المدرسة الوطنية للعلوم التطبيقية –أكادير
ⵜⵉⵎⵙⵙⵓⵜ ⵜⵏⵜⵔⴰⵏⵜ ⵏ ⵜⵓⵙⵙⵏⵉⵡⵉⵏ ⵜⵉⵖⵔⵉⵖⵉⵏ – ⴰⴳⴰⴷⵉⵔ.
ÉCOLE NATIONALE DES SCIENCES APPLIQUÉES - AGADIR

جامعة ابن زهر - أكادير
UNIVERSITÉ IBN ZOHR - AGADIR

LES COLLECTIONS EN JAVA

Abdellaziz Walid
a.walid@uiz.ac.ma
Département G-Info
2019-2020

Problématique

Question : On souhaite créer un objet qui va gérer un ensemble d'éléments ou d'objets de même type. Comment procéder ?

Exemple :

Un objet qui va gérer un ensemble d'objets de type classe Point.

```
class Point {
  private int x, y;
  public Point(int x, int y) { this.x = x; this.y = y; }
  public void affiche() {
    System.out.println("Point : "+ x +", "+ y);
  }
}
```

Solution : les tableaux

- Un tableau (array en anglais) est le type d'objet le plus simple à programmer pour gérer un ensemble d'éléments(d'objets) de même type.
- Exemple :

```
public class TabPoint {
  public static void main(String[] arg) {
    Point[] tp = new Point[3];
    tp[0] = new Point(1, 2);
    tp[1] = new Point(4, 5);
    tp[2] = new Point(8, 9);
    for (int i=0; i<tp.length; i++) tp[i].affiche();
  }
}
```

Synthèse sur les tableaux

- Avantages
  - accès par index (accès direct et rapide).
  - recherche efficace si le tableau est trié (dichotomie).

- Inconvénients
  - insertions et suppressions peu efficaces.
  - nombre d'éléments borné qui doit être défini à l'avance.
  - Les tableaux sont inadéquats pour gérer une quantité importante d'informations du même type quand leur nombre n'est pas connu à l'avance.

=> Chercher une solution adéquate pour résoudre les limitations inhérentes aux tableaux.

Solution alternative : Les collections

- Une collection est un objet qui regroupe de multiples éléments de même type dans une seule entité.
- regroupement mémoire d'éléments de même type
  - Exemples : promotion d'étudiants, sac de billes, ...
- Utilisation de collections pour :
  - stocker, retrouver et manipuler des données
  - Transmettre des données d'une méthode à l'autre
- Importance en conception : Relation entre classes
  - «… est une collection de … »
  - « … a pour composant une collection de … »
- Le JDK fournit plusieurs types et variantes de collections sous forme de classes et interfaces pour le but de :
  - adapter la structure collective aux besoins de la collection
  - ne pas reprogrammer les traitements répétitifs classiques (affichage, saisie, recherche d'éléments, …)
- Ces classes et interfaces sont dans le paquetage java.util

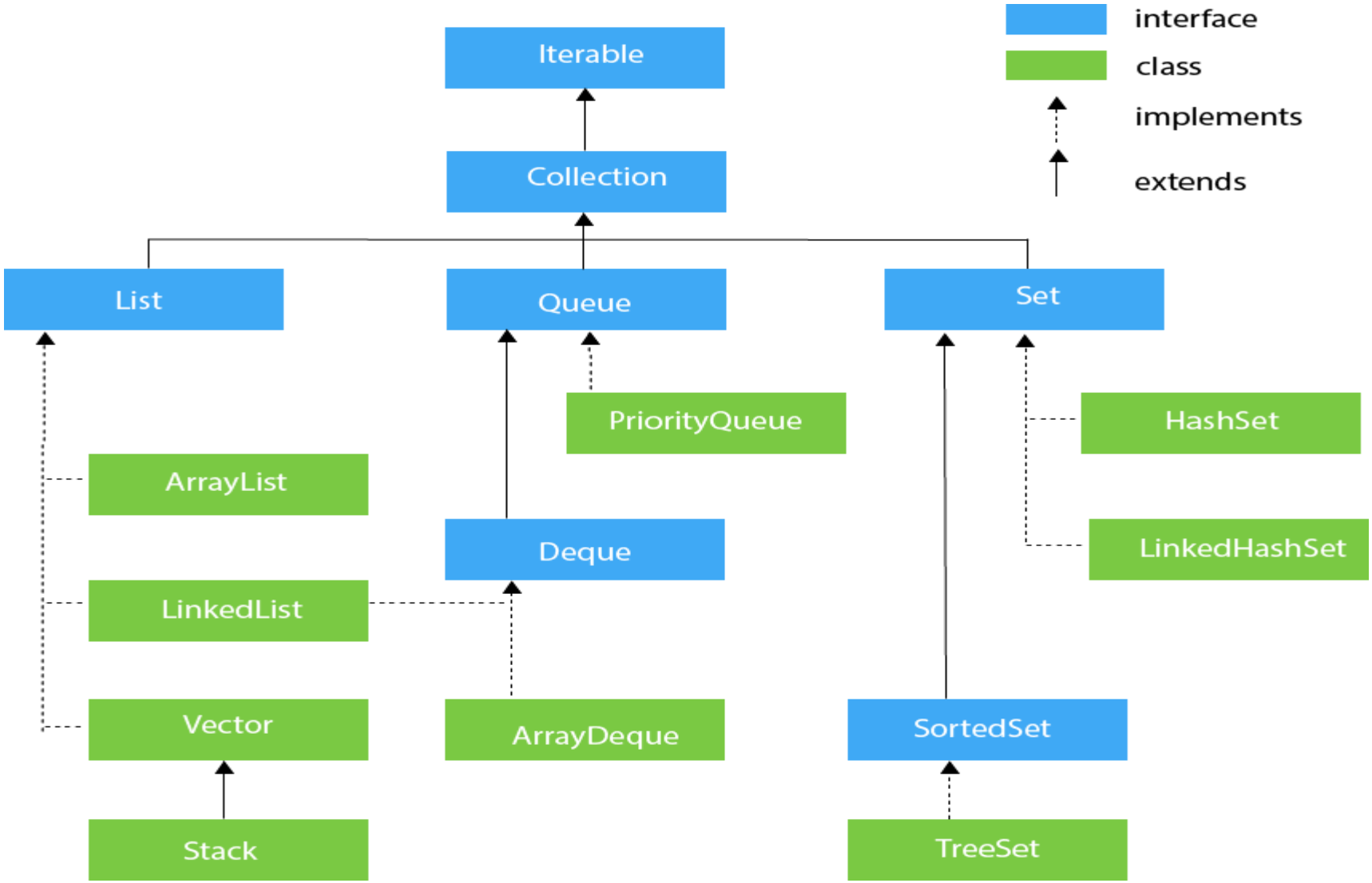
Les types de collection en Java

Deux hiérarchies principales :

