

## Les Collections

Rémi Forax  
forax@univ-mlv.fr

---

# Opération sur les tableaux

---

- La classe **java.util.Arrays** définit des méthodes statiques de manipulation des tableaux :
  - equals(), hashCode(), toString()
  - deepEquals(), deepHashCode(), deepToString()
  - binarySearch(), sort(), fill()
- Pour la copie de tableau, on utilise :
  - Object.clone(), System.arraycopy(), Arrays.copyOf()

# equals(), hashCode(), toString()

- Les méthodes equals(), hashCode(), toString() ne sont pas redéfinies sur les tableaux
- Arrays.equals(), Arrays.hashCode(), Arrays.toString() marchent pour Object et tous les types primitifs

```
int[] array=new int[]{2,3,5,6};
System.out.println(array);           // [I@10b62c9
System.out.println(Arrays.toString(array)); // [2, 3, 5, 6]
System.out.println(array.hashCode());  // 17523401
System.out.println(Arrays.hashCode(array)); // 986147
int[] array2=new int[]{2,3,5,6};
System.out.println(array2.hashCode()); // 8567361
System.out.println(Arrays.hashCode(array2)); // 986147
System.out.println(array.equals(array2)); // false
System.out.println(Arrays.equals(array,array2)); // true
```

---

# deepEquals, deepHashCode, etc

---

- Algorithmes recursifs sur les tableaux d'objets
- Le calcul est relancé si un tableau contient lui-même un tableau etc.
- `deepToString()` détecte les circularités

```
Object[] array3=new Object[]{2,3,null};  
array3[2]=array3;  
System.out.println(Arrays.deepToString(array3));  
// [2, 3, [...]]
```

# binarySearch, sort, fill

- Dichotomie (le tableau doit être trié)

```
binarySearch(byte[] a, byte key)
...
binarySearch(Object[] a, Object key)
<T> binarySearch(T[] a, T key, Comparator<? super T> c)
```

- Tri

```
sort(byte[] a)
sort(byte[] a, int fromIndex, int toIndex)
...
<T> sort(T[] a, Comparator<? super T> c)
<T> sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)
```

- Remplissage

```
fill(boolean[] a, boolean val)
fill(boolean[] a, int fromIndex, int toIndex, boolean val)
...
fill(Object[] a, Object val)
fill(Object[] a, int fromIndex, int toIndex, Object val)
```

---

# Ordre naturel

---

- Une classe peut spécifier un ordre naturel en implantant l'interface Comparable<T>

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

- T doit être la classe spécifiant l'ordre
- Valeur de retour de **compareTo(T t)** :
  - <0 si this est **inférieur** à t
  - ==0 si this est **égal** à t
  - >0 si this est **supérieur** à t

# compareTo et equals

- L'implantation de **compareTo** doit être compatible avec celle d'**equals** !!
- Si `o1.equals(o2)==true` alors  
    `o1.compareTo(o2)==0` (et vice versa)

```
public MyPoint implements Comparable<MyPoint> {  
    public MyPoint(int x,int y) {  
        this.x=x;  
        this.y=y;  
    }  
    public int compareTo(MyPoint p) {  
        int dx=x-p.x;  
        if (dx!=0)  
            return dx;  
        return y-p.y;  
    }  
    private final int x,y;  
}
```

**Attention BUG !!!**

# Example

- Il faut donc redéfinir aussi **equals**

```
public MyPoint implements Comparable<MyPoint> {  
    public MyPoint(int x,int y) {  
        this.x=x;  
        this.y=y;  
    }  
    @Override public boolean equals(Object o) {  
        if (!(o instanceof MyPoint))  
            return false;  
        MyPoint p=(MyPoint)o;  
        return x==p.x && y==p.y;  
    }  
    public int compareTo(MyPoint p) {  
        int dx=x-p.x;  
        if (dx!=0)  
            return dx;  
        return y-p.y;  
    }  
    private final int x,y;  
}
```



