Les collections Java

Plan

1. Introduction
2. Interfaces et implémentations
3. Algorithmes
4. Conclusion

Notion de *Collection*

Définition

- Une *collection* est un regroupement structuré de données
- En Java : c'est un **objet qui structure d'autres objets**
- Exemple : `ArrayList<E>, HashMap<K,V>`, etc.

L'API des collections Java en 2 packages

- `java.util` : les collections essentielles et le cadriciel associé
- `java.util.concurrent` : programmation concurrente

Exemples de modélisation

- Une main dans un jeu de cartes ⇒ ensemble de `n` cartes
- Un répertoire téléphonique ⇒ tableau associatif `nom` → `numéro`

Les *collections framework* orientés objet

Des interfaces

- Définissent les opérations possibles sur un type de collection
- E.g. `List<E>`, `Map<K,V>`, `Queue<E>`...

Des implémentations

- Les classes qui implémentent les interfaces
- E.g. `ArrayList<E>`, `HashMap<K,V>`, `ArrayDeque<E>`...

Des algorithmes

- Méthodes polymorphes manipulant les collections
- E.g. `Collections.sort(List<T> list)`

Exemples : ▸ STL en C++ ▸ collections SmallTalk et hors POO ▸ Python ...

Bénéfices

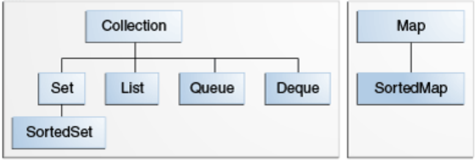
Utilisation

- Programmation **sans effort** : pas de code bas niveau, pas d'algo
- **Rapidité et qualité du code** : implémentations éprouvées

Standardisation

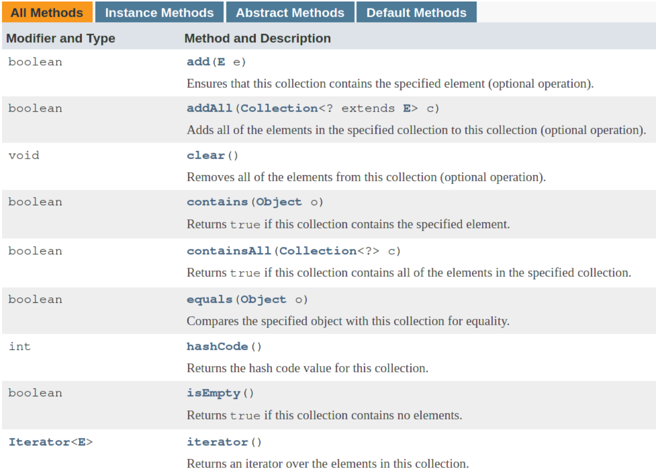
- **Interopérabilité** entre APIs développées indépendamment
- Facilite la prise en main de nouvelles APIs
- Facilite la création de nouvelles APIs
- Promeut la **réutilisation** des nouvelles collections développées

Les interfaces



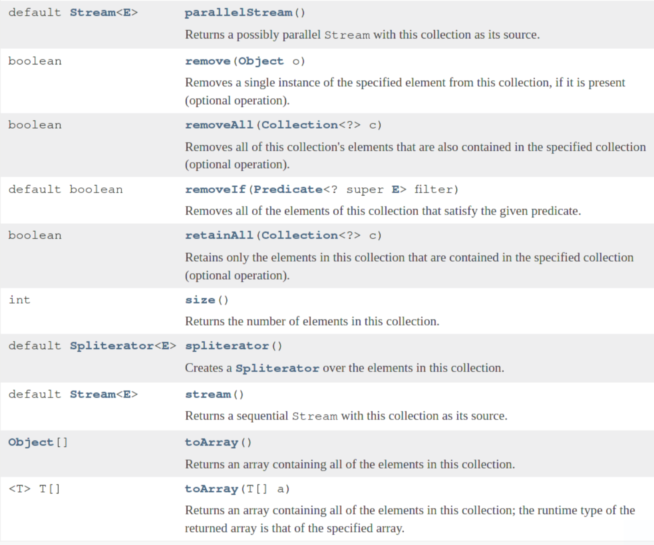
java.util.Collection<E>

Définit toutes les opérations possibles sur tout type de collection :



Modifier and Type	Method and Description
boolean	add(E e) Ensures that this collection contains the specified element (optional operation).
boolean	addAll(Collection<? extends E> c) Adds all of the elements in the specified collection to this collection (optional operation).
void	clear() Removes all of the elements from this collection (optional operation).
boolean	contains(Object o) Returns true if this collection contains the specified element.
boolean	containsAll(Collection<?> c) Returns true if this collection contains all of the elements in the specified collection.
boolean	equals(Object o) Compares the specified object with this collection for equality.
int	hashCode() Returns the hash code value for this collection.
boolean	isEmpty() Returns true if this collection contains no elements.
Iterator<E>	iterator() Returns an iterator over the elements in this collection.