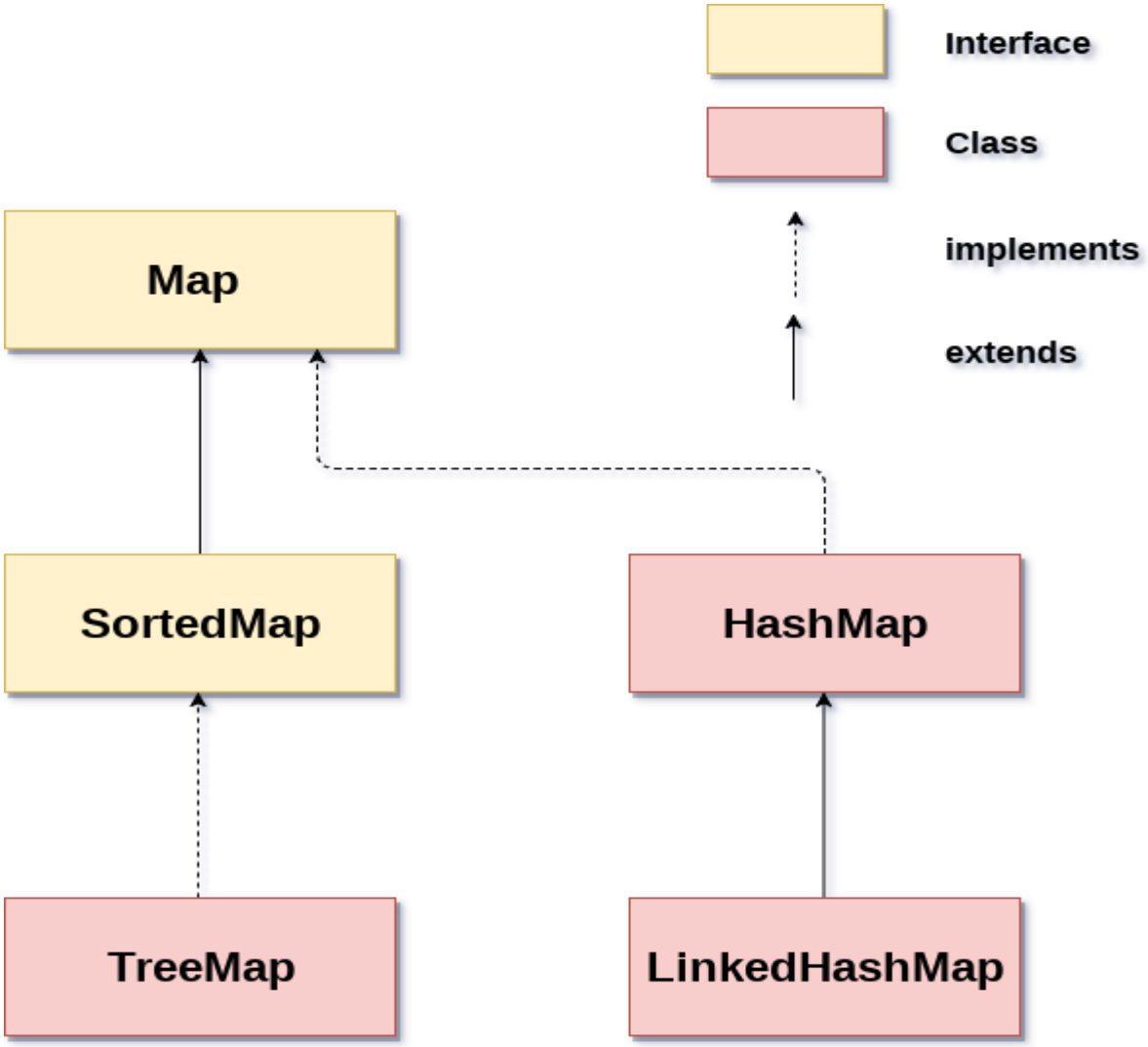
- Collection<E> : Objet(Collection) qui regroupe de multiple éléments(objets) de type E.

- Map<K,V> : Correspond aux collections indexées par des clés. Un élément (objet) de type V d'une Map est retrouvé rapidement si on connaît sa clé de type K.