---

# A quoi cela sert ?

---

- Par exemple, `java.util.Arrays.sort()` demande à ce que le tableau contienne des éléments mutuellement comparables

```
public MyPoint implements Comparable<MyPoint> {  
    public String toString() {  
        return "("+x+', '+y+')';  
    }  
    public static void main(String[] args) {  
        MyPoint[] points=new MyPoint[] {  
            new MyPoint(2,3), new MyPoint(3,3),  
            new MyPoint(3,2), new MyPoint(1,9)  
        };  
        Arrays.sort(points);  
        System.out.println(Arrays.toString(points));  
        // affiche (1,1) (1,9) (3,2) (3,3)  
    }  
}
```

---

# Comparaison externe

---

- L'interface `java.util.Comparator` permet de spécifier un ordre externe

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

- Un ordre externe est un ordre valable juste à un moment donné  
(rien de naturel et d'évident)
- La valeur de retour de **compare** suit les mêmes règles que **compareTo**

---

# Comparator

---

- L'implantation de **compare** doit être compatible avec celle d'**equals** !!
- Si `o1.equals(o2)==true` alors  
`compare(o1,o2)==0` (et vice versa)

```
Arrays.sort(points,new Comparator<MyPoint>() {  
    public int compare(MyPoint p1,MyPoint p2) {  
        int dx=x-p.x;  
        int dy=y-p.y;  
        return dx*dx+dy*dy;  
    }  
});  
System.out.println(Arrays.toString(points));  
// affiche (1,1) (3,2) (3,3) (1,9)
```

---

# Comparator inverse

---

- Il existe deux méthodes **static** dans la classe `java.util.Collections` qui renvoie un comparator correspondant à :
- L'inverse de l'ordre naturel

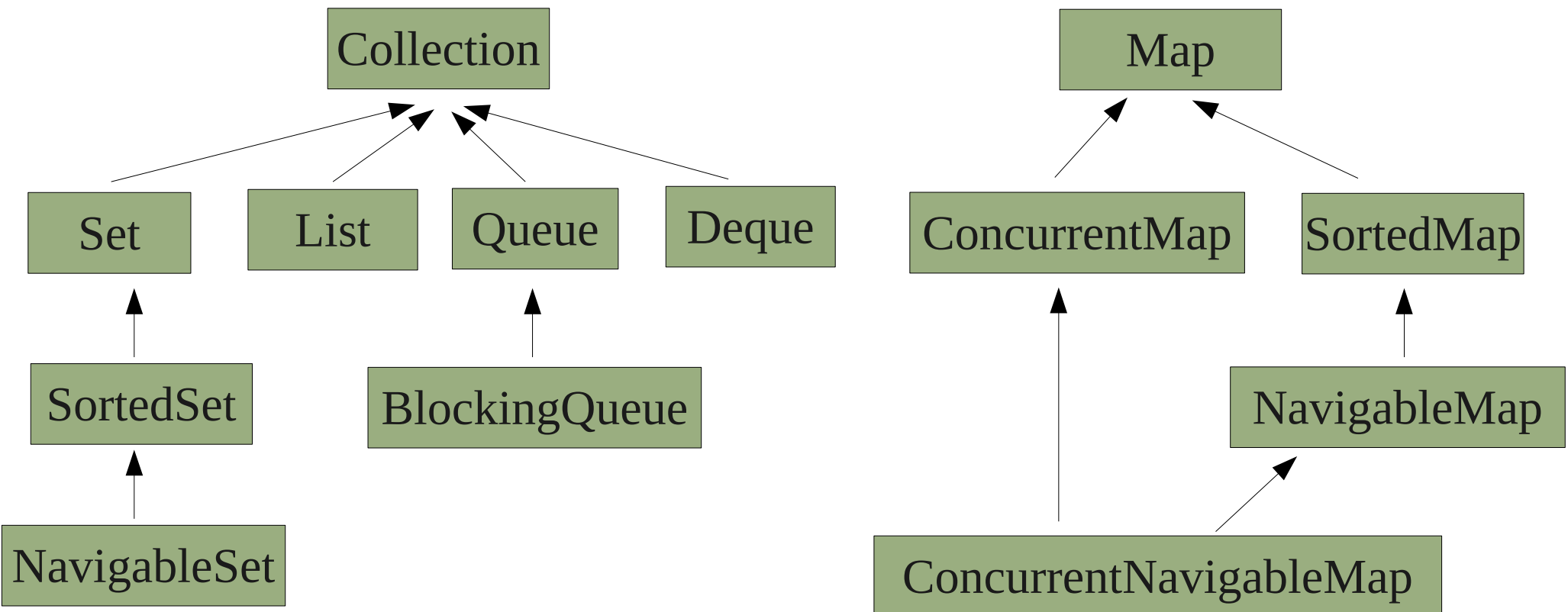
```
<T> Comparator<T> Collections.reverseOrder();
```

