Technologie Objet

—

Structures de données — Collections — Introduction aux patrons de conception

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

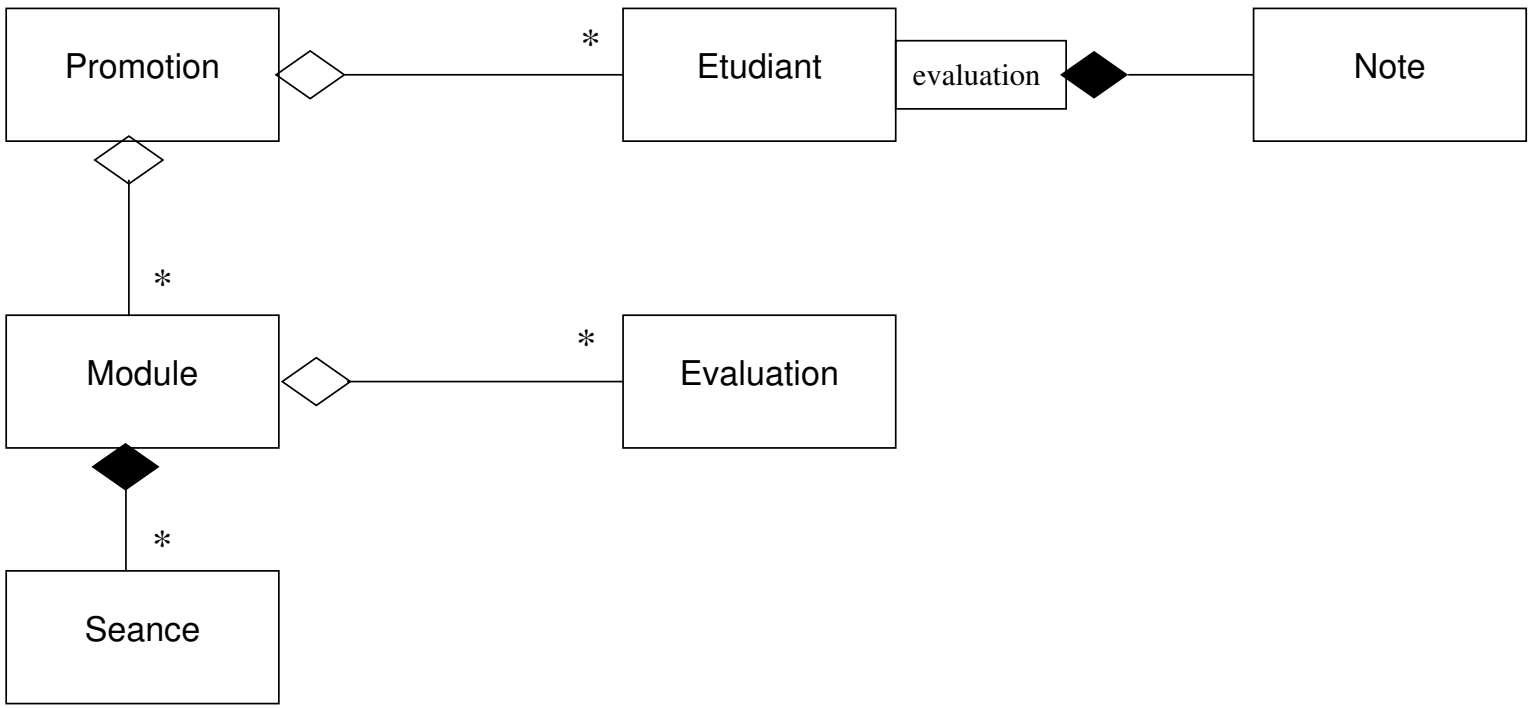
ENSEEIHT
Sciences du Numérique

Sommaire

Sommaire

Motivation

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



Objectifs de ce support

- Principales structures de données
- Techniques d'implantation
- **Généricité et sous-typage**
- **Classes internes** en Java
- **Les structures de données de l'API Java**
- **UML et structures de données**
- **Quelques patrons de conception**
- Une synthèse des concepts objets
- Un fil rouge pour illustrer ces différents aspects : les ensembles

Sommaire

- Historique
- Intérêt
- Définition
- Exemple : Composite
- Exemple : Commande
- Anti-patron

Historique

Notion de « **patron** » d'abord apparue en architecture :

- Christopher Alexander : « A Pattern Language : Towns, Buildings, Construction », 1977
- Il définit des « patrons » pour :
  - l'architecture des bâtiments
  - la conception des villes et de leur environnement

  *« Chaque modèle [patron] décrit un **problème qui se manifeste constamment** dans notre environnement, et donc **décrit le cœur de la solution** de ce problème, d'une façon telle que l'on peut **réutiliser cette solution** des millions de fois, sans **jamais le faire deux fois de la même manière**. » Christopher Alexander*

  Exemple : Une pièce doit être lumineuse et pas trop chaude en été.

Idée : appliquer la notion de patron à du logiciel : « design patterns »

- premiers patrons à partir de 1987 (partie de la thèse de Erich Gamma)
- puis Richard Helm, John Vlissides et Ralph Johnson (« Gang of Four, GoF »)
- 1er catalogue en 1993 : *Elements of Reusable Object-Oriented Software*

Vocabulaire : design patterns, patrons de conception, micro-architectures

Intérêt des patrons de conception

Pourquoi définir des patrons de conception

- Construire des systèmes plus extensibles, évolutifs, maintenables, réutilisables
- Capitaliser l'*expérience collective* des informaticiens
- Réutiliser les solutions qui ont fait leur preuve
- Identifier les avantages/inconvénients/limites de ces solutions
- Savoir quand les appliquer

Complémentaire avec les API

- une API propose des solutions directement utilisables
- un patron explique comment structurer son application ou une API

Patron de conception dans le cycle de développement

- intervient en conception détaillée
- reste indépendant du langage d'implantation

Qu'est ce qu'un patron de conception ?

Définition : Un *patron de conception* (*design pattern*) décrit une structure commune et répétitive de composants en interaction (la **solution**) qui résout un **problème de conception** dans un **contexte particulier**.

Plus court : solution éprouvée à des problèmes récurrents !

Quatre éléments principaux

- **nom :** un ou deux mots pour identifier le patron
- **problème :** situation où le problème s'applique
- **solution :** éléments de la conception, leurs relations et collaborations
  - la solution n'est pas forcément précise : idée d'architecture.
  - plusieurs variantes peuvent être possibles.
- **conséquences :** effets résultants et compromis induits (arbitrage !)

Un **bon patron de conception** :

- résout un problème
- correspond à une solution éprouvée
- favorise la réutilisabilité, l'extensibilité, etc.
- inclut une composante subjective : utilité, esthétique, etc.

Premier exemple

Problèmes similaires :

- Associer plusieurs objets géométriques pour les considérer comme un seul (Groupe)
- Définir un agenda hiérarchique pour gérer simultanément plusieurs agendas (AgendaHierarchique ou GroupeAgenda)
- Représenter un système de gestion de fichiers avec des dossiers et des fichiers.
- ...

Constations

- Ce sont des problèmes proches : représentation d'une structure arborescente
- On a adopté a peu près la même manière de les traiter
- Mais avec des variantes (pas les mêmes méthodes)

$\Longrightarrow$ Cette solution (expérience) est déjà capitalisée dans un patron de conception !

Patron de conception : Composite

Intention

- Organise des objets en des structures arborescentes : hiérarchies composant/composé.
- Permet au client de traiter uniformément un objet individuel ou les combinaisons de ceux-ci

Alias : —

Indications d'utilisation

- représentation de structures récursives d'éléments hétérogènes
- traitement uniforme d'un objet individuel ou d'une combinaison d'objets

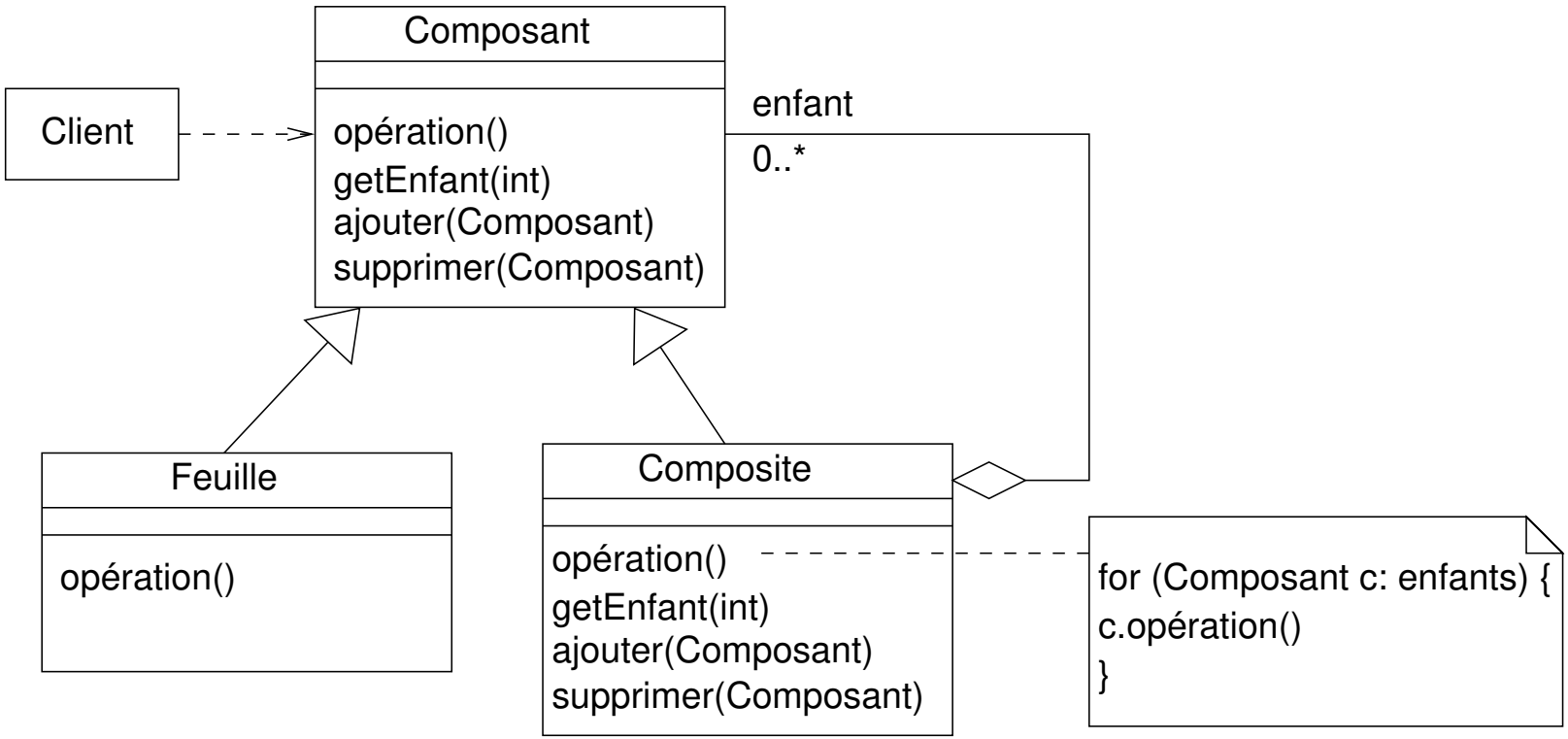
Exemples :

- Groupe d'objets géométriques dans éditeur de schémas mathématiques
- Agendas hiérarchiques

Conséquences :

+ le client est simple (traitement uniforme des objets composants ou composés)
+ facilite l'ajout de nouveaux types de composants
– difficile d'imposer des contraintes sur les compositions possibles

Composite : Diagramme de classe



Les opérations sur les composés peuvent ou non être remontées au niveau du composant

Exemple : Groupe d'objets géométriques.

