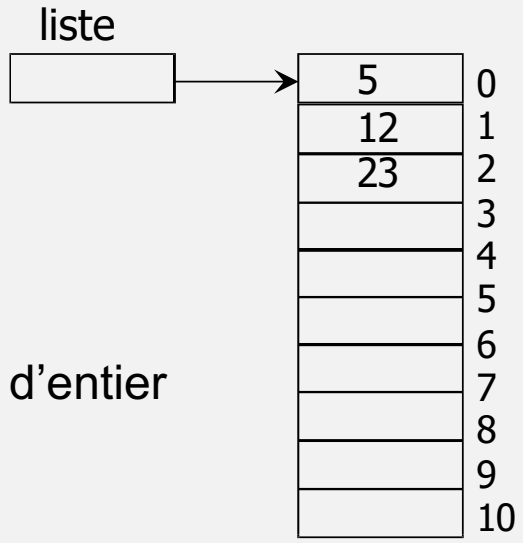
7 – Les Collections

Introduction

Tableaux de primitives



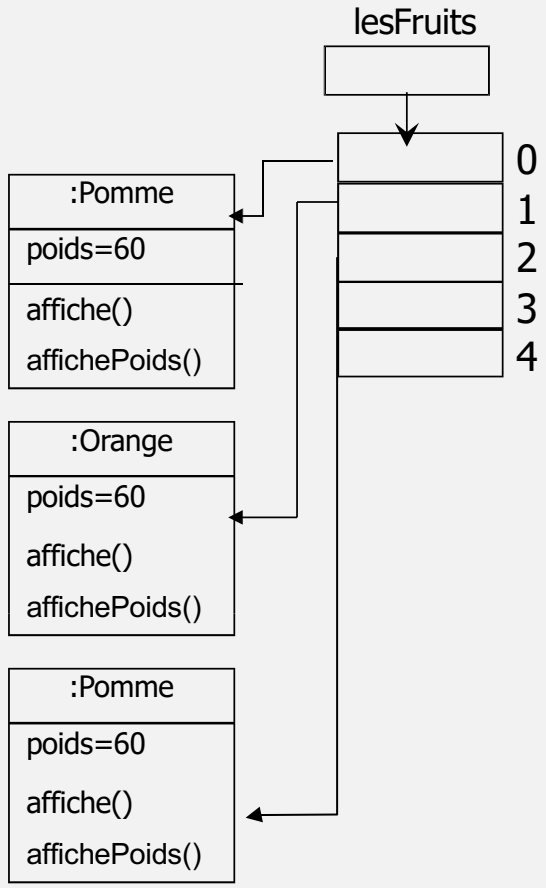
- Tableaux de primitives:
- Déclaration :
  - Exemple : Tableau de nombres entiers
    - `int[] liste;`
  - liste est un handle destiné à pointer vers un tableau d'entier
- Création du tableau
  - `liste = new int[11];`
- Manipulation des éléments du tableau:
  - `liste[0]=5; liste[1]=12; liste[3]=23;`
  - `for(int i=0;i<liste.length;i++){`
  - `System.out.println(liste[i]);`
  - `}`

Tableaux d'objets

- Déclaration :
  - Exemple : Tableau d'objets Fruit
  - `Fruit[]  lesFruits;`
- Création du tableau
  - `lesFruits = new Fruit[5];`
- Création des objets:
  - `lesFruits[0]=new Pomme(60);`
  - `lesFruits[1]=new Orange(100);`
  - `lesFruits[2]=new Pomme(55);`
- Manipulation des objets:

```
for(int i=0;i<lesFruits.length;i++){
  lesFruits[i].affiche();
  if(lesFruits[i] instanceOf Pomme)
((Pomme)lesFruits[i]).affichePoids();
 else
 ((Orange)lesFruits[i]).affichePoids();
 }
```

- Un tableau d'objets est un tableau de handles



Collections

Java propose l'API Collections qui offre un socle riche et des implémentations d'objets de type collection enrichies au fur et à mesure des versions de Java.

L'API Collections possède deux grandes familles chacune définies par une interface :

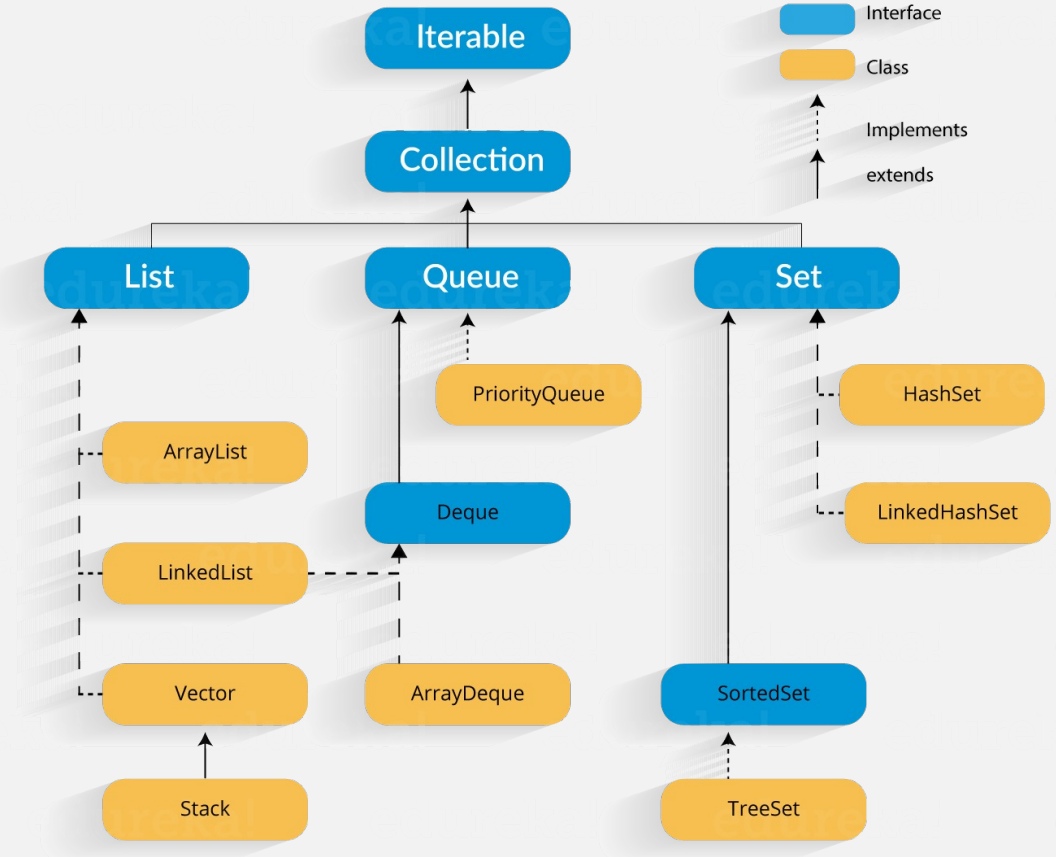
- **java.util.Collection** : pour gérer un groupe d'objets
- **java.util.Map** : pour gérer des éléments de type paires de clé/valeur
-