- L'inverse d'un ordre externe sur **T**

```
<T> Comparator<T> Collections.reverseOrder(Comparator<T> c);
```

# L'API des collections

- 2 paquets : `java.util`, `java.util.concurrent`
- 2 hiérarchies d'interfaces : `Collection`, `Map`



---

# Design

---

- Séparation Interface/implantation
- Plusieurs implantations pour une interface permet d'obtenir en fonction de l'algorithme que l'on veut écrire la meilleure complexité
- Deux classes contenant des algorithmes communs (méthode statique dans `java.util.Arrays` et `java.util.Collections`)

---

# Interfaces des collections

---

- Définition abstraite des collections :
  - **Collection** ensemble de données
  - **Set** ensemble de données sans doublon
  - **SortedSet** ensemble ordonné et sans doublon
  - **NavigableSet** ensemble ordonné, sans doublon avec précédent suivant
  - **List** liste indexée **ou** séquentielle
  - **Queue** file (FIFO) de données
  - **BlockingQueue** file bloquante de données
  - **Deque** double-ended queue

---

# Interfaces des maps

---

- Permet d'associer un objet (la clé) à un autre :
  - **Map** association sans relation d'ordre
  - **SortedMap** association avec clés triées
  - **NavigableMap** association avec clés triées avec suivant/précédent
  - **ConcurrentMap** association à accès concurrent



---

# Collection<E> et types paramétrés

---

- Toutes les collections sont des types paramétrés par le type des éléments (**E**) qu'elles contiennent
- Les collections sont des conteneurs homogènes
- Si l'on veut stocker des éléments de types différents, on utilise le super-type commun, voir Object

---

# Propriété des Collections

---

- Sauf exceptions :
  - Toutes les collections acceptent **null** comme un élément valide (mais ce n'est pas une bonne idée !)
  - Toutes les collections testent si un objet existe ou non par rapport à la méthode **equals()** de l'objet
  - Toutes les collections ne permettent pas les accès concurrents  
(par des threads différentes)

---

# Opérations optionnelles

---

- Certaines méthodes des collections sont des opérations optionnelles (signalées par un \*)
- Celles-ci peuvent ne pas être implantées par exemple pour définir des collections immutables (*read-only*)
- Les opérations optionnelles non implantées lèvent l'exception **UnsupportedOperationException**

---

# L'interface Collection

---

- Interface de base des ensembles, listes et files
- Opérations sur l'ensemble de données
  - **isEmpty/size/clear\***
  - **add\*/remove\*/contains**
  - Itérateur : **Iterator (hasNext,next,remove\*)**
  - Opération Groupées (*bulk*) :  
**addAll\*/removeAll\*/retainAll\*/containsAll**
  - Création d'un tableau : **toArray**

---

# Taille et effacement

---

- L'interface définit les méthodes :
  - **int size()** indiquant la taille d'une collection
  - **boolean isEmpty()** indiquant si une collection est vide.  
La méthode souvent déclarée dans une classe abstraite comme `size()==0`
  - **void clear\*()** permettant d'effacer les données de la collection

---

# Modification et test de contenu

---

- Les modifications et tests sont effectuées par :
  - **boolean add\*(E e)** ajoute un élément à la collection, true si la collection est modifiée
  - **boolean remove\*(Object o)** retire un objet, true si la collection est modifiée
  - **boolean contains(Object)** renvoie si la collection contient l'objet
- **remove()** et **contains()** prennent en paramètre des **Object** et non des **E** par compatibilité

# Éléments et equals

- Toutes les collections sauf exception testent si un objet existe en utilisant la méthode equals() de l'objet

```
public class MyPoint {
    public MyPoint(int x,int y) {
        this.x=x;
        this.y=y;
    }
    private final int x,y;
    public static void main(String[] args) {
        Collection<MyPoint> c=new ArrayList<MyPoint>();
        c.add(new MyPoint(1,2)); // true
        c.contains(new MyPoint(1,2)); // false
    }
}
```