Commande (Command)

Intention

- Découpler celui qui exécute le traitement et celui qui le définit. Réifier le traitement.
- Gestion possible des traitements en FIFO. Annulation possible.

Alias : Action, Transaction

Indications d'utilisation :

- réaliser un principe de *callback* : fournir le code à exécuter quand quelque chose se produit

  Exemple : Menu textuel (Invocateur) pour un éditeur d'une ligne (Récepteur)
- mémoriser les commandes à exécuter dans une file d'attente (et les exécuter en différé)

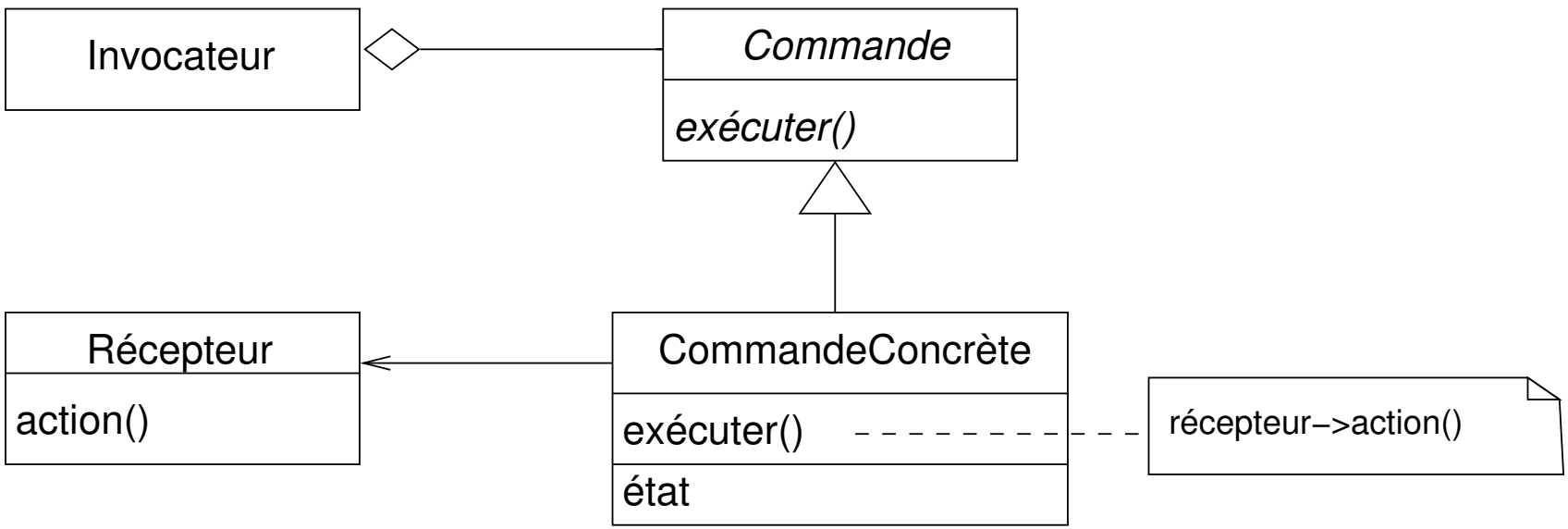
  Exemple : Un serveur d'impression qui reçoit les travaux à imprimer
- permettre de mémoriser les commandes et d'annuler leurs effets

  Exemple : L'éditeur d'une ligne pourrait permettre d'annuler les opérations réalisées
- structurer le système en opérations de haut niveau (macro-commandes : Composite !)

Exemples :

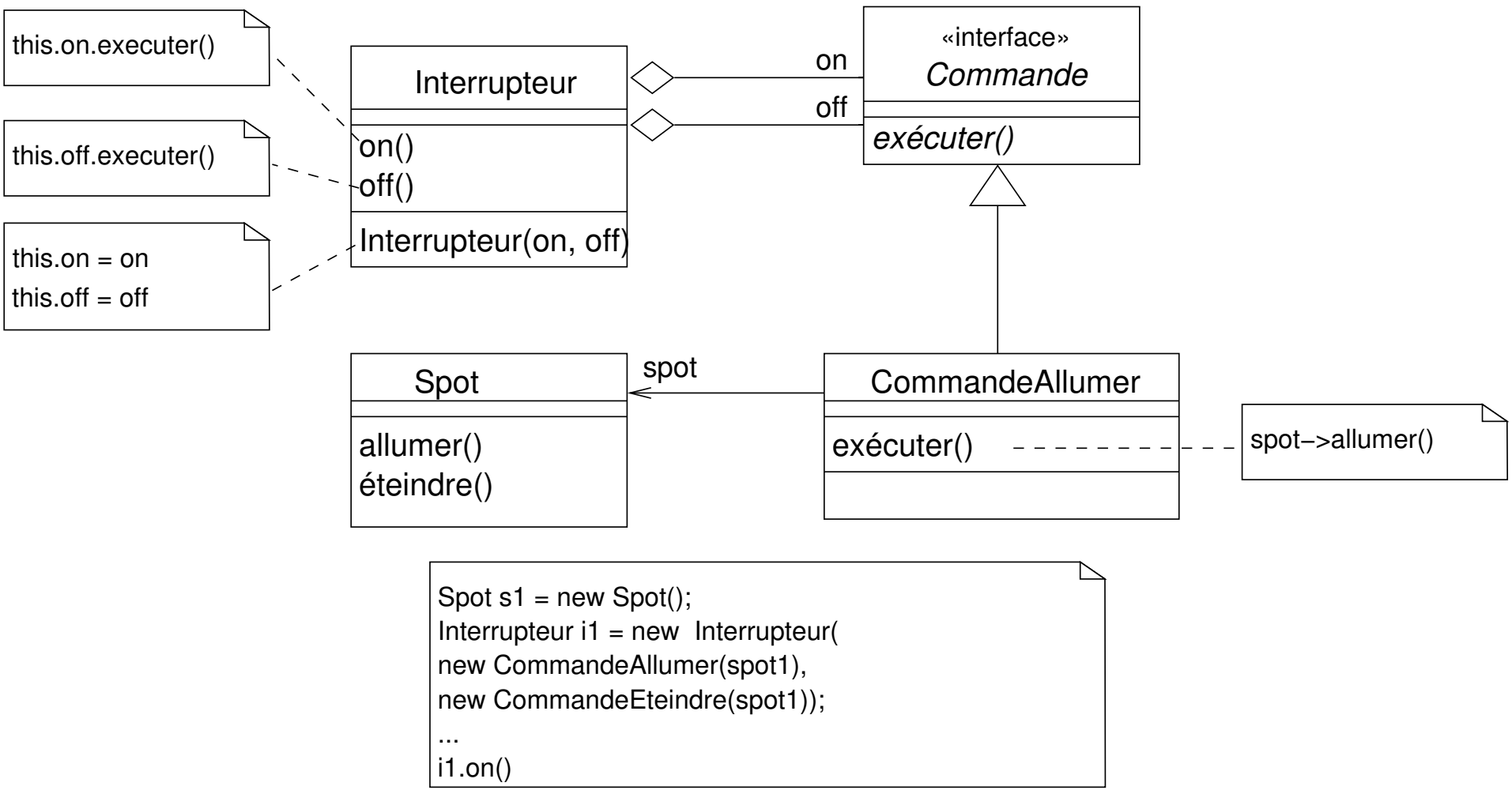
- Une télécommande qui pilote différents appareils (ventilateur, spot...) : examen 10/06/2013
- Un programme (séquence de tâches) qui pilote un robot : examen 24/05/2017
- Un interrupteur qui permet d'allumer/éteindre un spot, un ventilateur, etc.

Commande : Diagramme de classe



- L'Invocateur exécutera du code qui est spécifié par une Commande
- Une CommandeConcrète définit le code à exécuter,
  généralement en s'appuyant sur une action définie sur un Récepteur.
- Il s'agit de définir les commandes concrètes et les enregistrer auprès de l'Invocateur

Commande : Exemple d'un Interrupteur



Anti-patron (antipattern)

Anti-patron : Une pratique qui a plus d'inconvénients que d'avantages. À éviter donc !

Voir `http://fr.wikipedia.org/wiki/Antipattern`, `http://wiki.c2.com/?AntiPatternsCatalog`...

Quelques exemples ...

Marteau doré : Quand on n'a qu'un marteau, tous les problèmes ressemblent à des clous !
$\Rightarrow$ Utiliser le bon outil. Ne pas se limiter à ce qu'on connait déjà !

Vous n'en aurez pas besoin : Il ne faut pas implanter maintenant quelque chose dont on pense qu'il sera utile plus tard.

Action à distance : Le comportement d'une partie du programme dépend de manière difficile à identifier d'opérations exécutées dans d'autres parties (cause possible : variables globales)

Nombre magique : Utiliser des nombres non expliqués dans un algorithme
$\Longrightarrow$ Il faut utiliser des constantes bien nommées

Object orgy : Ne pas respecter le principe d'encapsultation.

Objet divin (God object) : Un objet (une classe) qui fait trop de choses !
$\Longrightarrow$ Répartir les responsabilités entre plusieurs objets (classes)

Sommaire

- Spécification d'un ensemble
- Utilisation d'un ensemble
- Quelques implantations
- Manipuler les éléments
- Classes internes
- Modifier pendant un parcours
- Et en Java ?

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

```java
public interface Ensemble<E> {
    /** Obtenir le nombre d'éléments dans l'ensemble.
     * @return nombre d'éléments dans l'ensemble.  */
    int cardinal();

    /** Savoir si un élément est présent dans l'ensemble.
     * @param x l'élément cherché
     * @return x est dans l'ensemble */
    boolean contient(E x);

    /** Ajouter un élément dans l'ensemble.
     * @param x l'élément à ajouter */
    void ajouter(E x);

    /** Enlever un élément de l'ensemble.
      * @param x l'élément à supprimer */
    void supprimer(E x);
}
```

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 helper @*/ 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 helper @*/ boolean contient(E x);
12
13     /** Ajouter un élément dans l'ensemble.
14      * @param x l'élément à ajouter */
15     //@ ensures contient(x);        // élément ajouté
16     void ajouter(E x);
17
18     /** Enlever un élément de l'ensemble.
19       * @param x l'élément à supprimer */
20     //@ ensures ! contient(x);      // élément supprimé
21     void supprimer(E x);
22 }
```

Rq : Les postconditions données ici sont partielles : limitées aux postconditions naturelles.

Compter le nombre de mots différents

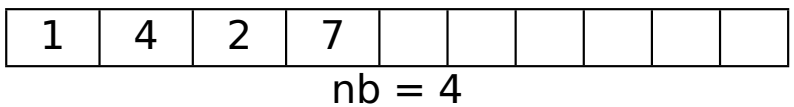
```
1  public class CompteurMots {
2     public static void main(String[] args) {
3        Ensemble<String> mots = new EnsembleTab<String>(100);
4        for (String mot : args) {
5           mots.ajouter(mot);
6        }
7        System.out.println("Nombre de mots différents : " + mots.cardinal());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

```
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
```

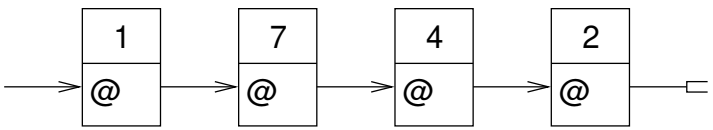
On anticipe un peu en créant EnsembleTab...

