Chapitre 4

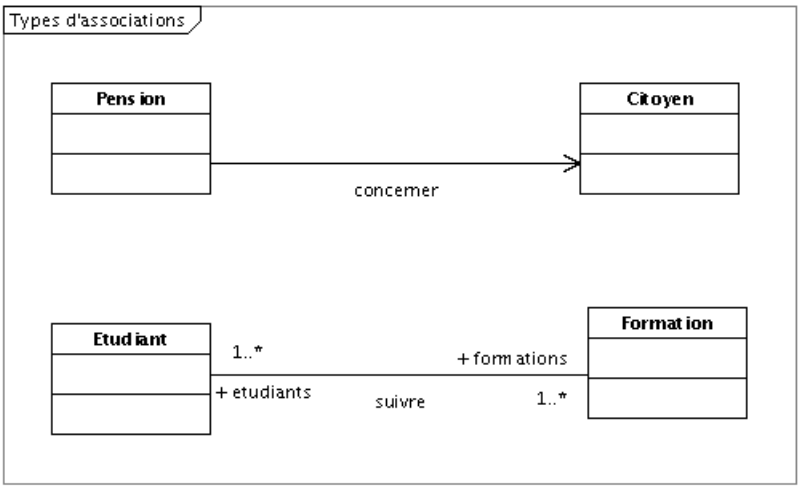
Tableaux et collections

Les associations entre classes définies dans un modèle correspondent à des variables dans le code des classes. On examine ici différents moyens de représenter ces associations :
– les tableaux, de taille fixe
– les collections, de taille variable, plus souples, mais plus complexes à mettre en œuvre. On s'attachera à décrire deux classes particulières de listes et d'ensembles.
Après quelques remarques sur la généricité, on abordera la création d'un type original de collection, utile pour modéliser le projet du chapitre précédent.

4.1 Associations, navigabilité et multiplicité

Nous avons vu précédemment comment deux classes pouvaient entrer en association. Nous devons spécifier les caractéristiques de l'association :

– sa **navigabilité** peut être unidirectionnelle ou bidirectionnelle. Cela nous permet de dire si un objet possédera un lien vers le ou les objets associés de l'autre classe.
– sa **multiplicité** doit être spécifiée sur chaque extrémité. Elle va avoir des conséquences directe sur la manière d'implémenter les liens entre les objets. Une multiplicité maximale de 1 permettra de représenter le lien à l'aide d'une simple variable d'instance. Une multiplicité maximale connue autorisera l'emploi d'un tableau, bien que ce ne soit pas la technique la plus simple. Une multiplicité maximale indéterminée forcera l'emploi d'une structure plus souple, comme une collection.



Dans les deux exemples ci-dessus, on constate que la navigabilité entre les pensions et les citoyens est unidirectionnelle. On peut trouver le titulaire d'une pension, mais il n'est pas

possible de savoir quelle(s) pension(s) touche un citoyen[1]. Par contre, la seconde association permet d'énumérer les formations suivies par un étudiant et les étudiants qui suivent une formation.

La classe Pension aura le squelette suivant :

```
public class Pension {

   public Citoyen citoyen;
 }
```

En contrepartie, la classe Citoyen ne possédera aucune référence à la classe Pension.

Voici ce que propose Poseidon comme implémentation de la classe Etudiant (celle de Formation sera ici rigoureusement symétrique) :

```
public class Etudiant {

    public java.util.Collection formations = new java.util.TreeSet();
 }
```

La suite de ce chapitre nous permettra de comprendre à quoi correspondent `Collection` et `TreeSet`. Nous verrons par ailleurs si ce choix nous convient.

4.2 Avantages des tableaux

Java offre une gestion confortable des tableaux. Les tableaux sont des données de types répétitifs qui permettent de rassembler plusieurs données sous un seul identificateur. Un indice, compris entre 0 et la taille du tableau (exclue), autorise l'accès individuel à chacune des cases du tableau. Formellement les tableaux rappellent les tableaux du C, mais en fait, ils s'en éloignent par de nombreux points :

- les tableaux sont des classes à part entière. En réalité, lorsqu'on crée une classe, on se donne également la possibilité de manipuler des tableaux contenant des objets de cette classe. Les objets instanciés de type tableau possèdent une variable d'instance publique (`length`) et une série de méthodes, héritées de `Object`.
- une variable de type tableau n'est qu'une référence sur un objet. Il faut donc faire appel à un constructeur implicite pour créer le tableau.

```
int[] nombres = new int[10];
String[] noms = new String[4];
```

- le tableau a une taille fixe, donnée en paramètre au constructeur, et elle ne peut être modifiée. On peut en connaître la valeur par l'attribut `length`. Par opposition au C, la taille du tableau est définie à l'exécution. Elle reste néanmoins impossible à modifier par la suite.
- un tableau créé sur base d'un type primitif contient des variables de ce type, initialisée à la valeur nulle correspondant à ce type. Il est également possible de fournir des valeurs lors de la construction du tableau :

[1] J'ai lu l'affirmation que ce serait la situation du système d'information de l'Etat belge. Ce point était dénoncé par l'auteur de l'article comme la cause de situations injustes.

```
int [] nombres = new int[10];
int[] nombres2 ={0,1,2,3,4,5,9,7,8,9};
```