java.util.Collection<E>



default Stream<E>	parallelStream() Returns a possibly parallel Stream with this collection as its source.
boolean	remove(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	removeAll(Collection<?> c) Removes all of this collection's elements that are also contained in the specified collection (optional operation).
default boolean	removeIf(Predicate<? super E> filter) Removes all of the elements of this collection that satisfy the given predicate.
boolean	retainAll(Collection<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	size() Returns the number of elements in this collection.
default Spliterator<E>	spliterator() Creates a Spliterator over the elements in this collection.
default Stream<E>	stream() Returns a sequential Stream with this collection as its source.
Object[]	toArray() Returns an array containing all of the elements in this collection.
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

java.util.Collection<E> remarques

Générique

- Nécessite de spécifier le type `E` des objets contenus

implémente `java.lang.Iterable<T>`

- définit des méthodes permettant une itération sur la collection

Certaines méthodes sont documentées *optional*

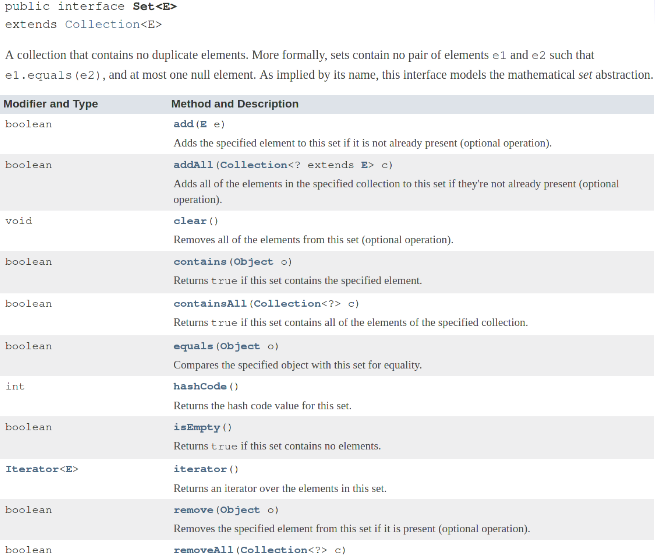
- Évite une exposition du nombre d'interfaces dans l'API
- Il existe des collections *immutable, fixed-size, append-only*…
- Elles ne permettent donc pas toutes certaines opérations
- ⇒ `java.lang.UnsupportedOperationException`

java.util.Collection<E>
Sous-types

Types de Collection aux comportements spécialisés

- Set<E> : collection non ordonnée d’objets non égaux
- SortedSet<E> : collection triée d’objets non égaux
- List<E> : collection d’objets indexés
- Queue<E> : file d’objets à traiter, e.g. en mode FIFO
- Deque<E> : file où on peut traiter le début ou la fin (e.g. LIFO)

Interface java.util.Set<E>



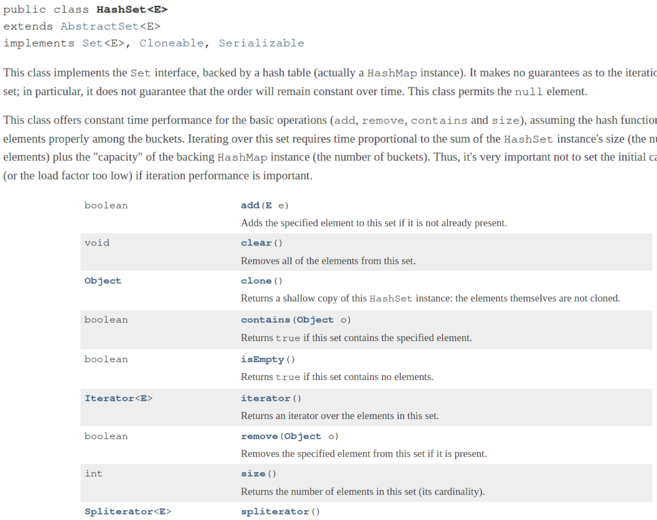
```
public interface Set<E>
extends Collection<E>
```

A collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical *set* abstraction.

Modifier and Type	Method and Description
boolean	**add**(E e) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(Object o) Returns true if this set contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this set contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this set for equality.
int	**hashCode**() Returns the hash code value for this set.
boolean	**isEmpty**() Returns true if this set contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present (optional operation).
boolean	**removeAll**(Collection<?> c)