Implantations possibles

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



- avec une liste chaînée : peu efficace
  - éléments chaînés entre eux

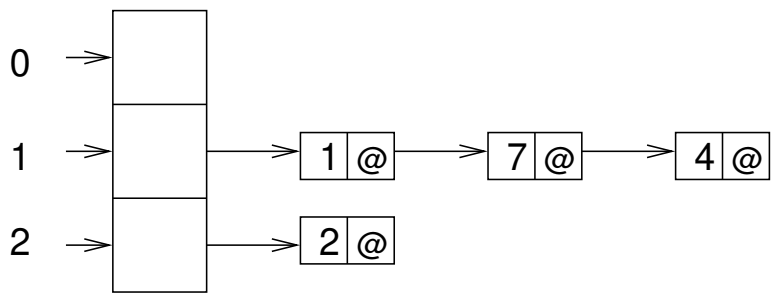


- avec un vecteur caractéristique : toutes les opérations en temps constant
  - l'élément sert d'indice
  - une case du tableau indique si l'élément en indice est présent ou non

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

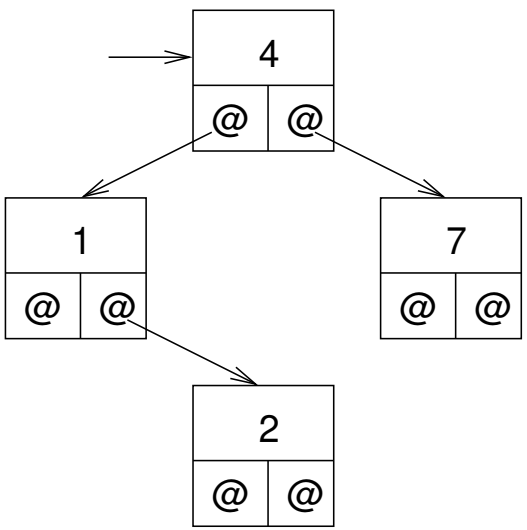
Implantations possibles (2)

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

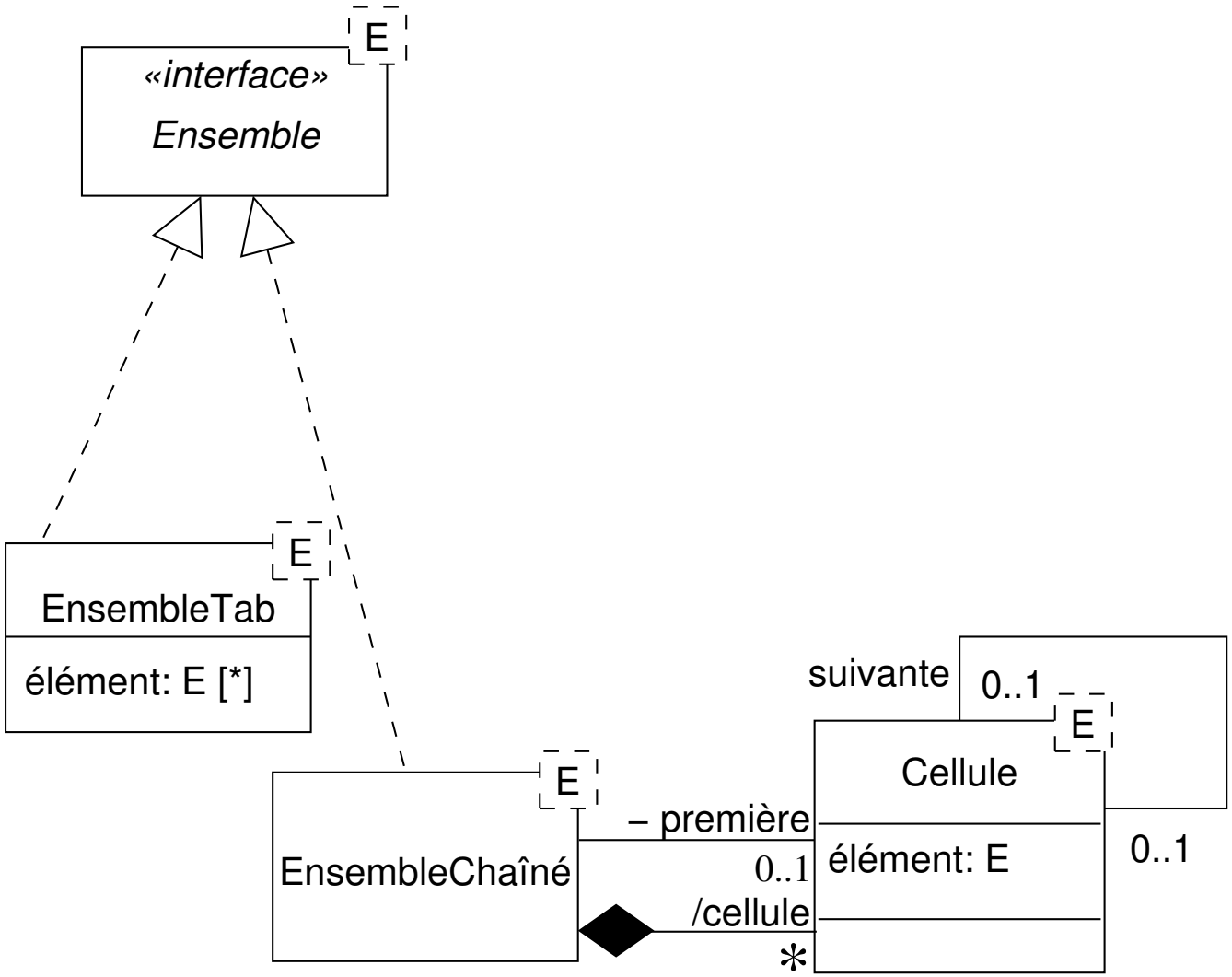


Implantations possibles (3)

- 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



Ensemble et ses réalisations EnsembleTab et EnsembleChaine



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     @Override 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     @Override public boolean contient(E x) {
26         return positionDe(x) < nb;
27     }
28
```

Implantation avec un tableau (2)

```
29      @Override 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      @Override public void ajouter(E x) {
39          if (!contient(x)) {
40              garantirNonPlein();
41              this.elements[nb++] = x;
42      }   }
43
44      /** Agrandir le tableau s'il est plein.  */
45      private void garantirNonPlein() {
46          if (this.nb >= this.elements.length) {
47              // agrandir le tableau
48              this.elements = java.util.Arrays.copyOf(this.elements,
49                      this.elements.length + 3);  // 3 arbitraire !
50  }   }   }
```

- Mettre à **null** la case du tableau est important pour ne pas garder une référence inutile sur un objet qui, sinon, ne pourrait pas être récupéré par le ramasse-miettes.
- Essayer d'agrandir le tableau est nécessaire car la spécification de ajouter dans l'interface Ensemble nous dit que l'on peut toujours ajouter un élément dans un ensemble.

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     @Override 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 }   }
```

- classe locale au paquetage (non publique)
- droit d'accès paquetage $\Longrightarrow$ accessible depuis le paquetage seulement
- ni accesseurs, ni modifieurs $\Longrightarrow$ code peu « naturel » 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    @Override public int cardinal() {
12       return nb;
13    }
14
15    @Override 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    @Override 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      @Override public void supprimer(E x) {
30         if (this.premiere != null) {
31            if (this.premiere.element.equals(x)) {
32               // Supprimer la première cellule
33               this.premiere = this.premiere.suivante;
34               this.nb--;
35            } else {
36               // Chercher la cellule avant l'élément à supprimer
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   }  }  }  }  }
```

- Comme pour `EnsembleTab`, on utilise `equals()` qui correspond à l'*égalité logique*.
- Pour `supprimer`, il faut distinguer le cas où on supprimer la première cellule ou une cellule suivante. Pour unifier, il serait possible d'utiliser une sentinelle, une première cellule dont la valeur ne serait pas utilisée.

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 ?

Nous illustrons ici des utilisations d'un ensemble. On ne doit donc pas spécifier les opérations demandées dans l'interface `Ensemble`. D'autant plus que certaines opérations imposent des contraintes sur le type des éléments de l'ensemble (réel, entier).

Afficher pour EnsembleTab et EnsembleChaine

```
1   public class Afficheur {
2
3       public static <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      public static <E> void afficher(EnsembleTab<E> ens) {
11          int i = 0;
12          while (i < ens.nb) {
13              System.out.println(ens.elements[i]);
14              i = i + 1;
15  }   }   }
```

- 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 (dû !) utiliser les types joker (<?>), voir T. 65
- **Question :** Quels sont les autres défauts de cette approche ?
- **Question :** Pourquoi avoir utilisé un **`while`** et non un **`for`** pour `afficher(EnsembleTab)` ?

Utilisation des méthodes afficher

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

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 donnera peut-être encore un ordre différent.
- Que se passe-t-il si on considère une nouvelle implantation de l'ensemble ?
  - Faut-il vraiment écrire une nouvelle méthode afficher ?
- Peut-on afficher un ensemble de type Ensemble<E> ?

Comment unifier les deux solutions ?

Les deux algorithmes ont la même structure.

- D'où l'utilisation du **while** au lieu du **for**.

On peut les unifier...

On obtient alors le même raffinage (algorithme) :