---

# Iterator

---

- Pour parcourir une collection, on utilise un objet permettant de passer en revue les différents éléments de la collection
- `java.util.Iterator<E>` définie :
  - **boolean hasNext()** qui renvoie vrai s'il y a un suivant
  - **E next()** qui renvoie l'élément courant et décale sur l'élément suivant
  - **void remove()** qui retire un élément précédemment envoyé par `next()`



---

# next() et NoSuchElementException

---

- L'opération **next()** est sécurisée en lève une exception runtime **NoSuchElementException** dans le cas où on dépasse la fin de la collection (c-a-d si **hasNext()** renvoie false)

```
public static void main(String[] args) {  
    Collection<Integer> c=new ArrayList<Integer>();  
    c.add(3);  
    Iterator<Integer> it=c.iterator();  
    it.next();  
    it.next(); // NoSuchElementException  
}
```

---

# Exemple d'iterateur

---

- Conceptuellement un iterateur s'utilise comme un pointeur que l'on décaler sur la collection

```
public static void main(String[] args) {  
    Collection<Integer> c=new ArrayList<Integer>();  
    c.add(3);  
    c.add(2);  
    c.add(4);  
    Iterator<Integer> it=c.iterator();  
    for(;it.hasNext();) {  
        System.out.println(it.next()*2);  
    } // affiche 6, 4, 8  
}
```

---

# Intérêt des itérateurs

---

- Parcours d'une collection est :
  - Pas toujours possible d'effectuer un parcours avec un index (Set, Queue, Deque)
  - Problème de complexité (List séquentiel)
- Les itérateurs offre un parcours garantie en  $O(n)$

---

# Modification et parcours

---

- Les itérateurs des collections sont *fail-fast* : si lors d'un parcours avec un itérateur, il y a une modification de la collection, l'itérateur lèvera alors une exception **ConcurrentModificationException**
- Il n'y a pas besoin de plusieurs threads pour lever cette exception
- Le fait que les itérateurs soient *fail-fast* ne veut pas dire que les collections peuvent être modifiés par des threads différentes

# Exemple de *fail-fast*

- Parcours + suppression par l'interface collection = **ConcurrentModificationException**

```
public class ConcurrentExceptionExample {
    public static void main(String[] args) {
        Set<String> set=new HashSet<String>();
        Collections.addAll(set,args);
        Iterator<String> it=set.iterator();
        for(;it.hasNext();) {
            String s=it.next();
            if ("toto".equals(s))
                set.remove(s);        // va causer une erreur à la prochaine itération
        }
    }
    // java ConcurrentExceptionExample hello world
    //Exception in thread "main" java.util.ConcurrentModificationException
    //    at java.util.HashMap$HashIterator.nextEntry(HashMap.java:787)
    //    at java.util.HashMap$KeyIterator.next(HashMap.java:823)
    //    at ConcurrentExceptionExample.main(ConcurrentExceptionExample.java:16)
}
```

---

# Même exemple qui marche

---

- Parcours + suppression mais par l'interface de l'iterateur (remove sur l'iterateur)

```
public class ConcurrentExceptionExample {  
    public static void main(String[] args) {  
        Set<String> set=new HashSet<String>();  
        Collections.addAll(set,args);  
        Iterator<String> it=set.iterator();  
        for(;it.hasNext();) {  
            String s=it.next();  
            if ("toto".equals(s))  
                it.remove();           // ok  
        }  
    }  
}
```

---

# Itérateur et concurrence

---

- Il y a trois sortes d'itérateurs
  - Les itérateurs normaux qui ne supportent pas la concurrence, il faut alors définir ceux-ci dans une section critique (cf cours concurrence)
  - Les itérateurs *weakly* consistant : les ajouts/modifications depuis la création ne sont pas garanties d'être visible
  - Les *snapshots* itérateurs : effectue une copie de la collection lors de la création de l'itérateur (l'itérateur est *read-only*, le **remove** n'est pas supporté)