Exemple d'implémentation de `Set<E>`

`java.util.HashSet<E>`



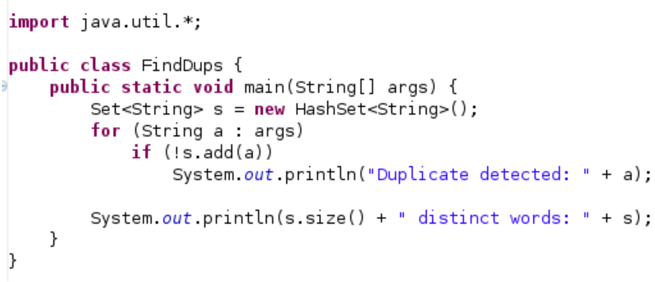
```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

This class implements the `Set` interface, backed by a hash table (actually a `HashMap` instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the `null` element.

This class offers constant time performance for the basic operations (`add`, `remove`, `contains` and `size`), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the `HashSet` instance's size (the number of elements) plus the "capacity" of the backing `HashMap` instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

`boolean`	**`add`**`(E e)` Adds the specified element to this set if it is not already present.
`void`	**`clear`**`()` Removes all of the elements from this set.
`Object`	**`clone`**`()` Returns a shallow copy of this `HashSet` instance: the elements themselves are not cloned.
`boolean`	**`contains`**`(Object o)` Returns `true` if this set contains the specified element.
`boolean`	**`isEmpty`**`()` Returns `true` if this set contains no elements.
`Iterator<E>`	**`iterator`**`()` Returns an iterator over the elements in this set.
`boolean`	**`remove`**`(Object o)` Removes the specified element from this set if it is present.
`int`	**`size`**`()` Returns the number of elements in this set (its cardinality).
`Spliterator<E>`	**`spliterator`**`()`

Exemple d'utilisation de `java.util.HashSet<E>`



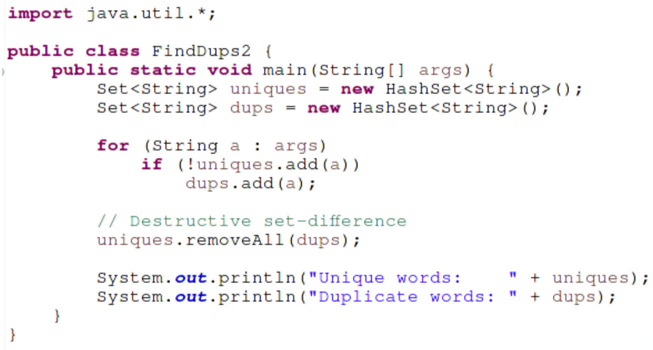
```
import java.util.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplicate detected: " + a);

        System.out.println(s.size() + " distinct words: " + s);
    }
}
```

```
java FindDups i came i saw i left
```

⇒

```
4 distinct words: [left, came, saw, i]
```

Exemple d’utilisation de `java.util.HashSet<E>`



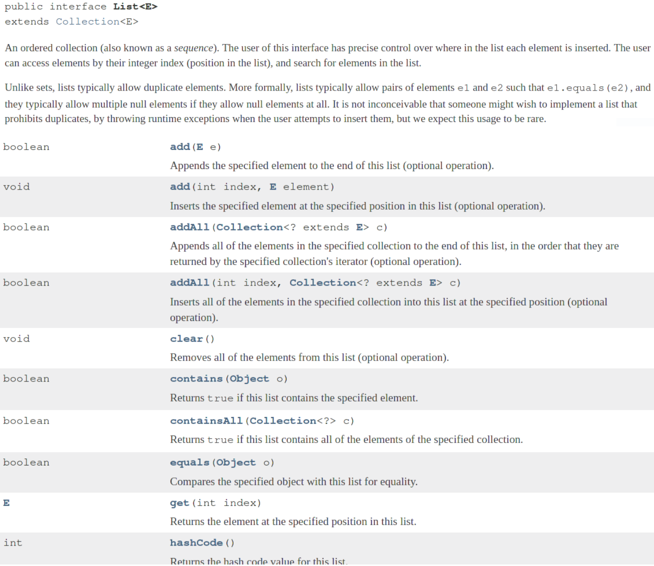
```java
import java.util.*;

public class FindDups2 {
    public static void main(String[] args) {
        Set<String> uniques = new HashSet<String>();
        Set<String> dups = new HashSet<String>();

        for (String a : args)
            if (!uniques.add(a))
                dups.add(a);

        // Destructive set-difference
        uniques.removeAll(dups);

        System.out.println("Unique words:    " + uniques);
        System.out.println("Duplicate words: " + dups);
    }
}
```

```
java FindDups2 i came i saw i left
```

⇒

```
Unique words: [left, saw, came]
Duplicate words: [i]
```

Interface `java.util.List<E>`



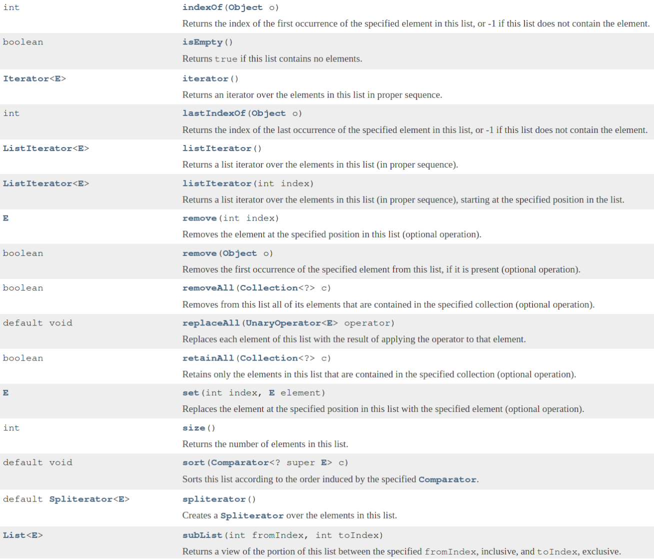
```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements `e1` and `e2` such that `e1.equals(e2)`, and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