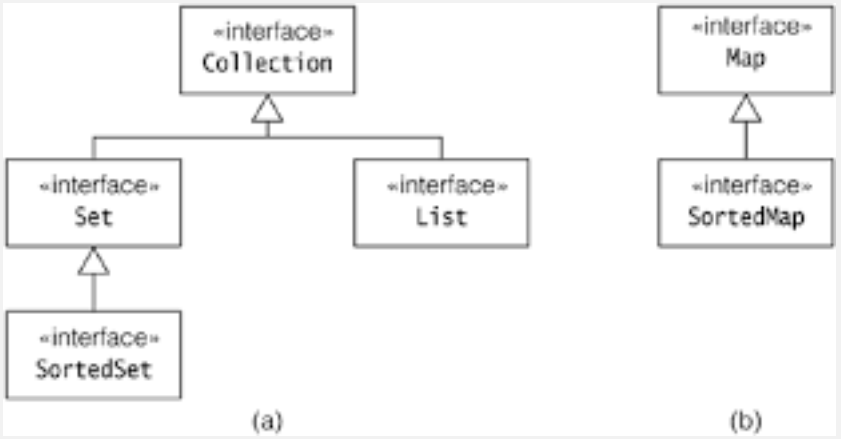
Collections

l'API Collections définit enfin :

- deux interfaces pour le parcours de certaines collections : Iterator et ListIterator.
- une interface et une classe pour permettre le tri de certaines collections : Comparable et Comparator
- des classes utilitaires : Arrays, Collections

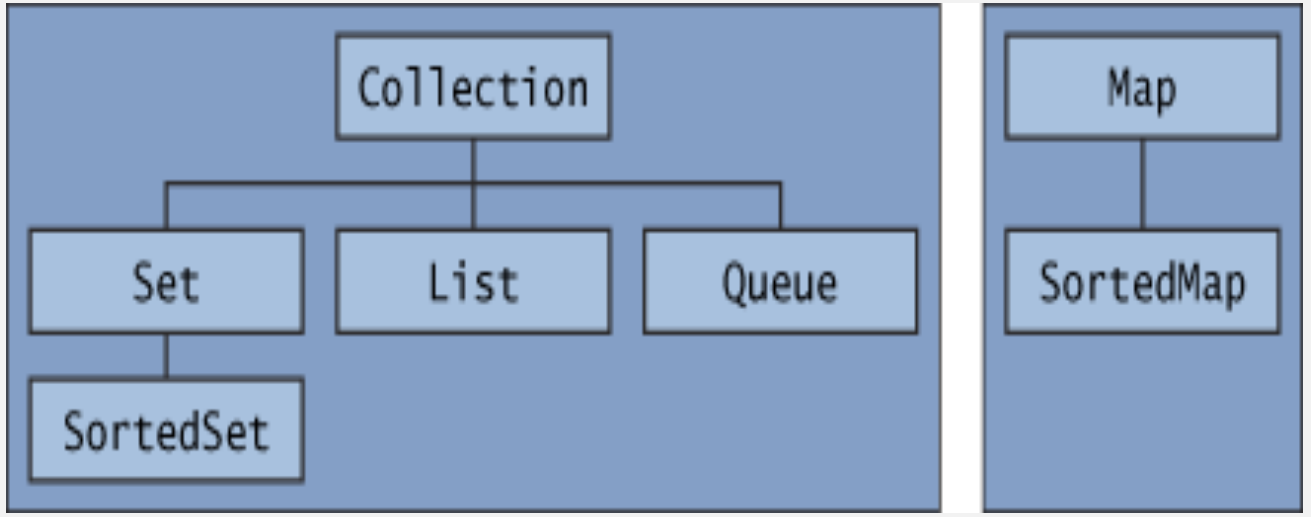
Collections





Collection interfaces

- Les interfaces de collection principales encapsulent différents types de collections. Ils représentent les types de données abstraites qui font partie de l'infrastructure de collections. Ce sont des interfaces donc elles ne fournissent pas d'implémentation !



