou plusieurs types
- la programmation par contrat pour traduire (partiellement) la partie axiomatique
- une ou plusieurs réalisations correspondant à des choix d'implantation :
  - les attributs
  - les algorithmes des opérations
  - les constructeurs