---

# Iterable & syntaxe foreach

---

- L'interface Collection hérite de l'interface Iterable

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

- Permet d'utiliser des collections avec for(:)

```
public static void main(String[] args) {  
    Iterable<Integer> iterable=new ArrayList<Integer>();  
    ...  
    for(int i:iterable)  
        System.out.println(i*2);  
} // affiche 6, 4, 8  
}
```



# Exemple d'Iterable

- Exemple d'Iterable

```
public static Iterable<Integer> range(final int begin, final int end) {
    return new Iterable<Integer>() {
        public Iterator<Integer> iterator() {
            return new Iterator<Integer>() {
                private int index=begin;
                public void remove() { // optionnel
                    throw new UnsupportedOperationException();
                }
                public boolean hasNext() {
                    return index<=end;
                }
                public Integer next() {
                    if (!hasNext())
                        throw NoSuchElementException("index>=end");
                    return index++;
                }
            };
        }
    };
}

public static void main(String[] args) {
    for(int i:range(2,5))
        System.out.printf("%d ",i);
    // affiche 2 3 4 5
}
```

---

# Enumeration

---

- Ancienne version des itérateurs

```
interface Enumeration<T> {  
    boolean hasMoreElements()  
    T nextElement()  
}
```

- Utilisé uniquement par compatibilité avec les anciennes APIs
- Méthodes pratiques
  - <T> Enumeration<T>  
Collections.enumeration(Collection<T> c)
  - <T> List<T> Collections.list(Enumeration<T> e)

---

# Ecrire un itérateur filtrant

---

- On ne veut sélectionner que des objets correspondant à un critère

```
interface Filter<E> {  
    boolean accept(E element);  
}
```

- On veut écrire la méthode suivante :

```
public static <T> Iterator<T> filterIterator(  
    Collection<T> coll, Filter<? super T> filter) {  
    ...  
}
```

- L'astuce consiste à écrire hasNext() de façon à avoir un coup d'avance

# hasNext() fait le boulot

- Si la collection ne contient pas null

```
public static <T> Iterator<T> filterIterator(
    Collection<T> coll, final Filter<? super T> filter) {

    final Iterator<T> it=coll.iterator();
    return new Iterator<T>() {
        public boolean hasNext() {
            if (element!=null)
                return true;
            while(it.hasNext()) {
                T e=it.next();
                if (filter.accept(e)) {
                    element=e;
                    return true;
                }
            }
            return false;
        }

        public T next() {
            if (!hasNext())
                throw new NoSuchElementException();
            T e=element;
            element=null;
            return e;
        }
        ...
        private T element;
    };
}
```

---

# Opérations groupées

---

- Les collections permettent les opérations groupées suivantes :
  - **addAll\***(Collection<? extends E>), **removeAll\***(Collection<?>) permettent d'ajouter/enlever tous les éléments d'une collection
  - **containsAll**(Collection<?>) vrai si tous les éléments de la collection argument sont contenus
  - **retainAll\***(Collection<?>) retient dans la collection courante seulement les éléments présent dans la collection en argument

---

# Collection & Tableau

---

- Les méthodes **toArray()** permettent de créer un tableau contenant les mêmes éléments que la collection
  - Avantages :
    - tableau indépendant de la collection
    - tableaux plus rapides à parcourir que les collections
  - Inconvénient :
    - duplique les données (allocation inutile)

# toArray()

- `Object[] toArray()` renvoie un tableau d'objet contenant les éléments
- `<T> T[] toArray(T[] array)` remplit le tableau avec les éléments, si le tableau n'est pas assez grand, un nouveau tableau de T de la bonne taille est créé par reflexion

```
public static void main(String[] args) {  
    List<String> list=Arrays.asList(args);  
    Collection<?> c=list;  
    String[] array=(String[])c.toArray(); // ClassCastException  
    Date[] array2=c.toArray(new Date[c.size()]); // ArrayStoreException  
    String[] array3=c.toArray(new String[c.size()]); // ok  
}
```

