Chapitre 7

Entré et sortie, fichiers, sauvegarde

On appelle entrée/sortie tout échange de données entre le programme et une source :

- entrée : au clavier, lecture d'un fichier, communication réseau
- sortie : sur la console, écriture d'un fichier, envoi sur le réseau

L'orientation objet du langage permet de regrouper des opérations similaires. JAVA utilise la notion de *flux* (*stream* en anglais) pour abstraire toutes ses opérations. Ici, le flux est un flux de données d'une source vers une destination. Le flux a donc une *direction*, on parle toujours soit d'un flux en entrée (qui pars d'une source et qui arrive dans le programme pour être traité) soit d'un flux en sortie (données traitées par le programme et qui sont envoyées à une destination). La nature de la source et de la destination peuvent être très différentes : un fichier, la console, le réseau. Le nature des données qui transitent dans le flux peut aussi varier : on distinguera principalement un flux de `bytes` (une suite de huit `0` ou `1`) ou un flux de caractères. La Figure 7.1 représente une partie de la hiérarchie de classes qui gère ces entrées/-sorties. La classe `File` contient des outils pour accéder aux informations d'un fichier, on en parlera plus tard. On peut voir deux classes abstraites qui manipulent les flux en entrée (`InputStream`) et les flux en sortie (`OutputStream`). On a représenté deux implémentations de chacune de ces classes abstraites selon la nature du flux (flux venant ou sortant d'un fichier ou d'un Objet). Finalement, on observe deux classes abstraites pour lire et écrire dans les flux (`Reader` et `Writer`). On a représenté deux implémentations qui diffèrent selon la nature du flux : un flux de `bytes` peut être traité par les classes `InputStreamReader` et `OutputStreamWriter` alors qu'un flux de caractères est traité par les classes `BufferedReader` et `BufferedWriter`. En résumé on obtient :

- Direction du Flux :
  - objets qui gèrent des flux d'entrée : `in`
    => `InputStream`, `FileInputStream`, `FileInputStream`
  - objets qui gèrent des flux de sortie : `out`
    => `OutputStream`, `FileOutputStream`, `FileOutputStream`
- Source du flux :
  - **fichiers** : on pourra avoir des flux vers ou à partir de fichiers
    => `FileInputStream` et `FileOutputStream`
  - **objets** : on pourra envoyer/recevoir un objet via un flux
    => `ObjectInputStream` et `ObjectOutputStream`

On peut considérer que les classes pour lire et écrire sont des outils pour transformer les flux. Un