– si le tableau est créé sur base d'une classe (et non d'un type primitif), il se compose en fait de simples références, par défaut nulles, sur des objets du type voulu. Ici encore, on peut procéder à une initialisation.

```
String[] noms = new String[4];
String[] noms2 = {'Matthieu','Marc','Luc','Jean'};
```

– chaque tableau constitue une classe originale. Il n'est donc pas possible d'écrire une méthode ayant un paramètre de type « tableau » qui accepterait n'importe quel tableau. Cela exclut, notamment, les tableaux basés sur les types primitifs. Il est, par contre, loisible de créer une méthode qui accepte un tableau d'`Objects` comme le montre l'exemple, peu utile, ci-dessous. La méthode renverra la taille de tout tableau passé en paramètre.

```
public static int tailleTableau(Object[] tableau){
        return tableau.length;}
```

– on peut créer des tableaux de tableaux, qui ne sont donc pas exactement des tableaux à deux dimensions. À la différence des tableaux du C, les tableaux internes peuvent avoir des tailles différentes, comme l'illustre cet exemple contenant les premières suites de Fibonacci[2] :

```
int[][] fib =
              {{0},
              {0, 1},
              {0, 1, 1},
              {0, 1, 1, 2,},
              {0, 1, 1, 2, 3},
              {0, 1, 1, 2, 3, 5},
              {0, 1, 1, 2, 3, 5, 8}
             };
```

4.3 Manipulation des tableaux

4.3.1 initialisation

On a vu plus haut l'initialisation d'un tableau à l'aide d'une série de valeur citées entre accolades. Comment faire pour réinitialiser un tableau déjà utilisé ? Une réaffectation à l'aide d'une paire d'accolades n'est pas possible. On peut créer un tableau anonyme, au moyen de la construction `new Type[]{liste de valeurs}` et l'affecter à la variable.

```
String[] strNombres= {"un","deux","trois","quatre","cinq"};
// illégal strNombres={"one","two","three","four","five","six"};
strNombres= new String[]{"one","two","three","four","five","six"};
```

[2] L'exemple est donné sur le site de Chuck ALISSON (*http ://www.freshsources.com*).

Notons que le tableau ne change pas de taille : on définit un nouveau tableau, celui qui contient des mots français est sans doute perdu et en passe d'être recyclé par le ramasse-miettes.

Dans tous les autres cas, l'initialisation du tableau devra se faire élément par élément, le plus souvent à l'aide d'une boucle.

4.3.2 accès aux éléments du tableau

tableaux simples

Une paire de crochets comprenant une valeur d'indice permet d'accéder aux éléments d'un tableau. La valeur de l'indice doit toujours être comprise entre 0 et la taille du tableau -1. Toute autre valeur déclenchera une exception. Dans cet exemple, que n'aurait pas désavoué Vivaldi, nous plaçons le nom des saisons dans un tableau, avant de les afficher de deux manières :

```
String[] saisons = new String[4];
saisons[0]="printemps";
saisons[1]="été";
saisons[2]="automne";
saisons[3]="hiver";
// affichage des saisons
for(int i=0;i<saisons.length;i++)
     System.out.println(saisons[i]);
System.out.println("2me version");
for(String s:saisons)
     System.out.println(s);
```

La seconde boucle d'affichage, introduite en Java 5, est plus simple à gérer. Les deux boucles sont ici équivalentes. Lorsqu'on entend modifier le tableau, ce n'est plus le cas. La boucle de Java 5 n'est utilisable que pour lire les données du tableau. La boucle d'initialisation suivante ne modifie pas le tableau :

```
int[] moyennes=new int[12];
for(int m:moyennes)
   m=1;
```

Cela s'explique par le fait que `m` est une copie de la valeur du tableau. Dans le cas d'un tableau d'objet, la copie de la valeur du tableau est une référence sur un objet. Si on modifie l'objet, la boucle est valable. Si on désire substituer une référence à un autre objet, cela ne marche de nouveau plus :

```
String[] copieSaisons=new String[4];
int n=0;
// boucle inutile
for(String s:copieSaisons)
     s=saisons[n++];
// méthode correcte
for(n=0;n<copieSaisons.length;n++)
     copieSaisons[n]=saisons[n];
```

tableaux de tableaux

L'accès aux éléments d'un tableau de tableaux se fait à l'aide de deux indices. La deuxième occurrence du nombre 5 du tableau `fib` se trouve dans la septième ligne (indice 6) et la sixième case (indice 5) :

```
int[][] fib =
             {{0},
             {0, 1},
             {0, 1, 1},
             {0, 1, 1, 2,},
             {0, 1, 1, 2, 3},
             {0, 1, 1, 2, 3, 5},
             {0, 1, 1, 2, 3, 5, 8}
            };
System.out.println(fib[6][5]);// affiche 5
```

4.3.3 comparaison

La méthode `equals` permet en principe de comparer des objets. Elle ne donne pas de résultats fiables avec les tableaux. Il faut utiliser une des méthodes statiques homonymes de la classe `Arrays` (du paquetage *java.util*)

```
int[] nombres1 ={0,1,2,3,4,5,6,7,8,9};
int[] nombres2 ={0,1,2,3,4,5,6,7,8,9};
// la méthode equals n'est pas fiable
System.out.println("nombres1␣et␣nombres2␣sont␣égaux␣:"+
         nombres1.equals(nombres2));//false
// on peut par contre se fier à Arrays.equals
System.out.println("nombres1␣et␣nombres2␣sont␣égaux␣:"+
         Arrays.equals(nombres1,nombres2));//true
```