```
1  public static <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  }   }
```

Comment le traduire en Java ?

Définir de nouvelles opérations

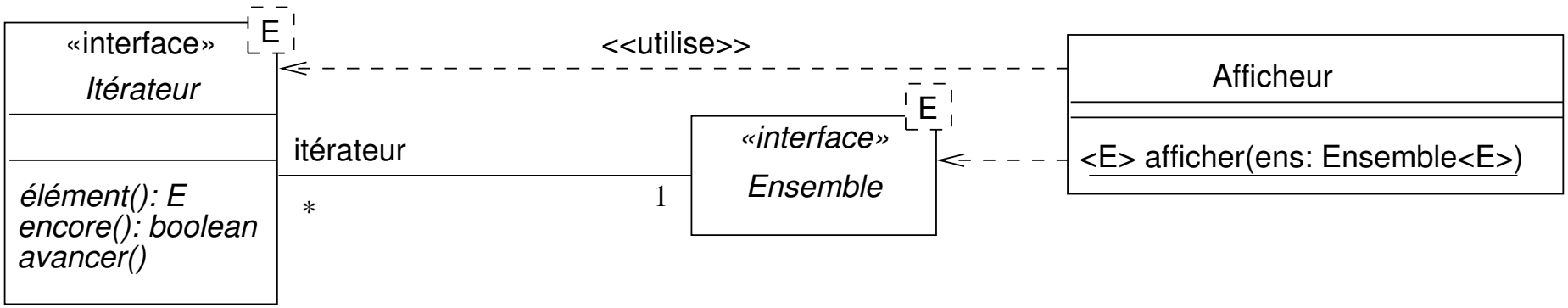
On identifie 4 nouvelles opérations :

- commencer / initialiser : `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 (sauf à ajouter d'autres méthodes)
   - Et donc 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
   - On a une bonne séparation des responsabilités (Ensemble : les données, Iterateur : le parcours)

Principe 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, list, arbre, entiers pairs, nombres premiers, etc.

Patron Itérateur (Iterator)

Intention

Fournit un moyen pour accéder séquentiellement à chacun des éléments d'un agrégat d'objets sans révéler la représentation interne de l'agrégat

Alias : Curseur (Cursor)

Indications d'utilisation

- accéder aux éléments d'un agrégat sans révéler sa structure interne
- gérer simultanément plusieurs parcours sur des agrégats
- offrir une interface uniforme pour parcourir différents types d'agrégats

Exemples :

- Les itérateurs des collections Java

Conséquences :

+ possibilité de définir plusieurs parcours (infixe et préfixe par exemple)
+ simplification de l'interface de l'agrégat
+ parcours simultanés possibles sur un même agrégat
- peut violer l'encapsulation : donne accès aux éléments de l'agrégat

L'interface Iterateur

```
1  public interface Iterateur<E> {
2
3      /** Existe-t-il des éléments non encore consultés ? */
4      boolean encore();
5
6      /** Obtenir l'élément courant.
7       * @throws NoSuchElementException si plus d'éléments à parcourir
8       */
9      E element();
10
11     /** Avancer le curseur.
12      * @throws NoSuchElementException si plus d'éléments à parcourir
13      */
14     void avancer();
15 }
```

- Exceptions nécessaires car element() et avancer() ne sont possibles que s'il y a encore des éléments.

```
1  public interface Ensemble<E> {
2      ...
3
4      /** Un itérateur sur l'ensemble. */
5      Iterateur<E> iterateur();
6  }
```

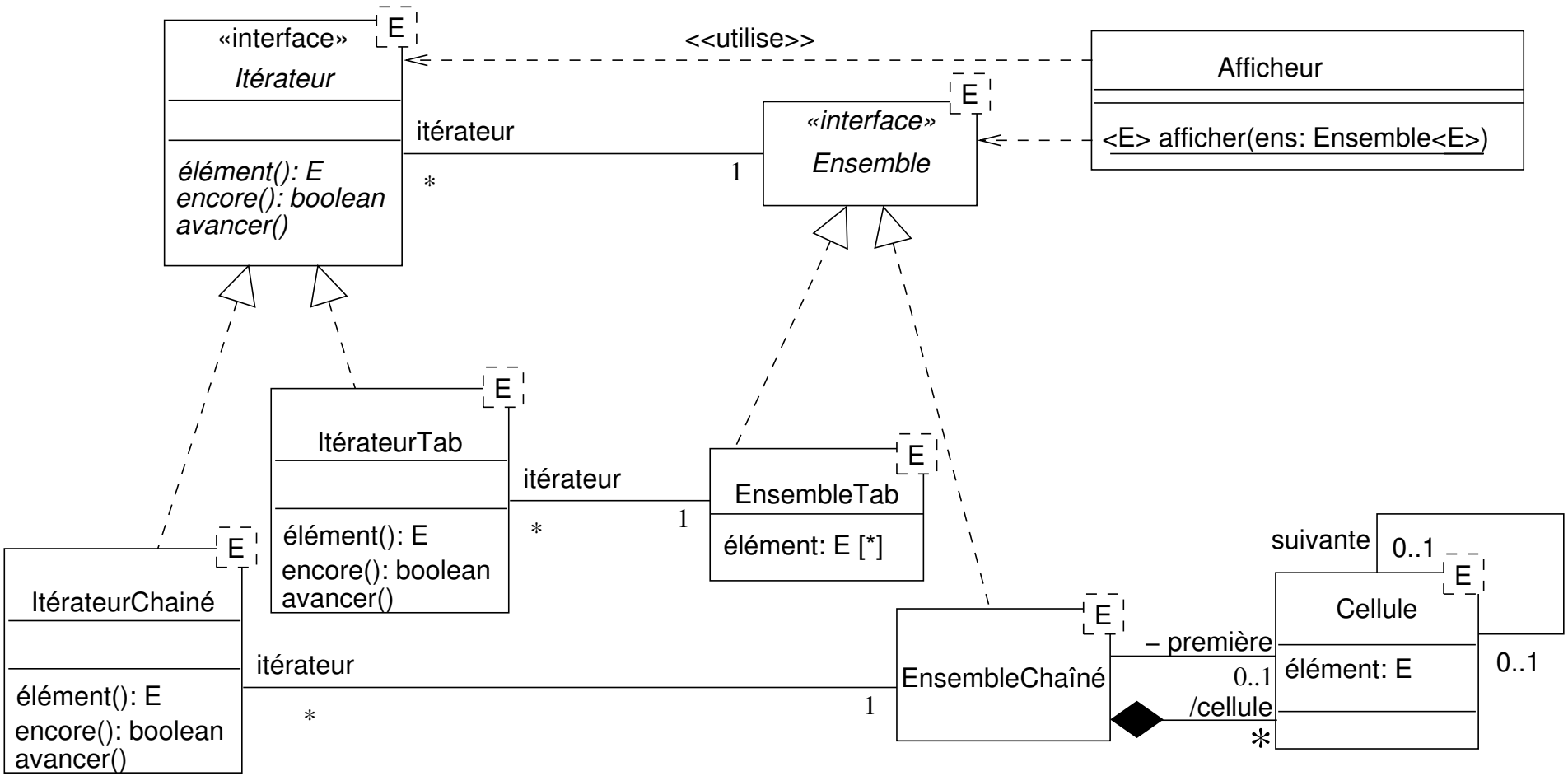
Utilisation de l'itérateur

```
1  public class Afficheur {
2
3      public static <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  }   }   }
```

qui correspond bien au raffinage :

```
1  public static <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  }   }
```

Architecture de la solution



- On définit une réalisation de `Iterateur` par réalisation de l'interface `Ensemble`.

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();
} } }
```

```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.

Du parcours dans EnsembleChaine à l'itérateur

```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();
} } }
```

```java
// pacourir dans EnsembleChaine
Cellule<E> curseur = ens.premiere;
while (curseur != null) {
  faireQuelqueChose(curseur.element);
  curseur = curseur.suivante;
}
```

```java
// parcourir avec l'itérateur
Iterateur<E> it = ens.iterateur();
while (it.encore()) {
  faireQuelqueChose(it.element());
  it.avancer();
}
```

Comparer les deux codes à droite puis, l'implantation des méthodes à gauche.
Le code des méthodes de IterateurChaine correspond au code d'un parcours classique.

Déterminer s'il y a des multiples : 2 parcours du même ensemble

- On peut utiliser simultanément plusieurs itérateurs !

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

- On tolère ici le **return** dans la boucle (éviter des conditions compliquées, code court)
- Utiliser un **foreach** serait plus judicieux (voir T. 58)

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) ?

- On ne peut plus garantir de parcourir une et une seule fois chaque élément !
- Conséquence : le considérer comme une erreur de programmation
- Idée : le détecter au plus tôt et le signaler (ConcurrentModificationException), d'où *fail-fast*

Solution : gérer un 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, l'exception `ConcurrentModificationException` est levée !

Question : Comment l'itérateur accéde-t-il à ce compteur ?

- L'itérateur doit avoir accès à l'ensemble et à son attribut privé compteur
- Mettre le compteur **`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 (statique, membre ou anonyme)

Itérateur pour ensemble chaîné : avec classe interne statique

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

Itérateur pour ensemble chaîné : avec classe interne statique (2)

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

- Une classe interne a accès à ce qui est privé de la classe englobante
- Une classe interne a un droit d'accès
  - Ici, privée car l'utilisateur de l'ensemble n'a pas à connaître l'itérateur concret !
- 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 Cellule<E> premiere;    // accès à la première cellule
3      private long nbModifications;   // nb de modifications faites sur la liste
4
5      @Override public Iterateur<E> iterateur() {
6          return new IterateurChaine();
7      }
8
9      private class IterateurChaine implements Iterateur<E> {
10         private long nbModifsAttendu;
11         private Cellule<E> curseur;
12
13         public IterateurChaine() {
14             this.curseur = premiere;
15             this.nbModifsAttendu = nbModifications;
16         }
17
18         @Override public boolean encore() {
19             this.checkModifications();
20             return this.curseur != null;
21         }
22
23         @Override public E element() {
24             this.checkModifications();
25             this.checkEncore();
26             return this.curseur.element;
27         }
```

Itérateur pour ensemble chaîné : avec classe interne membre (2)

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

- Le mot-clé **static** devant la classe interne a été supprimé.

$\Longrightarrow$ Un objet de la classe interne a accès à l'objet de la classe englobante qui l'a créé.

- premiere, nbModifications (attributs de EnsembleChaine) accessibles depuis l'itérateur
- Conseil : Ne pas mettre **this** (sauf si nécessaire) !

  this.premiere est une erreur, il faut écrire EnsembleChaine.**this**.premiere
- Pourquoi nommer la classe ?

Itérateur pour ensemble chaîné

avec classe anonyme

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

Itérateur pour ensemble chaîné (2)

avec classe anonyme

```
27              private void checkEncore() {
28                  if (! this.encore()) {
29                      throw new java.util.NoSuchElementException();
30              }   }
31
32              private void checkModifications() {
33                  if (nbModifications != this.nbModifsAttendu) {
34                      throw new java.util.ConcurrentModificationException();
35              }   }
36          };
37  }   }
```

- 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** (ou aurait pu être déclarés **final** depuis Java8)
- Pas forcément lisible, surtout si la classe anonyme est longue.
- **Remarque :** Il existe aussi des classes locales à des méthodes ou constructeurs : mêmes propriétés que les classes anonymes mais elles ont un nom.