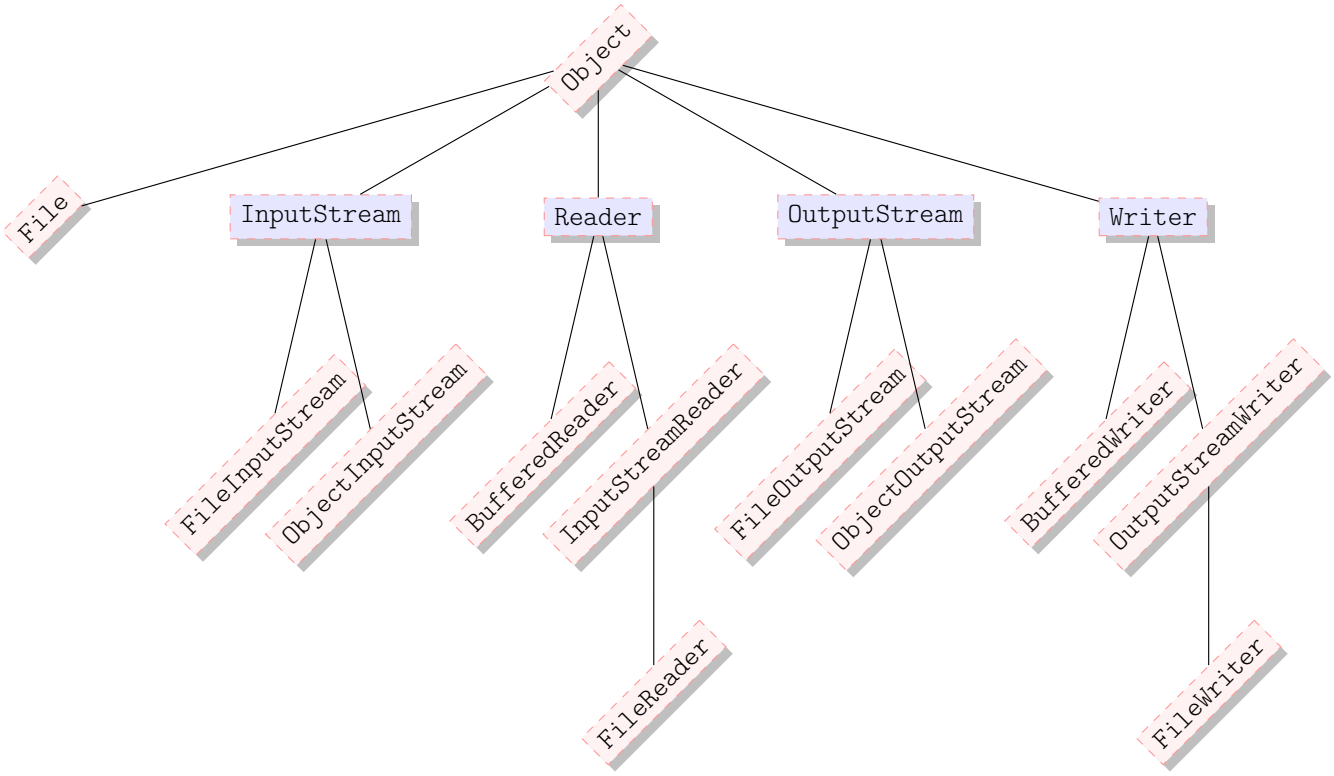


FIGURE 7.1 – Fragment de la hiérarchie de classes pour les entrées/sorties

`InputStreamReader` prend en entrée un flux de `bytes` et le transforme en un flux de caractères `char`. De manière inverse, le `OutputStreamWriter` transforme un flux de caractères en flus de `bytes`. Lorsqu'on a un flux de `char` en entrée, on peut alors utiliser la classe `BufferedReader` pour lire, via une mémoire tampon, ce qui se trouve dans le flux. De même pour la sortie, on peut écrire en sortie dans `BufferedWriter` qui utiliser une mémoire tampon et générer le flux de `char` en sortie. La Figure 7.2 résume ces étapes.

Notez bien que les opérations de lecture et d'écriture peuvent échouer pour des raisons multiples. Pour certaines, on peut avoir un contrôle (par exemple le fichier n'existe pas, on n'a pas d'accès en lecture ou écriture dans le répertoire), d'autres sont en dehors de tout contrôle (mauvais secteur du disque). Ainsi, toutes les opérations de lecture ou d'écriture sont susceptibles de lever des exceptions, qu'il faudra capturer. Il existe un hiérarchie d'exceptions dédiées aux erreurs d'entrée/sortie : ces exceptions héritent de la classe `IOException`.

1 Lire depuis la console, afficher sur la console

Pour lire, la console peut être accédée via ce que l'on appelle l'entrée « standard »`System.in`, qui est un objet de type `InputStream`. La classe `Scanner` est une classe très utile pour récupérer ce qui est tapé dans la console, en particulier récupérer et traduire automatiquement des entiers `int`, des nombres à virgule (`float` ou `double`), des chaînes de caractères (`String`).

