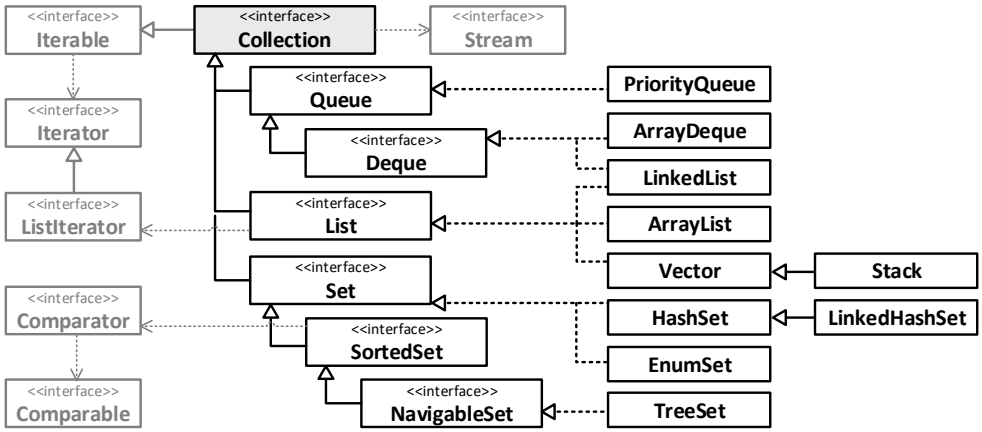
Chapitre 4

Compléments d'API

4.1 Collections

Les collections usuelles sont fournies par les éléments du package `java.util`. Elles constituent une hiérarchie dont la racine est l'interface `Collection` :



4.1.1 Interface `Collection`

L'interface `Collection` rassemble des éléments d'un même type, *a priori* sans contrainte d'unicité, et *a priori* sans interdiction d'y intégrer la référence `null`. Ses méthodes sont réparties en quatre catégories :

```
public interface Collection<E> extends Iterable<E> { // extrait :
 // (1) Opérations de consultation des éléments :
   int      size();                       // nombre d'éléments
   boolean  isEmpty();                    // test de collection vide
   boolean  contains(Object o);           // test d'appartenance
   boolean  containsAll(Collection<?> c); // test d'inclusion
   <T> T[]  toArray(T[] a); // collecte des éléments dans un tableau
 // (2) Opérations de mise à jour de la collection :
   boolean add(E e);                            // this union {e}
   boolean addAll(Collection<? extends E> c); // this union c
   boolean remove(Object o);                    // this - {o}
   boolean removeAll(Collection<?> c);          // this - c
   void clear();                                // this - this
   boolean retainAll(Collection<?> c);          // this inter c
   default boolean removeIf(Predicate<? super E> filter) {/*...*/}
 // (3) Opérations de parcours héritées de Iterable :
   Iterator<E> iterator(); // création d'un itérateur sur this
   default void forEach(Consumer<? super E> action) {/*...*/}
 // (4) Opération de parcours par streams :
   default Stream<E> stream() {/*...*/} // création d'un stream
}
```

Exemple d'utilisation :

```
Collection<Integer> c = /* création d'une collection vide */ ;
c.add(5); c.add(8); c.add(-1); c.add(-8); c.add(2); c.add(-3);
// c contient ( 5  8 -1 -8  2 -3 )
c.removeIf(i -> i<0); // c contient ( 5  8  2 ) ; résultat : true
```

