Les Collections

La notion de collection

- Une *collection* est un objet qui regroupe d'autres objets formant un groupe logique :
  - une main de poker -> une collection de cartes
  - un dossier mail -> une collection de lettres
  - un répertoire téléphonique -> une collection de noms associés à des numéros
- Vous connaissez déjà des objets de ce type :
  - ArrayList
  - HashTable
- On peut parler de ***structures de données***

Les collections en informatique

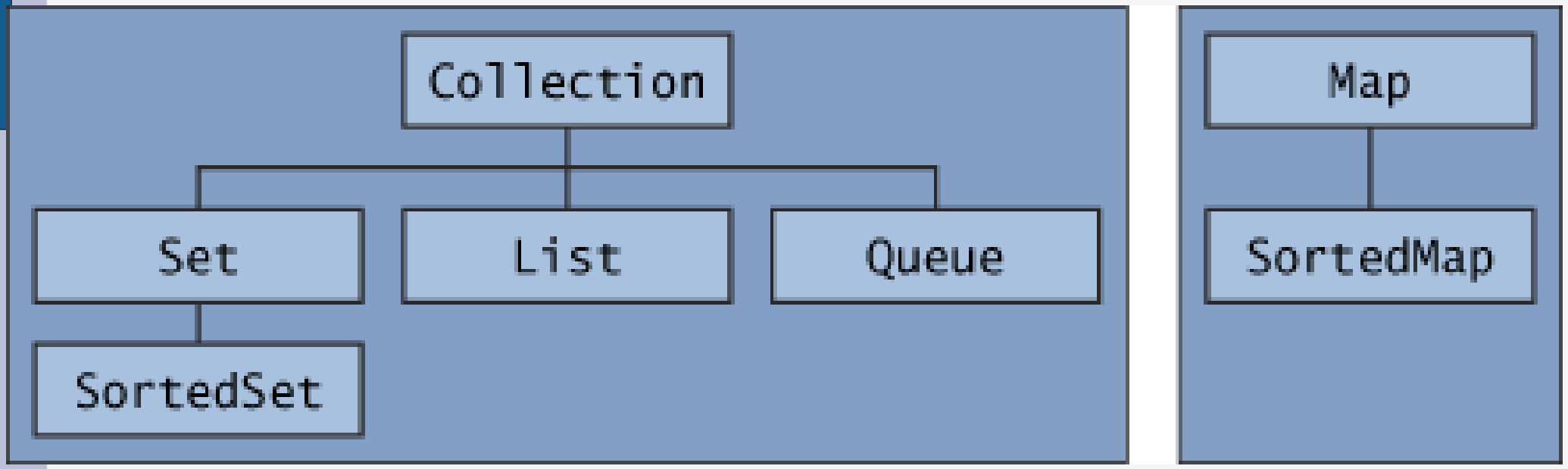
- Un framework informatique manipulant des collections contient :
  - des ***interfaces*** : types de donnée abstraits permettant de manipuler des collections sans connaître le détail de leur implémentation
  - des ***implémentations*** : des ***classes*** concrètes qui implémentent les interfaces : des structures de données réutilisables
  - des ***algorithmes*** : méthodes permettant des opérations utiles sur les collections (chercher, classer, ajouter, etc.). Ils sont polymorphiques : applicables sur différentes implémentations d'une interface
- La STL de C++, la hiérarchie des collections en smalltalk, ...

Bénéfices

- Réduire les efforts de programmation
  - pas de programmation « bas niveau »
- Rapidité et qualité des programmes
  - l'utilisation des collections garantie l'utilisation d'algorithmes éprouvés
- Interopérabilité simplifiée
  - des programmes indépendants manipulant les mêmes structures de données sont beaucoup plus simples à connectés

Les interfaces du JDK

- java.util



Caratéristiques communes

- Toutes les collections du JDK sont génériques :
  - *public interface Collection<E>*
- Pour réduire le nombre des interfaces du JDK, il n'y a pas d'interface spécifique pour chaque variante d'une collection (immuables, taille fixe, ajout uniquement, etc.)
- Ainsi, les opérations définies par une interface sont dites « optionnelles »
  - une implémentation particulière ne fournit pas forcément toutes les opérations d'une interface
  - Si une opération non supportée est invoquée, alors la collection renvoie une *UnsupportedOperationException*

java.util

Interface Collection<E>

Method Summary

boolean	**add**(E o) 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.
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 this collection's elements that are also contained in the specified collection (optional operation).
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.
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

Interface Set<E>

Method Summary

boolean	**add**(E o) 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) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this set (its cardinality).
Object[]	**toArray**() Returns an array containing all of the elements in this set.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

java.util

Class HashSet<E>

java.lang.Object
  └ java.util.AbstractCollection<E>
    └ java.util.AbstractSet<E>
      └ **java.util.HashSet<E>**

Constructor Summary

HashSet()
Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75).