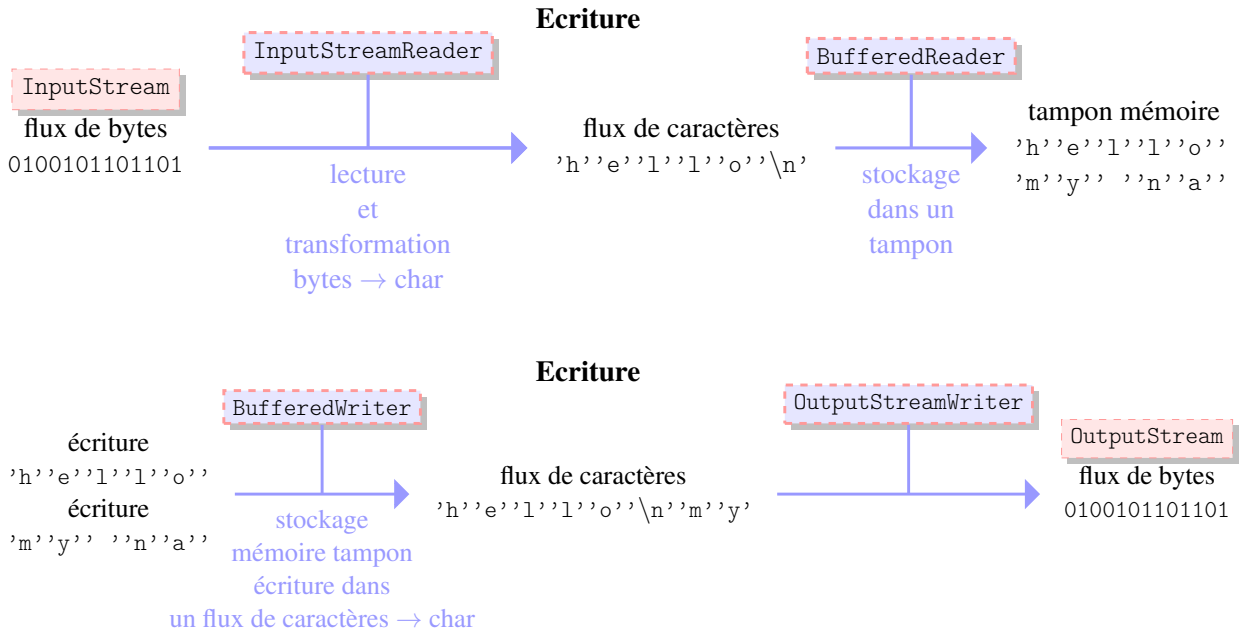


FIGURE 7.2 – Encapsulation BufferedReader(InputStreamReader) et BufferedWriter(OuputStreamWriter)

```
1 Scanner scan = new Scanner(System.in) ;
2 int n = scan.nextInt() ;
3 double x = scan.nextDouble() ;
4 String s = scan.nextLine() ;
```

Pour l'écriture, on accède la console via ce qui est appelé la « sortie standard » `System.out`, qui est de type `PrintStream` (lui-même héritant de `OutputStream`). Donc la fameuse expression `System.out.println( a_string)` ; est simplement l'écriture d'une chaîne de caractère dans un flux de `char` vers la sortie standard.

2 Lecture/Ecriture d'un fichier

La classe `File` permet d'obtenir des informations diverses sur les fichiers, on donne des exemples dans la Table 7.1. C'est aussi grâce à cette classe qu'on peut lire le contenu d'un fichier ou accéder à ce fichier pour écrire. Un fichier est une suite de `0` et `1` enregistrée sur le disque. Ainsi, pour écrire ou lire dans un fichier, JAVA utilise un flux de `bytes`. Pour écrire ou lire un fichier avec des `char`, on utilisera un objet `BufferedReader` ou `BufferedWriter`.

— En lecture : la classe `FileReader` va lire le fichier et placer le contenu dans un flux de `bytes`.
— En écriture : la classe `FileWriter` va prendre prendre un flux de bytes en entrée et écrire le flux dans le fichier

/

Dans l'exemple suivant, on va lire le premier octet d'un fichier (i.e., les 8 premières `bytes`) qui se nomme `ex.txt`. On instancie un objet de type `File` qui va accéder au fichier `ex.txt`. On va créer un flux en lecture à partir de cet objet. On aurait pu faire cela en deux étapes, mais ici, on le fait en une

— nom, chemin absolu, répertoire parent
— s'il existe un fichier d'un nom donné en paramètre
— droit : l'utilisateur a-t-il le droit de lire ou d'écrire dans le fichier
— la nature de l'objet (fichier, répertoire)
— la taille du fichier
— obtenir la liste des fichiers
— effacer un fichier
— créer un répertoire
— accéder au fichier pour le lire ou l'écrire

TABLE 7.1 – Information accessible depuis la classe `File`

seule étape en encapsulant l'instance de `File`. Ensuite, on lit le flux de `bytes` à l'aide de la méthode `read`. Evidemment, on obtient des `bytes` que l'on range dans un tableau. Une fois la lecture effectuée, on peut traiter l'information, ici, on se contente simplement d'afficher à l'écran la chaîne de caractère qui correspond à cet octet. On n'oublie bien sûr pas de fermer le flux avec la méthode `close`.

```
1 FileInputStream fis =
2                 new FileInputStream(new File(" ex.txt")) ;
3 byte[] huitLettres = new byte[8] ;
4 int nbLettreLues = fis.read(huitLettres) ;
5 for(int i=0 ;i<8 ;i++)
6    System.out.println(Byte.toString(huitLettres[i])) ;
7 fis.close() ;
```

Notez que dans ce code, on n'a pas tenu compte des exceptions :

— l'instanciation de l'objet `FileInputStream` peut lever une exception de type `FileNotFoundException`
— l'appel à la méthode `read` peut lever une exception de type `IOException`.
— la fermeture du flux avec la méthode `close` peut aussi lever une exception de typeIOException.

