

# Les collections et tables associatives en Java

L3 SPI parcours Informatique

Marie-Laure Nivet, [nivet\\_m@univ-corse.fr](mailto:nivet_m@univ-corse.fr)

# Une collection, un conteneur, qu'est ce que c'est ?

- Un objet conteneur permettant de regrouper plusieurs éléments, un objet qui contient d'autres objets
- Utilisé pour stocker (ajouter, enlever), retrouver (accéder, chercher) et manipuler (parcourir les éléments) des données ou pour transférer un ensemble de données d'une méthode à une autre
- Exemple d'utilisation
  - Une main au poker (collection de cartes)
  - Une boîte aux lettres (collection de lettres)
  - Un répertoire téléphonique (collection de n° de tel, associés à des personnes)

# De quoi parle-t-on quand on parle de collections en Java ?

- Du framework Collection
- De la hiérarchie Collection : conteneurs de valeurs
  - `java.util.Collection`
  - `Set`
  - `List`
  - `Queue`
- De la hiérarchie Map : tables associatives
- Des algorithmes généraux disponibles, des classes utilitaires

Il existe des collections dédiées à la gestion des accès concurrents dans le package `java.util.concurrent`, non abordés ici : par exemple `CopyOnWriteArrayList`, `ConcurrentHashMap`, `CopyOnWriteArraySet` qui permettent des modifications pendant leur parcours.

Ossature, charpente d'une application. On trouve aussi  
« bibliothèque de classes spécialisées »

# Le framework Collection

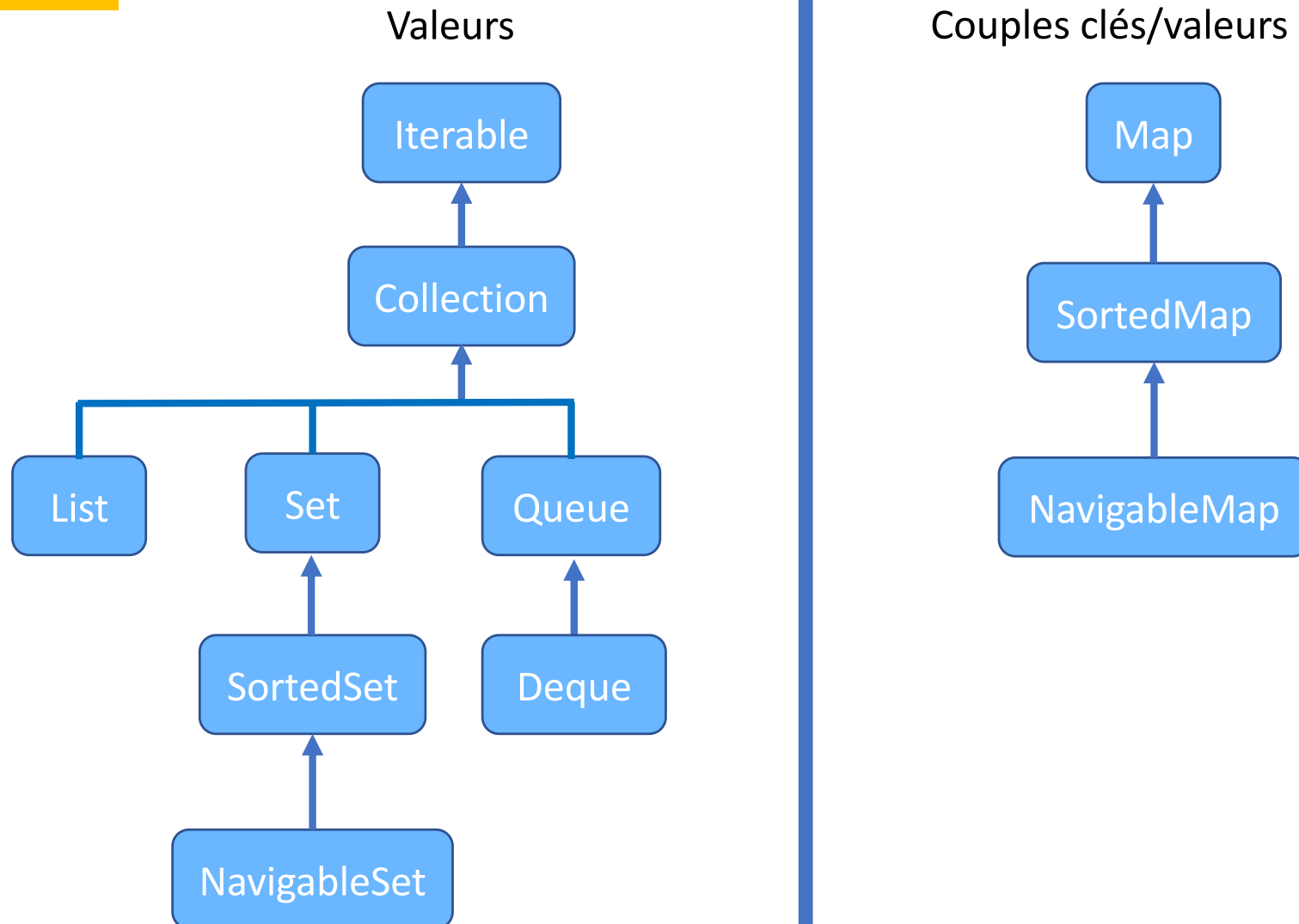
- Définition : architecture unifiée pour la représentation et la manipulation des collections d'objets
- Contenu
  - Interfaces : type de données abstrait permettant la manipulation des collections indépendamment de leur représentation
  - Classes abstraites : implémentations partielles des interfaces
  - Implémentation concrètes des interfaces Collection
  - Algorithmes de recherche et de tri s'appliquant aux objets issus de classes implémentant les interfaces collection.

# Comment se servir du Framework Collection ?

- Choisir un comportement de structure de données en identifiant les interfaces adaptées au problème
- Choisir une implémentation concrète de classe implémentant les interfaces en question en fonction
  - Des performances attendues
  - Du comportement face aux accès concurrents
- Allez voir du côté des Classes outils pour tout ce qui concerne les tris, opérations spécifiques et les ponts entre collections

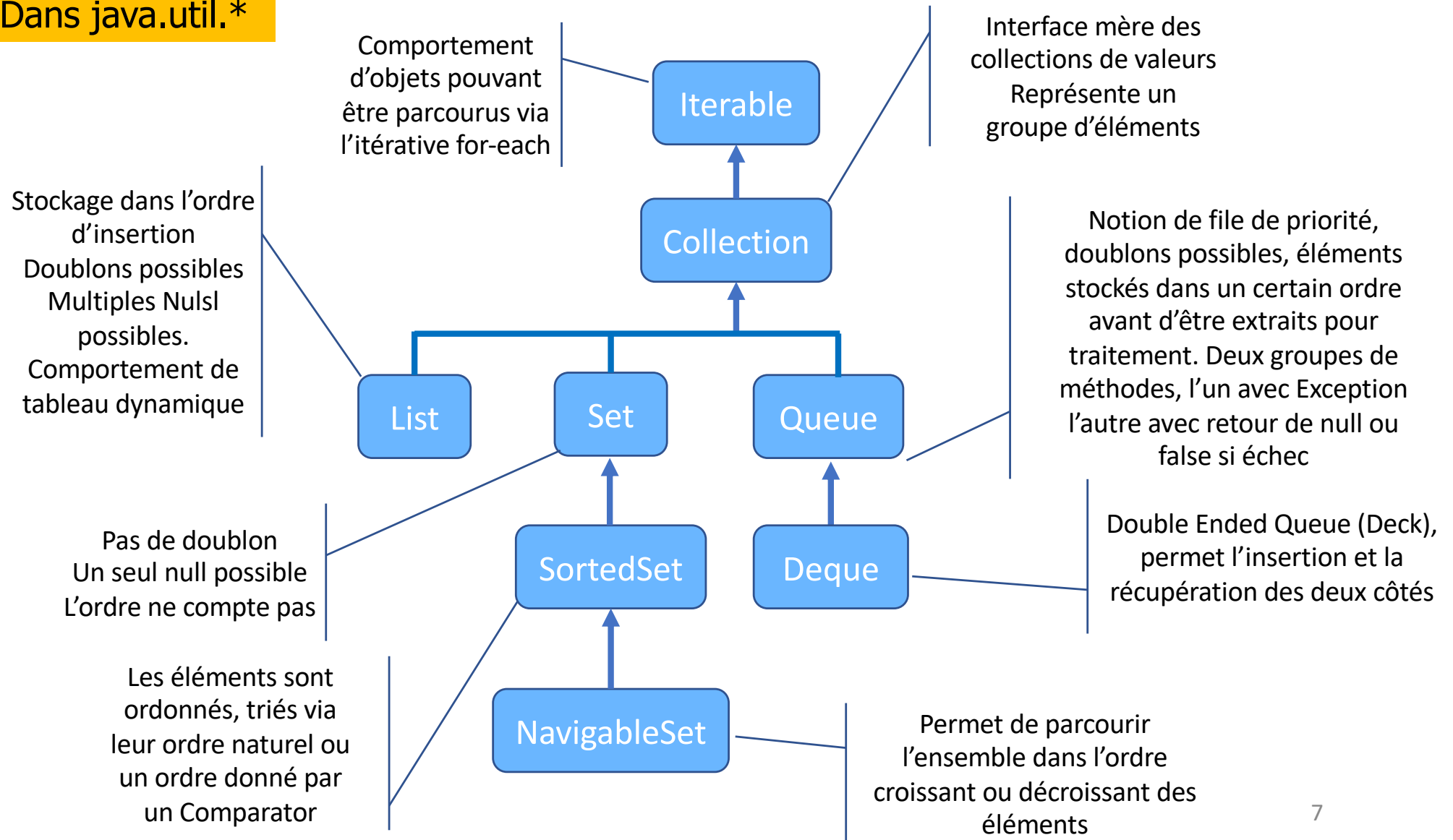
# Vue d'ensembles de la hiérarchie des interfaces Collection et tables associatives