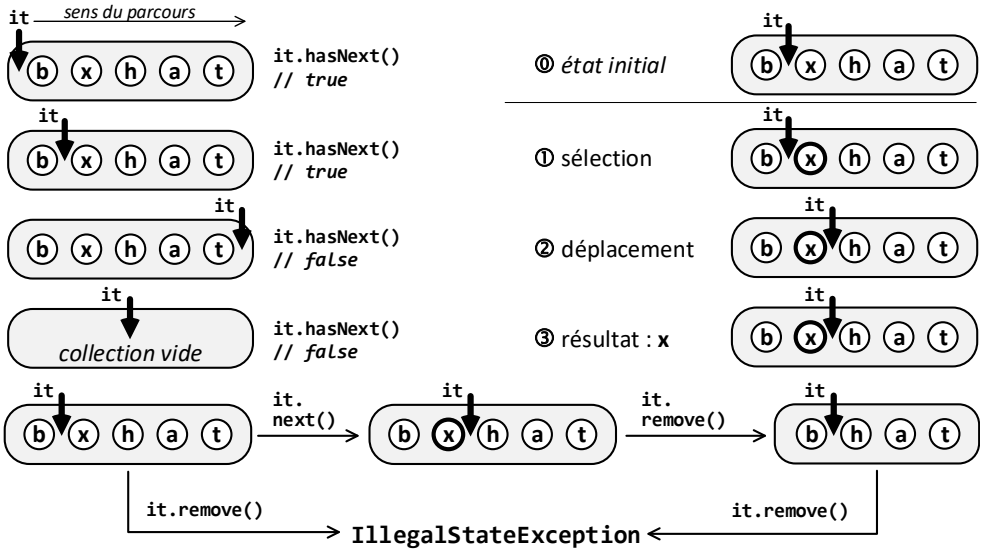
4.1.2 Parcours de collection par itérateur

La méthode `iterator` renvoie un objet de type `Iterator` qui représente un *itérateur* positionné *au début* de la collection :

```
public interface Iterator<E> { // extrait :
   boolean hasNext(); // indique s'il existe un prochain élément
   E next(); // renvoie le prochain élément et déplace l'itérateur
   void remove(); // supprime le dernier élément renvoyé par next
}
```

Définition Un itérateur est une structure associée à une collection, qui permet de la parcourir séquentiellement, et qui permet certaines mises à jour. Pour les n éléments d'une collection, il y a $n+1$ positions possibles pour un itérateur associé.

Utilisation Les méthodes `hasNext` et `next` permettent de déplacer l'itérateur. La méthode `remove` permet de supprimer un élément. Illustration :



Exemple, parcours de collection avec suppression des nombres négatifs ou pairs :

```
Collection<Integer> c = /* ... */ ;
for (Iterator<Integer> it = c.iterator(); it.hasNext(); ) {
   Integer i = it.next(); // un seul appel à next par boucle !
   if (i<0 || i%2==0) it.remove();
}
```

4.1.3 Manipulation de collection par *streams*

Principes généraux Un *stream* est une *vue* sur des données appelées *source* (*e.g.* une collection) qui ne stocke pas de données, qui ne permet pas de modifier la source, et qui ne permet de lire qu'une fois chaque donnée de la source.

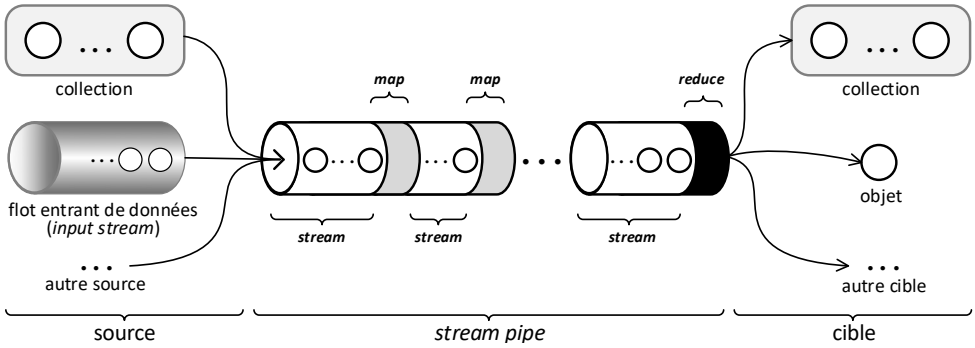
Opérations Deux types d'opérations peuvent être appliquées sur un *stream* :

- les opérations intermédiaires (*map*) qui renvoient un nouveau *stream*
- les opérations terminales, ou réductions (*reduce*), qui produisent une valeur