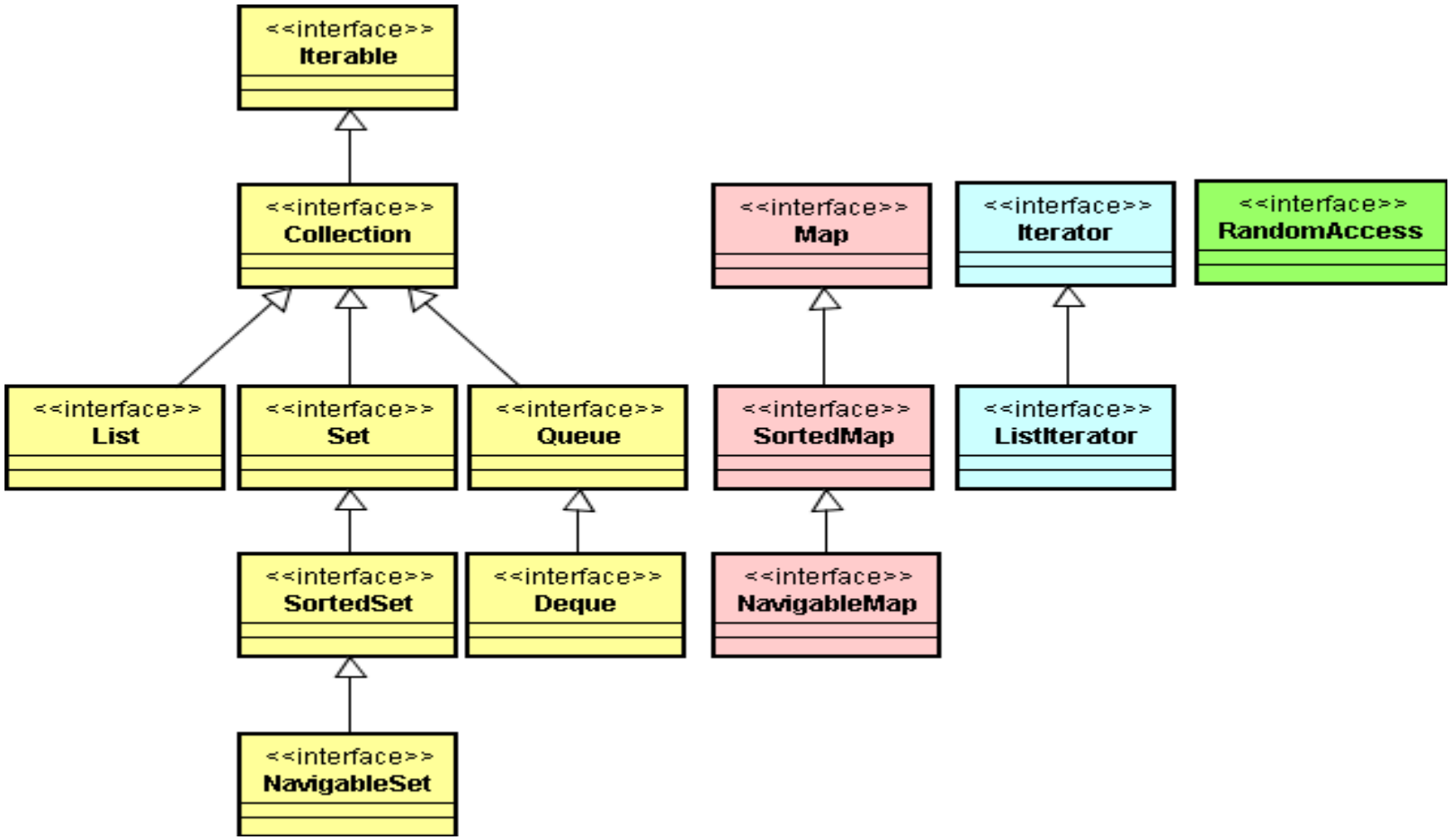
Hierarchy of Collection Framework



Hierarchy of Map Framework



Les interfaces Java liées au différents types de collections



Iterable Interface

- L'interface Iterable est l'interface racine pour toutes les classes de collection.
- L'interface Collection étend l'interface Iterable et, par conséquent, toutes les sous-classes de l'interface Collection implémentent également l'interface Iterable.
- Il ne contient qu'une seule méthode abstraite qui renvoie l'objet Iterator sur les éléments de type E.

```
public interface Iterable<E> {
  Iterator<E> iterator();
}
```

Iterator Interface

- Un itérateur (Iterator) est un objet sachant comment accéder aux éléments d'une collection un par un et qui connait leur position dans la collection.

- Il n'y a que trois méthodes dans l'interface Iterator.

```
public interface Iterator<E> {
  E next();
  boolean hasNext();
  void remove();
}
```

No.	Method	Description
1	public boolean hasNext()	Il retourne vrai si l'itérateur a plus d'éléments sinon il retourne faux.
2	public Object next()	Il renvoie l'élément et déplace le pointeur du curseur sur l'élément suivant.
3	public void remove()	Il supprime le dernier élément renvoyé par l'itérateur.

Iterator Interface

Exemple :

```
Collection<String> ville=new ArrayList<String>();
ville.add("Casablanca");//Adding object in arraylist
ville.add("Marrakech");
ville.add("Agadir");
Iterator itr=ville.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
```

- Heureusement, depuis Java SE 5.0, nous pouvons utiliser un raccourci élégant et bien plus concis au moyen de la boucle for sans utiliser l'objet Iterator :

for (variable : collection) instruction

Exemple :

```
for (String E : ville) {
System.out.println(E);
}
```

Collection interface

- L’interface Collection est l’interface implémentée par toutes les classes du Framework de la collection.

- Il déclare les méthodes que chaque collection aura.

- En d'autres termes, nous pouvons dire que l'interface Collection crée la base sur laquelle repose l'infrastructure de la collection.