Il faudrait donc modifier ce code pour prendre en compte ces exceptions (soit en utilisant un bloc `try` ... `catch` soit en délégant la capture de l'exception à une méthode appelante, voir Section 6).

Dans l'exemple suivant, on affiche le contenu d'un fichier sur la console. On accède au fichier en instanciant un objet de type `File`, on utilise un flux de byte pour lire le fichier avec un objet `FileReader`, et on utilise un objet de type `BufferedReader` pour transformer le flux de `bytes` en flux de `chars`. On lit le fichier ligne par ligne en utilisant la méthode `readLine()`. De nouveau, cet exemple ne prend pas en compte les exceptions.

```
1 BufferedReader reader =
2    new BufferedReader(new FileReader(new File("ex.txt"))) ;
3 String line = reader.readLine() ;
4 while(line != null){
5    System.out.println(line) ;
6    line = reader.readLine() ;
7 }
8 reader.close() ;
```

Ci-dessous on combine les deux exemples en traitant les exceptions avec un bloc `try` . . . `catch`.

```
1  try {
2     FileInputStream fis = new FileInputStream(new File("test.txt")) ;
3     byte[] buf = new byte[8] ;
4     int nbRead = fis.read(buf) ;
5     System.out.println("nb bytes read : " + nbRead) ;
6     for (int i=0 ;i<8 ;i++)
7       System.out.println(Byte.toString(buf[i])) ;
8     fis.close() ;
9
10    BufferedReader reader =
11        new BufferedReader(new FileReader(new File("test.txt"))) ;
12    String line = reader.readLine() ;
13    while (line!= null){
14      System.out.println(line) ;
15      line = reader.readLine() ;
16    }
17    reader.close() ;
18 } catch (FileNotFoundException e) {
19    e.printStackTrace() ;
20 }
21 catch(IOException e){
22    e.printStackTrace() ;
23 }
```

3 Lecture/Ecriture d'un objet : la sérialisation

Le mécanisme de « sérialisation »permet de stocker et transmettre une instance de classe. Pour se faire, la classe en question doit implémenter l'interface `Serializable`. D'une façon surprenante, cette interface *n'*a *pas* de méthode. On peut considérer qu'elle n'a qu'un rôle de *marqueur* pour indiquer que cet objet peut être sérialisé. C'est JAVA qui se charge de traduire l'objet dans un format qui n'est pas lisible par le programmeur. Pour se faire :

— en lecture : un objet `ObjectOutputStream` traduit l'objet sérialisable en un flux de `bytes`.
— en écriture : un objet `ObjectInputStream` traduit un flux de `bytes` dans un objet.

Dans l'expemple qui suit, on va sérialiser un objet de type `IrreductibleGaulois` pour l'écrire dans un fichier et le lire par la suite. Pour la l'écriture, on utilise un `ObjectOutputStream` et on appelle la méthode `writeObject()` pour traduire l'objet. Pour la lecture, on utilise `ObjectInputStream` et sa méthod `readObject`. Notez que lors de la lecture, le compilateur ne peut savoir le type de l'objet qui est lu. C'est pour cela qu'on utilise un cast explicite, le programmeur sachant que l'objet est bien du type `IrreductibleGaulois`.

```
1  IrreductibleGaulois panoramix =
2          new IrreductibleGaulois("Panoramix", 1.75) ;
3
4  ObjectOutputStream oos =
5          new ObjectOutputStream(
6            new FileOutputStream(
7              new File("panoramix.txt"))) ;
8
9  oos.writeObject(panoramix) ;
10 oos.close() ;
11
12 ObjectInputStream ois =
13         new ObjectInputStream(
14           new FileInputStream(
15             new File("panoramix.txt"))) ;
16
17 IrreductibleGaulois copyPanoramix =
18         (IrreductibleGaulois) ois.readObject() ;
19 System.out.println(copyPanoramix.nom) ;
20
21 ois.close() ;
```

Notez que dans cet exemple, le code n'est pas correct car il manque la gestion des exceptions.

Evidemment, pour qu'une classe soit « sérialisable », il serait souhaitable que tous les objets qui la composent soit sérialisable. Or, il se peut qu'un attribut ne le soit pas. Dans ce cas, on peut utiliser le mot clé `transient` pour indiquer de ne pas enregistrer cet attribut.