Dans `java.util.*`

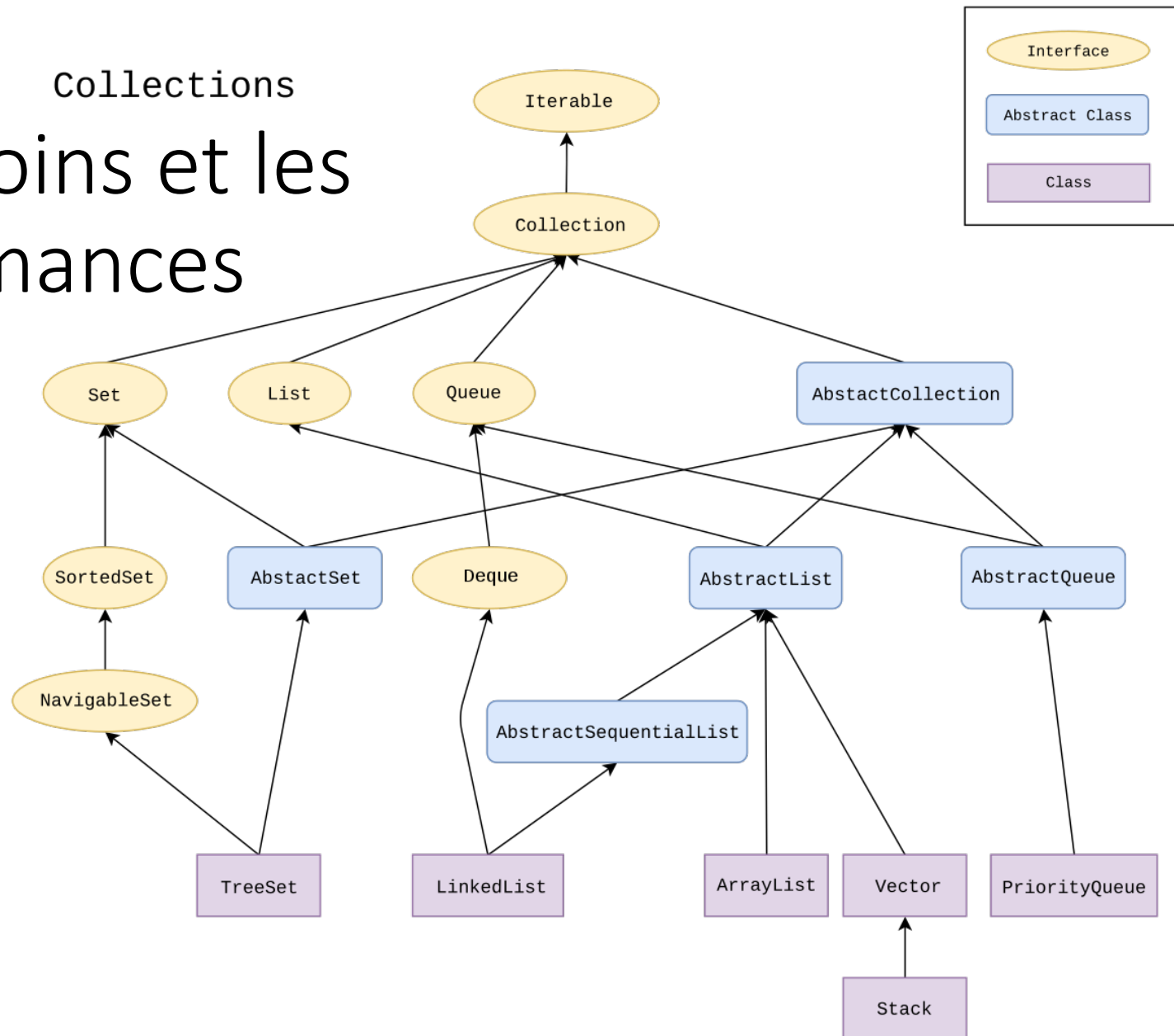


# Vue d'ensembles de la hiérarchie des interfaces Collection, stockage valeurs

Dans `java.util.*`

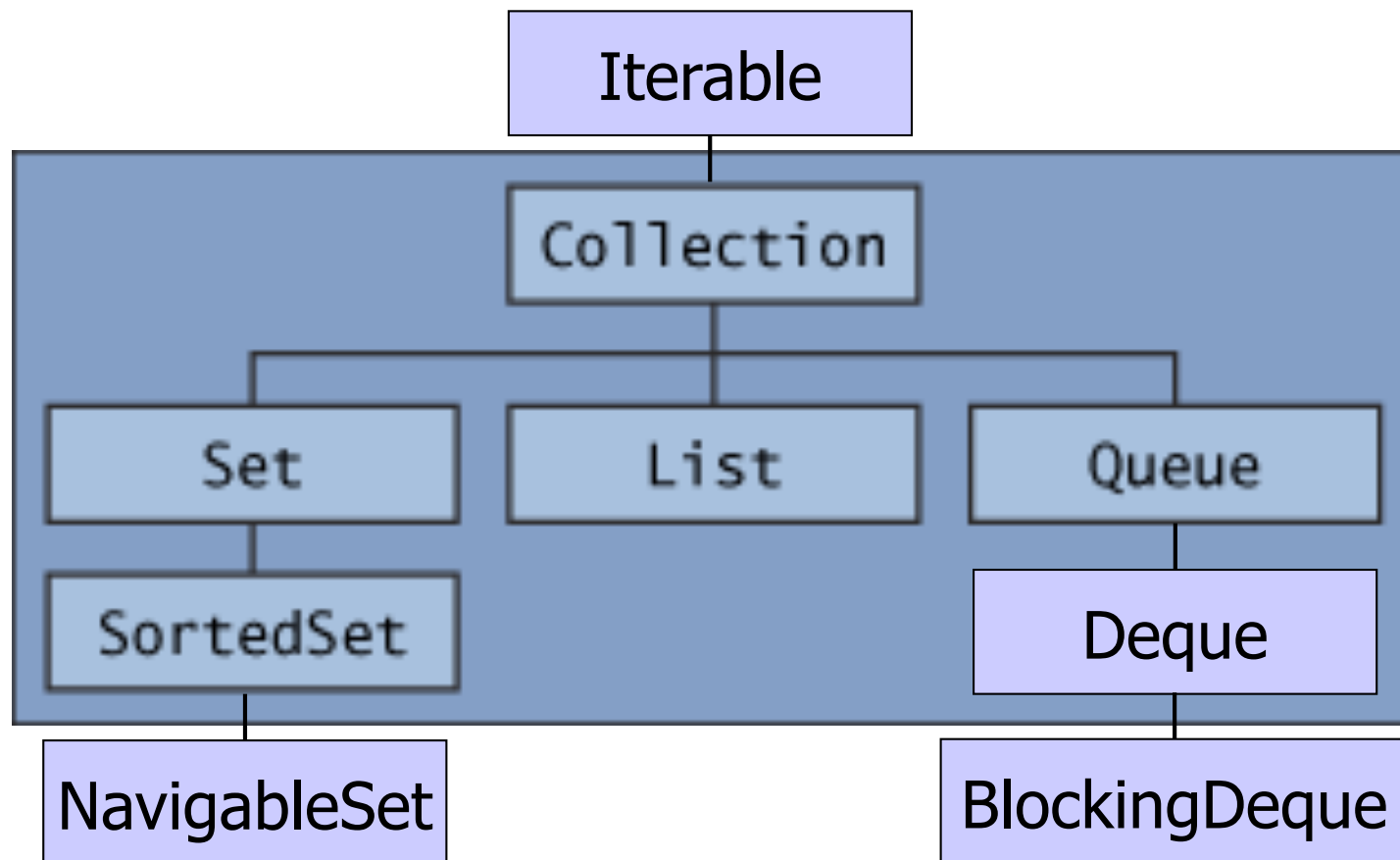


# Des classes abstraites pour faciliter l'implémentation et des classes concrètes selon les besoins et les performances





# L'interface Collection



# Description générale

- Représente un groupe d'objets, les éléments de la `Collection<E>`.
- Utilisée lorsqu'on recherche le maximum de généralité dans la manipulation des `Collection`
- Toutes les implémentations générales des collections ont un constructeur prenant une `Collection<E>` en argument → constructeur par conversion
- Certaines opérations (méthodes) proposées par les interfaces sont déclarées optionnelles
  - Certaines implémentation (classe implémentant l'interface) peuvent ne pas supporter des opérations (méthodes).
  - Traduit par la levée d'une `UnsupportedOperationException`
- Généricité depuis 1.5

# Code de l'interface Collection JDK 15

<https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Collection.html>

Retourne vrai si la collection change

```
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  
    default boolean removeIf(Predicate<? super E> filter)  
    Iterator<E> iterator();  
    default Spliterator<E> spliterator();  
    default Stream<E> parallelStream();  
    default Stream<E> stream();  
    // 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  
    Object[] toArray();  
    default <T> T[] toArray(IntFunction<T[]> generator)  
    <T> T[] toArray(T[] a);  
}
```

Méthode implémentée de Iterable (lambda expression)

```
default void forEach(Consumer<? super T> action)
```

# Utilisation de l'interface Collection comme d'un type

- Toujours rester le plus général possible, manipuler les collection d'objets via des références à des Objets Collection

```
// Création d'une instance d'ArrayList et manipulation
// de celle-ci via une référence au type Collection
Collection c1 = new ArrayList();
// Utilisation des méthodes de l'interface Collection et du
// polymorphisme. Par exemple, utilisation de la méthode add()
// C'est l'implémentation de la classe ArrayList qui va être
// utilisée (suivant l'implémentation la duplication sera ou
// non permise.
boolean b1 = c1.isEmpty();
boolean b2 = c1.add(new Integer(1));
```

Ancienne  
mouture, avant  
généricité

# Exercice

- Version générique classe

`TestCollectionGenerique`

- Créer une classe de Test

`TestCollectionGenerique` avec une méthode `main`

- Déclarer une référence à une Collection de chaines de caractères nommée `c`
- Mettre dans `c` une nouvelle `ArrayList` de chaines de caractères
- Ajouter deux chaines de caractères à `c`
- Afficher cette collection via sa référence

# Comment parcourir une collection ?

Il y a trois façons de parcourir les éléments d'une collection

- Instruction for (foreach)

```
for (typeElement element : nomCollection){  
    //traitement effectué sur element  
}
```

```
for (Object o : collection){  
    System.out.println(o);  
}
```

- Utilisation d'un objet Iterator
- Opérations d'agrégation basées sur la manipulation des Stream ou flux de données, apparus avec JSE 8 et le forEach de Iterable

# L'objet Iterator

- Objet retourné par la méthode `iterator()` de l'interface `Collection`
- Très similaire à une `Enumeration` avec pour différence qu'il est possible pour l'appelant d'enlever des éléments durant l'itération (impossible avec une `Enumeration`)
- Code de l'interface `Iterator`

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

Id à `Enumeration.hasMoreElements()`  
`Enumeration.nextElement`

Ôte le dernier elt retourné par `next`.