Collections

- Une collection est un tableau dynamique d'objets de type Object.
- Une collection fournit un ensemble de méthodes qui permettent:
  - D'ajouter un nouveau objet dans le tableau
  - Supprimer un objet du tableau
  - Rechercher des objets selon des critères
  - Trier le tableau d'objets
  - Contrôler les objets du tableau
  - Etc…
- Dans un problème, les tableaux peuvent être utilisés quand la dimension du tableau est fixe.
- Dans le cas contraire, il faut utiliser les collections
- Java fournit plusieurs types de collections:
  - ArrayList
  - Vector
  - Iterator
  - HashMap
  - Etc…
- Dans cette partie du cours, nous allons présenter uniquement comment utiliser les collections ArrayList, Vector, Iterator et HashMap

Collections

- Java fournit plusieurs types de collections:
  - ArrayList
  - Vector
  - Iterator
  - HashMap
  - Etc…
- **Il existe *trois composants qui étendent l'interface de Collection*, à savoir la List, Queue et les Map.**

Collections

Collection	Ordonné	Accès direct	Clé / valeur	Doublons	Null	Thread Safe
ArrayList	Oui	Oui	Non	Oui	Oui	Non
LinkedList	Oui	Non	Non	Oui	Oui	Non
HashSet	Non	Non	Non	Non	Oui	Non
TreeSet	Oui	Non	Non	Non	Non	Non
HashMap	Non	Oui	Oui	Non	Oui	Non
TreeMap	Oui	Oui	Oui	Non	Non	Non
Vector	Oui	Oui	Non	Oui	Oui	Oui
Hashtable	Non	Oui	Oui	Non	Non	Oui
Properties	Non	Oui	Oui	Non	Non	Oui
Stack	Oui	Non	Non	Oui	Oui	Oui
CopyOnWriteArrayList	Oui	Oui	Non	Oui	Oui	Oui
ConcurrentHashMap	Non	Oui	Oui	Non	Non	Oui
CopyOnWriteArraySet	Non	Non	Non	Non	Oui	Oui

public interface Collection<E> extends Iterable<E>

```java
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);        //optional
    boolean retainAll(Collection<?> c);        //optional
    void clear();                              //optional

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Iterable

Il existe trois manières d'itérer les objets de Iterable .

1. Utilisation de la boucle for améliorée (boucle for-each)
2. Utilisation de la boucle Iterable forEach
3. Utilisation de l' interface **Iterator<T>**

Iterator

- **Iterator** est une interface qui itère les éléments. Il permet de parcourir la liste et de modifier les éléments.
- ***L'interface Iterator a trois méthodes qui sont mentionnées ci-dessous*** :
    1. **public boolean hasNext()** — Cette méthode renvoie true si l'itérateur a plus d'éléments.
    2. **public object next()** — Il renvoie l'élément et déplace le pointeur du curseur vers l'élément suivant.
    3. **public void remove()** — Cette méthode supprime les derniers éléments renvoyés par l'itérateur.

Iterator

- La collection de type Iterator du package java.util est souvent utilisée pour afficher les objets d'une autre collection
- En effet il est possible d'obtenir un iterator à partir de chaque collection.
- Exemple :
  - Création d'un vecteur de Fruit.

```
Vector<Fruit> fruits=new Vector<Fruit>();
```

  - Ajouter des fruits aux vecteur

```
fruits.add(new Pomme(30));
fruits.add(new Orange(25));
fruits.add(new Pomme(60));
```

  - Création d'un Iterator à partir de ce vecteur

```
Iterator<Fruit> it=fruits.iterator();
```

  - Parcourir l'Iterator:

```
while(it.hasNext()){
  Fruit f=it.next();
  f.affiche();
}
```

List

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

List — une collection ordonnée (parfois appelée séquence). Les listes peuvent contenir des éléments en double. L'utilisateur d'une liste a généralement un contrôle précis sur l'endroit où chaque élément est inséré dans la liste et peut accéder aux éléments par leur index entier (position). Si vous avez utilisé Vector, vous connaissez la saveur générale de List.

Collection ArrayList

- ArrayList est une classe du package java.util, qui implémente l'interface List.
- Déclaration d'une collection de type List qui devrait stocker des objets de type Fruit:
  - `List<Fruit> fruits;`
- Création de la liste:
  - `fruits=new ArrayList<Fruit>();`
- Ajouter deux objets de type Fruit à la liste:
  - `fruits.add(new Pomme(30));`
  - `fruits.add(new Orange(25));`
- Faire appel à la méthode affiche() de tous les objets de la liste:
  - En utilisant la boucle classique for