Chapitre 8

Notion de types paramétrés et Collections

1 Type paramétré

```
17 | IrreductibleGaulois copyPanoramix = (IrreductibleGaulois) ois.readObject() ;
```

Dans le chapitre précédent, on a utilisé l'instruction ci-dessus : à partir d'un `ObjectInputStream`, on a récupéré un objet. Dans l'exemple, on savait qu'on allait récupérer un objet de type `IrreductibleGaulois`, mais au moment de la compilation, on ne peut pas toujours vérifier ces transformations (pour rappel, on appelle cette transformation un transtypage explicite, « cast » en anglais). Ce n'est qu'au moment de l'exécution que l'on va détecter une erreur. Pour renforcer la sécurité, JAVA offre la possibilité d'utiliser un type paramétré.

1.1 Exemple

Commençons par un exemple dans lequel on nous demande de coder une structure de données très simple : la liste chaînée. Pour commencer, on va faire une liste chaînée de `String`. On veut donc faire une liste dans laquelle chaque `String` est encapsulé dans un noeud. Le noeud contient la valeur et un lien sur le noeud suivant. Pour se faire, on écrit une classe `Noeud` et une classe `ListeChainee` comme suit

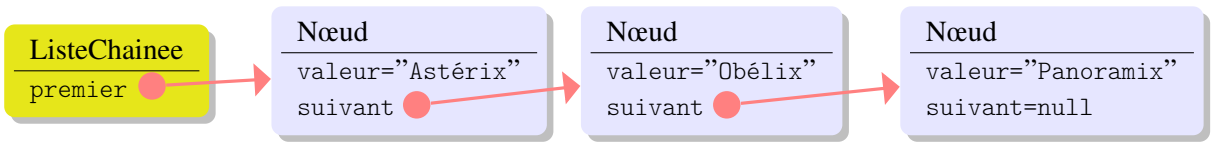
```
1  public class Noeud {
2     String valeur ;
3     Noeud suivant ;
4
5     public Noeud(String val){
6       valeur = val ;
7     }
8
9     public void setSuivant(Noeud next){
10      suivant = next ;
11    }
12 }
```

```java
public class ListChainee {
   Noeud premier ;

   public ListeChainee(){
     premier = null ;
   }

   public void add(String val){
     Noeud nouveau = new Noeud(val) ;
     if (premier == null)
       premier = nouveau ;
     else {
       Noeud dernier = premier ;
       while(dernier.suivant != null)
         dernier = dernier.suivant ;
       dernier.suivant = nouveau ;
     }
   }
}
```

Pour implémenter une liste contenant trois chaînes de caractères, par exemple {"Astérix","Obélix","Panoramix"}, en mémoire, on aura quatre objets que l'on peut représenter comme suit :



Maintenant, on voudrait faire une liste de `Personnages` à la place d'une liste de `String`.

— Une possibilité est de faire une nouvelle classe `NoeudPersonnage` dont les valeurs sont de type `Personnage` et faire de même pour une classe `ListeChaineePersonnages`. On devrait donc avoir une classe `NoeudType` par chaque type dont on voudrait faire une liste... pas très pratique. Imaginez que l'on veuille ensuite modifier la classe `Noeud` (par exemple ajouter une méthode, ou bien implémenter une liste avec une référence sur le noeud précédent), et il faudra faire le changement sur toutes les versions...

— Une autre possibilité est de modifier la classe `Noeud` et mettre `Object` à la place de String. On pourra mettre tous les types possibles dans un nœud, ce qui fonctionnera bien. Cependant, il faudra faire des transtypages explicites pour récupérer la valeur du nœud. Il faudra aussi faire bien attention à ce que l'on met dans la liste (on pourrait alors facilement mélanger des `Strings`, des `Personnages`, des `Integers`, etc... dans une même liste et il faudra un peu de travail utiliser son contenu par la suite).