Peut être appelée une seule fois par appel de `next`. Sinon il y a levée d'exception

# Utilisations d'un Iterator

```
Iterator<Object> i = c.iterator();  
while (i.hasNext()) {  
    Object o = i.next();  
    //faire quelque chose avec o  
}
```

```
for (Iterator<Object> i = c.iterator(); i.hasNext(); ) {  
    Object o = i.next();  
    //faire quelque chose avec o  
}
```



# Parcourir une collection, opérations d'agrégation sur Streams (depuis JSE 8)

Exemple : itération sur une collection d'objets via un Stream et impression des noms des éléments rouges

```
myShapesCollection.stream()  
    .filter(e -> e.getColor() == Color.RED)  
    .forEach(e -> System.out.println(e.getName()));
```

Idem mais profitant du multi-cœur sur une grande collection d'objets

```
myShapesCollection.parallelStream()  
    .filter(e -> e.getColor() == Color.RED)  
    .forEach(e -> System.out.println(e.getName()));
```

Pour plus d'informations et exemple sur les opérations d'agrégation et les streams  
cf. <https://docs.oracle.com/javase/tutorial/collections/streams/index.html>

# forEach de Iterable

Méthode implémentée de Iterable (lambda expression)

```
default void forEach(Consumer<? super T> action)
```

Cette méthode parcourt la collection courante et applique une expression lambda, son paramètre, de type `Consumer<E>` sur chacun des éléments

```
c.forEach(e -> /*traiter l'element e */);
```

```
Collection<Person> roster; ...  
roster.stream()  
    .filter(e -> e.getGender() == Person.Sex.MALE)  
    .forEach(e -> System.out.println(e.getName()));
```

```
for (Person p : roster) {  
    if (p.getGender() == Person.Sex.MALE) {  
        System.out.println(p.getName());  
    }  
}
```

# Parcourir une collection, opérations d'agrégation sur Streams (depuis JSE 8)

Exemple : Conversion des éléments d'une collection  
de String en Strings séparées par une virgule

```
String joined = elements.stream()  
    .map(Object::toString)  
    .collect(Collectors.joining(", "));
```

Sommer les salaires de tous les employés

```
int total = employees.stream()  
    .collect(Collectors.summingInt(Employee::getSalary));
```

Ici il est question de l'utilisation de lambda expressions

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

Nous y reviendrons dans le cours Concepts des langages.

# Exercice

- Reprendre et modifier la classe

`TestCollectionGenerique`

- Déclarer une référence à une Collection de chaines de caractères nommée `c`
- Mettre dans `c` une nouvelle `ArrayList` de chaines de caractères
- Récupérer les paramètres passés en ligne de commande (stockés dans le tableau d'argument du `main`) et les stocker dans `c`
  - Les paramètres seront « 33 un deux trois quatre 22 »
- Affichage
  - Parcourir et **afficher les chaines de deux caractères** avec les :
    - agregate opération (JSE 8)
    - avec une boucle `for`
    - avec un `iterator`
  - Afficher simplement la collection `c` avec une instruction `println`

# Comment choisir entre les deux méthodes : for ou iterator ?

- Vous devez utiliser un `iterator` à la place d'un `for` lorsque :
  - Vous avez besoin d'enlever l'élément courant, par extension lorsque vous souhaitez faire une opération de filtrage
  - Vous avez besoin de remplacer les éléments d'une liste ou d'un tableau pendant que vous les parcourez
  - Vous avez besoin de parcourir plusieurs collections en parallèle

Pour les opérations d'agrégation sur Stream, nous y reviendrons

# Exemple d'utilisation d'une collection pour un filtrage

- Principe :
  - Parcours de la collection
  - Utilisation de la méthode remove pour ôter les éléments ne satisfaisant pas une condition donnée

```
static void filter(Collection<?> c) {  
    for (Iterator<?> i=c.iterator(); i.hasNext(); )  
        if (!cond(i.next())) i.remove();  
}
```

Utilisation du polymorphisme : ce code marche quelque soit la l'implémentation de la Collection à partir du moment ou elle supporte l'opération remove.

# Parcourir pour filtrage

- Depuis la version JDK 8 on peut également utiliser les prédicats, lambda expression

```
public boolean removeIf(Predicate<? super E> filter)
```

```
package l3.poo.cours.collections;
import java.util.ArrayList;
import java.util.Collection;
public class MainCollection {
    public static void main(String[] args) {
        Collection<Integer> cInteger = new ArrayList<>();
        cInteger.add(1);cInteger.add(10); cInteger.add(11);
        cInteger.add(100);cInteger.add(101);cInteger.add(1000);

        cInteger.add(1001);cInteger.add(10000);cInteger.add(10001);
        System.out.println("Avant filtrage " + cInteger);
        cInteger.removeIf(e -> (e%2 == 0));
        System.out.println("Après filtrage " + cInteger);
    }
}
```

Ici il est question de l'utilisation de lambda expressions

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

# Exercice

- Reprendre et modifier la classe

`TestCollectionGenerique`

- Déclarer une référence à une Collection de chaînes de caractères nommée `c`
- Mettre dans `c` une nouvelle `ArrayList` de chaînes de caractères
- Récupérer les paramètres passés en ligne de commande (stockés dans le tableau d'argument du `main`) et les stocker dans `c`
  - Les paramètres seront « 33 un deux trois quatre 22 »
- Modifiez la collection `c` de façon à n'y laisser que les chaînes de plus de 2 caractères
- Affichez là...



# Les traitements généraux

- Les "*bulk operations*" permettent d'opérer des traitement sur tous les éléments d'une collection en une seule instruction.
- `boolean containsAll(Collection c)` : retourne vrai si l'instance courante de `Collection` (la cible) contient tous les éléments de `c`.
- `boolean addAll(Collection c)` : ajoute tous les éléments de `c` à la `Collection` cible.
- `boolean removeAll(Collection c)` : ôte de la `Collection` cible tous les éléments contenus dans `c`.

# Les traitements généraux

- `boolean retainAll(Collection c)` : ôte de la Collection cible tous les éléments qui ne sont pas également présents dans `c`. Conserve seulement les éléments présents dans `c`.
- `void clear()` : ôte tous les éléments de la Collection.
- Exemple d'utilisation
  - Ôter un élément `e` d'une Collection, `c`:

```
c.removeAll(Collections.singleton(e));
```

- Ôter les référence nulles d'une Collection, `c`:

```
c.removeAll(Collections.singleton(null));
```

Retourne un ensemble (Set) contenant seulement l'élément spécifié

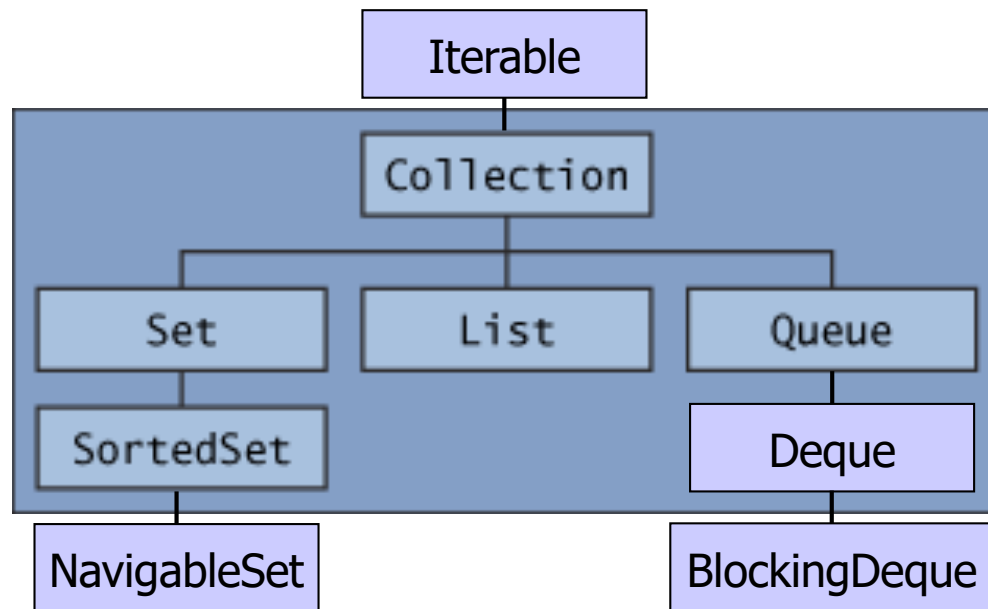
# Passerelles vers les tableaux

- Les méthodes `toArray` permettent de faire des passerelles avec les outils des autres API qui attendent des tableaux en entrée.
- Exemple d'utilisation

```
Object[] a = c.toArray();
```

```
String[] a = c.toArray(new String[0]);
```

# L'interface List



Les listes...

# Description générale d'une liste

- Collection ordonnée (une *séquence*).
  - $\langle a, b, c \rangle$ ,  $\langle c, b, a \rangle$  et  $\langle b, c, a \rangle$  ne sont pas les mêmes listes
- Structure de donnée linéaire
- Peut contenir des doublons
- Opérations classiques + opérations spécifiques :
  - Accès indexé, commençant à 0
  - Recherche et retour d'index
  - Parcours itératif adapté à la nature séquentielle de la liste
  - Traitement par intervalle

# Le code de l'interface, ajout des accès indexés

```
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  
  
    // Bulk Operations  
    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);
```