4.3.4 copie

La réalisation d'une copie d'objet n'est pas toujours simple en Java. On pourra distinguer trois manières d'opérer une copie d'un tableau :

copie par simple clonage

Une classe tableau hérite d'`Object` la capacité de se cloner. En clonant un tableau formé de données primitives, on obtient un deuxième tableau différent du premier. Toute modification du deuxième tableau laissera intact le premier tableau. Voici un exemple où nous allons trier la copie (technique du tri par insertion) :

```
int[] nombres=new int[]{78,14,21,-7,12};
int[] copies= nombres.clone();
for(int i=1;i<copies.length;i++) {
    int j=i;
    while(j>0 && copies[j-1]>copies[j]){
        int swap=copies[j-1];
        copies[j-1]=copies[j];
```

```
            copies[j]=swap;
            j--;
        }
    }
    System.out.println(Arrays.toString(nombres));//[78, 14, 21, -7, 12]
    System.out.println(Arrays.toString(copies)); //[-7, 12, 14, 21, 78]
```

Ce clonage ne convient pas nécessairement à des tableaux d'objets. En effet, les tableaux d'objets contiennent en réalité des références sur les objets. Si on se contente d'inverser les différents éléments ou si on remplace un élément par une référence vers un autre objet, le tableau clôné et l'original restent bien différents. Par contre, si on applique une de ses méthodes à un des objets de la copie, il y a de fortes chances pour l'original soit également modifié.

Dans ce premier exemple, nous allons cloner un tableau contenant des comptes. Si nous remplaçons le premier élément de la copie par un intrus, nous constatons que l'original reste intact. Mais si nous modifions l'adresse du titulaire du compte de l'original, nous voyons que la même modification se retrouve dans la copie.

```
Compte c;
Personne t;
Compte[] original = new Compte[5];
for(int i=0;i<5;i++){
     c=new Compte();
     t= new Personne();
     t.setAdresse(String.format("Adresse␣du␣titulaire␣%d", i));
     c.setTitulaire(t);
     original[i]=c;
}
Compte[] copie = original.clone();
// Placement d'un intrus dans la copie
c=new Compte();
t=new Personne();
t.setAdresse("Adresse␣de␣l'intrus");
c.setTitulaire(t);
copie[0]=c;
// Original non modifié
System.out.println(original[0].getTitulaire().getAdresse());
System.out.println(copie[0].getTitulaire().getAdresse());
// Manipulation du second élément de l'original
original[1].getTitulaire().setAdresse("rue␣du␣Temple");
// La copie a été modifiée
System.out.println(original[1].getTitulaire().getAdresse());
System.out.println(copie[1].getTitulaire().getAdresse());
```

Les anglo-saxons parlent de *shallow*[3] *copy*.

copie par la méthode `arraycopy`

La classe `System` possède une méthode statique `arraycopy` qui produit un clonage partiel d'un tableau, en introduisant des paramètres spécifiant les indices initiaux et le nombre d'éléments à copier. L'appel à cette méthode produit un résultat identique.

```
System.arraycopy(evan1, 0, evan2, 0, evan1.length);
```

[3] Peu profond, superficiel.

copie par clonage *profond*

Si on veut cloner un tableau en maintenant des copies totalement séparées, il convient de le faire à la main[4]. Voici une manière de réaliser l'opération (la boucle crée explicitement de nouveaux objets). On surdéfinit les méthodes `clone()` des objets impliqués (le code qui suit est limité aux champs concernés dans l'exemple).

```
// Dans la classe Compte
   @Override
    public Compte clone(){
        Compte copie = new Compte();
        copie.setTitulaire(this.getTitulaire().clone());
        return copie;
    }
// Dans la classe Personne
    @Override
    public Personne clone(){
        Personne p = new Personne();
        p.setAdresse(this.getAdresse());
        return p;
    }
```

On notera qu'on copie l'adresse de la personne de manière *shallow*, mais que ce ne prête pas à conséquence, puisque les instances de la classe `String` ne sont pas susceptibles d'être modifiées. Il reste à simuler à la main le clonage du tableau en utilisant les méthodes de clonage redéfinies.

```
for(int i=0;i<5;i++){
     copie[i]=original[i].clone();
}
```

La littérature technique parle ici de *deep copy*. Cette fois, les deux tableaux peuvent vivre des vies indépendantes.

4.3.5 autres richesses de la classe `Arrays`

Il a été question plus haut de la classe `Arrays`, qui fournit de nombreuses méthodes statiques pour manipuler des tableaux. En général, chaque méthode est surchargée pour convenir aux différents types de tableaux primitifs et aux tableaux d'objets.

`list()` permet de transformer un tableau en liste (voir plus loin dans les collections).

`binarySearch()` effectue une recherche de position d'un élément dans le tableau (il faut fournir un `Comparator` pour les tableaux d'objets).