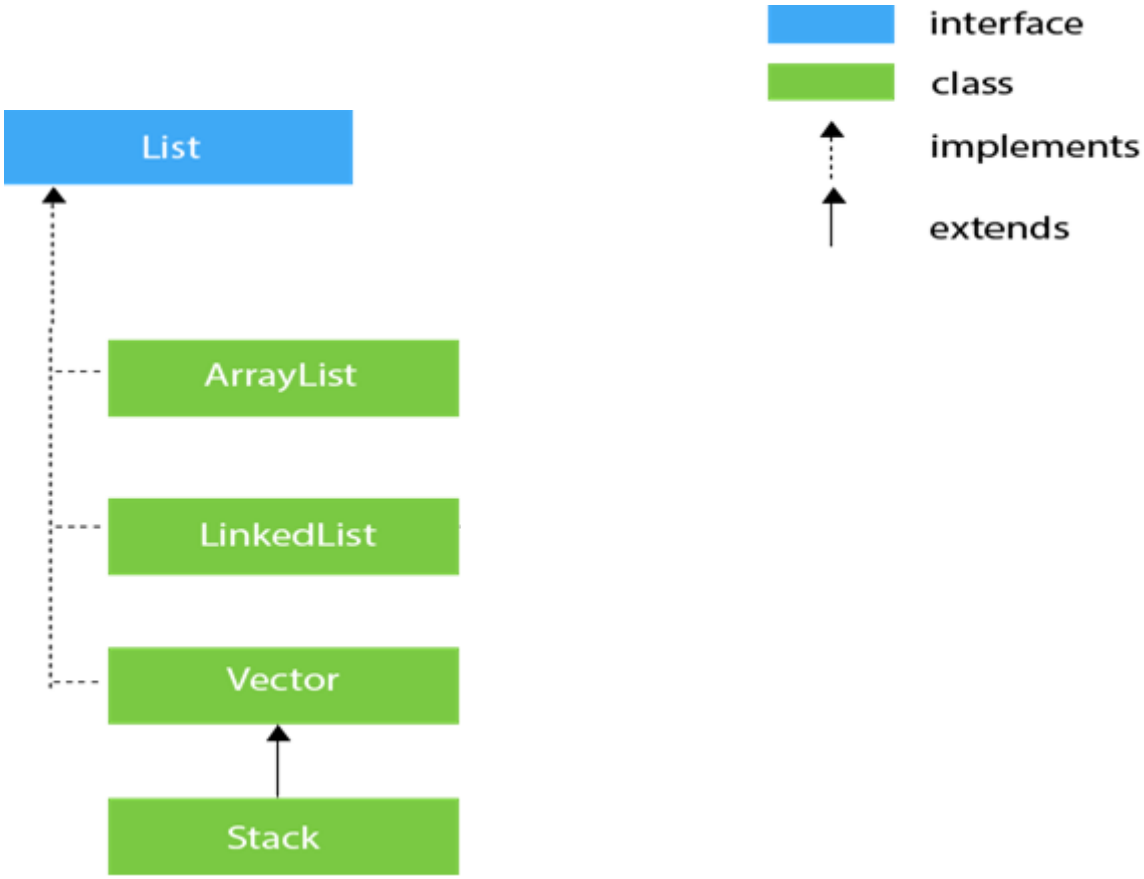
Les méthodes de l'interface Collection

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.
10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.
15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.

List Interface

- L'interface List est l'interface enfant de l'interface Collection.
- Il défend une structure de données de type liste dans laquelle nous pouvons stocker la collection d'objets commandée.
- Il peut avoir des valeurs en double.
- L'interface List est implémentée par les classes ArrayList, LinkedList, Vector et Stack.
- Pour instancier l'interface List, il faut utiliser:
  - List <data-type> list1= **new** ArrayList();
  - List <data-type> list2 = **new** LinkedList();
  - List <data-type> list3 = **new** Vector();
  - List <data-type> list4 = **new** Stack();
- Il existe diverses méthodes dans l'interface List qui peuvent être utilisées pour insérer, supprimer et accéder aux éléments de la liste.

Classes qui implémentent l'interface List



ArrayList Classe : Tableau dynamique

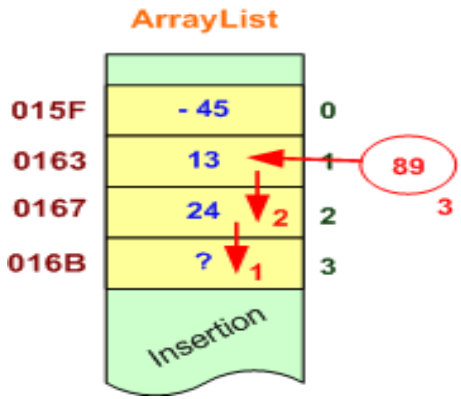

```
package test;
import java.util.*;
public class Test {
public static void main(String[] args) {
ArrayList<String> villes=new ArrayList<String>(); //Creating arraylist
villes.add("Casablanca");//Adding object in arraylist
villes.add("Marrakech");
villes.add("Agadir");
villes.add("Tanger");
villes.add("Casablanca");
villes.add(1,"Rabat");
//Traversing list through Iterator
Iterator itr=villes.iterator();
while(itr.hasNext()){
System.out.println(itr.next());  }  }}
```

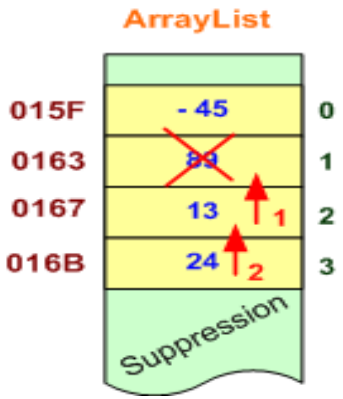
Résultats de l'exécution :

Casablanca
Rabat
Marrakech
Agadir
Tanger
Casablanca

ArrayList Class : Tableau dynamique

- La classe ArrayList offre des fonctionnalités d'accès rapide comparables à celles d'un tableau d'objets.
- Il permet des accès efficaces à un élément de rang donné, c'est-à-dire un accès direct vers l'objet désiré, au travers d'un indice, comme dans le cas d'un tableau d'objets.
- En outre, cette classe offre plus de souplesse que les tableaux d'objets dans la mesure où sa taille (son nombre d'éléments) peut varier au fil de l'exécution.
- L'insertion ou la suppression d'un objet à une position donnée ne pourra plus se faire aussi rapidement puisque toutes les cases suivantes du tableau devront être décalées.



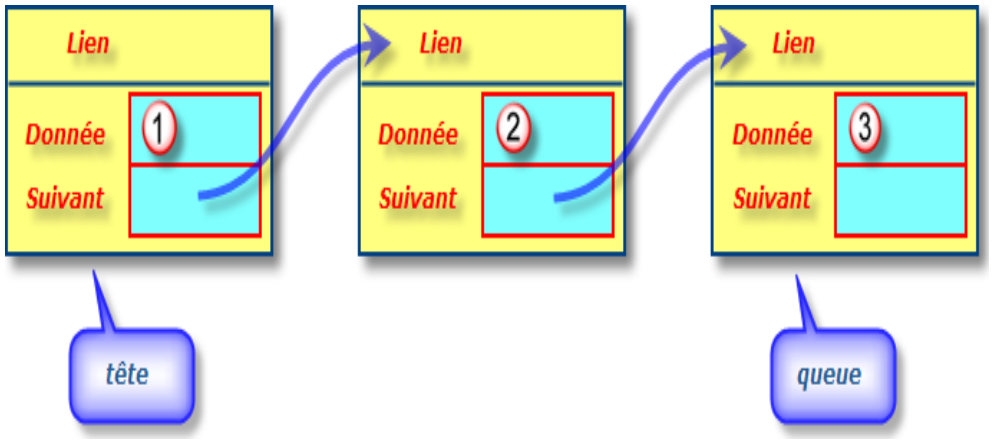


LinkedList Class : Liste chaînée


```
package test;
import java.util.*;
public class Test {
public static void main(String[] args) {
// TODO Auto-generated method stub
LinkedList<String> ville=new LinkedList<String>();
ville.add("Casablanca");
ville.add("Marrakech");
ville.add("Agadir");
ville.add("Tanger");
ville.add("Casablanca");
ville.add(2, "Rabat");
//Traversing list through Iterator
Iterator itr=ville.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}}
```