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      class M {   // classe interne membre
12          private int y = 7;
13          void m() {
14              System.out.println(x);          // 4    x de E
15              System.out.println(y);          // 7    y de M
16              System.out.println(E.this.y);   // 12   y de E
17  }   }   }
18
19  public class ExempleClasseInternes {
20      public static void main(String[] args) {
21          E.S s = new E.S();  // OK
22          // E.M m1 = new E.M();  // Erreur !
23              // error: an enclosing instance that contains E.M is required
24          E e = new E();
25          E.M m2 = e.new M(); // OK : m2 est lié à e
26          m2.m(); // dans m() : this == m2, E.this == e !
27  }   }
```

Classes internes : les principes sur un exemple, explications

La classe englobante E contient deux classes internes, l'une de classe S, l'autre d'instance M.

La classe interne S est de classe (**static**), donc :

- Son nom est E.S et on peut faire **new** E.S() pour en créer un objet.
- Elle ne peut pas référencer les membres d'instance de la classe englobante.
  - C'est le même principe que pour les méthodes de classe.
- Similaire à une classe externe, sauf qu'elle a accès aux membres de classe privés de E

La classe interne M est d'instance (pas de **static**), donc :

- On ne peut pas faire **new** E.M() car, étant d'instance, ses objets doivent être associés à un objet E.
- On fait donc e.**new** M() avec e un objet de type E.
  Ainsi l'objet de type M est associé à l'objet associé à e et aura accès à ses membres d'instance.
- Similaire à une classe externe, à la quelle on aurait fourni un objet de type E (avec en plus l'accès aux éléments privés sur cet objet).

```
1  class M {
2      private E e;
3      private int y = 12;
4      public M(E e)        { this.e = e; }
5      void m() {
6          ... y ... // le y de M => 12
7          ... e.y ... // le y de e => 7
8  }    }
```

Comment faire pour modifier l'ensemble pendant un parcours ?

Exercice 1 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

```
Exception in thread "main" java.util.ConcurrentModificationException
    at EnsembleChaine$IterateurChaine.checkModifications(EnsembleChaine.java:103)
    at EnsembleChaine$IterateurChaine.avancer(EnsembleChaine.java:90)
    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 le dernier élément retourné par `element()`
  - elle utilise le supprimer de l'ensemble et met à jour sa copie du compteur de modifications

```
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

Exercice 2 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 !
- Il faut juste avoir accès à un itérateur.
- On le spécifie dans une interface `Iterable` (abstraction).

```
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  }
```

- L'interface `Ensemble` hérite alors de `Iterable`.
- Plus généralement, les structures de données sont sous-types de cette 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      boolean hasNext();
5
6      /** Returns the next element in the iteration.
7       * @return the next element in the iteration.
8       * @exception NoSuchElementException iteration has no more elements.  */
9      E next();
10
11     /** Removes from the underlying collection the last element returned by the
12      * iterator (optional operation).  This method can be called only once per
13      * call to <tt>next</tt>.  The behavior of an iterator is unspecified if
14      * the underlying collection is modified while the iteration is in
15      * progress in any way other than by calling this method.
16      *
17      * @exception UnsupportedOperationException if the <tt>remove</tt>
18      *        operation is not supported by this Iterator.
19
20      * @exception IllegalStateException if the <tt>next</tt> method has not
21      *        yet been called, or the <tt>remove</tt> method has already
22      *        been called after the last call to the <tt>next</tt> method.  */
23     void remove();
24 }
```

- Seulement 3 opérations car `next()` fait à la fois `element()` et `avancer()`.
- L'opération `remove()` peut ne pas être définie (`UnsupportedOperationException`).

Iterable et foreach

- L'interface `java.lang.Iterable<E>` spécifie l'accès à un itérateur.

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

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 (avec les API Java)

```
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         System.out.println("ens = " + ens);
17 }   }
```

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

Sommaire

- Généricité et sous-typage
- Type Joker
- Effacement de type

Généricité et sous-typage

Exercice 3

Y a-t-il une relation de sous-typage entre `EnsembleTab<Object>` et `EnsembleTab<String>` ?
Si oui, dans quel sens ?

Généricité et sous-typage

La réponse est NON car sinon on arriverait à une incohérence. La preuve :

```
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  }   }
```

Ici, on pourrait donc mettre un nombre réel (`Double`) dans une liste de chaînes (`String`).

Mais le compilateur signale une erreur... qui démontre qu'il n'y a pas sous-typage :

```
TestErreurGeneriticiteSousTypage.java:4: error: incompatible types:
     EnsembleTab<String> cannot be converted to EnsembleTab<Object>
        EnsembleTab<Object> lo = ls;
                                 ^
1 error
```

Conclusion : Même si B est sous-type de A, C<B> n'est pas sous-type de C<A>.

Cas des tableaux

```
1  public class ExempleSousTypageTableau {
2      public static void main(String[] args) {
3          String[] ts = new String[10];
4          Object[] to = ts;
5          to[0] = "texte";
6          to[1] = 15.5;   // en fait new Double(15.5);
7          String s = ts[0];
8          System.out.println(s);
9  }   }
```

Pas d'erreur de compilation ! Surprenant et peu logique d'après T. 61.

Résultat de l'exécution :

```
Exception in thread "main" java.lang.ArrayStoreException: java.lang.Double
    at ExempleSousTypageTableau.main(ExempleSousTypageTableau.java:6)
```

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`)

Exercice 4 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  }   }
```

```
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 }
```

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

4.2. 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. 61.
- Une solution consiste 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  }   }
```

- 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  }   }
```

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

- Le compilateur interdit d'ajouter un objet dans une liste d'inconnus !
  - Normal : le type n'est pas connu, donc aucune garantie que c'est possible
  - Exemple : OK :`copier(new EnsembleTab<Object>(), new Object[] {"texte", 15.5});`
    KO :`copier(new EnsembleTab<String>(), new Object[] {"texte", 15.5});`
- La solution consiste à expliciter le paramètre de généricité :

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

  - OK (T = String) : `copier(new EnsembleTab<String>(), new String[] { "un", "deux"});`
  - OK (T = Object) : `copier(new EnsembleTab<Object>(), new String[] { "un", "deux"});`
  - Erreur : `copier(new EnsembleTab<String>(), new Object[] { "texte", 15.5 });`
    car T = String imposé par le premier paramètre et Object[] n'est pas sous-type de String[] !

Exercice 5 Écrire une méthode qui ajoute dans un ensemble les éléments d'un autre.

Ajouter dans un ensemble les éléments d'un autre

Première solution ?

```
1  public static <T> void ajouterTous(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  }   }
```

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

```
ajouterTous(new EnsembleTab<Point>(), new EnsembleTab<PointNomme>())
```

Deuxième solution ?

```
1  public static <T1, T2>
2  void ajouterTous(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  }   }
```

Et non !

Ajouter dans un ensemble les éléments d'un autre (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 ajouterTous(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  }   }
```

Pourquoi deux types ?

```
1  public static <T>
2  void ajouterTous(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  }   }
```

Exemple : `<Point>ajouterTous(new EnsembleTab<Object>(), new EnsembleTab<PointNomme>())`

- **? extends T :** Le type joker est tout sous-type de T : T est la **borne supérieure**
- **? super T :** Le type joker est tout super-type de T : T est la **borne inférieure**
- Ici mettre **extends** ou **super** suffit, sauf si un autre paramètre imposait une valeur à T

Effacement de type

Question : Que se passe-t-il si dans une même application on utilise `EnsembleTab<String>`, `EnsembleTab<Integer>`, `EnsembleTab<Object>`...?

- A-t-on plusieurs .class : `EnsembleTab<String>.class`, `EnsembleTab<Integer>.class`...?
- Non, il n'y en a qu'un seul : `EnsembleTab.class`.

Effacement de type : Les paramètres de généricité ne sont connus que du compilateur et n'apparaissent plus dans le code intermédiaire. C'est l'*effacement de type* (*type erasure*).

Conséquence : Pour un paramètre de généricité T, les phrases suivantes sont interdites et provoquent une erreur de compilation :

```
1  p instanceof T
2  new T()
3  new T[10]
4  class A extends T { ... }
5  class A implements T { ... }
```

Type brute (*raw type*)

Définition : On appelle **type brute** (*raw type*) une classe (ou une interface) générique pour laquelle on ne précise pas la valeur des paramètres de généricité.

```
1  Ensemble<String> ens1 = new EnsembleTab<>();    // types complets : E = String
2  Ensemble ens2 = new EnsembleTab();      // types brutes : E = Objet
```

Remarques :

1. La valeur d'un paramètre de généricité pour un type brute est sa borne supérieure
   - Pour Ensemble, la valeur de E est Object
   - Pour EnsembleOrdonne, la valeur de E serait Comparable car généricité contrainte : `class EnsembleOrdonne<E extends Comparable<E>> ...`
2. Les types brutes existent pour des raisons de compatibilité :
   - Avant Java5 et la généricité, on utilisait le sous-typage pour avoir une classe « générique »
   - On avait donc un Ensemble d'Object
   - On pouvait ajouter n'importe quel objet (pas de contrôle de type !)
   - Lors de la récupération d'un élément, il fallait le transtyper
3. Éviter au maximum les types brutes pour que le compilateur puisse vérifier le programme !

Danger des types bruts

```
1  import java.util.*;
2  public class ExempleListRawType {
3     public static void main(String[] args) {
4        List ls = new ArrayList();  // Une liste de String
5        ls.add("ok ?");      // ajouter String : ok
6        System.out.println("ls = " + ls);
7        ls.add(new Date());  // ajouter Date : Erreur !
8        System.out.println("ls = " + ls);
9
10       for (int i = 0; i < ls.size(); i++) {  // Afficher la liste
11          System.out.println(i + " --> " + ls.get(i));
12       }
13       String deuxieme = (String) ls.get(1);  // Récupérer le deuxième élément
14 }  }
```

```
1  ls = [ok ?]
2  ls = [ok ?, Sun Feb 25 18:01:06 CET 2024]
3  0 --> ok ?
4  1 --> Sun Feb 25 18:01:06 CET 2024
5  Exception in thread "main" java.lang.ClassCastException: class java.util.Date cannot
       be cast to class java.lang.String (java.util.Date and java.lang.String are in
       module java.base of loader 'bootstrap')
6      at ExempleListRawType.main(ExempleListRawType.java:13)
```

- Exception levée ligne 13 (récupération de l'élément) alors que l'erreur est ligne 7 (ajout).

Explications

- `ls` devrait représenter une liste de String. Seul le commentaire permet de le savoir !
- List et ArrayList : types brutes ⇒ E = Object ⇒ **pas de contrôle de type !**
  - On peut ajouter dans `ls` des String, mais aussi des Date, en fait tout Object !
- Si on veut récupérer un élément en tant que String, il faut transtyper !
  - C'est à ce moment qu'une erreur pourrait être détectée (ici Date au lieu de String)
  - L'erreur est en fait lors de l'ajout de la Date dans la liste (liste de String !)
  - Conclusion : l'erreur est détectée loin de son origine !
- Le compilateur nous a averti (avec les détails si on utilise l'option indiquée) :

```
1  Note: ExempleListRawType.java uses unchecked or unsafe operations.
2  Note: Recompile with -Xlint:unchecked for details.
```

```
1  ExempleListRawType.java:5: warning: [unchecked] unchecked call to add(E) as a member of the raw type List
2          ls.add("ok ?");     // ajouter String : ok
3                 ^
4    where E is a type-variable:
5      E extends Object declared in interface List
6  ExempleListRawType.java:7: warning: [unchecked] unchecked call to add(E) as a member of the raw type List
7          ls.add(new Date()); // ajouter Date : Erreur !
8                 ^
9    where E is a type-variable:
10     E extends Object declared in interface List
11 2 warnings
```

Généricité = Contrôle de type = Erreur signalée par le compilateur

```
1  import java.util.*;
2  public class ExempleList {
3     public static void main(String[] args) {
4        List<String> ls = new ArrayList<String>();  // Une liste de String (pléonasme !)
5        ls.add("ok ?");      // ajouter String : ok
6        System.out.println("ls = " + ls);
7        ls.add(new Date());  // ajouter Date : Erreur !
8        System.out.println("ls = " + ls);
9
10       for (int i = 0; i < ls.size(); i++) {  // Afficher la liste
11          System.out.println(i + " --> " + ls.get(i));
12       }
13       String deuxieme = ls.get(1);  // Récupérer le deuxième élément
14 }  }
```

```
1  ExempleList.java:7: error: incompatible types: Date cannot be converted to String
2          ls.add(new Date()); // ajouter Date : Erreur !
3                 ^
4  Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
5  1 error
```

- Le compilateur signale bien l'erreur sur la ligne 7.