`equals()` a été évoqué plus haut

`fill()` donne la possibilité d'initialiser un tableau avec une même valeur ou des valeurs croissantes dans une certaine plage

`hashCode()` renvoie un code de hachage plus approprié que celui de la méthode héritée de `Object`.

[4]Lorsqu'on définit une classe normale, on peut toujours envisager de surdéfinir la méthode `clone()` pour garantir la réalisation d'un clonage effectif. Dans le cas des tableaux, nous n'avons aucun moyen de réaliser une surdéfinition de la méthode `clone()`.

`sort()` produit un nouveau tableau trié (c'est une quatrième méthode de clonage, qui présente les mêmes problèmes que ceux que nous avons évoqués plus haut). Les tableaux d'objets demandent ici aussi un comparateur.

`toString()` transforme le tableau en une chaîne (il nous a été bien utile pour afficher le contenu des tableaux).

4.4 Les collections dans l'API Java

On a vu qu'un des inconvénients majeurs des tableaux résidait dans leur incapacité à faire varier leur nombre d'éléments. Java a proposé dès le début des classes plus souples permettant de répondre à cette objection, notamment avec la classe `Vector`. On a introduit plus tard l'interface `Collection` qui sert de base à tout un ensemble de classes fortement structurées.

`AbstractCollection` est une classe abstraite fondamentale et sert de classe-mère à toutes les collections. Elle implémente les interfaces `Collection` et `Iterable`, qui lui donnent l'essentiel de ses méthodes. Bien que abstraite, cette classe possède un grand nombre de méthodes déjà implémentées qu'il ne sera pas toujours nécessaire de surdéfinir dans ses descendants. La lecture de son code, abondamment commenté, permet de prendre la mesure de l'intelligence des concepteurs de Java.

Trois classes abstraites dérivent de `AbstractCollection` :

- `AbstractList` implémente l'interface `Liste` et fournit des collections dont les éléments ont une position dans la liste (qui ne dépend pas nécessairement de l'ordre d'introduction). Ils peuvent y figurer plusieurs fois. Parmi les listes, la plus connue est sans conteste `ArrayList`.
- `AbstractSet` est le prototype des ensembles (interface `Set`). Comme le nom l'indique, les éléments doivent être uniques. L'un des ensembles les plus employés se définit par la classe `TreeSet`, qui impose à ses éléments de posséder une méthode `compareTo()`, définie par l'interface `Comparable`.
- `AbstractQueue` est le dernier venu (en Java 5). Il sert de classe-mère à différentes queues, dont la plupart sont intégrées à des mécanismes internes de Java. Il n'en sera pas question plus avant.

La place et le temps me manquent pour détailler toutes les collections de Java, leurs différences et leurs nombreuses méthodes. Je me contenterai de quelques exemples d'utilisation .

4.4.1 Vivaldi sous forme de `ArrayList`

L'utilisation d'un `ArrayList` n'est pas vraiment difficile. Les méthodes `add()`, `size()`, `contains()` et `remove()` ont une signification assez évidente. Voici quelques manipulations sur des chaînes reprenant le nom des saisons. On notera la possibilité d'introduire le printemps après les autres saisons, en profitant d'une surcharge de la méthode `add()` qui spécifie un indice.

```
ArrayList<String> saisons = new ArrayList<String>();
saisons.add("été");
saisons.add("automne");
saisons.add("hiver");
saisons.add(0, "printemps");
System.out.println(saisons.toString());
System.out.println("Contient␣'été'␣:"
    +saisons.contains("été"));//true
```