L'index de début est 0

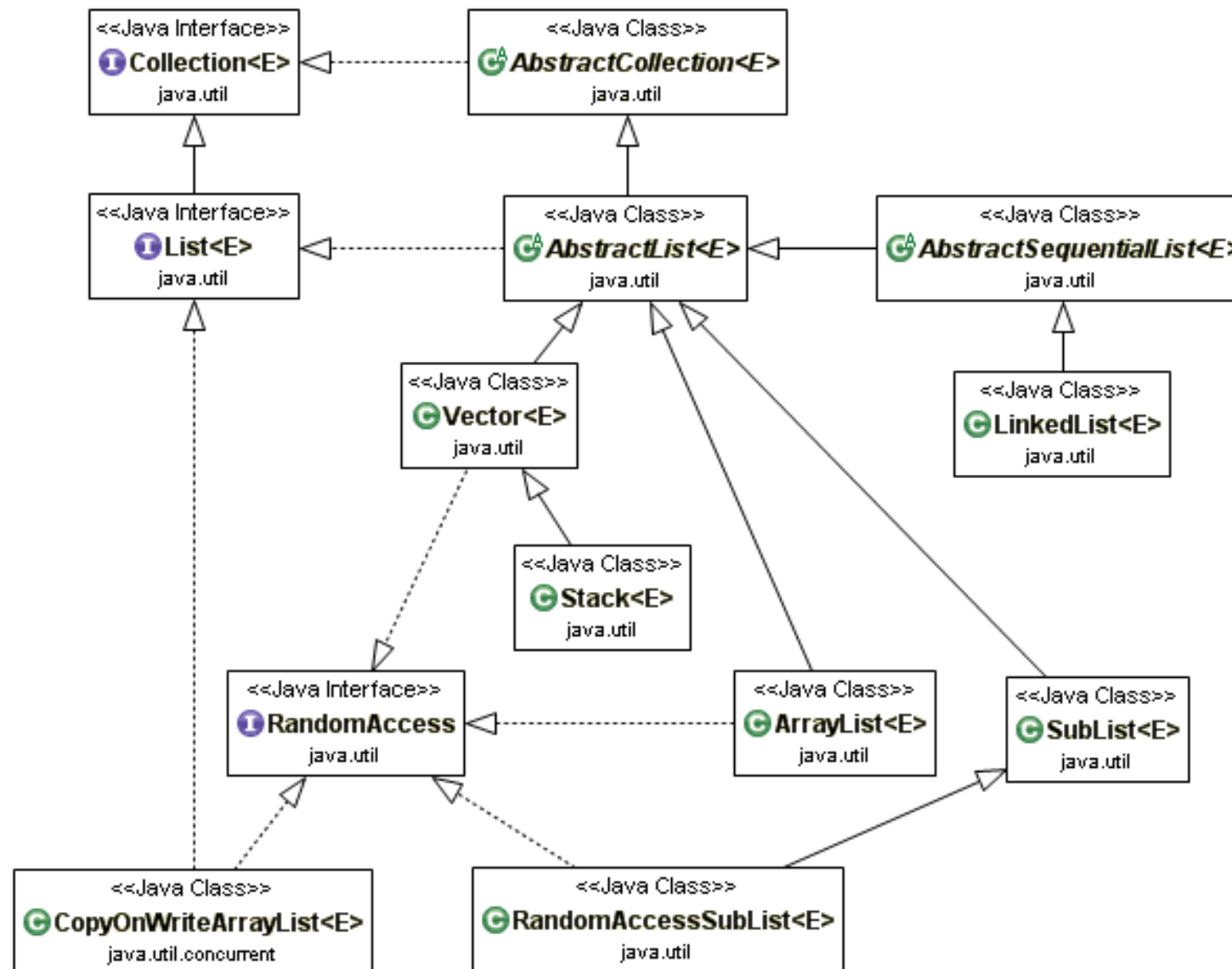
inclus      exclus

**La méthode héritée**

`boolean add(Object element)`  
ajoute l'élément en fin de la liste

# Vision globale de la hiérarchie des Listes

<https://www.jmdoudoux.fr/java/dej/chap-collections.htm>



# Les classes implémentant `List`

- `ArrayList<E>`
  - implémentation sous la forme de tableaux redimensionnable de l'interface `List<E>`.
  - contexte `monothread`
  - bonnes performance d'accès
- `LinkedList<E>`
  - Implémentation sous forme d'une liste doublement chaînée
  - non `synchronized`, contexte `monothread`
  - bonnes performances sur l'insertion
- `Vector<E>` (Historique)
  - Tableau redimensionnable
  - `Synchronized`, `thread-safe`
- `Stack<E>` (Historique)
  - `Synchronized`, `thread-safe`
  - Last In First Out
- `java.util.concurrent.CopyOnWriteArrayList<E>`
  - Variante `thread-safe` de `ArrayList<E>`



# Choix en termes de contexte et de performances

- Si pas d'accès concurrent : ArrayList ou LinkedList
  - Si ajout et/ou suppression en fin de liste : ArrayList
  - Si ajout et/ou suppression n'importe où : LinkedList
- Si accès concurents : CopyOnWriteArrayList ou Vector ou Collections.synchronizedList()

	get	add	contains	next	remove(0)	iterator.remove
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
CopyOnWriteArrayList	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

# Parcours de liste via l'interface `ListIterator`

- En plus des parcours déjà vus pour les collection on ajoute le parcours via `ListIterator`
- Interface fille de `Iterator`

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

- Permet de
  - parcourir la liste dans les deux directions,
  - modifier la liste pendant le parcours
  - obtenir la position courante de l'itérateur

# Code de ListIterator

```
public interface ListIterator<E> extends Iterator<E> {  
    boolean hasNext();  
    E next();  
    boolean hasPrevious();  
    E previous();  
    int nextIndex();  
    int previousIndex();  
    void remove(); // Optional  
    void set(E o); // Optional, last elt return by next  
    void add(E o); // Optional, before current cursor position  
}
```

# Exemple de code, parcours d'une liste

## Parcours à l'envers

```
for  
(ListIterator<Type> i=l.listIterator(l.size()-1);i.hasPrevious();) {  
    Type f = i.previous(); // ou int index = i.previousIndex();  
    ...  
}
```

Retourne un itérateur positionné à l'index spécifié

## Parcours à l'endroit

```
for (ListIterator<Type> i=l.listIterator();i.hasNext(); ) {  
    Type f = i.next(); // ou int index = i.nextIndex();  
    ...  
}
```

Retourne un itérateur positionné au début de la liste

# Exercice

Écrire une méthode de classe générique prenant en paramètre une liste d'éléments d'un type E, deux entiers i et j et effectuant la permutation dans la liste des valeurs d'index i et j

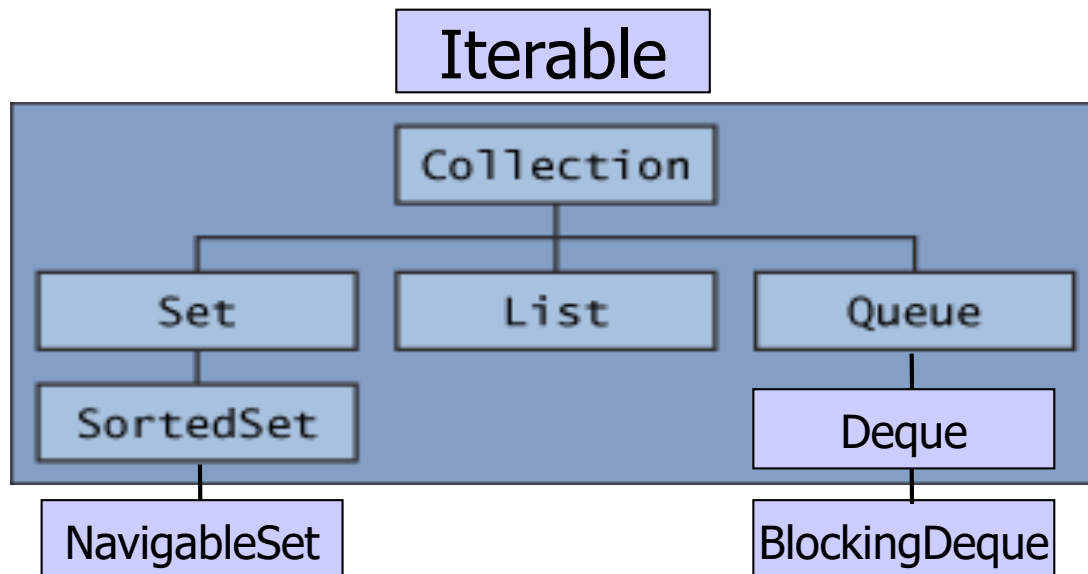
# Exercice

- Écrire une méthode de classe générique prenant en paramètre une liste d'éléments d'un type E, et deux éléments de ce même type E, permettant de remplacer dans la liste de départ toutes les occurrences de la première valeur par la deuxième
- Attention cela doit également marcher si on demande à remplacer une valeur nulle...

# Exercice

- Écrire une méthode de classe générique prenant en paramètre une liste d'éléments d'un type E, un élément de ce même type E et une autre liste d'éléments du type E, permettant de remplacer dans la liste de départ toutes les occurrences de la valeur par les éléments de la deuxième liste.
- Attention cette méthode doit également fonctionner si la valeur recherchée est la valeur nulle.

# L'interface Set



Les ensembles...



# Description générale de Set

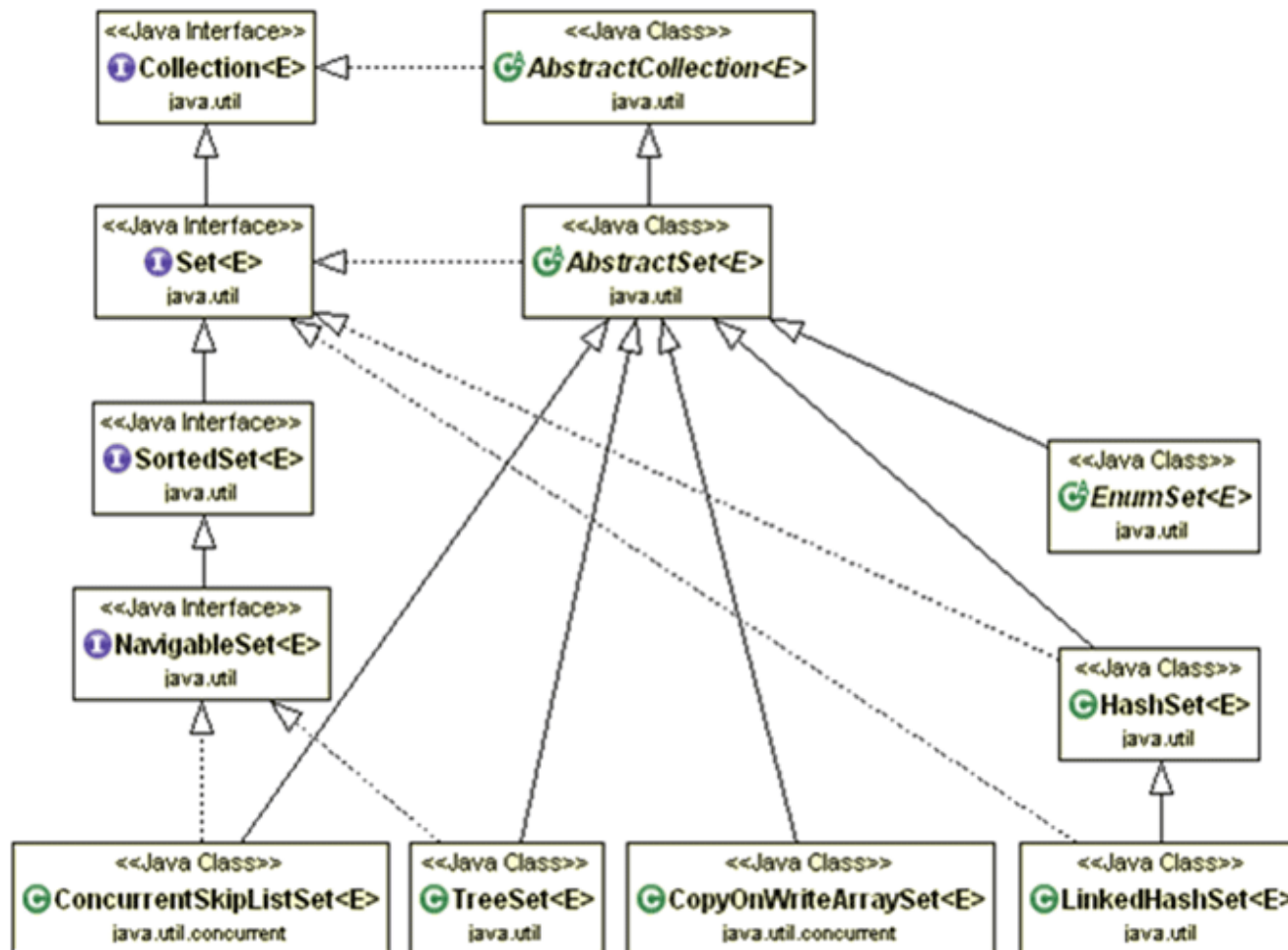
- Interface fille de `Collection`
- Aucune méthode ajoutée
- `Collection` qui **ne peut contenir de doublons** en accord avec le concept mathématique d'ensemble
  - Restriction à l'ajout via un test d'égalité (au sens d'`equals`) avant insertion
  - Redéfinition obligatoire de `equals` et `hashCode`
  - Attention : la non duplication d'objets n'est pas assurée si les objets sont modifiés après l'ajout
- Deux instances d'ensemble sont égales si elles contiennent les même éléments
  - L'ordre n'est pas important :  $\{a,b,c\}$  est le même ensemble que  $\{b,a,c\}$