Évitez de mélanger types brutes et génériques

```
1  import java.util.*;
2  public class ExempleListDanger {
3     public static void main(String[] args) {
4        List<String> ls = new ArrayList<String>();  // Une liste de String (pléonasme !)
5        ls.add("ok ?");      // ajouter String : ok
6        System.out.println("ls = " + ls);
7        List l = ls;   // perte de l'information de type
8        l.add(new Date());   // ajouter Date : Erreur !
9        System.out.println("ls = " + ls);
10
11       for (int i = 0; i < ls.size(); i++) {  // Afficher la liste
12          System.out.println(i + " --> " + ls.get(i));
13       }
14       String deuxieme = ls.get(1);  // Récupérer le deuxième élément
15 }  }
```

```
1  ls = [ok ?]
2  ls = [ok ?, Sun Feb 25 18:01:09 CET 2024]
3  0 --> ok ?
4  Exception in thread "main" java.lang.ClassCastException: class java.util.Date cannot
       be cast to class java.lang.String (java.util.Date and java.lang.String are in
       module java.base of loader 'bootstrap')
5      at ExempleListDanger.main(ExempleListDanger.java:12)
```

Explications

- Dès qu'on passe par un type brute, l'information de généricité est perdue
- Le compilateur ne pourra plus faire de contrôle de type
  - Pas d'erreur sur l'ajout de la date !
- Mais il le signale explicitement par un avertissement

```
1  Note: ExempleListDanger.java uses unchecked or unsafe operations.
2  Note: Recompile with -Xlint:unchecked for details.
```

```
1  ExempleListDanger.java:8: warning: [unchecked] unchecked call to add(E) as a member
       of the raw type List
2          l.add(new Date());  // ajouter Date : Erreur !
3               ^
4    where E is a type-variable:
5      E extends Object declared in interface List
6  1 warning
```

- Le compilateur ajoute **toujours** un transtypage quand on récupère un objet d'un type générique
  - D'où l'exception signalée dans le **for**
  - Si on avait utilisé `l.get(i)`, il n'y aurait pas eu d'erreur dans le **for**
  - mais elle aurait eu lieu lors de l'initialisation de deuxieme !

Comment détecter l'erreur dès l'ajout dans la liste ?

Utiliser Collections.checkedList qui vérifie que les éléments ajoutés sont du type attendu (T. 104 et 118) !

```
1  import java.util.*;
2  public class ExempleCheckedListDanger {
3     public static void main(String[] args) {
4        List<String> ls = Collections.checkedList(new ArrayList<String>(), String.class);
5        ls.add("ok ?");      // ajouter String : ok
6        System.out.println("ls = " + ls);
7        List l = ls;   // perte de l'information de type
8        l.add(new Date());   // ajouter Date : Erreur !
9        System.out.println("ls = " + ls);
10
11       for (int i = 0; i < ls.size(); i++) {  // Afficher la liste
12          System.out.println(i + " --> " + ls.get(i));
13       }
14       String deuxieme = ls.get(1);  // Récupérer le deuxième élément
15 }  }
```

```
1  ls = [ok ?]
2  Exception in thread "main" java.lang.ClassCastException: Attempt to insert class java
       .util.Date element into collection with element type class java.lang.String
3      at java.base/java.util.Collections$CheckedCollection.typeCheck(Collections.java
           :3049)
4      at java.base/java.util.Collections$CheckedCollection.add(Collections.java:3097)
5      at ExempleCheckedListDanger.main(ExempleCheckedListDanger.java:8)
```

Que peut vérifier le compilateur concernant la généricité ?

Si le compilateur connait les types, il peut vérifier les affectations et les transtypages !

```
1  import java.util.*;
2  public class TypesConnus {
3      public static void main(String[] args) {
4          List<String> ls = new ArrayList<>();
5          List<String> l = ls;
6          List<Integer> li1 = l;
7          List<Integer> li2 = (List<Integer>) l;
8  }   }
```

```
1  TypesConnus.java:6: error: incompatible types: List<String> cannot be converted to
       List<Integer>
2          List<Integer> li1 = l;
3                              ^
4  TypesConnus.java:7: error: incompatible types: List<String> cannot be converted to
       List<Integer>
5          List<Integer> li2 = (List<Integer>) l;
6                                              ^
7  2 errors
```

Que peut vérifier le compilateur concernant la généricité ? (2)

Si le compilateur ne connait pas la valeur des paramètres de généricité, il ne peut plus contrôler les types et le signale par un avertissement.

```
1  import java.util.*;
2  public class TypesInconnus {
3      public static void main(String[] args) {
4          List<String> ls = new ArrayList<>();
5
6          List l = ls;    // perte de la valeur du paramètre de généricité E
7          List<Integer> li1 = l;                  // unchecked conversion
8          List<Integer> li2 = (List<Integer>) l;  // unchecked cast
9
10         Object o = ls;  // perte du type réel et donc de la valeur de E
11         List<Integer> li3 = (List<Integer>) o;  // unchecked cast
12         List<Integer> li4 = (List) o;           // unchecked conversion
13 }   }
```

Que peut vérifier le compilateur concernant la généricité ? (3)

```
1  TypesInconnus.java:7: warning: [unchecked] unchecked conversion
2          List<Integer> li1 = l;                  // unchecked conversion
3                              ^
4    required: List<Integer>
5    found:    List
6  TypesInconnus.java:8: warning: [unchecked] unchecked cast
7          List<Integer> li2 = (List<Integer>) l;  // unchecked cast
8                                                  ^
9    required: List<Integer>
10   found:    List
11 TypesInconnus.java:11: warning: [unchecked] unchecked cast
12         List<Integer> li3 = (List<Integer>) o;  // unchecked cast
13                                                 ^
14   required: List<Integer>
15   found:    Object
16 TypesInconnus.java:12: warning: [unchecked] unchecked conversion
17         List<Integer> li4 = (List) o;           // unchecked conversion
18                             ^
19   required: List<Integer>
20   found:    List
21 4 warnings
```

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

Exercice 7 Proposer une réalisation de file en utilisant des structures chaînées. Estimer le temps d'exécution de chaque opération.

Exercice 8 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. 26)

- 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. 28)

- 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).

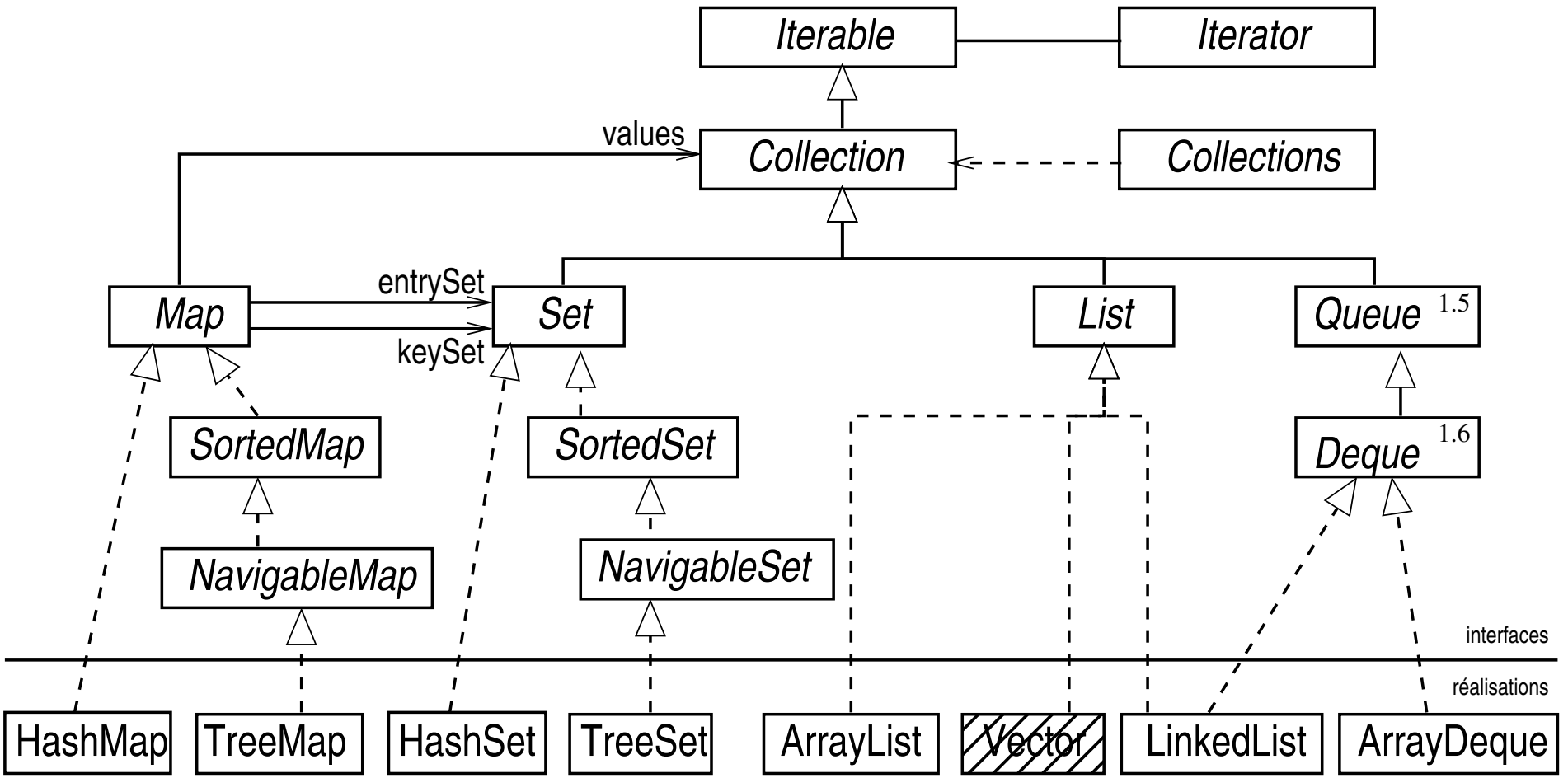
Principaux constituants :