HashSet(Collection<? extends E> c)
Constructs a new set containing the elements in the specified collection.

HashSet(int initialCapacity)
Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and default load factor, which is 0.75.

HashSet(int initialCapacity, float loadFactor)
Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and the specified load factor.

HashSet

Method Summary

boolean	**add**(E o) 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).

Methods inherited from class java.util.AbstractSet

equals, hashCode, removeAll

Methods inherited from class java.util.AbstractCollection

addAll, containsAll, retainAll, toArray, toArray, toString

Methods inherited from class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Set

addAll, containsAll, equals, hashCode, removeAll, retainAll, toArray, toArray

exemple

```
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

Duplicate detected: i

Duplicate detected: i

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

java.util.List

Method Summary

boolean	**add**(E o) 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.
int	**indexOf**(Object o) Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this 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)

java.util.List

int	**lastIndexOf**(Object o) Returns the index in this list of the last occurrence of the specified element, or -1 if this list does not contain this element.
ListIterator<E>	**listIterator**() Returns a list iterator of the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this 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 in this list of the specified element (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this list all the elements that are contained in the specified collection (optional operation).
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.
List<E>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array.

java.util

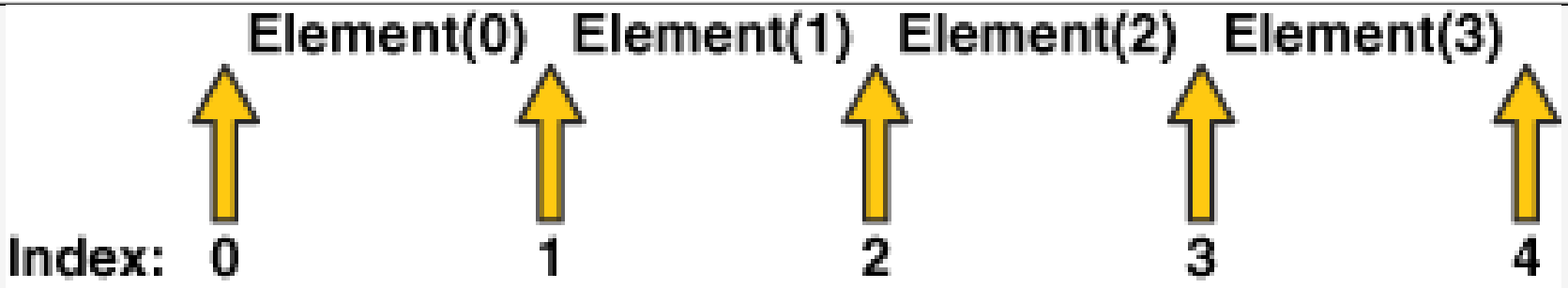
Interface ListIterator<E>

All Superinterfaces:

Iterator<E>

Method Summary

void	**add**(E o) 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.
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.
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 o) Replaces the last element returned by `next` or `previous` with the specified element (optional operation).



java.util.SortedSet

- SortedSet : Interface définie pour des collections où les éléments sont classés suivant un ordre particulier.
- Implémentations : La classe ***TreeSet***

```
import java.util.*;