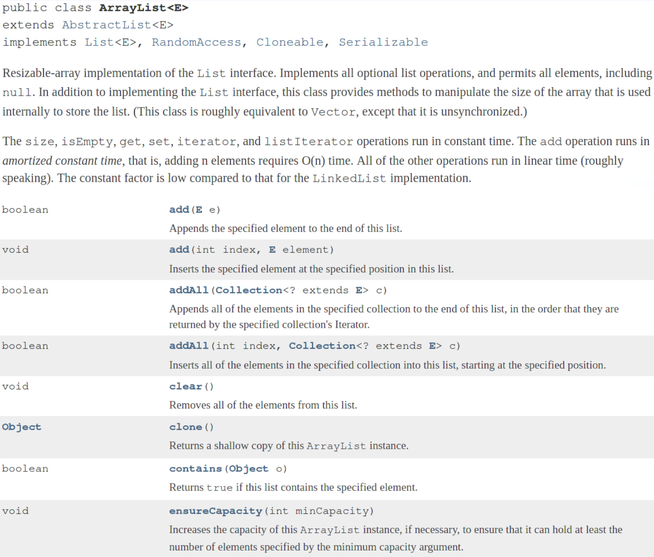
`boolean`	**`add`**`(E e)` Appends the specified element to the end of this list (optional operation).
`void`	**`add`**`(int index, E element)` Inserts the specified element at the specified position in this list (optional operation).
`boolean`	**`addAll`**`(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
`boolean`	**`addAll`**`(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
`void`	**`clear`**`()` Removes all of the elements from this list (optional operation).
`boolean`	**`contains`**`(Object o)` Returns `true` if this list contains the specified element.
`boolean`	**`containsAll`**`(Collection<?> c)` Returns `true` if this list contains all of the elements of the specified collection.
`boolean`	**`equals`**`(Object o)` Compares the specified object with this list for equality.
`E`	**`get`**`(int index)` Returns the element at the specified position in this list.
`int`	**`hashCode`**`()` Returns the hash code value for this list.

Interface `java.util.List<E>`



int	indexOf(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	isEmpty() Returns true if this list contains no elements.
Iterator<E>	iterator() Returns an iterator over the elements in this list in proper sequence.
int	lastIndexOf(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	listIterator() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	listIterator(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	remove(int index) Removes the element at the specified position in this list (optional operation).
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	removeAll(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
default void	replaceAll(UnaryOperator<E> operator) Replaces each element of this list with the result of applying the operator to that element.
boolean	retainAll(Collection<?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	size() Returns the number of elements in this list.
default void	sort(Comparator<? super E> c) Sorts this list according to the order induced by the specified Comparator.
default Spliterator<E>	spliterator() Creates a Spliterator over the elements in this list.
List<E>	subList(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

Implémentation `java.util.ArrayList<E>`



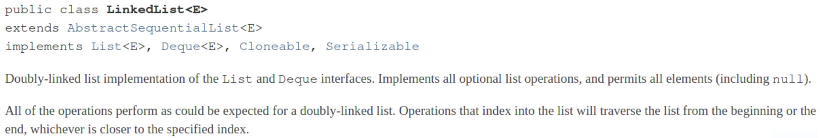
```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

`boolean`	**`add`**`(E e)` Appends the specified element to the end of this list.
`void`	**`add`**`(int index, E element)` Inserts the specified element at the specified position in this list.
`boolean`	**`addAll`**`(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
`boolean`	**`addAll`**`(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list, starting at the specified position.
`void`	**`clear`**`()` Removes all of the elements from this list.
`Object`	**`clone`**`()` Returns a shallow copy of this `ArrayList` instance.
`boolean`	**`contains`**`(Object o)` Returns `true` if this list contains the specified element.
`void`	**`ensureCapacity`**`(int minCapacity)` Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.

Autre implémentation `java.util.LinkedList<E>`



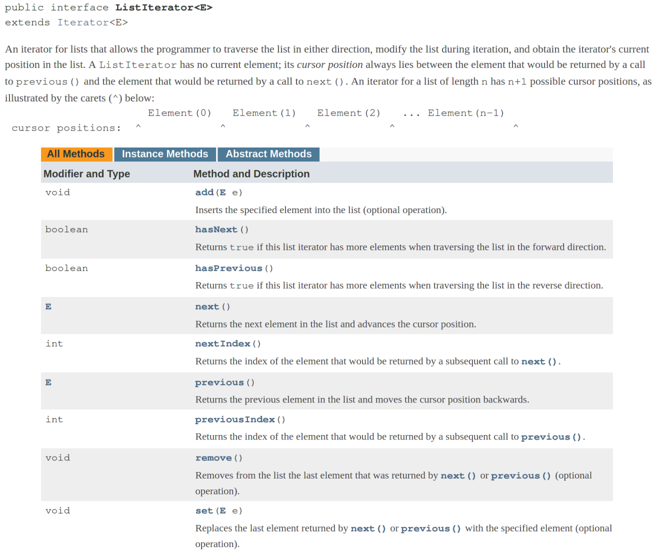
```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Doubly-linked list implementation of the `List` and `Deque` interfaces. Implements all optional list operations, and permits all elements (including `null`).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