- **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 structures de données 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 est le super-type des structures de données de Java (sauf Map)

Les collections sont **paramétrées par E**, type des éléments (depuis Java5)

Une collection autorise ou non plusieurs **occurrences** d'un même élément (List vs Set)

Les éléments peuvent être **désignés par une position** ou non (List vs Set)

Les éléments peuvent être **ordonnés** ou non (Set vs SortedSet)

Pour **limiter le nombre d'interfaces**, plusieurs choix ont été faits :

- Les méthodes de modifications ne sont pas implantées sur une collection immuables
  - ⇒ Elles lèvent l'exception : `UnsupportedOperationException`
- Certaines collections peuvent interdire l'élément **`null`**
  - ⇒ Les méthodes d'ajout lèvent alors l'exception `NullPointerException`
- Certaines collections peuvent contrôler le type des éléments ajoutés
  - ⇒ Les méthodes d'ajout lèvent alors l'exception `ClassCastException`

Règle : 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>* : Opérations

```
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 cette collection
boolean contains(Object o)    // est-ce que la collection contient o ?
boolean containsAll(Collection<?>       //  ''  ''  tous les éléments de c ?
boolean isEmpty()                       // vrai ssi la collection est vide
Iterator<E> iterator()                  // un itérateur sur la collection
boolean remove(Object o)                // supprimer l'objet o (collection changée ?)
boolean removeAll(Collection<?> c) // supprimer de this tous les éléments de c
boolean retainAll(Collection<?> c) // ne conserver que les éléments aussi dans 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
```

- le booléen des opérations de modification permet de savoir si la collection a changé
- pourquoi `contains(Object)` et pas `contains(E)` ? Précondition plus faible !
- toArray(T[]) : choisir le type du tableau dans lequel récupérer les éléments de la collection

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

Structure de données où chaque élément peut être identifié par sa position.

Ajoute des opérations relatives à la position (et en précise d'autres) :

```
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 Java5
- 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

$\Longrightarrow$ L'utilisateur peut choisir son style de programmation (conditions ou exceptions)

- 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 getOrDefault(Object k, V vDefault)    // retourne vDefault si k absent
V putIfAbsent(K key, V vDefault)    // associe vDefault à key, si key absente
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 ?

// les relations vers les collections dans le diagramme de classe
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 }   }   }
```

Remarque : Expliciter le type de e dans le dernier **for** est lourd et nuit à la lisibilité. On peut le remplacer par le mot-clé var (java10/JEP286) :

```
for (var e : occ.entrySet()) {
    System.out.println(e.getKey() + " --> " + e.getValue());
}
```

Exemple d'utilisation des Map (exécution)

```
1  > java -ea CompteNbOccurrences A B UN C A C D dix E dix A A UN D E
2  occ = {A=4, B=1, dix=2, C=2, D=2, E=2, UN=2}
3  clés = [A, B, dix, C, D, E, UN]
4  valeurs = [4, 1, 2, 2, 2, 2, 2]
5  entrées = [A=4, B=1, dix=2, C=2, D=2, E=2, UN=2]
6  A --> 4
7  B --> 1
8  dix --> 2
9  C --> 2
10 D --> 2
11 E --> 2
12 UN --> 2
```

Question : Comment afficher les chaînes dans l'ordre lexicographique ?

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 (TreeSet, TreeMap, etc.) s'appuient :

- soit sur un comparateur explicite fourni en paramètre du constructeur
- soit, à défaut, sur l'ordre naturel du type des éléments

Si ni l'un ni l'autre n'existe, il y aura une erreur à l'exécution !

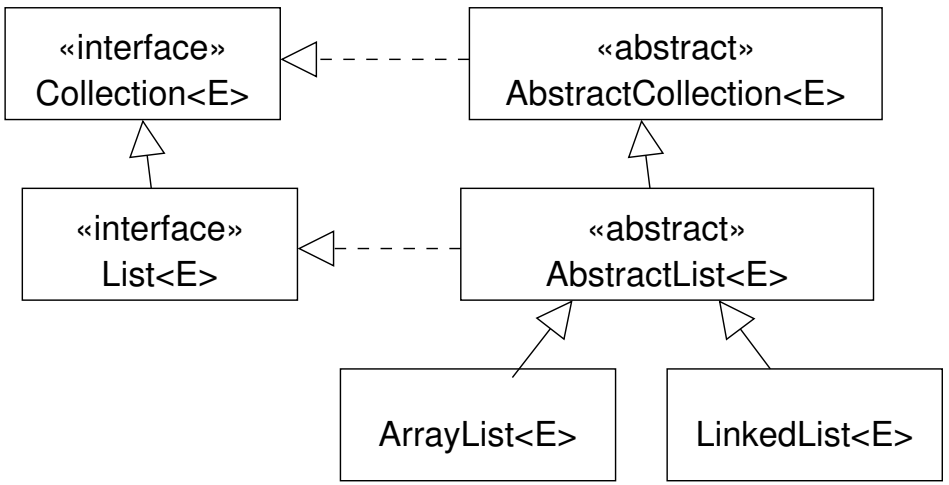
Principales réalisations

Interface	**Réalisations**				
	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 réalisations.
  $\Longrightarrow$ 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 (voir T. 112)
  - *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 (nécessitent une relation d'ordre).

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

Sommaire

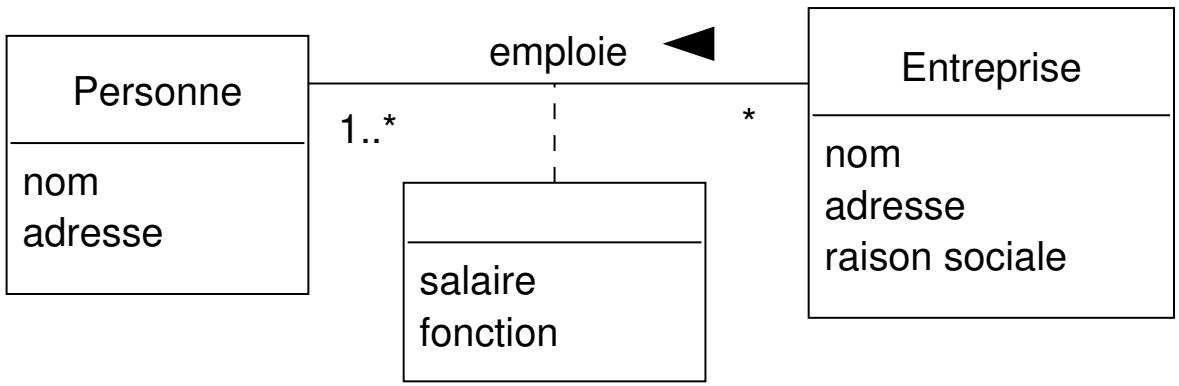
UML : Choix de la structure de données adaptée

Le diagramme UML peut être complété pour préciser la structure de données adaptée grâce aux contraintes placées aux extrémités des relations de délégation :

- 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

UML : Attribut et classe d'association

Définition : Un attribut d'association est un attribut qui caractérise la relation et pas seulement une de ses classes extrémités.



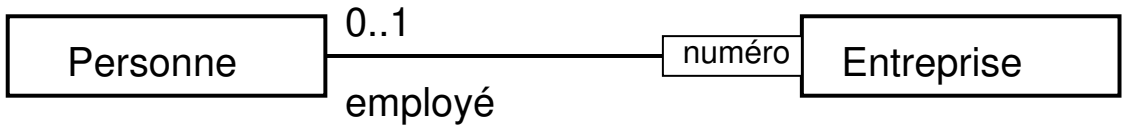
Remarque : Dans le cas d'une multiplicité 1, il est possible *mais non souhaitable* d'attacher l'attribut d'association à la classe (salaire sur Personne si elle ne peut travailler que dans une seule entreprise).

Remarque : Les attributs d'association peuvent être promus au rang de classe. Ici, une classe Poste, avec des méthodes telles que travailler, etc.

Traduction en code : Plusieurs possibilités : dans une classe, dans l'autre, dans les deux, en réifiant la relation, etc.

UML : Qualificatif

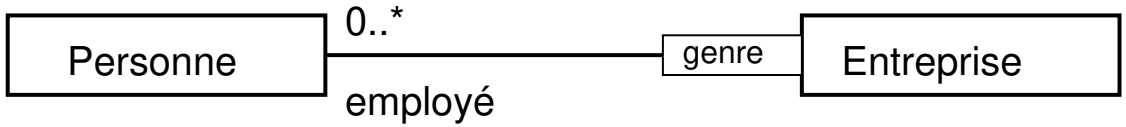
Un *qualificatif* est un attribut spécial placé sur une extrémité d'une relation.



Le qualificatif « numéro » est une clé de la classe Entreprise qui permet d'atteindre un jeu d'objet Personne (ici au plus un).

Intérêt : La qualification améliore la précision sémantique de la relation :

- elle *réduit la multiplicité* effective ($*$ est transformée en `0..1`, `numéro` est ici équivalent à une clé au sens base de données)
  Le qualificatif *genre* (femme ou homme) réduit la multiplicité qui reste $*$



- elle *améliore la navigation* dans le réseau des objets (pour désigner une personne, il suffit d'avoir son numéro)

Remarque : Ceci permet de montrer que le numéro de la personne dans l'entreprise est unique. Comment l'indiquer si numéro est un attribut de Personne ou de la relation ?

Traduction en code : Un qualificatif UML peut se traduire par un tableau associatif (Map)

Sommaire

- Interface de marquage
- Adaptateur
- Procuration (Proxy)

Efficacité de List.get(int)

Exercice 9 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

Solution : Utiliser une interface de marquage.