public class Test2 {
    public static void main(String[] args) {
        SortedSet<String> s = new TreeSet<String>();
        s.add("test");
        s.add("abc");
        s.add("01");
        System.out.println(s);
    }
}
```

> [01, abc, test]

La classe java.util.Collections

Method Summary

static <T> boolean	**addAll**(Collection<? super T> c, T... a) Adds all of the specified elements to the specified collection.
static <T> int	**binarySearch**(List<? extends Comparable<? super T>> list, T key) Searches the specified list for the specified object using the binary search algorithm.
static <T> int	**binarySearch**(List<? extends T> list, T key, Comparator<? super T> c) Searches the specified list for the specified object using the binary search algorithm.
static <E> Collection<E>	**checkedCollection**(Collection<E> c, Class<E> type) Returns a dynamically typesafe view of the specified collection.
static <E> List<E>	**checkedList**(List<E> list, Class<E> type) Returns a dynamically typesafe view of the specified list.
static <K,V> Map<K,V>	**checkedMap**(Map<K,V> m, Class<K> keyType, Class<V> valueType) Returns a dynamically typesafe view of the specified map.
static <E> Set<E>	**checkedSet**(Set<E> s, Class<E> type) Returns a dynamically typesafe view of the specified set.
static <K,V> SortedMap<K,V>	**checkedSortedMap**(SortedMap<K,V> m, Class<K> keyType, Class<V> valueType) Returns a dynamically typesafe view of the specified sorted map.
static <E> SortedSet<E>	**checkedSortedSet**(SortedSet<E> s, Class<E> type) Returns a dynamically typesafe view of the specified sorted set.
static <T> void	**copy**(List<? super T> dest, List<? extends T> src) Copies all of the elements from one list into another.
static boolean	**disjoint**(Collection<?> c1, Collection<?> c2) Returns `true` if the two specified collections have no elements in common.
static <T> List<T>	**emptyList**() Returns the empty list (immutable).
static <K,V> Map<K,V>	**emptyMap**() Returns the empty map (immutable).
static <T> Set<T>	**emptySet**() Returns the empty set (immutable).

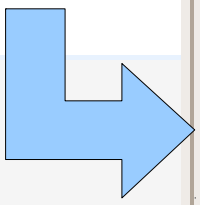
static <T> Enumeration<T>	**enumeration**(Collection<T> c) Returns an enumeration over the specified collection.
static <T> void	**fill**(List<? super T> list, T obj) Replaces all of the elements of the specified list with the specified element.
static int	**frequency**(Collection<?> c, Object o) Returns the number of elements in the specified collection equal to the specified object.
static int	**indexOfSubList**(List<?> source, List<?> target) Returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static int	**lastIndexOfSubList**(List<?> source, List<?> target) Returns the starting position of the last occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static <T> ArrayList<T>	**list**(Enumeration<T> e) Returns an array list containing the elements returned by the specified enumeration in the order they are returned by the enumeration.
static <T extends Object & Comparable<? super T>> T	**max**(Collection<? extends T> coll) Returns the maximum element of the given collection, according to the *natural ordering* of its elements.
static <T> T	**max**(Collection<? extends T> coll, Comparator<? super T> comp) Returns the maximum element of the given collection, according to the order induced by the specified comparator.
static <T extends Object & Comparable<? super T>> T	**min**(Collection<? extends T> coll) Returns the minimum element of the given collection, according to the *natural ordering* of its elements.
static <T> T	**min**(Collection<? extends T> coll, Comparator<? super T> comp) Returns the minimum element of the given collection, according to the order induced by the specified comparator.
static <T> List<T>	**nCopies**(int n, T o) Returns an immutable list consisting of n copies of the specified object.
static <T> boolean	**replaceAll**(List<T> list, T oldVal, T newVal) Replaces all occurrences of one specified value in a list with another.
static void	**reverse**(List<?> list) Reverses the order of the elements in the specified list.

static <T> Comparator<T>	**reverseOrder**(Comparator<T> cmp) Returns a comparator that imposes the reverse ordering of the specified comparator.
static void	**rotate**(List<?> list, int distance) Rotates the elements in the specified list by the specified distance.
static void	**shuffle**(List<?> list) Randomly permutes the specified list using a default source of randomness.
static void	**shuffle**(List<?> list, Random rnd) R[interface in java.util]ute the specified list using the specified source of randomness.
static <T> Set<T>	**singleton**(T o) Returns an immutable set containing only the specified object.
static <T> List<T>	**singletonList**(T o) Returns an immutable list containing only the specified object.
static <K,V> Map<K,V>	**singletonMap**(K key, V value) Returns an immutable map, mapping only the specified key to the specified value.
static <T extends Comparable<? super T>> void	**sort**(List<T> list) Sorts the specified list into ascending order, according to the *natural ordering* of its elements.
static <T> void	**sort**(List<T> list, Comparator<? super T> c) Sorts the specified list according to the order induced by the specified comparator.
static void	**swap**(List<?> list, int i, int j) Swaps the elements at the specified positions in the specified list.
static <T> Collection<T>	**synchronizedCollection**(Collection<T> c) Returns a synchronized (thread-safe) collection backed by the specified collection.
static <T> List<T>	**synchronizedList**(List<T> list) Returns a synchronized (thread-safe) list backed by the specified list.
static <K,V> Map<K,V>	**synchronizedMap**(Map<K,V> m) Returns a synchronized (thread-safe) map backed by the specified map.
static <T> Set<T>	**synchronizedSet**(Set<T> s) Returns a synchronized (thread-safe) set backed by the specified set.
static <K,V> SortedMap<K,V>	**synchronizedSortedMap**(SortedMap<K,V> m) Returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
static	**synchronizedSortedSet**(SortedSet<T> s)

exemple