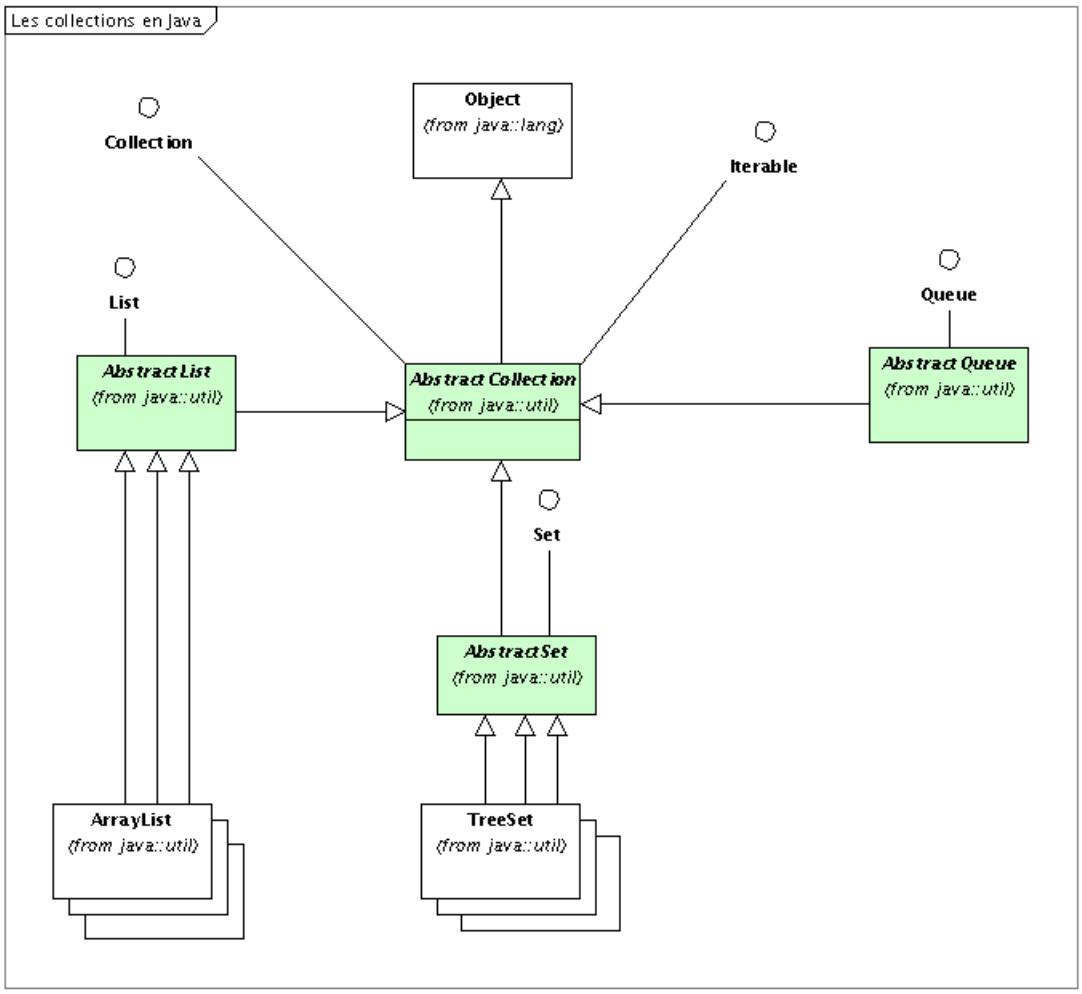


FIG. 4.1 – Aperçu sommaire des collections en Java

```
System.out.println("Contient␣'summer'␣:"
    +saisons.contains("summer"));//false
System.out.println("Réussite␣du␣retrait␣de␣'summer'␣:"
    +saisons.remove("summer"));//false
saisons.add("summer");
System.out.println("Réussite␣du␣retrait␣de␣'summer'␣:"
    +saisons.remove("summer"));//true
System.out.printf("Il␣y␣a␣%d␣saisons.\n", saisons.size());
```

4.4.2 Vivaldi dans un `TreeSet`

Pour examiner la spécificité de la manipulation d'un `TreeSet`, je vais développer une classe particulière, `Saison`, parce que l'utilisation des chaînes nous forcerait à ranger les saisons par ordre alphabétique. Un `TreeSet` impose la détermination d'un critère de classement, implémenté dans la méthode `compareTo()`. Cette méthode admet un objet comme paramètre et doit pouvoir gérer la comparaison d'une instance quelconque avec n'importe quel objet. L'usage de la généricité rend la comparaison d'une instance de saison avec un char d'assaut peu probable, mais la norme nous impose de pouvoir dire comment réagir dans un tel cas. J'ai choisi de dire qu'une saison serait plus petite que n'importe quel objet d'une autre classe.

`compareTo()` fonctionne de manière similaire à la fonction `cmp` du langage C : on

suppose une soustraction entre les deux éléments à comparer. On renvoie :

- une valeur positive pour signifier que le premier élément est « plus grand » que le second
- une valeur négative si le premier est plus petit
- 0 si les éléments sont égaux.

Pour les comparaisons légitimes, j'ai choisi de comparer la date de début de la saison. Voici le code de la classe `Saison`. Je ne donnerai pas de détails sur le fonctionnement des dates, qui feront l'objet d'un prochain chapitre. On constate la présence de deux variables d'instance : `nom` et `dateDebut`. Je n'ai pas prévu d'accesseurs pour les manipuler, vu qu'il s'agit plus d'un exercice que d'une classe utile. La méthode `toString()` les utilise pour afficher la saison.

```
public class Saison implements Comparable{

  /**
   * le nom de la saison
   */
  private String nom;
  /**
   * la date de début de la saison
   */
  private Date dateDebut;
  /**
   * Constructeur
   * @param nom initialise le nom de la saison
   * @param mois mois où commence la saison (1-12)
   * Attention, j'ai utilisé une approche intuitive de la notion
   * de mois (Java les numérote de 0 à 11)
   */

  public Saison(String nom, int mois){
      this.nom=nom;
      Date date= new Date();
      Calendar cal= new GregorianCalendar(2005,mois-1,21);
      dateDebut=cal.getTime();    }

  /**
   * Comparaison rendue nécessaire par l'implémentation de
   * l'interface Comparable
   * @return 0 si les objets sont égaux,
   *         1 si le premier est le plus grand,
   *         -1 si le premier est plus petit
   * @param o Objet deuxième terme de la compaison
   */
  public int compareTo(Object o) {
      if(o.getClass()!=this.getClass())
          return -1;
      else
          return this.dateDebut.compareTo(((Saison)o).dateDebut);
  }

  /**
   * Permet d'afficher la saison sous forme lisible
```

```
     * @return Le nom et la date de début de la saison
     */
    public String toString(){
        Calendar cal = new GregorianCalendar();
        cal.setTime(dateDebut);
        return nom+"<"+(cal.get(MONTH)+1)+">";
    }
}
```