La solution offerte par JAVA est la possibilité d'ajouter un type en paramètre de la classe. Ainsi, on va pouvoir avoir une seule implémentation de la classe `Noeud` qui va prendre en paramètre une classe, que l'on va noter `E` et on notera `Noeud<E>`. On va écrire les deux classes comme suit.

```java
public class Noeud<E> {
   E valeur ;
   Noeud<E> suivant ;

   public Noeud(E val){
     valeur = val ;
   }

   public void setSuivant(Noeud<E> next){
     suivant = next ;
   }
}
```

```java
public class ListChainee<E> {
   Noeud<E> premier ;

   public ListeChainee(){
     premier = null ;
   }

   public void add(E val){
     Noeud<E> nouveau = new Noeud<E>(val) ;
     if (premier == null)
       premier = nouveau ;
     else {
       Noeud<E> dernier = premier ;
       while(dernier.suivant != null)
         dernier = dernier.suivant ;
       dernier.suivant = nouveau ;
     }
   }

   public E get(int index){
     int i=0 ;
     Noeud<E> courant=premier ;
     while(courant.suivant != nll && i<index){
       i++ ;
       courant = courant.suivant ;
     }
     if(index == i) // on a trouvé l'élément numéro i
       return courant ;
     else
       return null ;
     }
   }
}
```

Avec cette implémentation, on va pouvoir utiliser des listes de `Personnage`s, des listes de `String`, `Integer`, etc. Pour faire une liste de trois `Personnage`s, on peut écrire le code suivant :

```
1 IrreductibleGaulois asterix = new IrreductibleGaulois("Astérix") ;
2 IrreductibleGaulois obelix = new IrreductibleGaulois("Obélix") ;
3 Gaulois Informatix = new Gaulois("Informatix") ;
4 ListeChainee<Gaulois> liste = new ListeChainee<Gaulois>() ;
5 liste.add(asterix) ;
6 liste.add(obelix) ;
7 liste.add(informatix) ;
```

Même si Astérix et Obélix sont des irréductibles gaulois, ils sont donc a fortiori des gaulois, et on peut donc manipuler une liste de gaulois. Maintenant, puisqu'il n'y a pas besoin de transtypage explicite, le compilateur peut vérifier si les types sont correctement utilisés.

Si on veut manipuler une liste de `Integer`, il faut ajouter des objets de type `Integer`, et quand on utilise un élément de la liste, on récupère un objet de type `Integer` et on doit utiliser une méthode comme `intValue()` pour obtenir la valeur de l'`Integer`. A priori, cela n'est pas très pratique, mais JAVA offre une fonctionalité qui fait automatiquement les conversions (ce qui est appelé *autoboxing*).

```
1 ListeChainee<Integer> maListe = new ListeChainee<Integer>() ;
2 //old style
3 maListe.add(new Integer(7)) ;
4 Integer sept = maListe.get(1) ;
5 System.out.println(sept.intValue()) ;
6 //new style
7 maListe.add(6) ;
8 int six = maListe.get(2) ;
```

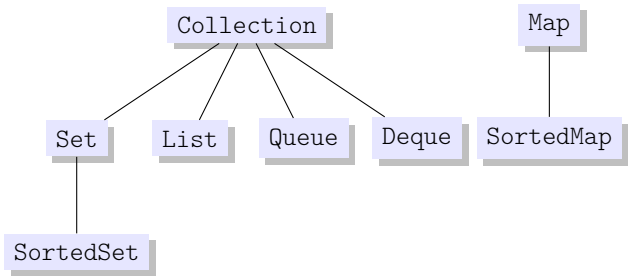
1.2 Types paramétrés

Pour ce cours, on en restera à cette introduction préliminaire des types paramétrés, car il reste beaucoup à dire. Ce sujet sera vu plus en détail lors du cours *Programmation Java Avancée*.

2 Collections

Les listes, les ensembles, les piles, les files d'attente sont des objets qui regroupent plusieurs éléments en une seule entité. Ces structures partagent un certain nombre de choses. On peut poser le même genre de questions : est-ce que la structure contient des éléments ? combien ? Certaines opérations sont similaires : on peut ajouter ou enlever un élément à la structure, on peut vider la structure. On peut aussi parcourir les éléments contenus dans la structure. On peut avoir des implémentations différentes de chacune de ces structures, par exemple on a vu l'exemple de la liste chaînée, mais on pourrait avoir une implémentation basée sur un tableau, ou un liste doublement chainée (avec une référence sur l'élément précédent).

Comment peut-on manipuler toutes ces structures ? Une solution – qui devrait paraître naturelle maintenant – est d'utiliser une hiérarchie d'interfaces, et c'est ce que JAVA propose.