```
for(int i=0;i<fruits.size();i++){
  fruits.get(i).affiche();
}
```

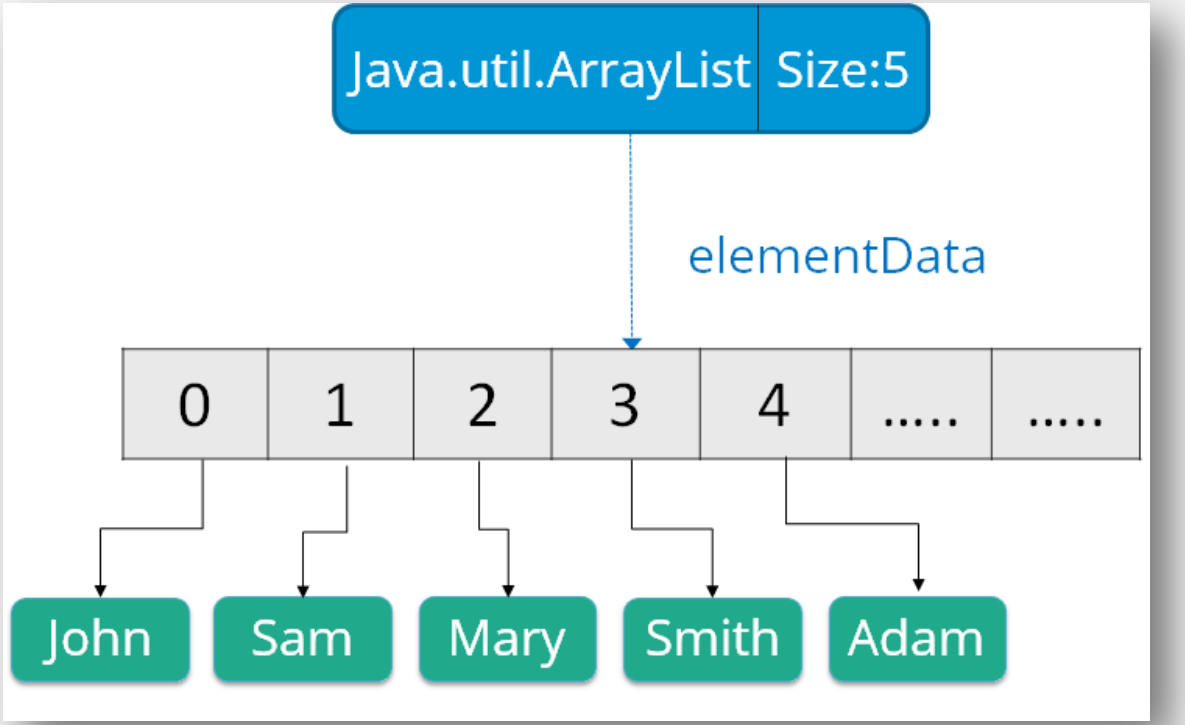
  - En utilisant la boucle for each

```
for(Fruit f:fruits)
  f.affiche();
```

- Supprimer le deuxième Objet de la liste
  - `fruits.remove(1);`

Exemple d'utilisation de ArrayList

```java
import java.util.ArrayList;
import java.util.List;

public class App1 {
    public static void main(String[]    args) {
// Déclaration d'une liste de type  Fruit
    List<Fruit> fruits;
// Création de la   liste
    fruits=new ArrayList<Fruit>();
// Ajout de 3 objets Pomme, Orange  et Pomme à  la liste
    fruits.add(new Pomme(30));
    fruits.add(new Orange(25));
    fruits.add(new Pomme(60));
// Parcourir    tous    les objets
      for(int i=0;i<fruits.size();i++){
// Faire appel à la méthode affiche() de chaque Fruit de la  liste
        fruits.get(i).affiche();
       }
// Une  autre   manière plus simple pour parcourir une liste
      for(Fruit f:fruits) // Pour chaque    Fruit de la liste
        f.affiche(); // Faire appel à la méthode affiche() du Fruit f
    }
}
```



Collection Vector

- Vector est une classe du package java.util qui fonctionne comme ArrayList
- Déclaration d'un Vecteur qui devrait stocker des objets de type Fruit:
  - `Vector<Fruit> fruits;`
- Création de la liste:
  - `fruits=new Vector<Fruit>();`
- Ajouter deux objets de type Fruit à la liste:
  - `fruits.add(new Pomme(30));`
  - `fruits.add(new Orange(25));`
- Faire appel à la méthode affiche() de tous les objets de la liste:
  - En utilisant la boucle classique for

```
for(int i=0;i<fruits.size();i++){
  fruits.get(i).affiche();
}
```

  - En utilisant la boucle for each

```
for(Fruit f:fruits)
  f.affiche();
```

- Supprimer le deuxième Objet de la liste
  - `fruits.remove(1);`

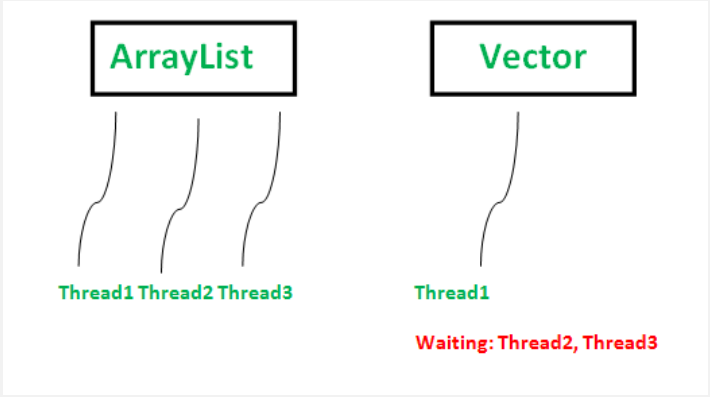
Exemple d'utilisation de Vector