# Les classes génériques du JDK implémentant Set

- `HashSet<E>` : stockage des éléments dans une table de hachage
  - Contexte mono thread
  - Possibilité d'ajout d'un élément null
  - meilleures performances (temps constant des opérations de base), aucune garantie concernant l'ordre de parcours
- `TreeSet<E>` : stockage des éléments dans un arbre binaire (red-black tree)
  - Contexte mono thread
  - impossibilité d'ajout d'un élément null
  - éléments ordonnés suivant leur valeur (ordre naturel, interface `Comparable`),
  - beaucoup plus lent que `HashSet`
- `LinkedHashSet<E>` : stockage des éléments dans une table de hachage avec une liste chaînée
  - un peu plus lent que `HashSet`, éléments ordonnés suivant leur ordre d'insertion
- `EnumSet<E extends Enum<E>>` : tous les éléments de la collection doivent appartenir à la même énumération
- `java.util.concurrent.CopyOnWriteArraySet<E>` et `java.util.concurrent.ConcurrentSkipListSet<E>`

# Vision globale de la hiérarchie des Set

<https://www.jmdoudoux.fr/java/dej/chap-collections.htm>



# Exemple d'ajout

```
public class MyOwnUtilityClass {  
    public static void checkDuplicate(Set s, String[] args){  
        for (int i=0; i<args.length; i++)  
            if (!s.add(args[i]))  
                System.out.println("Duplicate detected: "+args[i]);  
        System.out.println(s.size()+" distinct words detected: "+s);  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Set s = new HashSet(); // Ordre non garanti  
        MyOwnUtilityClass.checkDuplicate(s, args);  
        s = new TreeSet(); // Ordre naturel  
        MyOwnUtilityClass.checkDuplicate(s, args);  
        s = new LinkedHashSet(); // Ordre d'insertion  
        MyOwnUtilityClass.checkDuplicate(s, args);  
    }  
}
```

# Résultat d'exécution

- Les paramètres sont passés dans l'ordre suivant :  
2, 3, 4, 1, 2
- Ordre de stockage obtenu suivant les différentes implémentations
  - `java.util.HashSet` [3, 2, 4, 1]
  - `java.util.TreeSet` [1, 2, 3, 4]
  - `java.util.LinkedHashSet` [2, 3, 4, 1]

# Exercice, exemple

- Construire une collection sur la base d'une autre nommée  $C$  en enlevant tous les doublons

# Exercice : code 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 detected: "+s);
    }
}
```

Référence via  
le type interface

Exécutez ce code via la ligne de commande :

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

Qu'obtenez vous à l'affichage ?

# L'interface SortedSet

- Décrit le comportement des ensembles ordonnés en garantissant le parcours dans l'ordre ascendant
- Les éléments contenus dans l'ensemble doivent être munis d'une relation d'ordre pour pouvoir comparer les éléments entre eux via :
  - Un objet `Comparator<E>`
  - Une implémentation de `Comparable<E>` pour la classe dont sont issus les éléments



# SortedSet<E>

```
public interface SortedSet<E> extends Set<E>{  
    E first()  
    E last()  
    SortedSet headSet(E toElement)  
    SortedSet tailSet(E fromElement)  
    SortedSet subSet(E fromElement, E toElement)  
    Comparator< ? super E> comparator()  
}
```

# Opérations générales équivalence avec les ensembles

- `s1.containsAll(s2)`
  - Vrai si `s2` est un **sous-ensemble** de `s1`.
- `s1.addAll(s2)`
  - Effectue l'**union** de `s1` et `s2` et met le résultat dans `s1`.
- `s1.retainAll(s2)`
  - Effectue l'**intersection** de `s1` et `s2` et met le résultat dans `s1`.
- `s1.removeAll(s2)`
  - Effectue la **différence** entre `s1` et `s2` et met le résultat dans `s1` (ensemble contenant les éléments de `s1` qui ne sont pas dans `s2`)

# Exercice

- Écrire un code permettant de faire la différence symétrique non destructive entre deux ensembles  $e1$  et  $e2$ , et de stocker le résultat dans un ensemble  $e3$
- Explication des termes :
  - Différence symétrique entre  $e1$  et  $e2$  :  $e3$  doit contenir les éléments contenus dans  $e1$  ou dans  $e2$ , mais pas ceux contenus dans les deux
  - Non destructive :  $e1$  et  $e2$  doivent rester inchangés

# NavigableSet

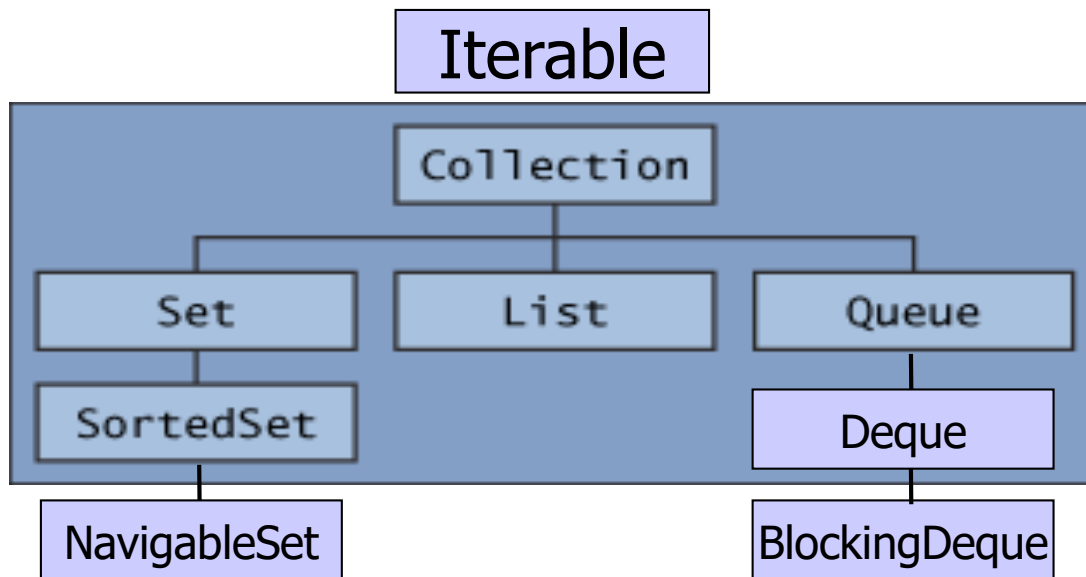
- Permet la navigation
  - Accès et parcours selon ordre croissant ou décroissant
  - Les méthodes lower (plus petit), floor (plus petit ou égal), ceiling (plus grand ou égal), higher (plus grand) retournent les éléments ... qu'un élément donné ou null sinon
  - Les méthodes poolFirst et poolLast retourne et ôte le plus petit ou plus grand élément de l'ensemble

# Choix des Set en terme de performances

	add()	contains()	next()	thread-safe
HashSet	$O(1)$	$O(1)$	$O(h/n)$	Non
LinkedHashSet	$O(1)$	$O(1)$	$O(1)$	Non
CopyOnWriteArraySet	$O(n)$	$O(n)$	$O(1)$	Non
EnumSet	$O(1)$	$O(1)$	$O(1)$	Oui
TreeSet	$O(\log n)$	$O(\log n)$	$O(\log n)$	Oui
ConcurrentSkipListSet	$O(\log n)$	$O(\log n)$	$O(1)$	Oui

<https://www.jmdoudoux.fr/java/dej/chap-collections.htm>

# L'interface Queue



Les files...

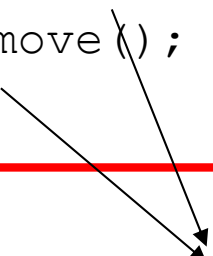
*Prononcez "Deck" pour  
Double Ended Queue*

# Description générale

- `Collection` permettant de stocker les éléments préalablement à un traitement
- Interface fille de `Collection`
  - Méthodes supplémentaires d'insertion, de suppression et d'inspection
- Généralement les éléments sont ordonnés en respectant le principe FIFO (first-in-first-out)
  - Possibilité d'utiliser un autre principe
  - Obligation de définir une propriété d'ordonnancement
- Comportement général
  - Quelque soit l'ordre, la tête de queue est l'élément qui sera retourné (enlevé) par un appel à `remove` ou `poll`
  - Dans une FIFO, tous les nouveaux éléments sont insérés à la fin de la queue, les autres types de queue peuvent utiliser d'autres règles de placement

# Code de l'interface Queue