⇒ sera, par exemple, plus rapide qu'une `ArrayList` si on travaille fréquemment avec des indices se trouvant à l'intérieur de la liste

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



```
public interface ListIterator<E>
extends Iterator<E>
```

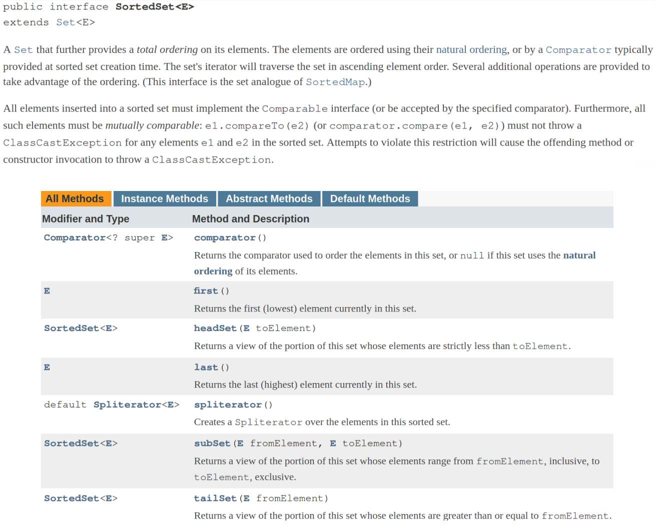
An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list. A `ListIterator` has no current element; its *cursor position* always lies between the element that would be returned by a call to `previous()` and the element that would be returned by a call to `next()`. An iterator for a list of length `n` has `n+1` possible cursor positions, as illustrated by the carets (`^`) below:

```
                      Element(0)   Element(1)   Element(2)   ... Element(n-1)
 cursor positions:  ^            ^            ^            ^                  ^
```

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
`void`	**`add(E e)`** Inserts the specified element into the list (optional operation).
`boolean`	**`hasNext()`** Returns `true` if this list iterator has more elements when traversing the list in the forward direction.
`boolean`	**`hasPrevious()`** Returns `true` if this list iterator has more elements when traversing the list in the reverse direction.
`E`	**`next()`** Returns the next element in the list and advances the cursor position.
`int`	**`nextIndex()`** Returns the index of the element that would be returned by a subsequent call to **`next()`**.
`E`	**`previous()`** Returns the previous element in the list and moves the cursor position backwards.
`int`	**`previousIndex()`** Returns the index of the element that would be returned by a subsequent call to **`previous()`**.
`void`	**`remove()`** Removes from the list the last element that was returned by **`next()`** or **`previous()`** (optional operation).
`void`	**`set(E e)`** Replaces the last element returned by **`next()`** or **`previous()`** with the specified element (optional operation).

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



```
public interface SortedSet<E>
extends Set<E>
```

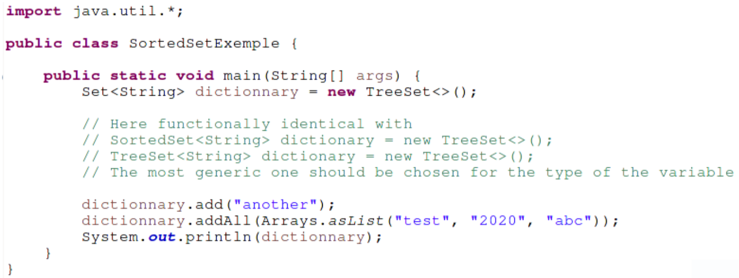
A `Set` that further provides a *total ordering* on its elements. The elements are ordered using their natural ordering, or by a `Comparator` typically provided at sorted set creation time. The set's iterator will traverse the set in ascending element order. Several additional operations are provided to take advantage of the ordering. (This interface is the set analogue of `SortedMap`.)

All elements inserted into a sorted set must implement the `Comparable` interface (or be accepted by the specified comparator). Furthermore, all such elements must be *mutually comparable*: `e1.compareTo(e2)` (or `comparator.compare(e1, e2)`) must not throw a `ClassCastException` for any elements `e1` and `e2` in the sorted set. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a `ClassCastException`.

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
`Comparator<? super E>`	`comparator()` Returns the comparator used to order the elements in this set, or `null` if this set uses the **natural ordering** of its elements.
`E`	`first()` Returns the first (lowest) element currently in this set.
`SortedSet<E>`	`headSet(E toElement)` Returns a view of the portion of this set whose elements are strictly less than `toElement`.
`E`	`last()` Returns the last (highest) element currently in this set.
`default Spliterator<E>`	`spliterator()` Creates a `Spliterator` over the elements in this sorted set.
`SortedSet<E>`	`subSet(E fromElement, E toElement)` Returns a view of the portion of this set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive.
`SortedSet<E>`	`tailSet(E fromElement)` Returns a view of the portion of this set whose elements are greater than or equal to `fromElement`.