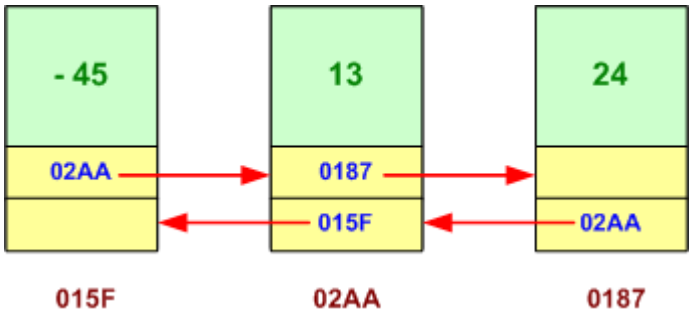
Résultats de l'exécution :

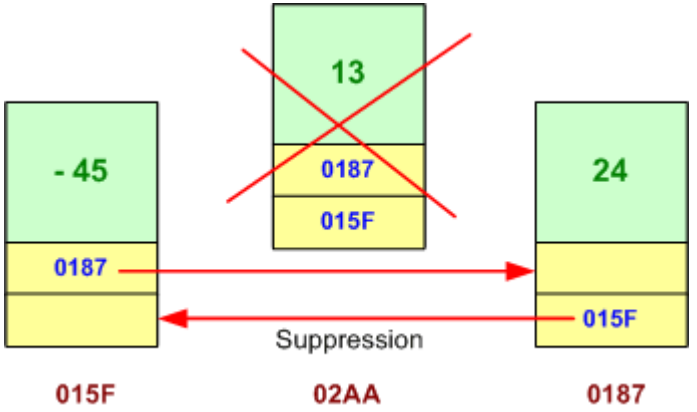
Casablanca
Marrakech
Rabat
Agadir
Tanger
Casablanca



LinkedList Class

- Il existe une autre structure de données très répandue, la liste chaînée, qui permet de résoudre les problèmes des tableaux dynamiques (ArrayList).
- une liste chaînée stocke chaque objet avec un lien qui y fait référence. Chaque lien possède également une référence vers le lien suivant de la liste.
- Avec Java, chaque élément d'une liste chaînée possède en fait deux liens, c'est-à-dire que chaque élément est aussi relié à l'élément précédent.
- La bibliothèque Java possède la classe LinkedList prête à l'emploi et qui implémente donc la liste "doublement chaînée".





Vector Class : Les vecteurs


```
package test;
import java.util.*;
public class Test {
public static void main(String[] args) {
// TODO Auto-generated method stub
Vector<String> ville=new Vector<String>();
ville.add("Casablanca");
ville.add("Marrakech");
ville.add("Agadir");
ville.add("Tanger");
ville.add("Casablanca");
ville.insertElementAt("Rabat", 4);;
//Traversing list through Iterator
Iterator itr=ville.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}  }}
```

Résultats de l'exécution :

Casablanca
Marrakech
Agadir
Tanger
Rabat
Casablanca

- Vector utilise un tableau dynamique pour stocker les éléments de données. C'est similaire à ArrayList.
- Il est synchronisé et contient de nombreuses méthodes qui ne font pas partie du Framework Collection.

Stack Class : les piles (LIFO)


```
package test;
import java.util.*;

public class Test {

public static void main(String[] args) {
// TODO Auto-generated method stub
Stack<String> ville=new Stack<String>();
ville.push("Casablanca"); //Empiler la pile
ville.push("Marrakech");
ville.push("Agadir");
ville.push("Tanger");
ville.push("Casablanca");
ville.push("Rabat");
ville.pop(); //dépiler la pile
//Traversing list through Iterator
Iterator itr=ville.iterator();
while(itr.hasNext()){
System.out.println(itr.next()); } }}
```

Résultats de l'exécution :

Casablanca
Marrakech
Agadir
Tanger
Casablanca

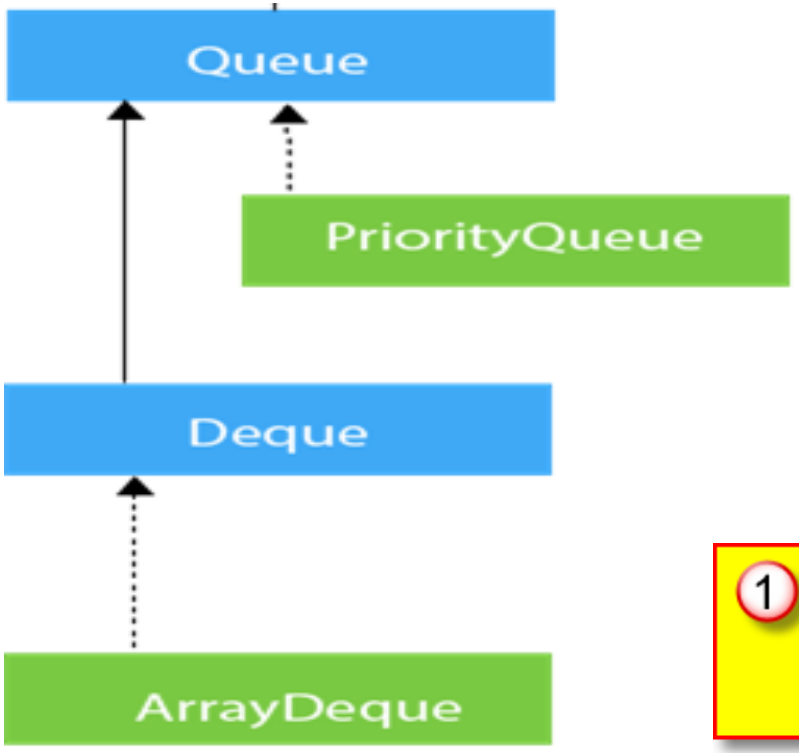
- Stack est la sous-classe de Vector.
- Il implémente la structure de données (LIFO), à savoir la pile.
- La pile contient toutes les méthodes de la classe Vector et fournit également ses méthodes telles que boolean push (), boolean peek (), boolean push (objet o), qui définit ses propriétés.