```
public interface Queue<E> extends Collection<E> {  
    E element(); //retourne la tête  
    boolean offer(E o); //faux si l'ajout a échoué, sur Q bornée  
    E peek(); //retourne la tête  
    E poll(); // enlève et retourne la tête de queue  
    E remove(); // enlève et retourne la tête de queue  
}
```



Comportement différent sur Queue vide,  
poll retourne null, remove lance une exception  
NoSuchElementException



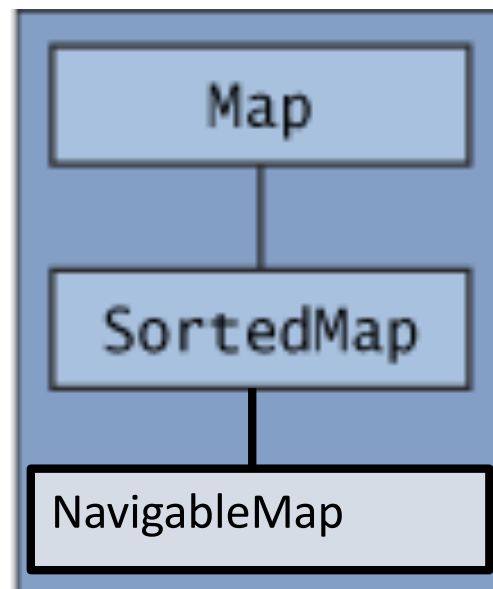
# Les classes génériques du JDK implémentant Queue

- Utilisation commune
  - `LinkedList`, liste chaînée, FIFO
  - `PriorityQueue`, basée sur une structure de données tas, à la construction on doit spécifier la règle d'ordonnancement des éléments
- Utilisation concurrente (synchronisation des accès), interface `BlockingQueue` (wait si vide ou pleine)
  - `LinkedBlockingQueue` , liste chaînée, FIFO
  - `ArrayBlockingQueue` , FIFO de taille bornée modélisée dans un tableau
  - `PriorityBlockingQueue` , structure en Tas (cf. prec)
  - `DelayQueue` , structure en Tas, ordre défini sur le temps
  - `SynchronousQueue` , suit les principes du rendez-vous

# L'interface Deque dit « *Deck* »

- Nouvelles interfaces
  - Deque : Double ended queue
    - Queue qui supporte l'insertion et le retrait des éléments aux deux extrémités
    - Interface fille de l'interface Queue
  - BlockingDeque
    - Attente du fait que la Deque ne soit plus vide lors du retrait d'un élément, ou qu'elle ne soit plus pleine lors de l'ajout
    - Interface fille de Deque et de BlockingQueue

# L'interface Map



Tables associatives,  
Dictionnaires,  
Tables de hachage

# Définition, tables associatives (dictionnaire)

- Type de données associant à une clé une valeur
- Chaque clé est associée à une seule valeur
- Généralisation du concept de tableau :
  - Tableau : association entre index successifs entiers et valeurs
  - Table associative : association entre une clé d'un type quelconque à une valeur
- Une façon d'implanter une table associative est de créer une liste chaînée de paires clé-valeurs
  - Inefficace mais simple !
- Une autre façon est d'utiliser une table de hachage

# Différences par rapport aux tableaux

## Intérêts des tables de hachage ?

Tableau indexé

Index	Valeur
0	12,5
1	8
...	...
N	17

Tableau associatif

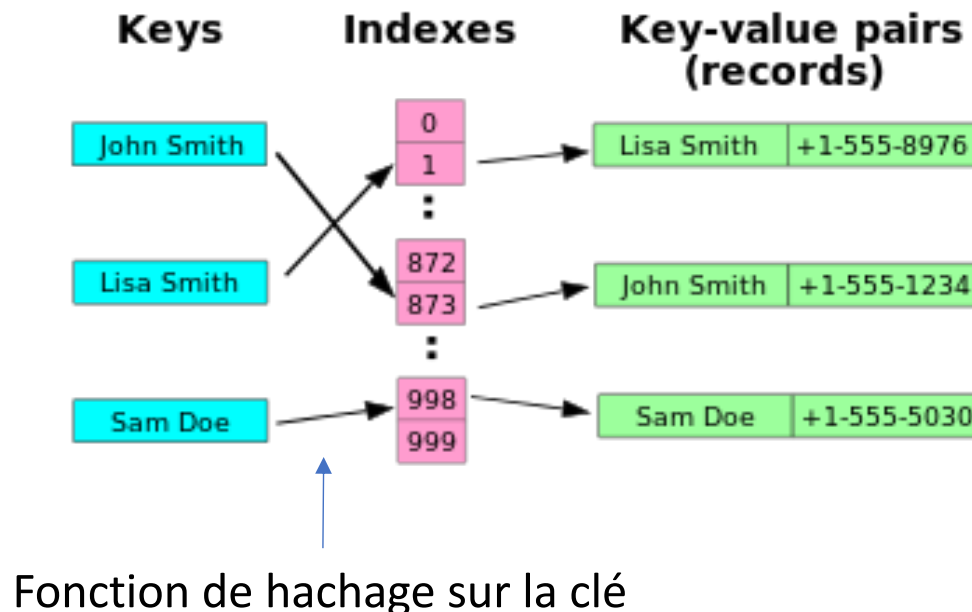
Clé	Valeur
`Paul`	12,5
`Jean`	8
...	...
`Etienne`	17

``Paul`.hashCode() = 0`

Bonnes performances en terme d'insertion, de recherche et de retrait, proche de  $O(1)$  et toujours  $< O(n)$

# Tables associatives vs Tables de Hachage

- Une table de hachage est une implémentation d'une table associative ou on utilise une fonction de hachage pour obtenir l'indice auquel est stocké le couple clé-valeur.



# Description générale

- Concept des tables de Hachage
  - Correspond aux collections indexées par des clés, groupe de couples objet-clé
  - Il ne peut y avoir deux fois la même clé dans la table
  - Une clé correspond à un et un seul objet
  - En fonction de la clé on obtient un indice permettant de ranger le couple objet-clé dans la table
    - Via la fonction de hachage → `hashCode()`
  - Généralisation du concept de tableau

Un tableau est une table de hachage ou la clé est l'indice  
Cf. [https://www.youtube.com/watch?v=KyUTuwz\\_b7Q](https://www.youtube.com/watch?v=KyUTuwz_b7Q)

# Principe du Hachage

- Hatables and Hash Functions :  
[https://www.youtube.com/watch?v=KyUTuwz\\_b7Q](https://www.youtube.com/watch?v=KyUTuwz_b7Q)
- La classe Object propose une implémentation par défaut de la méthode hashCode() qui renvoie la référence de l'objet sous la forme d'une valeur de type int.
  - On peut toujours récupérer la valeur du hashcode par défaut via System.identityHashCode
- Règles à respecter dans la redéfinition de la fonction de Hachage en Java
  - Redéfinition de equals = redéfinition de hashcode
  - Si deux objets sont égaux, ils doivent avoir le même hashcode. L'inverse n'est pas vrai !

Les conseils avisés de JM Doudoux

[https://www.jmdoudoux.fr/java/dej/chap-techniques\\_java.htm](https://www.jmdoudoux.fr/java/dej/chap-techniques_java.htm)



# Règles de redéfinition de hashCode() en Java

## equals

```
public boolean equals(Object obj)
```

Indicates whether some other object is "equal to" this one.

The equals method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value *x*, *x.equals(x)* should return *true*.
- It is *symmetric*: for any non-null reference values *x* and *y*, *x.equals(y)* should return *true* if and only if *y.equals(x)* returns *true*.
- It is *transitive*: for any non-null reference values *x*, *y*, and *z*, if *x.equals(y)* returns *true* and *y.equals(z)* returns *true*, then *x.equals(z)* should return *true*.
- It is *consistent*: for any non-null reference values *x* and *y*, multiple invocations of *x.equals(y)* consistently return *true* or consistently return *false*, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value *x*, *x.equals(null)* should return *false*.

The equals method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values *x* and *y*, this method returns *true* if and only if *x* and *y* refer to the same object (*x == y* has the value *true*).

Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes.

### Parameters:

*obj* - the reference object with which to compare.

### Returns:

*true* if this object is the same as the *obj* argument; *false* otherwise.

### See Also:

`hashCode()`, `HashMap`

# Règles de redéfinition de hashCode en Java

## hashCode

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by [HashMap](#).

The general contract of hashCode is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

### Returns:

a hash code value for this object.

### See Also:

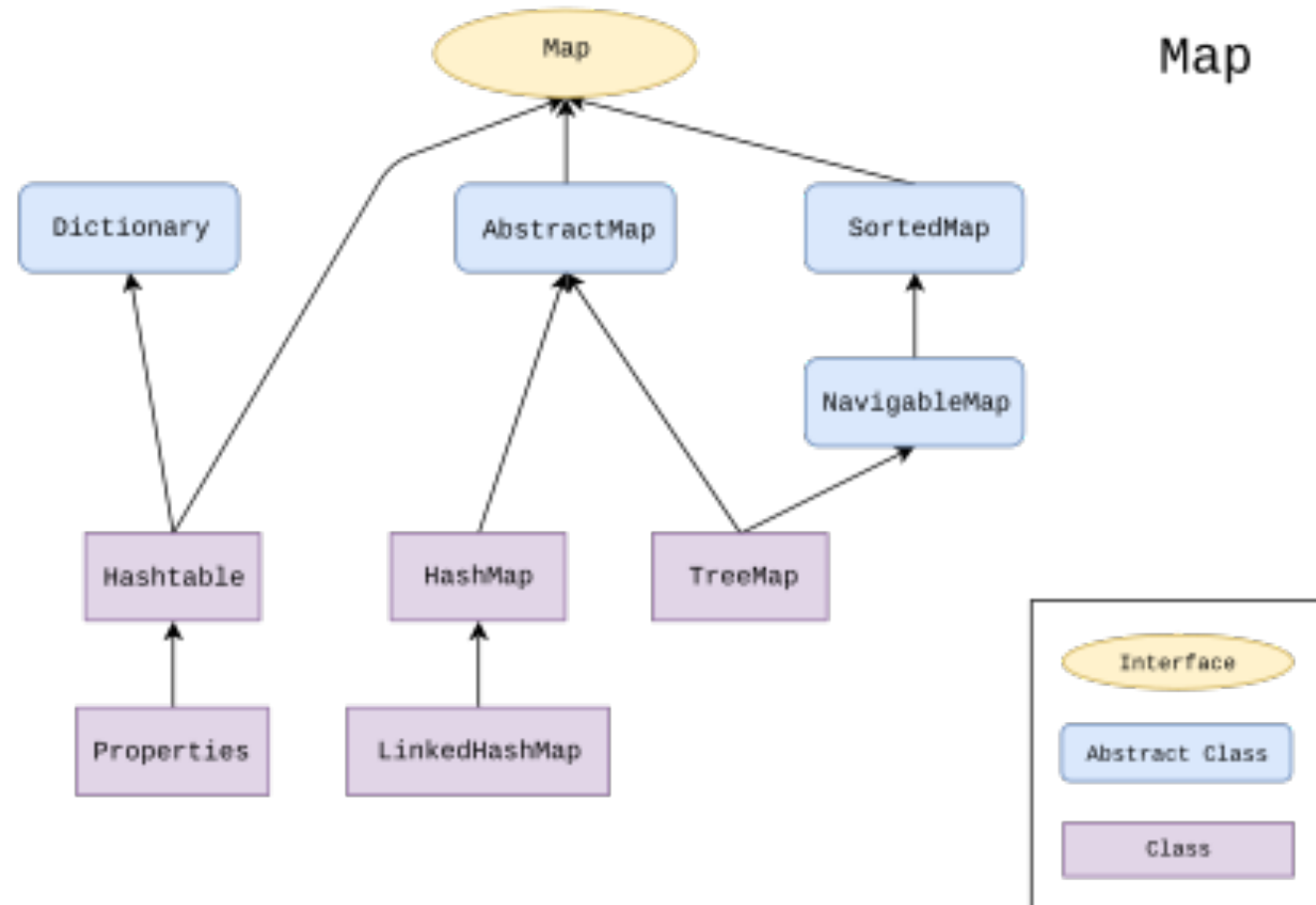
[equals\(java.lang.Object\)](#), [System.identityHashCode\(java.lang.Object\)](#)

# Code de l'interface Map