```java
import java.util.Vector;  public class App2 {

public static void main(String[]    args) {
// Déclaration d'un vecteur de type Fruit
    Vector<Fruit> fruits;
// Création du vecteur
    fruits=new Vector<Fruit>();
// Ajout de 3 objets Pomme, Orange  et Pomme au vecteur
    fruits.add(new Pomme(30));
    fruits.add(new Orange(25));
    fruits.add(new Pomme(60));
// Parcourir    tous    les objets
  for(int i=0;i<fruits.size();i++){
// Faire appel à la méthode affiche() de chaque Fruit
    fruits.get(i).affiche();
  }
// Une  autre   manière plus simple pour parcourir un   vecteur
  for(Fruit f:fruits) // Pour chaque    Fruit du veceur
    f.affiche(); // Faire appel à la méthode affiche() du Fruit f
  }
}
```

ArrayList vs Vector

N°	ArrayList	Vector
1.	ArrayList n'est pas synchronisé.	Le vecteur est synchronisé.
2.	ArrayList incrémente 50 % de la taille actuelle du tableau si le nombre d'éléments dépasse sa capacité.	Les incréments vectoriels de 100 % signifient que la taille du tableau double si le nombre total d'éléments dépasse sa capacité.
3.	ArrayList n'est pas une classe héritée. Il est introduit dans JDK 1.2.	Vector est une classe héritée.
4.	ArrayList est rapide car non synchronisé.	Vector est lent parce qu'il est synchronisé, c'est-à-dire que dans un environnement multithreading, il maintient les autres threads dans un état exécutable ou non jusqu'à ce que le thread actuel libère le verrou de l'objet.
5.	ArrayList utilise l'interface Iterator pour parcourir les éléments.	Un vecteur peut utiliser l'interface Iterator ou l'interface Enumeration pour parcourir les éléments.

ArrayList vs Vector



Performances	ArrayList est plus rapide. Comme il n'est pas synchronisé, alors que les opérations vectorielles ralentissent les performances car elles sont synchronisées (thread-safe), si un thread travaille sur un vecteur, il a acquis un verrou sur celui-ci, ce qui oblige tout autre thread voulant travailler dessus à doivent attendre que le verrou soit libéré.
Croissance des données	ArrayList et Vector **augmentent et diminuent de manière dynamique** pour maintenir une utilisation optimale du stockage, mais la façon dont ils se redimensionnent est différente. ArrayList incrémente 50 % de la taille actuelle du tableau si le nombre d'éléments dépasse sa capacité, tandis que vector incrémente 100 % - doublant essentiellement la taille actuelle du tableau.
Traversal	Vector peut utiliser à la fois Enumeration et Iterator pour parcourir les éléments vectoriels, tandis que ArrayList ne peut utiliser que **Iterator** pour parcourir.
Applications	Vector peut utiliser à la fois Enumeration et Iterator pour parcourir les éléments vectoriels, tandis que ArrayList ne peut utiliser que **Iterator** pour parcourir.

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

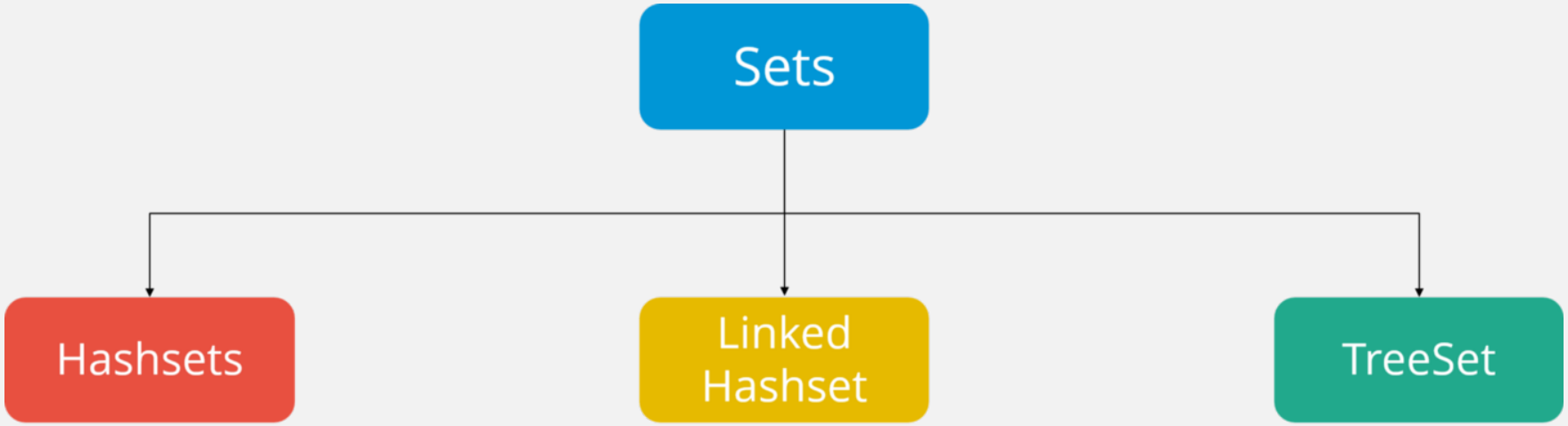
```java
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index,
        Collection<? extends E> c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

Set

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

Set — une collection qui ne peut pas contenir d'éléments en double. Cette interface modélise l'abstraction des ensembles mathématiques et est utilisée pour représenter des ensembles, tels que les cartes comprenant une main de poker, les cours constituant l'emploi du temps d'un étudiant ou les processus exécutés sur une machine.

Set<E>
extends Collection<E>



Set<E>
extends Collection<E>

- La classe Java HashSet crée une collection qui utilise une table de hachage pour le stockage.
- HashSet ne contient que des éléments uniques et hérite de la classe AbstractSet et implémente l'interface Set.
- Aussi, il utilise un mécanisme de *hachage* pour stocker les éléments.

Set<E>

extends Collection<E>

Voici quelques-unes des méthodes de la classe Java HashSet :

Method	Description
boolean add(Object o)	Adds the specified element to this set if it is not already present.
boolean contains(Object o)	Returns true if the set contains the specified element.
void clear()	Removes all the elements from the set.
boolean isEmpty()	Returns true if the set contains no elements.
boolean remove(Object o)	Remove the specified element from the set.
Object clone()	Returns a shallow copy of the HashSet instance: the elements themselves are not cloned.
Iterator iterator()	Returns an iterator over the elements in this set.
int size()	Return the number of elements in the set.