Voici une utilisation de la classe `Saison` dans un `TreeSet`. On retrouve les méthodes `add()`, `remove()` et `contains()`.

```
TreeSet<Saison> saisons = new TreeSet<Saison>();
Saison printemps=new Saison("printemps", 3);
saisons.add(printemps);
saisons.add(new Saison("hiver", 12));
saisons.add(new Saison("automne", 9));
saisons.add(new Saison("été",21,7));
// le prochain ajout sera ignoré
saisons.add(printemps);
System.out.println(saisons);
//[printemps<3>, été<7>, automne<9>, hiver<12>]
saisons.remove(printemps);
// deuxième suppression ignorée
saisons.remove(printemps);
System.out.println(saisons);
saisons.add(printemps);
for(Saison s: saisons)
         System.out.println(s);
System.out.println("Recherche␣de␣printemps␣réussie␣:"
   +saisons.contains(printemps));// TRUE
Saison intrus=new Saison("floréal", 4);
System.out.println("Recherche␣de␣floreal␣réussie␣:"
   +saisons.contains(intrus)); // FALSE
```

Notons que l'utilisation d'une fausse saison ayant la même date de début qu'une saison présente dans le tableau provoque une réponse vraie pour la méthode `contains()`.

```
Saison imposteur=new Saison("germinal",3);
System.out.println("Recherche␣de␣germinal␣réussie␣:"
   +saisons.contains(imposteur)); /TRUE
```

La méthode `remove()` ne fonctionne pas de la même façon que le méthode `contains()`. Elle se base sur la méthode `equals()` qui n'est en principe pas prévue pour tenir compte des dates. Il faut surdéfinir cette dernière :

```
/**
 * Version surdéfinie de equals, pour tenir compte du critère
 * de classement défini dans compareTo()
 * @param o Objet avec lequel effectuer la comparaison
 * @return vrai si les objets ont la même date
 */
```

```
public boolean equals(Object o){
    return this.compareTo(o)==0;
}
```

On constate l'absence d'une méthode `get()`. Il n'y a aucun moyen d'accéder au *nième* élément de l'ensemble. On peut par contre obtenir un moyen d'accès très efficace grâce aux méthodes `headSet()`, `tailSet()`, et `subSet()`, qui renvoient respectivement des sous-ensembles triés se terminant par une limite, commençant par une limite ou compris entre deux limites. Les méthodes `first()`, `last()` et `size()` pourront également être utiles. Attention, les trois méthodes d'obtention des sous-ensembles ne produisent pas des copies. Toute manipulation sur ces sous-ensembles se répercute sur l'original.

```
SortedSet ss = saisons.tailSet(new Saison("summer", 7));
System.out.println(ss);
//[été<7>, automne<9>, hiver<12>]
System.out.println("Premier élément de l'ensemble trié "+ss.first());
ss.remove(new Saison("winter",21,12));
System.out.println(ss);//[été<7>, automne<9>]
System.out.println(saisons);//[printemps<3>, été<7>, automne<9>]
```

4.4.3 parcourir une collection

Nous avons vu que les tableaux pouvaient se parcourir à l'aide de deux types de boucles : une boucle avec indice et une boucle « for-each ». Qu'en est-il des collections ?

boucles for-each de Java 5

Comme il fallait s'y attendre, les boucles de Java 5 fonctionnent très bien. Il n'y a aucune différence entre une liste et un ensemble[5].

```
// parcours for-each
for(String s : saisons)
    System.out.println(s);
```

gestion d'un indice

Les listes fonctionnent également très bien avec un indice, qu'il est facile de limiter entre 0 et la taille -1. Par contre, nous ne disposons de rien de semblable pour les ensembles.

```
// parcours indexé
for(int i=0;i<saisons.size();i++)
    System.out.println(saisons.get(i));
```

[5]Pour l'exemple avec les ensembles, il faudra évidemment utiliser le bon type, à savoir Saison.

utilisation d'un itérateur

Comment faisaient nos grands-pères pour énumérer les éléments d'un ensemble avant l'arrivée des boucles de Java 5 ? Ils disposaient heureusement des itérateurs. Un itérateur, comme nous le verrons lors de l'implémentation d'une paire, est intimement lié à la conception d'une collection. C'est une classe spécialisée par une interface comprenant trois méthodes :

- `hasnext()` renvoie vrai s'il reste des éléments à parcourir
- `next()` renvoie l'élément suivant (un `Object` pour nos ancêtres, un objet du type précisé par la déclaration de la collection en Java 5)
- `remove()` supprime l'élément qui a été listé par le dernier next(). Son utilité ne semble pas primordiale pour l'utilisateur de la collection, mais il permet en fait de réaliser l'implémentation de la méthode `remove()` de la collection.

```
// parcours avec un itérateur
Iterator<Saison> iterateur=saisons.iterator();
while(iterateur.hasNext())
     System.out.println(iterateur.next());
```

En fait, les boucles de Java 5 font un usage masqué des itérateurs . Désormais, seuls les programmeurs qui créent de nouveaux types de collections ont encore besoin de se soucier des itérateurs.