— `Collection` : Tout en haut de la hiérarchie se trouve l'interface `Collection`. C'est le plus petit dénominateur commun, une collection rassemble simplement un nombre d'éléments. Certains types de collections autoriseront des doublons, pas d'autres, certains types sont ordonnés. On retrouve dans cette interface des méthodes de base pour parcourir, ajouter, enlever des éléments.
— `Set` : cette interface représente un *ensemble* au sens mathématique, et donc, ce type de collection n'admet *aucun* doublon.
— `List` : cette interface représente une *séquence* d'éléments. Ainsi, l'ordre dans lequel on ajoute ou on enlève des éléments est important. On peut avoir des doublons dans la séquence.
— `Queue` : cette interface représente une *file d'attente*. Dans une file d'attente, l'élément qui est en tête est particulièrement important. En fait si important qu'on peut avoir l'image suivante d'une file d'attente : il y l'élément en tête et il y a les éléments qui suivent, dont on ne préocuppe pas. On a généralement deux types de file d'attente : celle ou le premier entré est en tête de file et celui ou le dernier entré est celui en tête de file. L'ordre dans lequel les éléments sont ajoutés ou enlevés est important et il peut y avoir des doublons.
— `Deque` : cette interface ressemble aux files d'attente, mais les éléments importants sont les éléments en tête et en queue.
— `Map` : cette interface représente une relation binaire (surjective) : chaque élément est associé à une clé et chaque clé est unique (mais on peut avoir des doublons pour les éléments).
— `SortedSet` est la version ordonnée d'un ensemble
— `SortedMap` est la version ordonnée d'une relation binaire ou les *clés* sont ordonnées.

On introduira de la notion d'ordre entre des objets plus tard dans ce document, mais elle sera vu pendant le cours *Programmation Java Avancée*.

Notez que ces interfaces sont génériques, i.e. on peut leur donner un paramètre pour indiquer qu'on a une collection de `Gaulois`, de `Integer`, de `String`, etc...

2.1 Méthodes de l'interface Collection

Toutes les classes qui implémentent l'interface `Collection` devront redéfinir un certain nombre de méthodes parmis lesquelles on retrouve des méthodes pour l'ajout, le retrait d'éléments, des méthodes pour savoir si un ou des éléments sont présents dans la collection, une méthode (`size`) pour connaitre le nombre d'éléments contenus dans la collection. On ne va pas expliquer en détail chaque méthode, mais vous trouverez ci-dessous une liste (non exhaustive) des méthodes de cette interface. Notez la présence de certaines méthodes génériques. On va expliquer dans la section suivante l'utilité de la méthode `iterator()`.

— `boolean add(E e)`
— `void clear()`
— `boolean contains(Object o)`
— `boolean equals(Object o)`
— `boolean isEmpty()`

— `Iterator<E> iterator()`
— `boolean remove(Object o)`
— `int size()`
— `Object[] toArray()`

2.2 Parcourir une collection

Il y a deux moyens de parcourir une collection : soit en utilisant une boucle « pour chaque élément », qui offre une version différente de la boucle `for` vue précédemment, soit en utilisant un objet appelé `iterator`.

En utilisant la généricité, le compilateur va connaître le type des éléments qui sont contenus dans la collection. Une collection étant un ensemble d'éléments, JAVA propose une façon simple d'accéder à chacun des éléments. Dans l'exemple ci-dessous, on a une collection `maCollection` qui contient des objets de type `E`. On va accéder à chaque élément de la collection `maCollection` en utilisant le mot-clé `for`, chaque élément sera stocké dans une variable `<nom>` de type `E` (évidemment).

```
1 Collection<E> maCollection ;
2 ...
3 for (E <nom> : maCollection)
4     // block d'instructions
```

Par exemple, dans l'exemple suivant, on parcours une liste de `Personnages` et on appelle la méthode `presentation()`.

```
1 LinkedList<Personnage> villageois = new LinkedList<Personnage>() ;
2 villageois.add(new Gaulois("Ordralfabétix")) ;
3 villageois.add(new Gaulois("Abraracourcix")) ;
4 villageois.add(new Gaulois("Assurancetourix")) ;
5 villageois.add(new Gaulois("Cétautomatix")) ;
6 for (Personnage p : villageois)
7     p.presentation() ;
8
```

Une autre solution est d'utiliser un objet dédié au parcourt d'éléments dans une collection : un objet qui implémente l'interface `Iterator`. Pour obtenir cet objet, on peut appeler la méthode `iterator()` qui se trouve dans l'interface `Collection`. L'interface `Iterator`, contient les deux méthodes suivantes :