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

```java
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
//optional
    boolean removeAll(Collection<?> c);
 //optional
    boolean retainAll(Collection<?> c);
 //optional
    void clear();
 //optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Remarque : rien n'a été ajouté à l'interface de collecte , sauf aucun doublon autorisé

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

```java
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

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

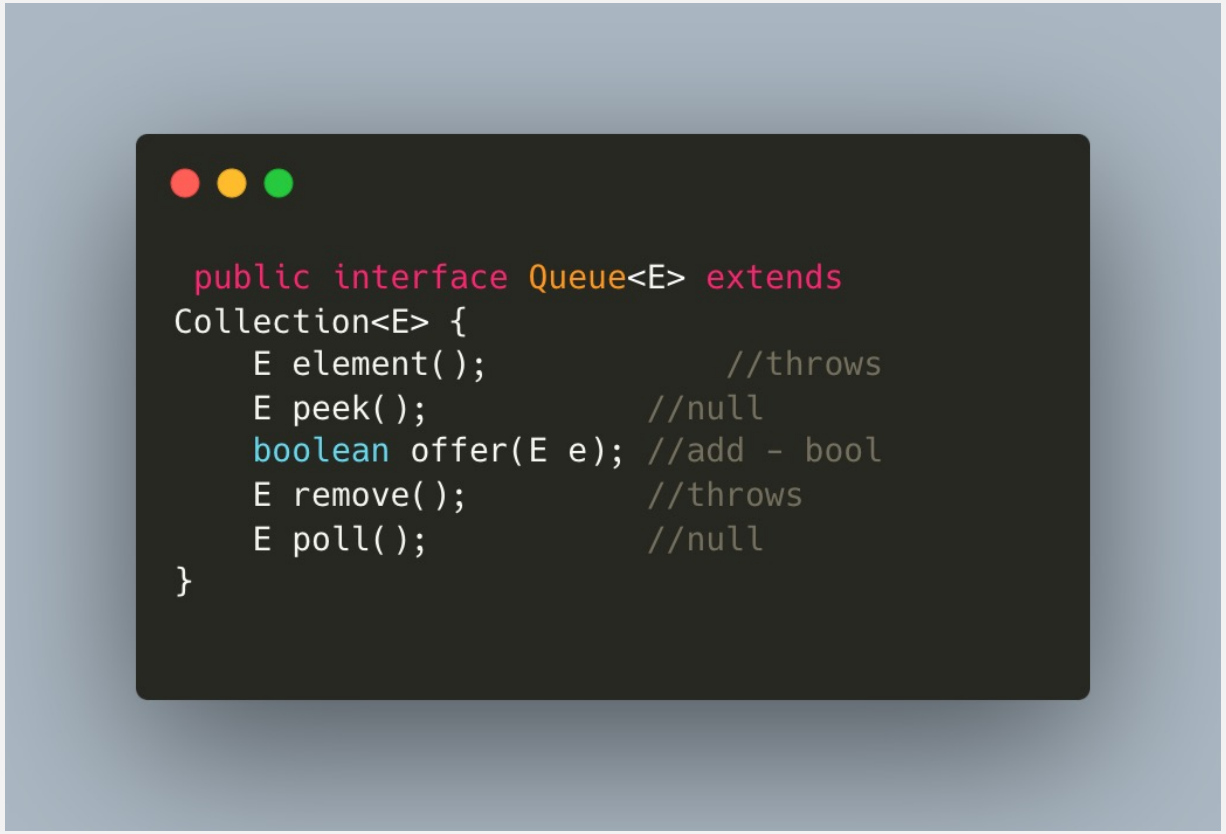
- **SortedSet —** un ensemble qui maintient ses éléments dans l'ordre croissant. Plusieurs opérations supplémentaires sont prévues pour profiter de la commande.
- Les ensembles triés sont utilisés pour les ensembles ordonnés naturellement, tels que les listes de mots et les rouleaux d'appartenance.

Queue

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

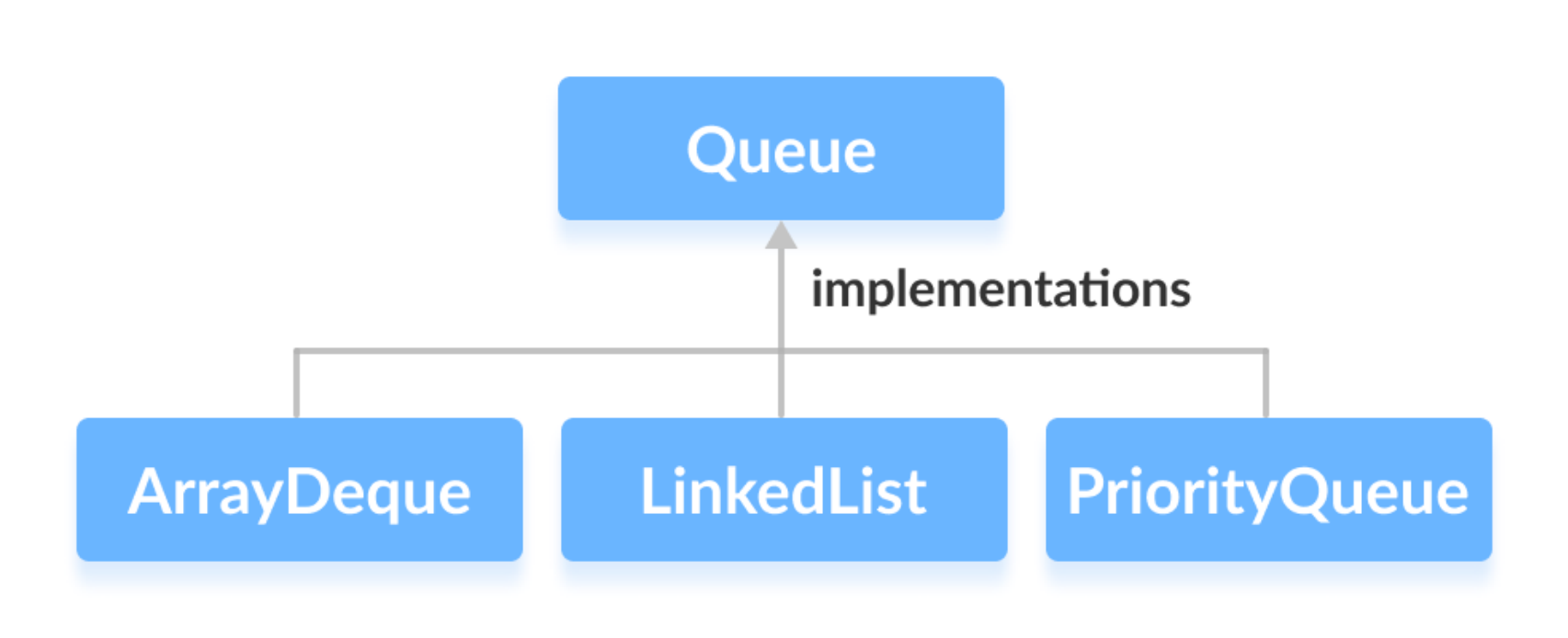
- **Queue —** une collection utilisée pour contenir plusieurs éléments avant le traitement. Outre les opérations de collecte de base, une file d'attente (Queue) fournit des opérations d'insertion, d'extraction et d'inspection supplémentaires.