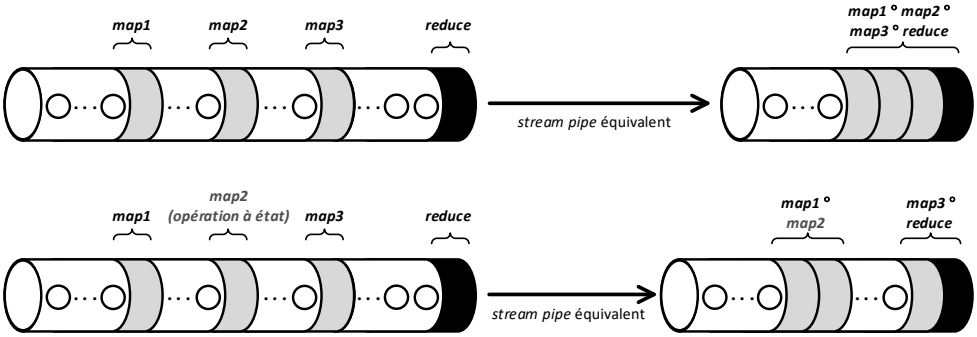
Types de *map* Les opérations à état (*stateful*) nécessitent la connaissance de toutes les valeurs d'un *stream* pour produire un nouveau *stream* (*e.g.* tri). Les autres opérations sont dites sans état (*stateless*, *e.g.* incrémentation).

Types de *reduce* Une réduction simple produit une valeur (*e.g.* somme d'entiers). Une réduction mutable produit un conteneur modifiable de valeurs (*e.g.* concaténation dans un `StringBuffer`).

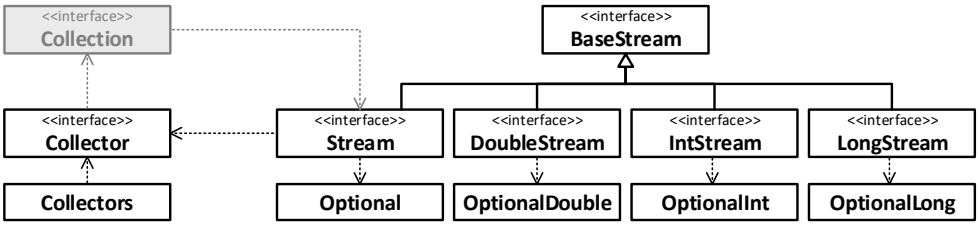
Évaluation paresseuse Un *stream pipe* est constitué d'une série de *map* et d'un *reduce* appliqué à un *stream* :



Les *map* ne sont évaluées qu'à l'évaluation du *reduce*, ce qui permet de les combiner et ainsi d'éviter des calculs intermédiaires inutiles :



API Les *streams* sont représentés par l'interface `BaseStream` et ses dérivées. Les classes `Optional` désignent des résultats de réductions calculables ou non. L'interface `Collector` permet de construire des résultats de réductions mutables :



Créations de *streams* Pour créer un *stream*, on peut utiliser la méthode `stream` de l'interface `Collection`, mais aussi la méthode `stream` de la classe `Arrays`, certaines méthodes de la classe `Random`, ou encore les méthodes statiques de l'interface `Stream`.

Manipulation de *stream* Les principales méthodes de `Stream` et `BaseStream` sont illustrées dans les exemples suivants. Le commentaire qui préfixe leur appel représente un *stream* d'entier. La façon dont il a été obtenu n'a pas d'importance :

```
(/* 5, -8, -1, -8 */).filter( i -> i<0 );       // -8, -1, -8
(/* 5, -8, -1, -8 */).map( i -> i+2 );          //  7, -6,  1, -6
(/* [1, 2], [3, 4]*/).flatMap(Arrays::stream); //  1,  2,  3,  4
(/* 5, -8, -1, -8 */).distinct();               //  5, -8  -1
(/* 5, -8, -1, -8 */).sorted();                 // -8, -8, -1,  5
(/* 5, -8, -1, -8 */).limit(2);                 //  5, -8
```