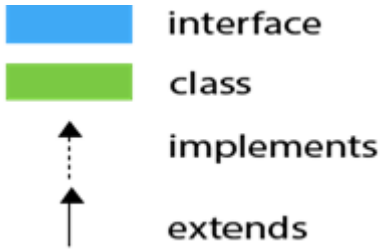
Queue interface : file FIFO

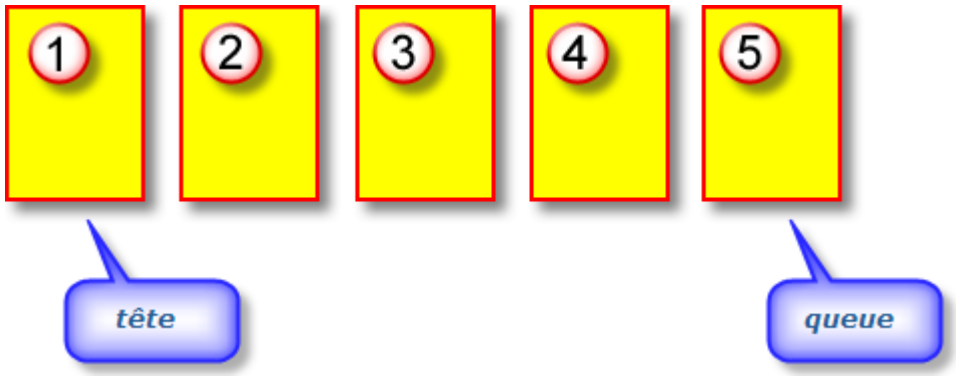
- Les queues implémentent les files d'attente.
- L'interface Queue conserve l'ordre du premier entré premier sorti. Il peut être défini comme une liste ordonnée utilisée pour contenir les éléments sur le point d'être traités.
- Il existe différentes classes telles que PriorityQueue, Deque et ArrayDeque qui implémente l'interface Queue.
- L'interface Queue peut être instanciée comme suit:

```
Queue< data-type > q1 = new PriorityQueue ();
Queue< data-type > q2 = new ArrayDeque ();
```

Classes qui implémentent l'interface Queue







PriorityQueue Class


```
package test;
import java.util.*;
public class Test {
public static void main(String[] args) {
PriorityQueue<Integer> number = new PriorityQueue<Integer>();
 number.add(2);  // Enfiler la file
number.add(3);
number.add(1);
number.add(4);
System.out.println("head:"+number.peek());
System.out.println("iterating the queue elements:");
Iterator itr=number.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
number.remove();  //défiler la fille
number.poll();
System.out.println("after removing two elements:");
Iterator<Integer> itr2=number.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());}}}
```

Résultats de l'exécution :

```
head:1
iterating the queue elements:
1
3
2
4
after removing two elements:
3
4
```

- La classe PriorityQueue implémente l'interface Queue.
- Il contient les éléments ou objets à traiter en fonction de leurs priorités.
- PriorityQueue n'autorise pas le stockage de valeurs NULL dans la file d'attente.

Interface Deque

- Une queue à deux extrémités permet d'ajouter ou de supprimer efficacement des éléments en début et à la fin.

- Java 6 a introduit une nouvelle interface Deque, dérivée de Queue, destinée à gérer les files d'attente à double entrée, c'est-à-dire dans lesquelles nous pouvons réaliser l'une des opérations suivantes à l'une des extrémités de la queue :
  - ajouter un élément,
  - examiner un élément,
  - supprimer un élément.

- Pour instancier une fille d’attente à double entrée :

```
Deque d = new ArrayDeque();
```

ArrayDeque Class


```
package test;
import java.util.*;
public class Test {
public static void main(String[] args) {
// TODO Auto-generated method stub
Deque<String> ville = new ArrayDeque<String>();
 ville.add("Casablanca");
ville.add("Marrakech");
ville.add("Rabat");
ville.addFirst("Tanger");
Iterator itr=ville.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} }
}
```

Résultats de l'exécution :

Tanger
Casablanca
Marrakech
Rabat

Les points importants concernant la classe ArrayDeque sont :

- Contrairement à Queue, nous pouvons ajouter ou supprimer des éléments des deux côtés.
- Les éléments nuls ne sont pas autorisés dans ArrayDeque.
- ArrayDeque est plus rapide que LinkedList et Stack.

Synthèse

- Les listes chaînées et les tableaux vous permettent de spécifier l'ordre dans lequel vous voulez organiser vos éléments.

- Cependant, si vous recherchez un élément particulier et que vous ne vous rappeliez pas sa position, vous aurez besoin de parcourir tous les éléments jusqu'à ce que vous trouviez l'élément recherché.

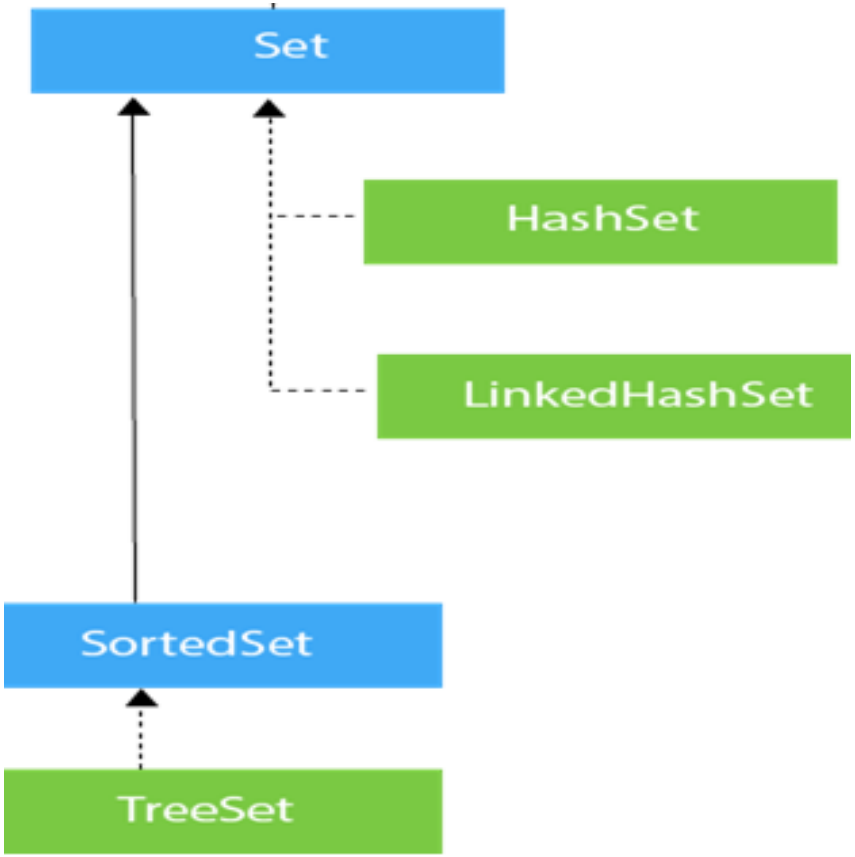
- Cela requiert beaucoup de temps, surtout lorsque la collection contient pas mal d'éléments. Si l'ordre des éléments n'a pas d'importance, il existe des structures de données qui vous permettent de retrouver un élément très rapidement.