Exemple avec `java.util.TreeSet<E>`

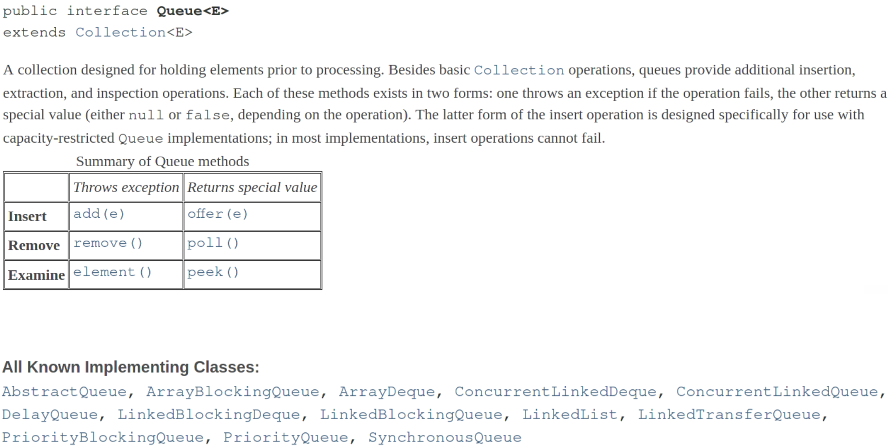


```
import java.util.*;

public class SortedSetExemple {

    public static void main(String[] args) {
        Set<String> dictionnary = new TreeSet<>();

        // Here functionally identical with
        // SortedSet<String> dictionnary = new TreeSet<>();
        // TreeSet<String> dictionary = new TreeSet<>();
        // The most generic one should be chosen for the type of the variable

        dictionnary.add("another");
        dictionnary.addAll(Arrays.asList("test", "2020", "abc"));
        System.out.println(dictionnary);
    }
}
```

⇒

```
[2020, abc, another, test]
```

java.util.Queue<E>



```
public interface Queue<E>
extends Collection<E>
```

A collection designed for holding elements prior to processing. Besides basic `Collection` operations, queues provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either `null` or `false`, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted `Queue` implementations; in most implementations, insert operations cannot fail.

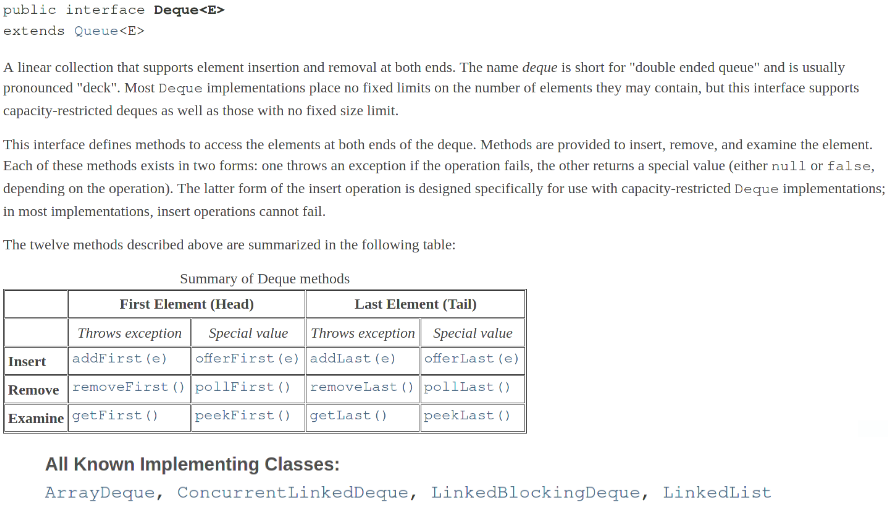
Summary of Queue methods

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

All Known Implementing Classes:

`AbstractQueue`, `ArrayBlockingQueue`, `ArrayDeque`, `ConcurrentLinkedDeque`, `ConcurrentLinkedQueue`, `DelayQueue`, `LinkedBlockingDeque`, `LinkedBlockingQueue`, `LinkedList`, `LinkedTransferQueue`, `PriorityBlockingQueue`, `PriorityQueue`, `SynchronousQueue`

java.util.Deque<E>



```
public interface Deque<E>
extends Queue<E>
```

A linear collection that supports element insertion and removal at both ends. The name *deque* is short for "double ended queue" and is usually pronounced "deck". Most `Deque` implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

This interface defines methods to access the elements at both ends of the deque. Methods are provided to insert, remove, and examine the element. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either `null` or `false`, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted `Deque` implementations; in most implementations, insert operations cannot fail.