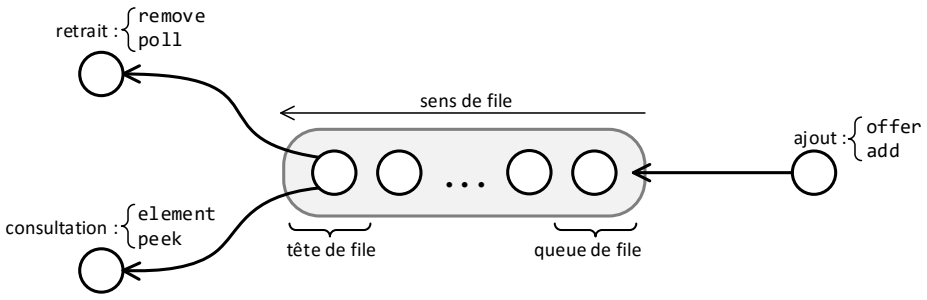
```
(/* 5, -8, -1, -8 */).anyMatch( i -> i<0 );                    // true
(/* 5, -8, -1, -8 */).reduce(0, (t,u)->t+u, (u1,u2) -> u1+u2));//-12
(/* 5, -8, -1, -8 */).reduce(0, (u1,u2) -> u1+u2));            //-12
(/* 5, -8, -1, -8 */).reduce((u1,u2) -> u1+u2)).get();         //-12
(/* 5, -8, -1, -8 */).min(Integer::compareTo).get();           // -8
```

```
Collection<Integer> valeurs = (/* 5, -8, -1, -8 */)
      .collect(ArrayList::new, Collection::add, Collection::addAll);
// Variante :
valeurs = (/* 5, -8, -1, -8 */).collect(Collectors.toList());
// Dans les deux cas, valeurs contient : [5, -8, -1, -8]
```

4.1.4 Interfaces dérivées de `Collection` et mises en œuvre

Interface `Queue` Représente une *file d'attente* (*i.e.* une structure munie d'un point d'accès en écriture nommé *queue* et d'un point d'accès en lecture nommé *tête*) :

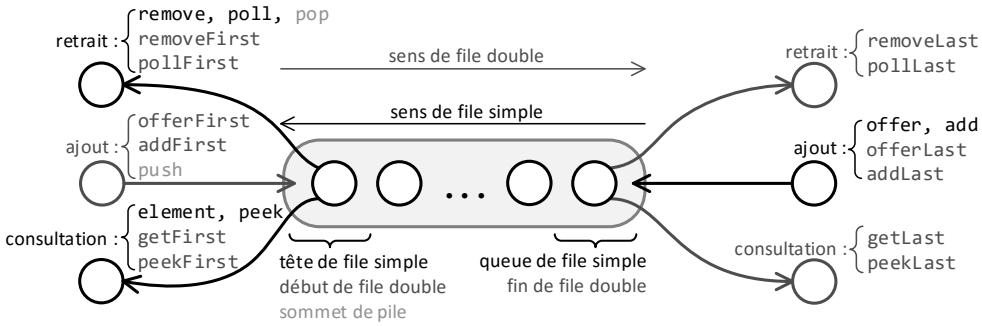
```
public interface Queue<E> extends Collection<E> {
   boolean offer(E e); // ajout en queue de file (alias de add)
   E remove();     // suppression de la tête de file, avec exception
   E poll();       // suppression de la tête de file
   E element();    // consultation de la tête de file, avec exception
   E peek();       // consultation de la tête de file
}
```



Interface `Deque` Représente une *file à double entrée*, dont les deux points d'accès permettent indifféremment d'ajouter ou de supprimer des éléments :

```
public interface Deque<E> extends Queue<E> {
   void addFirst(E e);      // ajout au début (pas de résultat)
   void addLast(E e);       // ajout en fin (pas de résultat)
   boolean offerFirst(E e); // ajout au début
   boolean offerLast(E e);  // ajout en fin (alias : add, offer)
   E removeFirst(); // suppression du 1er élément (alias remove)
   E removeLast();  // suppression du dernier élément
   E pollFirst();   // suppression du 1er élément (alias poll)
   E pollLast();    // suppression du dernier élément
   E getFirst();  // consultation du 1er élément (alias element)
   E getLast();   // consultation du dernier élément
   E peekFirst(); // consultation du 1er élément (alias peek)
   E peekLast();  // consultation du dernier élément
   // Opérations spécifiques de suppression d'un élément :
   boolean removeFirstOccurrence(Object o); // par le début
   boolean removeLastOccurrence(Object o);  // par la fin
   // Opérations de pile (sommet de pile = début de file) :
   void push(E e); // ajout au sommet de pile (alias addFirst)
   E pop(); // retrait sommet de pile (alias remove, removeFirst)
}
```

Une file à double entrée peut être utilisée comme une *pile* avec les méthodes push (ajout), pop (retrait) et peek (consultation). Ces trois méthodes sont liées à un même point d'accès, la *tête de file*, que l'on peut donc assimiler ici au *sommet* :