- L'inconvénient est que ces structures de données ne vous permettent pas de contrôler l'ordre des éléments. En effet, elles les organisent plutôt selon l'ordre qui leur permet de les retrouver facilement.

Set Interface : Ensemble non ordonné

- Set Interface en Java est présent dans le package java.util.
- Il étend l'interface Collection.
- Il représente l'ensemble non ordonné d'éléments, ce qui ne nous permet pas de stocker les éléments en double.
- Nous pouvons stocker au plus une valeur nulle dans Set.
- Set est implémenté par HashSet, LinkedHashSet et TreeSet.
- L'ensemble peut être instancié comme:
  - Set<data-type> s1 = **new** HashSet<data-type>();
  - Set<data-type> s2 = **new** LinkedHashSet<data-type>();
  - Set<data-type> s3 = **new** TreeSet<data-type>();

Les classes qui implémentent L'interface Set



HashSet


```
package test;
import java.util.*;

public class Test {

public static void main(String[] args) {
// TODO Auto-generated method stub
HashSet<String> ville=new HashSet<String>();
ville.add("Casablanca");
ville.add("Marrakech");
ville.add("Rabat");
ville.add("Tanger");
 ville.add("Casablanca");
Iterator itr=ville.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} }}
```

Résultats de l'exécution :

Tanger
Marrakech
Casablanca
Rabat

- HashSet recourt à une technique dite de hachage, qui mémorise ses éléments dans un ordre quelconque (et donc plus rapide si l'ordre n'a pas d'importance).

LinkedHashSet Class


```
package test;
import java.util.*;
public class Test {
public static void main(String[] args) {
// TODO Auto-generated method stub
LinkedHashSet<String> ville=new LinkedHashSet<String>();
ville.add("Casablanca");
ville.add("Marrakech");
ville.add("Rabat");
ville.add("Tanger");
ville.add("Casablanca");
Iterator itr=ville.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} }
}
```

Résultats de l'exécution :

Casablanca
Marrakech
Rabat
Tanger

- LinkedHashSet est un ensemble tout à fait particulier qui se souvient de l'ordre d'insertion des éléments.

SortedSet Interface

- SortedSet est l'alternative de l'interface Set qui fournit un classement total de ses éléments.
- Les éléments du SortedSet sont disposés dans l'ordre croissant (croissant).
- SortedSet fournit des méthodes supplémentaires qui inhibent l'ordre naturel des éléments.
- SortedSet peut être instancié sous la forme:

```
SortedSet<data-type> set = new TreeSet();
```

TreeSet Class


```
package test;
import java.util.*;
public class Test {
public static void main(String[] args) {
// TODO Auto-generated method stub
TreeSet<String> ville=new
TreeSet<String>();
ville.add("Marrakech");
ville.add("Rabat");
ville.add("Casablanca");
ville.add("Tanger");
ville.add("Casablanca");
Iterator itr=ville.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
} }
}
```

Résultats de l'exécution :

Casablanca
Marrakech
Rabat
Tanger

- TreeSet : qui utilise un arbre binaire pour ordonner complètement les éléments.
- TreeSet mémorise ses éléments dans l'ordre ascendant avec un arbre, pour maintenir plus rapidement le tri des éléments stockées.
- La classe TreeSet ressemble beaucoup à la classe HashSet, avec une amélioration supplémentaire.

Synthèse

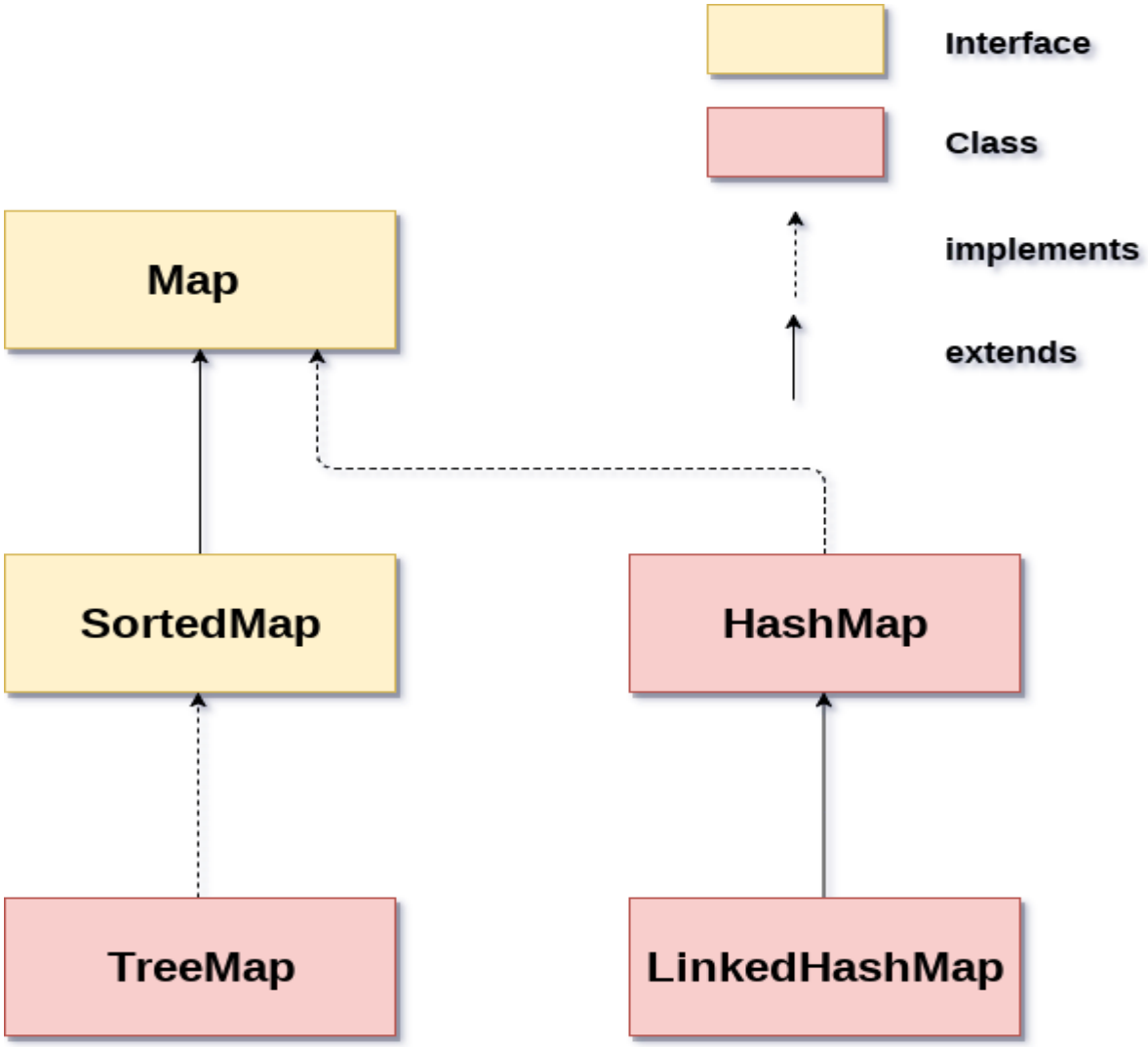
- un Set (ensemble) est une collection qui vous permet de retrouver rapidement un élément. Pour cela, il faut le connaître exactement, ce qui n'est pas souvent le cas.