```
public interface Map<K,V> {  
    //Basic Operation  
    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> t);  
    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,V> {  
    K getKey();  
    V getValue();  
    V setValue(V value);  
}
```

# Hiérarchie des Map



# Les classes génériques du JDK implémentant `Map<K, V>`

- `HashMap`
  - implémentation sous la forme d'une table de hachage
  - utilisation possible du null comme clé
  - pas de conservation de l'ordre
- `TreeMap`
  - implémentation sous la forme d'un arbre binaire équilibré (red-black tree)
  - stockage par valeur de clés croissantes (ordre des clés garanti)
  - pas de gestion de la synchronisation
- `LinkedHashMap`
  - pas de gestion de la synchronisation
  - conserve l'ordre d'insertion des clés
- `HashTable`
  - thread-safe, gestion de la synchronisation
  - null non utilisable comme clé
- ...

# Exercice : Calcul de fréquence

- Écrivez un programme permettant de calculer la fréquence de mots passés en paramètre d'un programme
- Exemple d'utilisation et de résultat

```
% java Freq if it is to be it is up to me to delegate
```

```
8 distinct words:
```

```
{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}
```

# L'interface SortedMap

- Table de Hachage qui est ordonnée suivant l'ordre des clés ascendantes
  - Équivalent pour les tables de hachage à l'interface SortedSet
- Utilisée lorsque le but est de maintenir une collection de couples clé/valeur avec ordre sur les clés
  - Dictionnaires
  - Annuaire téléphoniques

# Ponts entre Map et Collection

```
public Set keySet();  
    public Collection values();  
    public Set entrySet(); //Ensemble des couples clé-objet
```

Ce n'est pas un Set car il peut y avoir plusieurs fois la même valeur

- Intérêt du pont : pouvoir itérer sur les Map, c'est le seul moyen...

```
for (KeyType key : m.keySet())  
    System.out.println(key);
```

```
// Filter a map based on some property of its keys  
for (Iterator<Type> i=m.keySet().iterator(); i.hasNext();)  
    if (i.next().isBogus()) i.remove();
```

```
for (MapEntry<KeyType, ValType> e : m.entrySet())  
    System.out.println(e.getKey() + ": " + e.getValue());
```



# Exercice

- Écrire une instruction permettant de savoir si une Map donnée (m1) est une sous-Map d'une autre (m2), c'est-à-dire si m1 contient tous les couples clef/valeur de m2
- Écrire une instruction permettant de savoir si m1 et m2 contiennent des enregistrements pour les mêmes clés.

# Les multimap...

- Une *multimap* est comme une map qui peut associer à une clé plusieurs valeurs
- Pas de classe spécifique
- Technique :
  - Associer à chaque clé d'entrée une valeur sous forme de `List`
- Exercice d'utilisation
  - Rechercher dans un fichier de mots (un mot par ligne) tous les mots étant constitués des mêmes lettres et afficher ces mots s'il y en a plus d'un certain nombre (passé en paramètre)

# Raisonnement

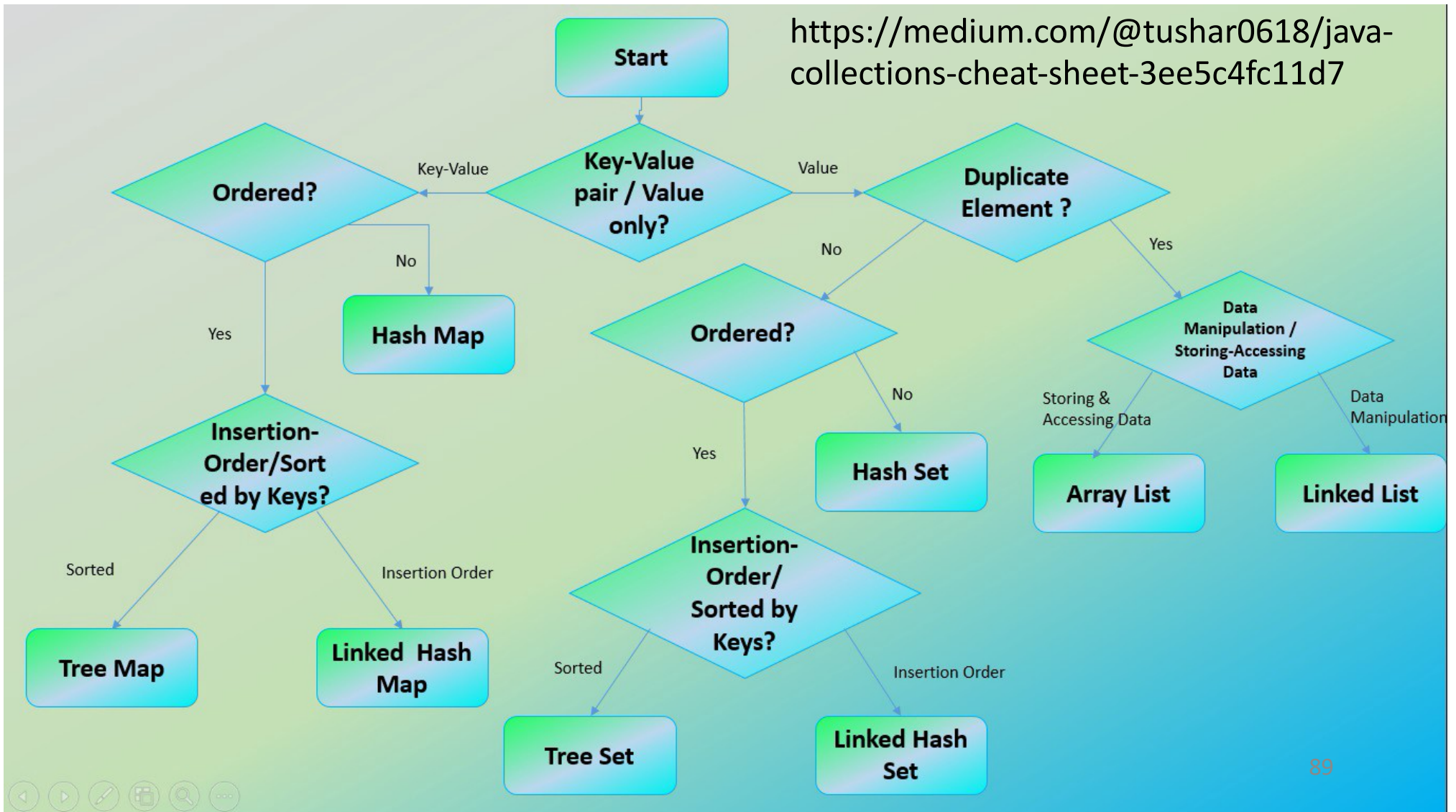
- Ouverture du fichier (nom passé en paramètre)
- Création d'une Map
- Pour chaque ligne lue
  - Classer les lettres du mot par ordre alphabétique et regarder dans la table s'il existe une entrée correspondante
    - Si oui, ajouter le mot à la liste existante
    - Si non, créer une nouvelle liste et ajouter le mot
- Afficher les listes de permutation lorsqu'elle ont plus de x éléments (x passé en paramètre)

# NavigableMap

- Permet la navigation
  - Parcours dans l'ordre croissant ou décroissant des clés
  - Les méthodes, `lowerEntry`, `floorEntry`, `ceilingEntry`, et `higherEntry` retournent des objets `Map.Entry` associés avec des clés respectivement plus petites, plus petites ou égales, plus grandes ou égales, et plus grandes qu'une clé données et null s'il n'y a pas de telle clé
  - Ces méthodes permettent de localiser les éléments

# Les questions à se poser...

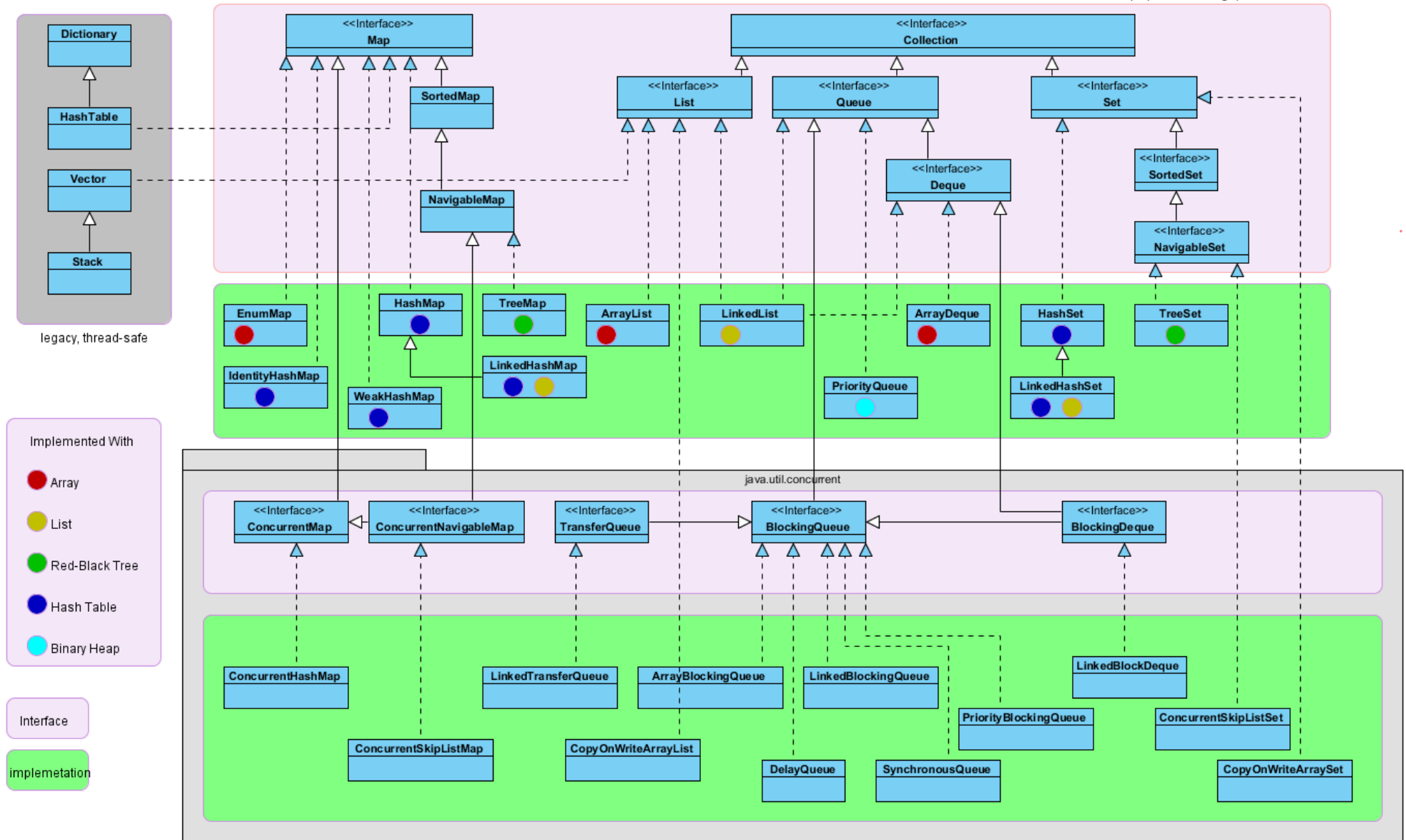
<https://medium.com/@tushar0618/java-collections-cheat-sheet-3ee5c4fc11d7>



# Pour résumer

Java Collection Cheat Sheet

<http://pierrchen.blogspot.com>



# JAVA COLLECTIONS

## Cheat Sheet

List	Add	Remove	Get	Contains	Next	Data Structure
ArrayList	O(1)	O(n)	O(1)	O(n)	O(1)	Array
LinkedList	O(1)	O(1)	O(n)	O(n)	O(1)	Linked List
CopyOnWriteArrayList	O(n)	O(n)	O(1)	O(n)	O(1)	Array
Set	Add	Remove	Contains	Next	Size	Data Structure
HashSet	O(1)	O(1)	O(1)	O(h/n)	O(1)	Hash Table
LinkedHashSet	O(1)	O(1)	O(1)	O(1)	O(1)	Hash Table + Linked List
EnumSet	O(1)	O(1)	O(1)	O(1)	O(1)	Bit Vector
TreeSet	O(log n)	O(log n)	O(log n)	O(log n)	O(1)	Redblack tree
CopyOnWriteArraySet	O(n)	O(n)	O(n)	O(1)	O(1)	Array
ConcurrentSkipListSet	O(log n)	O(log n)	O(log n)	O(1)	O(n)	Skip List
Map	Put	Remove	Get	ContainsKey	Next	Data Structure
HashMap	O(1)	O(1)	O(1)	O(1)	O(h / n)	Hash Table
LinkedHashMap	O(1)	O(1)	O(1)	O(1)	O(1)	Hash Table + Linked List
IdentityHashMap	O(1)	O(1)	O(1)	O(1)	O(h / n)	Array
WeakHashMap	O(1)	O(1)	O(1)	O(1)	O(h / n)	Hash Table
EnumMap	O(1)	O(1)	O(1)	O(1)	O(1)	Array
TreeMap	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Redblack tree
ConcurrentHashMap	O(1)	O(1)	O(1)	O(1)	O(h / n)	Hash Tables
ConcurrentSkipListMap	O(log n)	O(log n)	O(log n)	O(log n)	O(1)	Skip List
Queue	Offer	Peak	Poll	Remove	Size	Data Structure
PriorityQueue	O(log n)	O(1)	O(log n)	O(n)	O(1)	Priority Heap
LinkedList	O(1)	O(1)	O(1)	O(1)	O(1)	Array
ArrayDeque	O(1)	O(1)	O(1)	O(n)	O(1)	Linked List
ConcurrentLinkedQueue	O(1)	O(1)	O(1)	O(n)	O(1)	Linked List
ArrayBlockingQueue	O(1)	O(1)	O(1)	O(n)	O(1)	Array
PriorityBlockingQueue	O(log n)	O(1)	O(log n)	O(n)	O(1)	Priority Heap
SynchronousQueue	O(1)	O(1)	O(1)	O(n)	O(1)	None
DelayQueue	O(log n)	O(1)	O(log n)	O(n)	O(1)	Priority Heap
LinkedBlockingQueue	O(1)	O(1)	O(1)	O(n)	O(1)	Linked List



# Benchmarks performances





# Travailler avec les collections

Trier, mélanger, rechercher...

# Les classes abstraites pour faciliter vos implémentations

- On vous fournit des versions classes abstraites
  - AbstractCollection
  - AbstractSet
  - AbstractList
  - AbstractSequentialList
  - AbstractMap
- Aide à l'implémentation de futures classes
  - Réduction de l'effort d'écriture de code

# Les classes utilitaires

- Collections
  - Ensemble de méthodes de classe pour
    - remplissage
    - faire des opérations de tri, ou de "dérangement"
    - des recherches rapides dans des Collections triées
- Arrays
  - Ensemble de méthodes de classe pour
    - remplissage
    - faire des opérations de tri, ou de "dérangement"
    - des recherches rapides dans des tableaux triés
    - transformation de tableau en liste

# Le remplissage, les méthodes static `fill`

- Remplissage par duplication d'une même référence d'objet ou de valeur
  - `Arrays.fill(Object[] a, Object o)`
  - `Arrays.fill(typePrimitif[] a, typePrimitif i)`
  - `Collections.fill(List list, Object obj)` : **attention** remplissage et pas ajout...
- Pas de remplissage pour les Map et les Set

Thinking Java : Ensemble d'interface "Generator" proposées ainsi que d'autres méthodes `fill` pour faire du remplissage aléatoire...

# Les opérations de tri, les méthodes static `sort`

- Tri des tableaux de type primitifs
  - Classe `Arrays`
- Tri des Objets
  - Classe `Arrays` et `Collection`
  - Utilisable sur les collections d'instances de classe implémentant l'interface `Comparable`

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

`e1.compareTo(e2)` retourne  
un `int < 0` si `e1 < e2`  
`0` si `e1 = e2`  
un `int > 0` si `e1 > e2`

# Exemple de code