Principe : Définir une propriété sémantique booléenne sur une classe

Application :

- L'interface `RandomAccess` définit cette propriété (Java4).
- `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

Utilisation de RandomAccess : Collections.copy (spécification)

```
1  /**                 ------ from java/util/Collections.java (Java6) ------
2   * Copies all of the elements from one list into another.  After the
3   * operation, the index of each copied element in the destination list
4   * will be identical to its index in the source list.  The destination
5   * list must be at least as long as the source list.  If it is longer, the
6   * remaining elements in the destination list are unaffected. <p>
7   *
8   * This method runs in linear time.
9   *
10  * @param  dest The destination list.
11  * @param  src The source list.
12  * @throws IndexOutOfBoundsException if the destination list is too small
13  *         to contain the entire source List.
14  * @throws UnsupportedOperationException if the destination list's
15  *         list-iterator does not support the <tt>set</tt> operation.
16  */
17 public static <T> void copy(List<? super T> dest, List<? extends T> src) {
```

- Utilisation de **extends** et **super** pour les paramètres de généricité des deux listes
  - même si mettre les deux n'est pas nécessaire !
- Indication de la complexité dans le commentaire (linéaire)
- Mention des exceptions qui peuvent se produire (même si elles sont non vérifiées !)
- Utilisation de <p> (nouveau paragraphe en HTML)

Utilisation de RandomAccess : Collections.copy (implantation)

```
1   public static <T> void copy(List<? super T> dest, List<? extends T> src) {
2       int srcSize = src.size();
3       if (srcSize > dest.size())
4           throw new IndexOutOfBoundsException("Source does not fit in dest");
5
6       if (srcSize < COPY_THRESHOLD ||
7               (src instanceof RandomAccess && dest instanceof RandomAccess)) {
8           for (int i=0; i<srcSize; i++)
9               dest.set(i, src.get(i));
10      } else {
11          ListIterator<? super T> di=dest.listIterator();
12          ListIterator<? extends T> si=src.listIterator();
13          for (int i=0; i<srcSize; i++) {
14              di.next();
15              di.set(si.next());
16  }   }   }
17
18  private static final int COPY_THRESHOLD = 10;
```

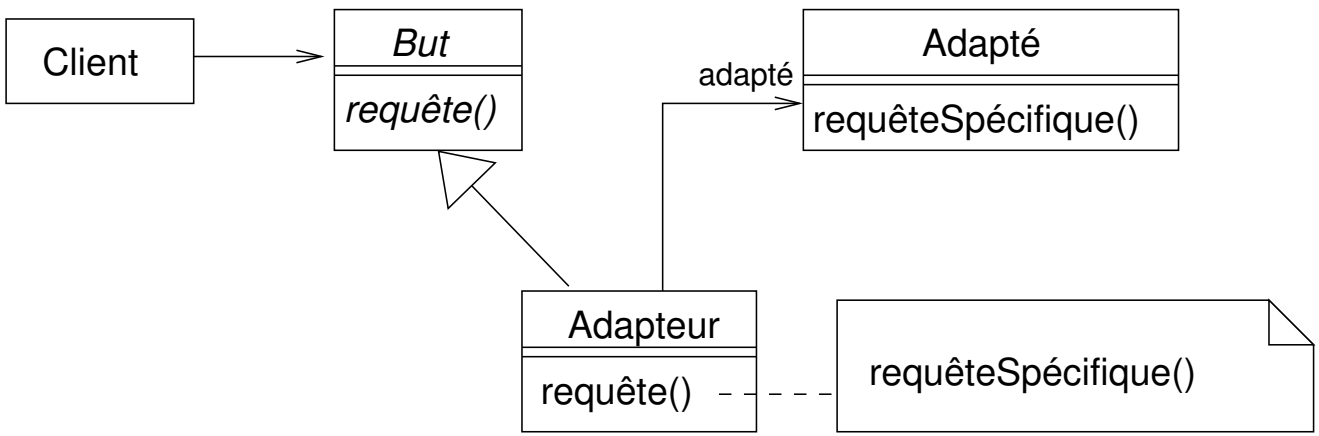
- L.7 : Test explicite que les listes sont RandomAccess ⇒ Utilisation de List.get et List.set (L.9)
- L.6 : Taille de la liste petite ⇒ Utilisation de List.get et List.set (compromis coût itérateur)
- L.10 : Utilisation d'un itérateur (pas de RamdomAccess et grande taille)
- L.4 : Détection explicite de `IndexOutOfBoundsException` (pas de **`else`**, saut de ligne)
  - Principe : soit on fait le traitement attendu complètement, soit on ne fait rien !
  - `UnsupportedOperationException` levée dès le premier appel à `set` (L.9 ou 15) : principe respecté
  - Remarque : le set de Iterator utilise celui de List
- L.2 Variable locale pour éviter d'accéder plusieurs fois `src.size()` (L.3, 6, 8, 13)

Utiliser un tableau là où une liste est attendue

Exercice 10 : 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

L'adaptateur ArrayList (classe interne de Arrays) et la méthode asList

```
1  private static class ArrayList<E> extends AbstractList<E>   // extrait source Java6
2      implements RandomAccess, java.io.Serializable
3  {
4      private final E[] a;
5      ArrayList(E[] array) {
6          if (array==null)
7              throw new NullPointerException();
8          a = array;
9      }
10     public int size()       { return a.length; }
11     public E get(int index) { return a[index]; }
12     public E set(int index, E element) {
13         E oldValue = a[index];
14         a[index] = element;
15         return oldValue;
16     }
17     ...
18 }
19 public static <T> List<T> asList(T... a) {
20     return new ArrayList<T>(a);
21 }
```

- Attention : ne pas confondre java.util.Arrays.ArrayList et java.util.ArrayList !
- Noter la réalisation de RandomAccess !

Exemple

```
1   import java.util.*;
2   public class SortTableau {
3       public static void main(String[] args) {
4           List<Integer> l = new ArrayList<Integer>();
5           Collections.addAll(l, 2, 4, 1, 8, 9, 3, 1);
6           System.out.println("l = " + l);
7           Collections.sort(l);
8           System.out.println("l = " + l);
9
10          Integer tab[] = { 2, 4, 1, 8, 9, 3, 1 };
11          List<Integer> lt = Arrays.asList(tab);
12          System.out.println("tab = " + Arrays.toString(tab));
13          Collections.sort(lt);
14          System.out.println("tab = " + Arrays.toString(tab));
15          System.out.println("La classe de lt = " + lt.getClass());
16  }   }
```

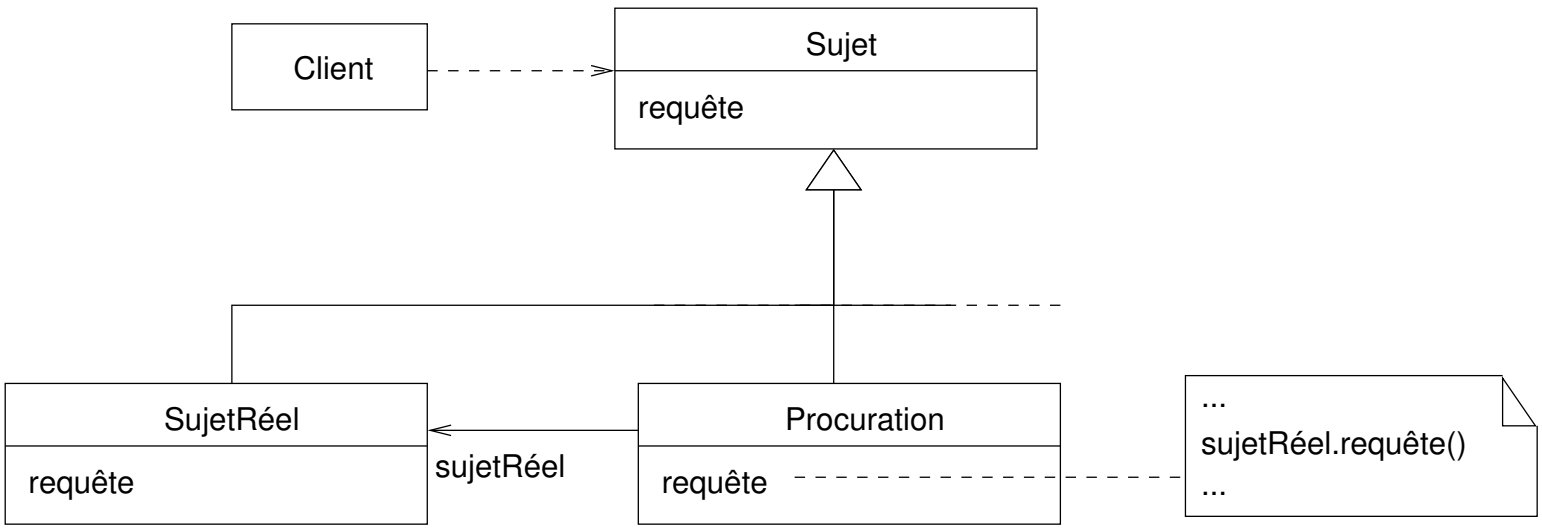
```
l = [2, 4, 1, 8, 9, 3, 1]
l = [1, 1, 2, 3, 4, 8, 9]
tab = [2, 4, 1, 8, 9, 3, 1]
tab = [1, 1, 2, 3, 4, 8, 9]
La classe de lt = class java.util.Arrays$ArrayList
```

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 11 : 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 Procuration ou Mandataire.



Le proxy UnmodifiableList (classe interne de Collections)

```
1  static class UnmodifiableList<E> extends UnmodifiableCollection<E> implements List<E> {
2      final List<? extends E> list;           // source Java 1.6
3
4      UnmodifiableList(List<? extends E> list) {
5          super(list);
6          this.list = list;
7      }
8
9      public boolean equals(Object o) {return o == this || list.equals(o);}
10     public int hashCode()       {return list.hashCode();}
11
12     public E get(int index) {return list.get(index);}
13     public E set(int index, E element) {
14         throw new UnsupportedOperationException();
15     }
16     public void add(int index, E element) {
17         throw new UnsupportedOperationException();
18     }
19     public E remove(int index) {
20         throw new UnsupportedOperationException();
21     }
22     public int indexOf(Object o)            {return list.indexOf(o);}
23     public int lastIndexOf(Object o)        {return list.lastIndexOf(o);}
24     ...
25 }
```

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 }   }
```

Exécution

```
l = [2, 4, 1, 8, 9, 3, 1]
taille = 7
premier = 2
rol = [4, 1, 8, 9, 3, 1]
Classe de rol : class java.util.Collections$UnmodifiableRandomAccessList
taille = 6
premier = 4
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.base/java.util.Collections$UnmodifiableList.remove(Collections.java:1318)
    at ProxyList.m1(ProxyList.java:6)
    at ProxyList.main(ProxyList.java:16)
```

UnmodifiableList ou UnmodifiableRandomAccessList ?

Ceci dépend si la liste fournie est elle-même un sous-type de RandomAccess ou non !

```
1   public static <T> List<T> unmodifiableList(List<? extends T> list) {
2       return (list instanceof RandomAccess ?
3                   new UnmodifiableRandomAccessList<T>(list) :
4                   new UnmodifiableList<T>(list));
5   }
6
7   ...
8
9   static class UnmodifiableRandomAccessList<E> extends UnmodifiableList<E>
10                                              implements RandomAccess
11  {
12      UnmodifiableRandomAccessList(List<? extends E> list) {
13          super(list);
14      }
15
16      public List<E> subList(int fromIndex, int toIndex) {
17          return new UnmodifiableRandomAccessList<E>(
18              list.subList(fromIndex, toIndex));
19  }   }
```

Le patron Proxy/Procuration

Intention :

Fournir à un client un mandaire (ou remplaçant) pour contrôler l'accès à un objet fournisseur

Indications d'utilisation :

- procuration à distance : représentant local d'un objet distant
  - pour les applications distribuées
  - faire un appel à distance comme un appel à distance
  - grâce à un proxy local qui gère (et abstrait) l'appel à distance
- procuration virtuelle : retarder l'allocation mémoire des ressources d'un objet jusqu'à son utilisation réelle
  - exemple : les images d'une application (ne pas ralentir le lancement de l'application)
  - cache proxy : stocker le résultat d'opérations coûteuses
- procuration de protection : contrôle les accès
  - L'exemple que l'on a vu
  - Mais aussi Collections.checkedList, etc.
- référence intelligente (*smart pointer*) : remplaçant d'un pointeur brut :
  - compteur de référence
  - Copy-on-write : retarder la copie d'un objet jusqu'à ce que ce soit nécessaire

  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