- En fait, nous disposons généralement de certaines informations importantes sur l'élément à rechercher, et il faut le retrouver à partir de ces informations.

- Solution : La structure de données cartes (ou map) permet d'effectuer ce genre de recherche. Une carte enregistre des paires clé/valeur. Une valeur peut être retrouvée à partir de la clé correspondante.

- Les cartes sont également appelées table associative, ou dictionnaire.

Hierarchy of Map Framework



Interface Map

- Une carte contient des valeurs sur la base d'une clé, c'est-à-dire d'une paire clé/valeur.
- Chaque paire clé et valeur est appelée entrée.
- Une carte (Map) contient des clés uniques.
- Une carte est utile si vous devez rechercher, mettre à jour ou supprimer des éléments à l'aide d'une clé.

HashMap Class


```
package test;
import java.util.*;

public class Test {

public static void main(String[] args) {
// TODO Auto-generated method stub
 HashMap<Integer,String> ville=new HashMap<Integer,String>();

ville.put(101,"Marrakech");
ville.put(11,"Rabat");
ville.put(12,"Agadir");
ville.put(102,"Casablanca");
ville.put(103,"Tanger");
for(Map.Entry<Integer,String> m:ville.entrySet()){
System.out.println(m.getKey()+" "  +m.getValue());
        }}}
```

Résultats de l'exécution :

```
101 Marrakech
102 Casablanca
103 Tanger
11 Rabat
12 Agadir
```

- La classe Java HashMap contient des valeurs basées sur la clé.
- La classe Java HashMap ne contient que des clés uniques.
- La classe Java HashMap peut avoir une clé NULL et plusieurs valeurs NULL.
- La classe Java HashMap n'est pas synchronisée.
- La classe Java HashMap ne maintient aucun ordre.

LinkedHashMap Class


```
package test;
import java.util.*;
public class Test {
public static void main(String[] args) {
// TODO Auto-generated method stub
 LinkedHashMap<Integer,String> ville=new
LinkedHashMap<Integer,String>();

ville.put(101,"Marrakech");
ville.put(100,"Rabat");
ville.put(107,"Casablanca");
ville.put(2,"Casablanca");
ville.put(103,"Tanger");
for(Map.Entry m:ville.entrySet()){
        System.out.println(m.getKey()+"
"+m.getValue());
       } }}
```

Résultats de l'exécution :

```
101 Marrakech
100 Rabat
107 Casablanca
2 Casablanca
103 Tanger
```

- Java LinkedHashMap n'est pas synchronisé.
- Java LinkedHashMap maintient l'ordre d'insertion.

Interface SortedMap

L'interface *SortedMap* définit un comportement de *Map* avec des entrées ordonnées.

Comparator<? super K> comparator()	Retourne le Comparator utilisé par le SortedMap, ou null si le SortedMap n'utilise pas de Comparator.
K firstKey()	Retourne la première clé du SortedMap (la plus petite) Lève une exception NoSuchElementException si le SortedMap est vide.
K lastKey()	Retourne la dernière clé du SortedMap (la plus grande) Lève une exception NoSuchElementException si le SortedMap est vide.
SortedMap<K, V> headMap(K jusqua)	Retourne l'ensemble des entrées dont les clés sont inférieures à jusqua.
SortedMap<K, V> subMap(K depuis, K jusqua)	Retourne l'ensemble des entrées dont les clés sont supérieures ou égales à depuis, et inférieures à jusqua.
SortedMap<K, V> tailMap(K depuis)	Retourne l'ensemble des entrées dont les clés sont supérieures ou égales à depuis.

TreeMap Class


```
package test;
import java.util.*;
public class Test {
public static void main(String[] args) {
// TODO Auto-generated method stub
TreeMap<Integer,String> ville=new
TreeMap<Integer,String>();
ville.put(101,"Marrakech");
ville.put(11,"Rabat");
ville.put(12,"Agadir");
ville.put(102,"Casablanca");
ville.put(103,"Tanger");
for(Map.Entry<Integer,String>
m:ville.entrySet()){
System.out.println(m.getKey()+" "
+m.getValue());
        }}}
```

Résultats de l'exécution :

```
11 Rabat
12 Agadir
101 Marrakech
102 Casablanca
103 Tanger
```

- Java TreeMap contient uniquement des éléments uniques.
- Java TreeMap ne peut pas avoir une clé NULL, mais peut avoir plusieurs valeurs NULL.
- Java TreeMap n'est pas synchronisé.
- Java TreeMap maintient l'ordre croissant.