```
import java.util.*;

public final class Name implements Comparable<Name> {

    private final String firstName, lastName;

    public Name(String firstName, String lastName) {
        if (firstName == null || lastName == null)
            throw new NullPointerException();

        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String firstName() { return firstName; }
    public String lastName() { return lastName; }

    public boolean equals(Object o) {
        if (!(o instanceof Name)) return false;
        Name n = (Name) o;
        return n.firstName.equals(firstName) && n.lastName.equals(lastName);
    }

    public int hashCode() {
        return 31*firstName.hashCode() + lastName.hashCode();
    }

    public String toString() {
        return firstName + " " + lastName;
    }

    public int compareTo(Name n) {
        int lastCmp = lastName.compareTo(n.lastName);
        return (lastCmp != 0 ? lastCmp : firstName.compareTo(n.firstName));
    }
}
```

```
public static void main(String[] args) {
    Name nameArray[] = {
        new Name("John", "Lennon"),
        new Name("Karl", "Marx"),
        new Name("Groucho", "Marx"),
        new Name("Oscar", "Grouch")
    };
    List<Name> names = Arrays.asList(nameArray);
    Collections.sort(names);
    System.out.println(names);
}
```

[Oscar Grouch, John Lennon, Groucho Marx, Karl Marx]

# Tri suivant un ordre quelconque

- Si vous souhaitez trier des éléments issus d'une classe m'implémentant par `Comparable`
- Ou si vous voulez effectuer un tri suivant une logique différente de celle impliquée par `Comparable`, tri dit naturel
- Vous devez utiliser un `Comparator`
- Un `Comparator` est un objet qui encapsule une relation d'ordre

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}  
//retourne un entier négatif, Zéro ou positif selon que o1 est  
//plus petit, égal ou plus grand que o2. Si le type d'un des  
//arguments est impropre il y a une ClassCastException.
```

# Exemple de cas d'utilisation

- Une classe `Employee` implémentant `Comparable` et basée sur l'ordre alphabétique des Noms
- On vous demande de trier les `Employees` sur la base de leur ancienneté, vous allez utiliser un `Comparator<Employee>`

```
public class Employee implements Comparable<Employee>{  
    public Name name() { ... }  
    public int number() { ... }  
    public Date hireDate() { ... }  
    ... }
```



# Comparator pour ordre suivant l'ancienneté

```
import java.util.*;
class EmpSort {
    static final Comparator<Employee> SENIORITY_ORDER =
        new Comparator<Employee>() {
            public int compare(Employee e1, Employee e2){
                return e2.hireDate().compareTo(e1.hireDate());
            }
        };
    // Employee Database
    static final Collection<Employee> employees = ... ;
    public static void main(String[] args) {
        List<Employee>e = new ArrayList<Employee>(employees);
        Collections.sort(e, SENIORITY_ORDER);
        System.out.println(e);
    }
}
```

# Exercice

- La classe String implémente l'interface Comparable et classe les String par ordre Lexicographique (les mots commençant par des majuscules en premier)
- Écrivez un Comparator qui ne tient pas compte de la casse
  - PS : méthode d'instance toLowerCase()

# Les algorithmes de recherche

- `Arrays.binarySearch`
- `Collection.binarySearch` sur les `List`
- Ne s'utilise que sur des structures qui ont été triées

# Extraits d'interview Java Collection...

- What is the typical use of Hashtable?
- I am trying to store an object using a key in a Hashtable. And some other object already exists in that location, then what will happen? The existing object will be overwritten? Or the new object will be stored elsewhere?
- What is the difference between the size and capacity of a Vector?
- Can a vector contain heterogenous objects?
- What is the difference between set and list?
- Which implementation of the List interface provides for the fastest insertion of a new element into the middle of the list?
  - a. Vector
  - b. ArrayList
  - c. LinkedList
  - d. None of the above
- What is difference between array & arraylist?

# Bibliographie

- Le toujours et plus que jamais excellent cours de Jean-Michel Doudoux

<https://www.jmdoudoux.fr/java/dej/chap-collections.htm>

- Le tutorial Oracle Java sur les collections (attention JDK 8)

<https://docs.oracle.com/javase/tutorial/collections>

- Aide mémoire Java, Granet, Vincent, Regourd, Jean-Pierre, Dunod, 2019, 333 p., 978-2-10-079038-8, sur ScholarVox

<http://univ.scholarvox.com.udcpp.idm.oclc.org/catalog/book/docid/88875305?searchterm=Aidem%C3%A9moire%20%20Java>

- Tables de Hachage

- Cours INSA Rouen

<http://asi.insa-rouen.fr/enseignement/siteUV/algo/cours/AlgoDynamique.pdf>

- Université de Montreal

<http://www.iro.umontreal.ca/~hamelsyl/hachage4.pdf>