---

# Note de design

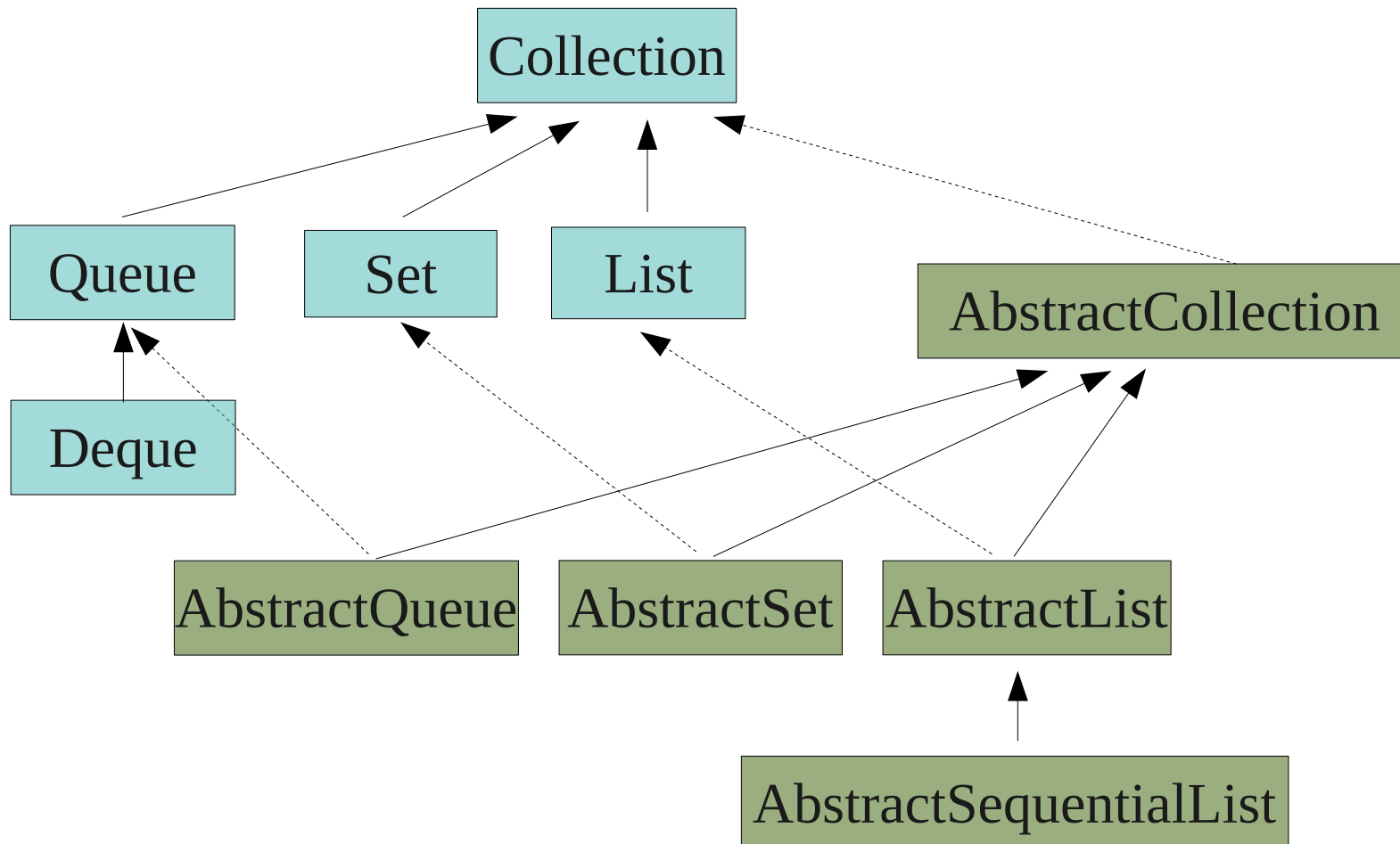
---

- En prenant un peu de recul, la différence entre un itérateur et une collection immutable est la présence de la méthode `size()`.
- Lorsque l'on architecture une classe :
  - Au lieu de retourner un itérateur
  - Il vaut mieux retourner une collection immutable agissant comme une vue.
- Cela évite les recopies inutiles !!