The twelve methods described above are summarized in the following table:

Summary of Deque methods

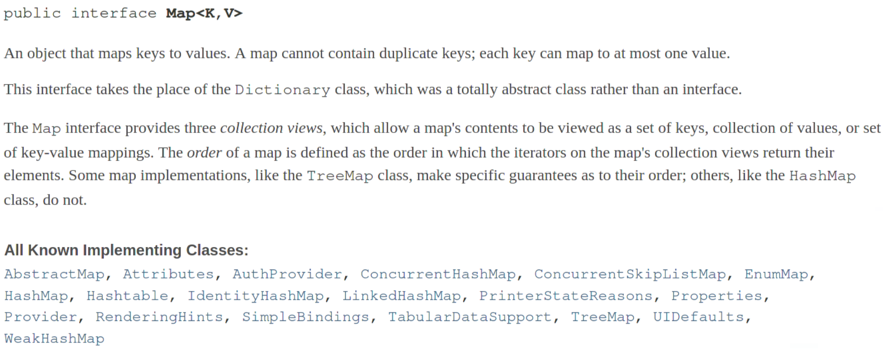
	First Element (Head)		**Last Element (Tail)**	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

All Known Implementing Classes:

`ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque, LinkedList`

java.util.Map<K,V>

Tableau associatif : clé → valeur



```
public interface Map<K,V>
```

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

The `Map` interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

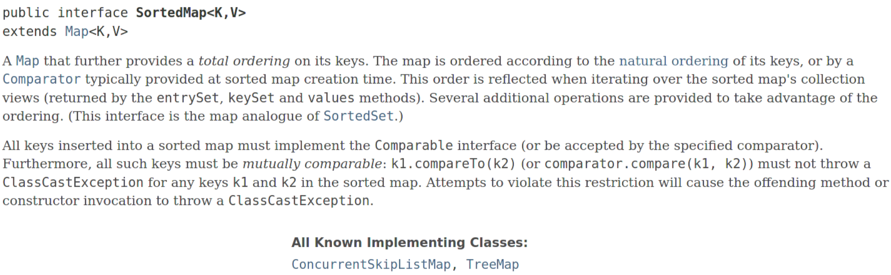
All Known Implementing Classes:

`AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap`

▸ API de Map<K,V>

java.util.SortedMap<K,V>

Tableau associatif trié sur les clés



```
public interface SortedMap<K,V>
extends Map<K,V>
```

A Map that further provides a *total ordering* on its keys. The map is ordered according to the natural ordering of its keys, or by a Comparator typically provided at sorted map creation time. This order is reflected when iterating over the sorted map's collection views (returned by the entrySet, keySet and values methods). Several additional operations are provided to take advantage of the ordering. (This interface is the map analogue of SortedSet.)

All keys inserted into a sorted map must implement the Comparable interface (or be accepted by the specified comparator). Furthermore, all such keys must be *mutually comparable*: k1.compareTo(k2) (or comparator.compare(k1, k2)) must not throw a ClassCastException for any keys k1 and k2 in the sorted map. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a ClassCastException.

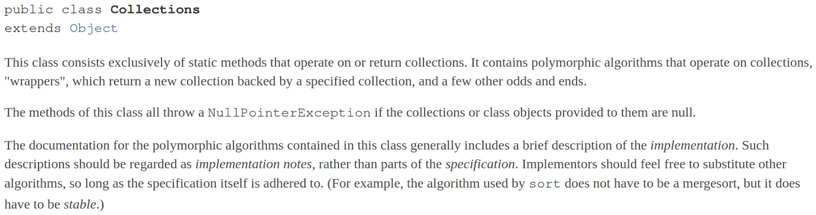
All Known Implementing Classes:

ConcurrentSkipListMap, TreeMap

▸ API de SortedMap<K,V>

La classe `java.util.Collections`

Fournit un grand nombre de méthodes statiques permettant de travailler avec les collections : `max, min, sort, reverse, search`...



```
public class Collections
extends Object
```

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

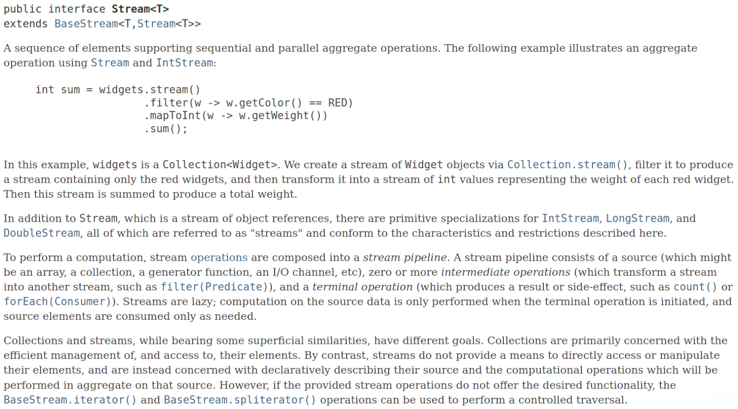
The methods of this class all throw a `NullPointerException` if the collections or class objects provided to them are null.

The documentation for the polymorphic algorithms contained in this class generally includes a brief description of the *implementation*. Such descriptions should be regarded as *implementation notes*, rather than parts of the *specification*. Implementors should feel free to substitute other algorithms, so long as the specification itself is adhered to. (For example, the algorithm used by `sort` does not have to be a mergesort, but it does have to be *stable*.)