```
import java.util.*;

public class Test2 {
    public static void main(String[] args) {
        List<String> l = new ArrayList<String>();
        l.add("test");
        l.add("01");
        l.add("abc");
        System.out.println(l);
        Collections.sort(l);
        System.out.println(l);
        Collections.shuffle(l);
        System.out.println(l);
    }
}
```



```
[test, 01, abc]
[01, abc, test]
[01, test, abc]
```

L'interface java.util.Queue<E>

java.util

Interface Queue<E>

Type Parameters:

E - the type of elements held in this collection

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Subinterfaces:

BlockingDeque<E>, BlockingQueue<E>, Deque<E>

All Known Implementing Classes:

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

---

```
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.

L'interface java.util.Queue<E>

Method Summary	
boolean	**add**(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.
E	**element**() Retrieves, but does not remove, the head of this queue.
boolean	**offer**(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
E	**peek**() Retrieves, but does not remove, the head of this queue, or returns `null` if this queue is empty.
E	**poll**() Retrieves and removes the head of this queue, or returns `null` if this queue is empty.
E	**remove**() Retrieves and removes the head of this queue.

L'interface java.util.Queue<E>

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

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out). Whatever the ordering used, the *head* of the queue is that element which would be removed by a call to `remove()` or `poll()`. In a FIFO queue, all new elements are inserted at the *tail* of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties.

L'interface java.util.Queue<E>

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

The `offer` method inserts an element if possible, otherwise returning `false`. This differs from the `Collection.add` method, which can fail to add an element only by throwing an unchecked exception. The `offer` method is designed for use when failure is a normal, rather than exceptional occurrence, for example, in fixed-capacity (or "bounded") queues.

The `remove()` and `poll()` methods remove and return the head of the queue. Exactly which element is removed from the queue is a function of the queue's ordering policy, which differs from implementation to implementation. The `remove()` and `poll()` methods differ only in their behavior when the queue is empty: the `remove()` method throws an exception, while the `poll()` method returns `null`.

The `element()` and `peek()` methods return, but do not remove, the head of the queue.

L'interface java.util.Deque<E>

java.util

Interface Deque<E>

Type Parameters:

`E` - the type of elements held in this collection

All Superinterfaces:

Collection<E>, Iterable<E>, Queue<E>

All Known Subinterfaces:

BlockingDeque<E>

All Known Implementing Classes:

ArrayDeque, LinkedBlockingDeque, LinkedList

```
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.

L'interface java.util.Deque<E>

	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()`

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.

L'interface java.util.Deque<E>

This interface extends the `Queue` interface. When a deque is used as a queue, FIFO (First-In-First-Out) behavior results. Elements are added at the end of the deque and removed from the beginning. The methods inherited from the `Queue` interface are precisely equivalent to `Deque` methods as indicated in the following table:

`Queue` Method	Equivalent `Deque` Method
`add(e)`	`addLast(e)`
`offer(e)`	`offerLast(e)`
`remove()`	`removeFirst()`
`poll()`	`pollFirst()`
`element()`	`getFirst()`
`peek()`	`peekFirst()`

L'interface java.util.Deque<E>

Deques can also be used as LIFO (Last-In-First-Out) stacks. This interface should be used in preference to the legacy `Stack` class. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque. Stack methods are precisely equivalent to `Deque` methods as indicated in the table below:

Stack Method	Equivalent `Deque` Method
`push(e)`	`addFirst(e)`
`pop()`	`removeFirst()`
`peek()`	`peekFirst()`

L'interface java.util.Deque<E>

This interface provides two methods to remove interior elements, `removeFirstOccurrence` and `removeLastOccurrence`.

Unlike the `List` interface, this interface does not provide support for indexed access to elements.

La classe java.util.ArrayDeque<E>

java.util

Class ArrayDeque<E>

java.lang.Object
  └ java.util.AbstractCollection<E>
    └ **java.util.ArrayDeque<E>**

Type Parameters:

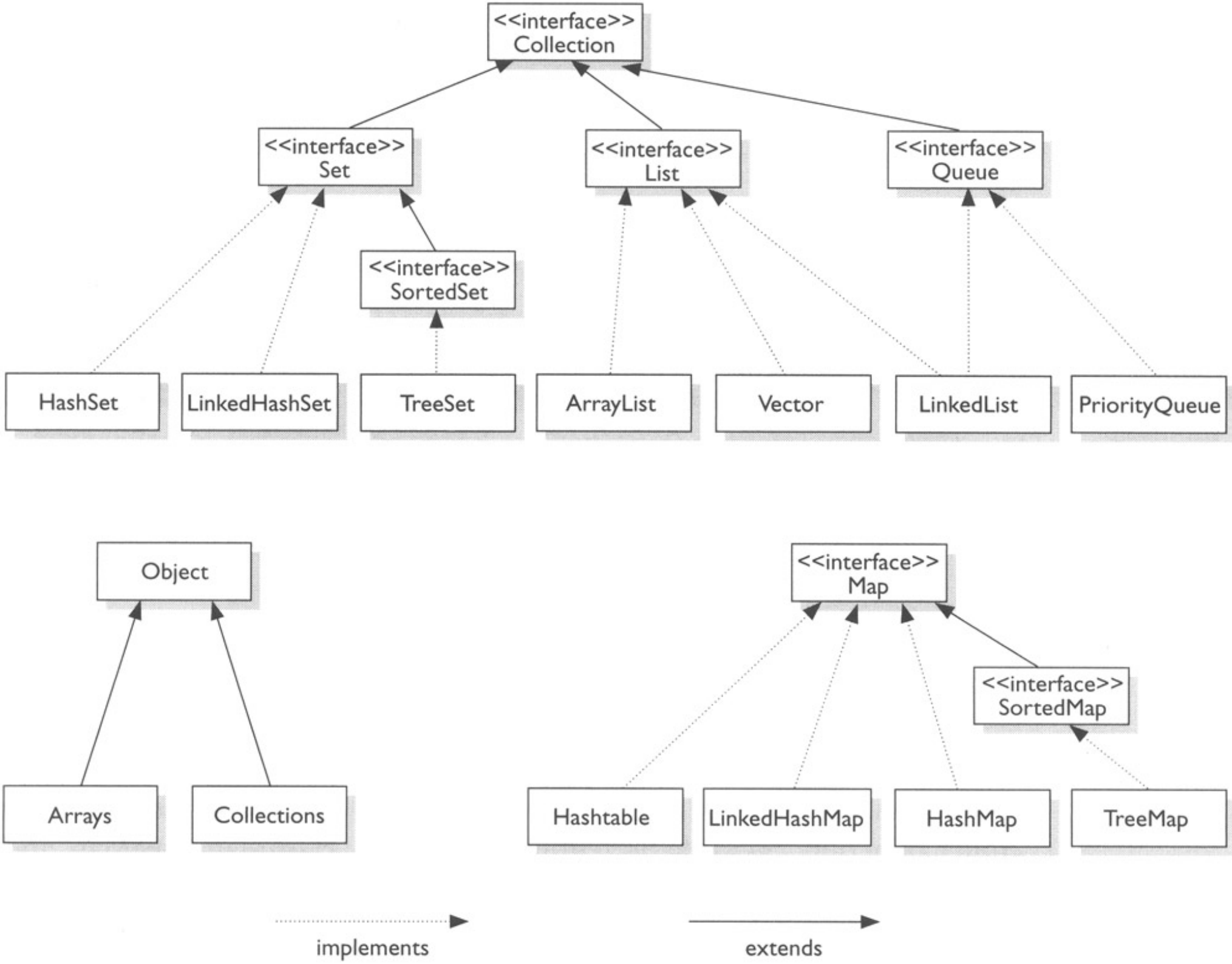
E - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, Queue<E>

```
public class ArrayDeque<E>
extends AbstractCollection<E>
implements Deque<E>, Cloneable, Serializable
```

Resizable-array implementation of the Deque interface. Array deques have no capacity restrictions; they grow as necessary to support usage. They are not thread-safe; in the absence of external synchronization, they do not support concurrent access by multiple threads. Null elements are prohibited. This class is likely to be faster than Stack when used as a stack, and faster than LinkedList when used as a queue.


<<interface>>
Collection
<<interface>>
Set
<<interface>>
List
<<interface>>
Queue
<<interface>>
SortedSet
HashSet
LinkedHashSet
TreeSet
ArrayList
Vector
LinkedList
PriorityQueue
Object
Arrays
Collections
<<interface>>
Map
<<interface>>
SortedMap
Hashtable
LinkedHashMap
HashMap
TreeMap
implements
extends