4.5 Approche de la généricité (Java 5)[6]

Il n'est pas possible de passer sous silence l'usage des classes génériques en parlant des collections. Avant Java5, toutes les collections étaient des collections d'objets. Voici un exemple de programme qui pouvait poser problème.

```
Collection pommes = new ArrayList();
pommes.add(new Pomme());
pommes.add(new Pomme());
pommes.add(new Scoubidou());
for(int i=0;i<pommes.size();i++){
   Pomme p = (Pomme)pommes.get(i);
   p.extraireJus();
}
```

L'ajout d'un élément étranger dans la collection de pommes passait parfaitement inaperçu, puisque la collection contenait des objets. C'est au moment de la récupération des données, qu'il fallait transformer l'objet récupéré par la méthode `get()` en pomme pour en extraire le jus. Cette opération de casting provoquait évidemment une erreur d'exécution. On pouvait bien sûr tester la classe et prendre des mesures à ce niveau, mais cela alourdissait terriblement le code.

L'utilisation de classes génériques simplifie énormément les choses. On voit parfaitement que l'erreur dans ce programme ne consiste pas à transformer l'objet en pomme au moment de l'extraction, mais réside dans l'introduction d'un scoubidou dans la collection. Le compilateur est parfaitement à même de savoir que l'objet introduit ne possède pas le bon type. Il suffit donc de spécifier le type des données à placer dans la collection au moment de sa déclaration.

[6]Ce paragraphe n'aborde qu'une toute petite partie de cette question.

```
Collection<Pomme> pommes = new ArrayList<Pomme>();
pommes.add(new Pomme());
pommes.add(new Pomme());
pommes.add(new Scoubidou()); // erreur de compilation
for(int i=0;i<pommes.size();i++){
   Pomme p = pommes.get(i); // casting implicite
   p.extraireJus();
}
```

L'introduction de la généricité se situe donc au niveau du code source et de la compilation. Le programme exécutable n'a aucune idée du type des éléments contenus dans la collection. Dans des aspects avancés de la généricité, cela permettra d'en comprendre certaines limites. Le code de la première version est toujours utilisable (compatibilité oblige), mais il déclenchera un avertissement :

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

4.6 Implémenter une paire

Dans l'exercice consacré au rallye, nous avions proposé plusieurs fois une multiplicité de 2 : pour le nombre de camions dans l'écurie, de voitures par camion et de pilotes par voiture. Nous avions partout utilisé des `ArrayList`, qui ne permettaient aucun contrôle sur le nombre effectif d'éléments dans la liste. Je me propose de réaliser un nouveau type de collection, nommé `Paire`, pour réaliser ce contrôle. Le cahier des charges sera le suivant :

- une paire pourra contenir de 0 à 2 éléments.
- il sera possible d'ajouter jusqu'à un maximum de deux éléments dans une paire (méthode `add()`). Tout ajout superflu sera ignoré. La méthode d'ajout renverra une valeur logique pour signifier si l'ajout a été effectivement réalisé. On ne pourra pas placer deux fois le même élément dans la paire.
- on pourra accéder aux éléments au moyen d'une méthode `get()` avec un paramètre 0 ou 1. Un indice hors plage ou ne correspondant pas à un élément présent provoquera le renvoi de null. On pourra aussi parcourir la paire avec un itérateur ou une boucle « for-each ». La position relative des éléments sera conditionnée par leur ordre d'introduction, mais jugée non significative.
- il sera possible de supprimer un élément dont on donne la référence.

La collection que nous allons manipuler est relativement basique. Nous pouvons la faire dériver d'une classe peu spécifiée : `AbstractCollection`. Celle-ci ne demande pas grand chose :

1. créer les variables contenant les données ;
2. fournir une surdéfinition de la méthode `size()` qui renverra le nombre d'éléments présents dans la paire ;
3. fournir une surcharge de la méthode `add()` (celle de la classe abstraite lève une exception de fonction non implémentée) ;
4. renvoyer un itérateur (dont le rôle est primordial dans le fonctionnement des autres méthodes) ;
5. nous y ajouterons une méthode `get()`, comme pour les listes, afin de pouvoir récupérer les données par leur indice.