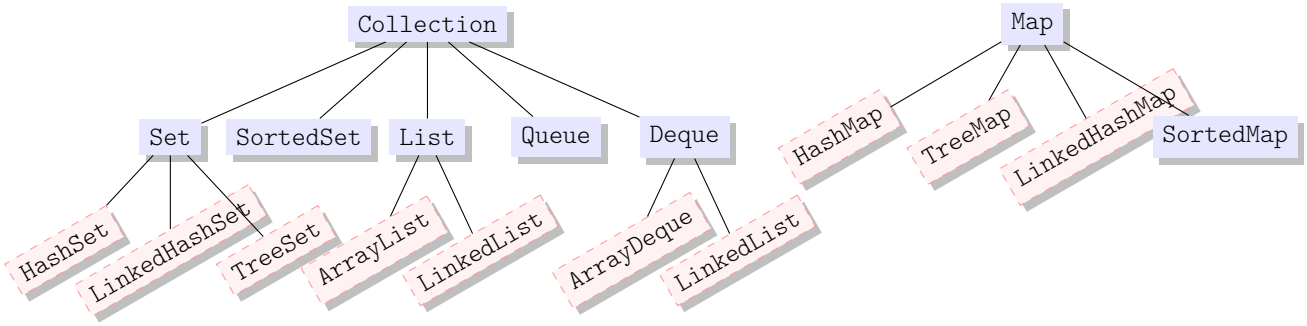
— `hasNext()` retourne un `boolean` qui indique s'il reste des éléments à visiter,
— `next()` donne accès à l'élément suivant, et `remove` permet d'enlever l'élément de la collection.

```
1 public interface Iterator<E> {
2     boolean hasNext() ;
3     E next() ;
4     void remove() ; //optional
5 }
```

En fait, l'utilisation d'une boucle « pour chaque »masque l'utilisation d'un `Iterator`. En général, l'utilisation de la boucle « pour chaque »est la plus simple. Vous devez utiliser un `Iterator` lorsque vous voulez enlever un élément ou lorsque vous voulez parcourir plusieurs collections en parallèle.

2.3 Implémentations

Pour chacune des interfaces, il existe plusieurs *implémentations*. Le diagramme ci-dessous indique les principales implémentations. Certaines sont basées sur des tableaux pour stocker les éléments (`ArrayList`, `ArrayDeque`), d'autres sur des arbres (`TreeSet`, `TreeMap`), d'autres avec des tables de Hash (`HashSet`, `HashMap`). Chaque implémentation a ses propres propriétés (avec ses propres avantages et défauts). Mieux vaut donc lire la documentation pour savoir quelle implémentation est plus appropriée à chaque utilisation. Pour des utilisations « simples » (où les collections ne contiennent pas un trop grand nombre d'objets) on ne verra guère de différences entre les implémentations. Nous ne prendrons pas le temps ici de comparer toutes ces implémentations.



2.4 Ordre

La *classe* `Collections` (à ne pas confondre avec l'interface `Collection`) est une classe qui contient beaucoup de méthodes statiques pour manipuler des collections passées en paramètres. Par exemple, il existe une méthode `sort(List<T> maListe)` qui va trier une liste `maListe`. Si la liste contient des dates, la méthode va trier en ordre chronologique, si la liste contient des `String`, la liste sera trier par ordre lexicograhique. Mais si vous voulez trier des `Gaulois` par quantité de sangliers consommés par an, vous pourrez aussi facilement utiliser la méthode `sort` ! Derrière tout cela, une nouvelle interface est utilisée : l'interface `Comparable`.

Cette interface ne contient qu'une seule méthode : la méthode `public int compareTo(T o)`. Cette méthode retourne un entier négatif si l'objet est plus petit que l'objet passé en paramètre, zéro s'ils sont égaux, et un entier positif si l'objet est plus grand que l'objet passé en paramètre.

Si vous regardez la documentation des classes `String`, `Integer`, `Double`, `Date`, `GregorianCalendar` et beaucoup d'autres, vous verrez que ces classes implémentent toutes l'interface `Comparable`. Ils ont donc une implémentation de la méthode `compareTo`.

Vous pouvez donc spécifier votre propre méthode `compareTo`. Par exemple pour la classe `Gaulois`, on pourrait écrire

```java
public class Gaulois extends Personnage implements Comparable<Gaulois>{
   String nom ;
   int quantiteSanglier ;
     ...

   public int compareTo(Gaulois ixis) {
     return this.quantiteSanglier - ixis. quantiteSanglier ;
   }
}
```

Attention, si vous essayez de trier une liste qui n'implémente pas l'interface `Comparable`, vous provoquerez une erreur (`ClassCastException`).