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

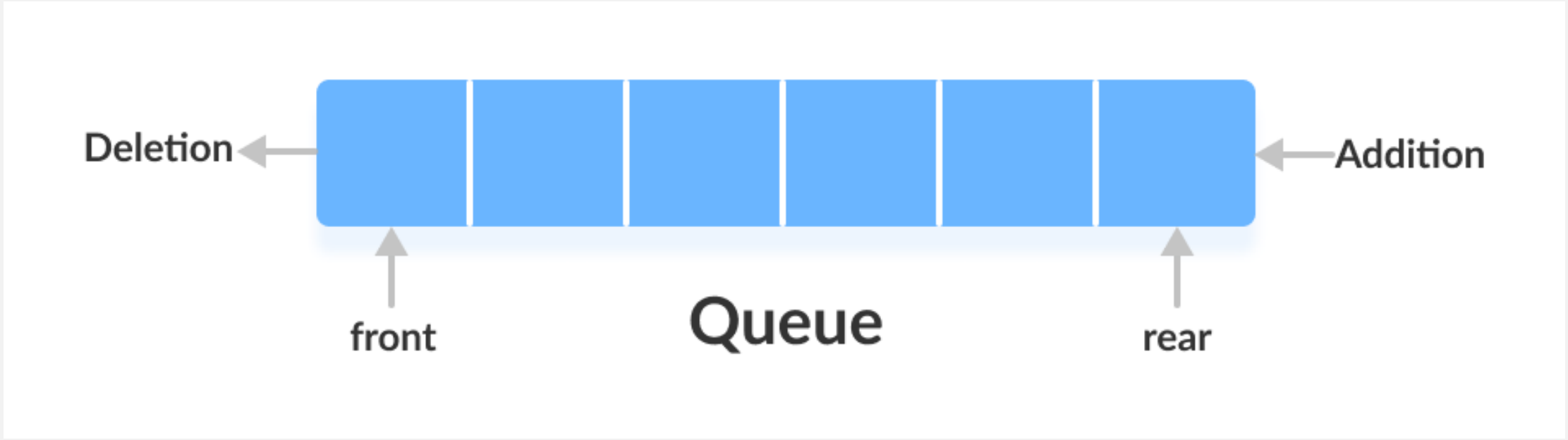


```
 public interface Queue<E> extends
Collection<E> {
    E element();             //throws
    E peek();              //null
    boolean offer(E e); //add - bool
    E remove();            //throws
    E poll();              //null
}
```

L’interface Queue



Fonctionnement Queue



Les méthodes de Queue

- **add()** - *Insère l'élément spécifié dans la file d'attente. Si la tâche réussit, add() renvoie true, sinon il lève une exception.*
- **offer()** - *Insère l'élément spécifié dans la file d'attente. Si la tâche réussit, offer() renvoie true, sinon false.*
- **element()** - *Renvoie la tête de la file d'attente. Lève une exception si la file d'attente est vide.*
- **peek()** - *Renvoie la tête de la file d'attente. Renvoie null si la file d'attente est vide.*
- **remove()** - *Renvoie et supprime la tête de la file d'attente. Lève une exception si la file d'attente est vide.*
- **poll()** - *Renvoie et supprime la tête de la file d'attente. Renvoie null si la file d'attente est vide.*

Les caractéristiques de Queue

- La file d'attente est utilisée pour insérer des éléments à la fin de la file d'attente et supprime depuis le début de la file d'attente. Il suit le concept FIFO.
- La file d'attente Java prend en charge toutes les méthodes de l'interface Collection, y compris l'insertion, la suppression, etc.
- LinkedList , ArrayBlockingQueue et PriorityQueue sont les implémentations les plus fréquemment utilisées.

Les caractéristiques de Queue

- Si une opération nulle est effectuée sur BlockingQueues, NullPointerException est levée.
- Les files d'attente disponibles dans le package java.util sont des files d'attente illimitées.
- Les files d'attente disponibles dans le package java.util.concurrent sont les files d'attente limitées.
- Toutes les files d'attente, à l'exception des Deques, prennent en charge l'insertion et la suppression respectivement à la fin et à la tête de la file d'attente. Les Deques prennent en charge l'insertion et le retrait d'éléments aux deux extrémités.

Map

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

- **Map —** un objet qui mappe les clés aux valeurs. Une carte ne peut pas contenir de clés en double ; chaque clé peut correspondre à au plus une valeur. Si vous avez utilisé Hashtable, vous connaissez déjà les bases de Map.