▸ `java.util.Collections`

L'interface `java.util.stream.Stream`

Les collections permettent de créer des `Stream` sur elles : une séquence d'éléments sur laquelle on peut appliquer divers traitement, possiblement chaînés.



```
public interface Stream<T>
extends BaseStream<T,Stream<T>>
```

A sequence of elements supporting sequential and parallel aggregate operations. The following example illustrates an aggregate operation using `Stream` and `IntStream`:

```
    int sum = widgets.stream()
                     .filter(w -> w.getColor() == RED)
                     .mapToInt(w -> w.getWeight())
                     .sum();
```

In this example, `widgets` is a `Collection<Widget>`. We create a stream of `Widget` objects via `Collection.stream()`, filter it to produce a stream containing only the red widgets, and then transform it into a stream of `int` values representing the weight of each red widget. Then this stream is summed to produce a total weight.

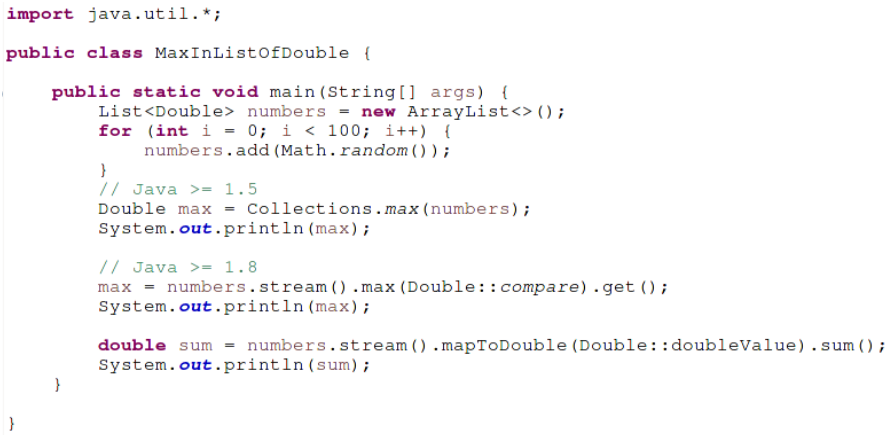
In addition to `Stream`, which is a stream of object references, there are primitive specializations for `IntStream`, `LongStream`, and `DoubleStream`, all of which are referred to as "streams" and conform to the characteristics and restrictions described here.

To perform a computation, stream operations are composed into a *stream pipeline*. A stream pipeline consists of a source (which might be an array, a collection, a generator function, an I/O channel, etc), zero or more *intermediate operations* (which transform a stream into another stream, such as `filter(Predicate)`), and a *terminal operation* (which produces a result or side-effect, such as `count()` or `forEach(Consumer)`). Streams are lazy; computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.

Collections and streams, while bearing some superficial similarities, have different goals. Collections are primarily concerned with the efficient management of, and access to, their elements. By contrast, streams do not provide a means to directly access or manipulate their elements, and are instead concerned with declaratively describing their source and the computational operations which will be performed in aggregate on that source. However, if the provided stream operations do not offer the desired functionality, the `BaseStream.iterator()` and `BaseStream.spliterator()` operations can be used to perform a controlled traversal.

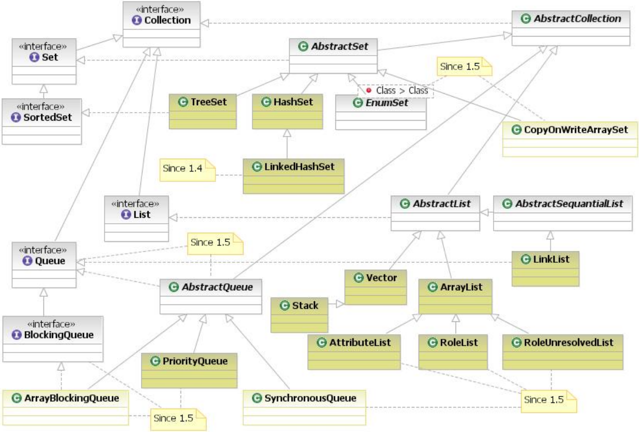
▸ `java.util.stream.Stream`

Exemple de code



```java
import java.util.*;

public class MaxInListOfDouble {

    public static void main(String[] args) {
        List<Double> numbers = new ArrayList<>();
        for (int i = 0; i < 100; i++) {
            numbers.add(Math.random());
        }
        // Java >= 1.5
        Double max = Collections.max(numbers);
        System.out.println(max);

        // Java >= 1.8
        max = numbers.stream().max(Double::compare).get();
        System.out.println(max);

        double sum = numbers.stream().mapToDouble(Double::doubleValue).sum();
        System.out.println(sum);
    }

}
```

Diagramme de classes partiel



▸ Image tirée du Wiki book sur les collections Java

▸ Description complète du cadriciel par Oracle

Conclusion

Philosophie

- Quelque soit le langage, il faut **utiliser les collections prédéfinies**
- **Il faut chercher jusqu'à trouver** celle qui répond parfaitement à vos besoins

Conséquences

- Probablement, **vous ne coderez jamais plus une nouvelle collection**
- **ni d'algorithme** dédié de type filtre, agrégation, mapping, etc.
- Vous écrirez la condition du filtre, la donnée à agréger, etc.
- Dans le cas contraire : vous faites fausse route !

Ce cours est basé sur le tutoriel d'Oracle : ▸ Oracle Java Tutorial