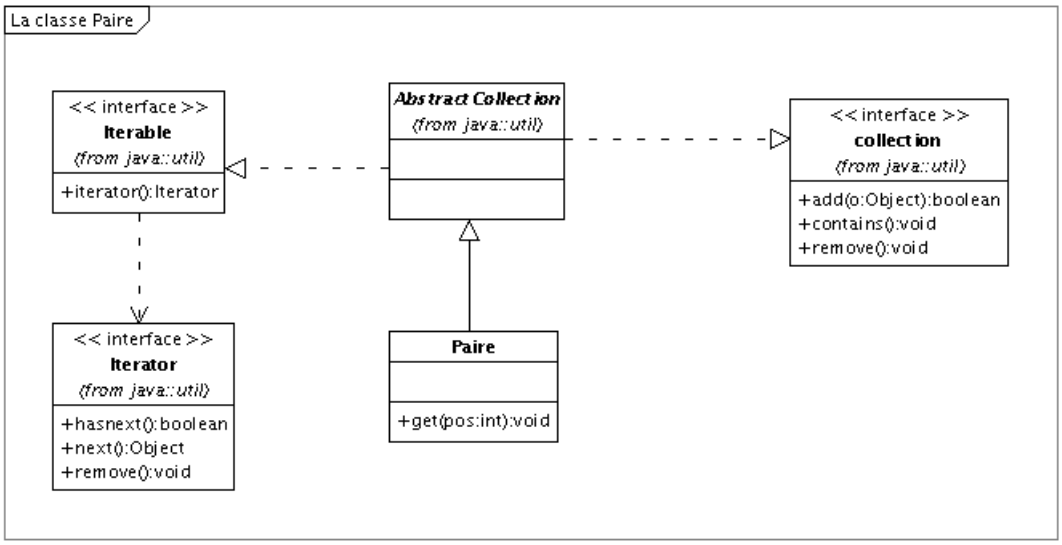


FIG. 4.2 – La classe Paire et sa classe-mère

4.6.1 déclaration de la classe

Nous allons créer une classe générique, afin de pouvoir spécifier les types d'objets qu'une instance de paire pourra contenir. En soi, ce n'est pas difficile. Cela va par contre rendre plus difficile la gestion des contenus. <E> représente le type des éléments introduits dans la paire. Nous importons deux classes pour servir de classe mère et pour implémenter l'itérateur.

```
import java.util.AbstractCollection;
import java.util.Iterator;
/**
 * Une paire est une collection de deux objets différents de type <E>
 */
public class Paire<E> extends AbstractCollection<E> {
...}
```

4.6.2 création des données et constructeur

Il faut en fait créer trois variables privées pour gérer la paire :

```
/**
 * Nombre d'éléments dans la paire
 */
private int taille=0;

/**
 * Variables contenant les deux données. On aurait pu aussi
 * utiliser un tableau
 */
private E premier,second;

/** Creates a new instance of Paire */
public Paire() {     }
```

4.6.3 méthodes surdéfinies

`size()`

Cette méthode ne présente aucune difficulté. Il suffit de renvoyer la valeur de la variable d'instance `taille`.

```
/**
 * Méthode surdéfinie:
 * Renvoie le nombre d'élément actuel de la paire
 * @return nombre d'éléments dans la paire
 */
public int size() {
     return taille;
}
```

`add()`

Trois cas peuvent se présenter : élément présent, plus de place, élément non présent. On incrémente si besoin est la variable `taille`.

```
/**
 * Méthode surchargée:
 * Ajouter un élément, à condition qu'il n'y soit pas déjà et qu'il
 * reste de place
 * @param e Elément à ajouter
 * @return Vrai si ajout réussi
 */
   public boolean add(E e){
       boolean success=true;
       switch(taille){
           case 0:
               premier=e;
               taille++;
               break;
           case 1:
               if(!premier.equals(e)){
                   e2=e;
                   taille++;
               }
               break;
           default:
               success=false;
       }
       return success;
   }
```

iterator()

Cette méthode renvoie une instance d'un itérateur conçu pour manipuler une paire. Nous verrons tout de suite après comment définir cette classe auxilaire.

```
/**
 * Méthode exigée par l'interface Collection: elle permet de définir
 * un itérateur sur les objets contenus dans la collection
 * @return Un itérateur sur l'objet Paire
 */
public Iterator<E> iterator() {
      return new IterPaire<E>();
}
```

4.6.4 méthode ajoutée

`get()`

Nous ajoutons cette méthode, inspirée des listes, pour permettre un accès individuel à chaque objet. On pourra également grâce à elle effectuer un parcours indicé de la collection.

```
/**
 * Récupérer un élément sur base de sa position
 * @param pos Position de l'élément à récupérer (0 ou 1)
 * @return Référence de l'élément récupéré ou null
 */
public E get(int pos){
    if(pos>-1 && pos<taille)
        if(pos==0)
            return premier;
        else
            return second;
     else
         return null;
    }
```

4.6.5 l'itérateur

L'itérateur va jouer un rôle crucial dans le fonctionnement de la classe. J'ai choisi ici de le définir comme une classe interne à l'intérieur de la classe *Paire*. En effet, il ne sert à rien de la rendre plus accessible, puisqu'on peut en obtenir une instance avec la méthode `iterator()`.

```
/**
 * classe interne permettant de parcourir la paire
 */
private class PaireIterator<E> implements Iterator<E>{

/**
 * Position du prochain élément à accéder par l'itérateur
 */
private int ptr;

/**
 * Constructeur de l'itérateur remettant next à 0
 */
```

```
IterPaire(){
     ptr=0;
}
/// Ici les autres méthodes
}
```

Nous avons vu plus haut que trois méthodes devaient être implémentées par l'itérateur.

hasNext()

En plaçant la classe IterPaire à l'intérieur de la classe Paire, nous nous donnons la possibilité d'accéder directement à ses variables d'instances.

```
/**
 * teste l'existance d'un élément suivant dans la paire
 * @return renvoie la référence de l'élément suivant s'il existe
 */
public boolean hasNext() {
     return (ptr<taille);
}
```

next()

```
/**
 * Accède au prochain élément de la paire
 * @return renvoie la référence de l'élément suivant s'il existe
 */

public E next() {
   if(hasNext())
      if(ptr++==0)
            return (E)premier;
      else
            return (E)second;
   else
      return null;
}
```

remove()

```
/**
 * Supprime l'élément en cours d'itération
 * (attention next est déjà incrémenté)
 */
public void remove() {
      if(ptr==2){
          second=null;
          taille--;
      }
      if(ptr==1){
```

```
            premier=second;
            second=null;
            taille--;
        }
}
```