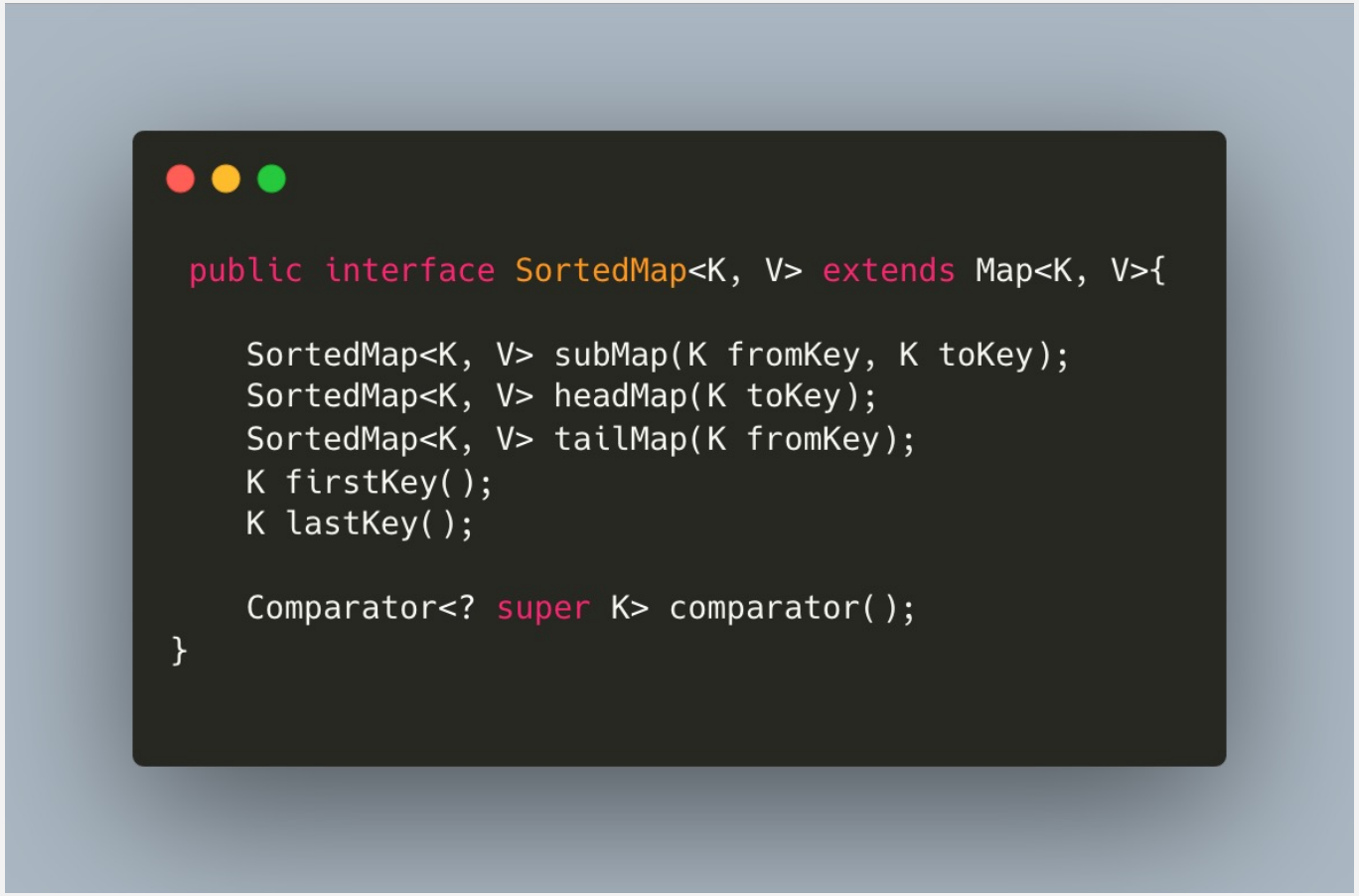
public interface **Map<K,V>**

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

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

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

- **SortedMap —** une carte qui maintient ses mappages dans l'ordre croissant des clés. Il s'agit de l'analogue Map de SortedSet. Les cartes triées sont utilisées pour les collections naturellement ordonnées de paires clé/valeur, telles que les dictionnaires et les annuaires téléphoniques.

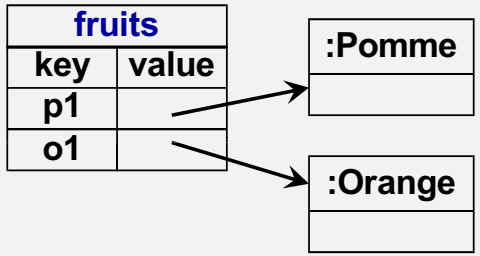
public interface **SortedMap<K,V>**
extends Map<K,V>



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

    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();

    Comparator<? super K> comparator();
}
```

Collection de type HashMap

- La collection HashMap est une classe qui implémente l'interface Map. Cette collection permet de créer un tableau dynamique d'objet de type Object qui sont identifiés par une clé.
- Déclaration et création d'une collection de type HashMap qui contient des fruits identifiés par une clé de type String :
  - `Map<String, Fruit> fruits=new HashMap<String, Fruit>();`
- Ajouter deux objets de type Fruit à la colelction
  - `fruits.put("p1", new Pomme(40));`
  - `fruits.put("o1", new Orange(60));`
- Récupérer un objet ayant pour clé "p1«
  - `Fruit f=fruits.get("p1");`
  - `f.affiche();`
- Parcourir toute la collection:




```
Iterator<String> it=fruits.keySet().iterator();
while(it.hasNext()){
  String key=it.next();
  Fruit ff=fruits.get(key);
  System.out.println(key);
  ff.affiche();
}
```