Interface `Set` Représente des *ensembles d'objets* qui ne peuvent pas contenir plusieurs occurrences d'un *même* objet (selon `equals`) :

```
public interface Set<E> extends Collection<E> { }
```

Interface `SortedSet` Utilise une *relation d'ordre* définie soit par l'interface `Comparable` (relation d'ordre *naturelle*, intrinsèquement liée aux objets d'un

certain type T), soit par l'interface Comparator (relation d'ordre quelconque sur des objets d'un certain type T) :

```
public interface Comparable<T> {
  int compareTo(T o);   /*(this.compareTo(o)  < 0) <=> (this< o)*/ }
```

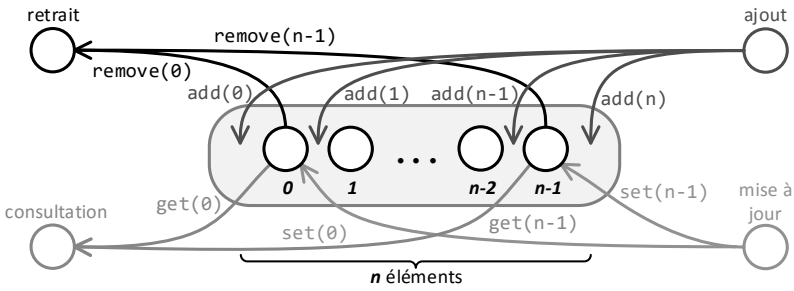
```
public interface Comparator<T> {
  int compare(T x, T y);/*(this.compare(x, y) < 0) <=> (x   < y)*/ }
```

```
public interface SortedSet<E> extends Set<E> {
   Comparator<? super E> comparator(); // comparateur utilisé
   SortedSet<E> subSet(E x, E y); // les éléments dans [x .. y[
   SortedSet<E> headSet(E e); // les éléments plus petits que e
   SortedSet<E> tailSet(E e); // les éléments plus grands que e
   E first(); // le plus petit élément
   E last();  // le plus grand élément
}
```

Interface NavigableSet Hérite de SortedSet et introduit des méthodes de parcours et d'extraction supplémentaires.

Interface List Représente une *liste indexée i.e.* une liste dont les n éléments sont désignés par des *indices* allant de 0 à $n-1$:

```
public interface List<E> extends Collection<E> {
   E get(int index);            // élément d'indice index
   E set(int index, E element); // mise à jour de l'élément index
   E remove(int index);         // suppression de l'élément index
   void add(int i, E element);  // insertion à la position i
   boolean addAll(int i, Collection<? extends E> c);
   List<E> subList(int i, int j); // extraction de sous-liste
   int indexOf(Object o);     // première occurrence de o
   int lastIndexOf(Object o); // dernière occurrence de o
   default void replaceAll(UnaryOperator<E> op) {/* ... */} // maj
   default void sort(Comparator<? super E> c)   {/* ... */} // tri
   ListIterator<E> listIterator();          // positionné au début
   ListIterator<E> listIterator(int index); // positionné sur index
}
```



Exemples d'utilisation des listes :

```
List<Integer> ls = /* création d'une liste vide */ ;
ls.add(1); ls.add(8); ls.add(5); ls.add(8); // ls : ( 1 8 5 8 )
ls.replaceAll( i -> (i%2==1 ? i+1 : i) );   // ls : ( 2 8 6 8 )
ls.replaceAll(Math::incrementExact);        // ls : ( 3 9 7 9 )
ls.sort( (i,j) -> i-j );                    // ls : ( 3 7 9 9 )
```

Interface `ListIterator` Itérateur de liste qui permet le parcours dans les deux sens, la mise à jour et l'insertion d'éléments, et la récupération d'indice :

```
public interface ListIterator<E> extends Iterator<E> {
   // méthodes hasNext, next et remove héritées de Iterator
   boolean hasPrevious(); // indique s'il existe un élément *avant*
   E previous();          // renvoie l'élément *avant*, et déplace
   int nextIndex();       // renvoie l'indice de next
   int previousIndex();   // renvoie l'indice de previous
   void set(E e);         // met à jour le dernier next OU previous
   void add(E e);         // insère un élément
}
```