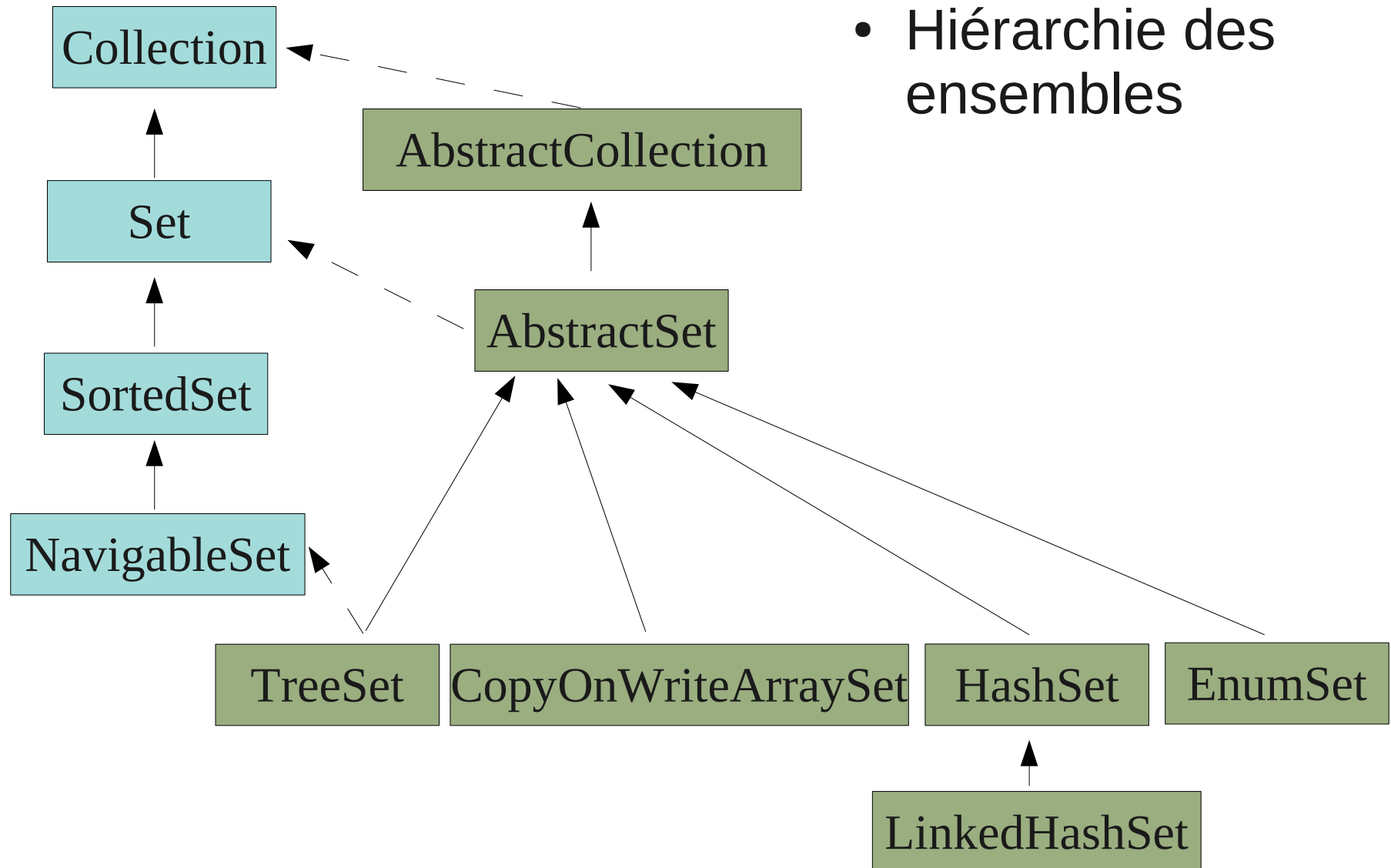
# Implantation des collections

- Chaque collection possède une classe abstraite



# Set

- Hiérarchie des ensembles



---

# L'interface Set et implantations

---

- L'interface **Set** définit un ensemble ne possédant pas de doublon. Pas de méthode supplémentaire par rapport à Collection
  - Les différentes implantations :
    - HashSet
    - LinkedHashSet
    - CopyOnWriteArraySet
    - EnumSet (2 implantations)
    - TreeSet
    - ConcurrentSkipListSet

---

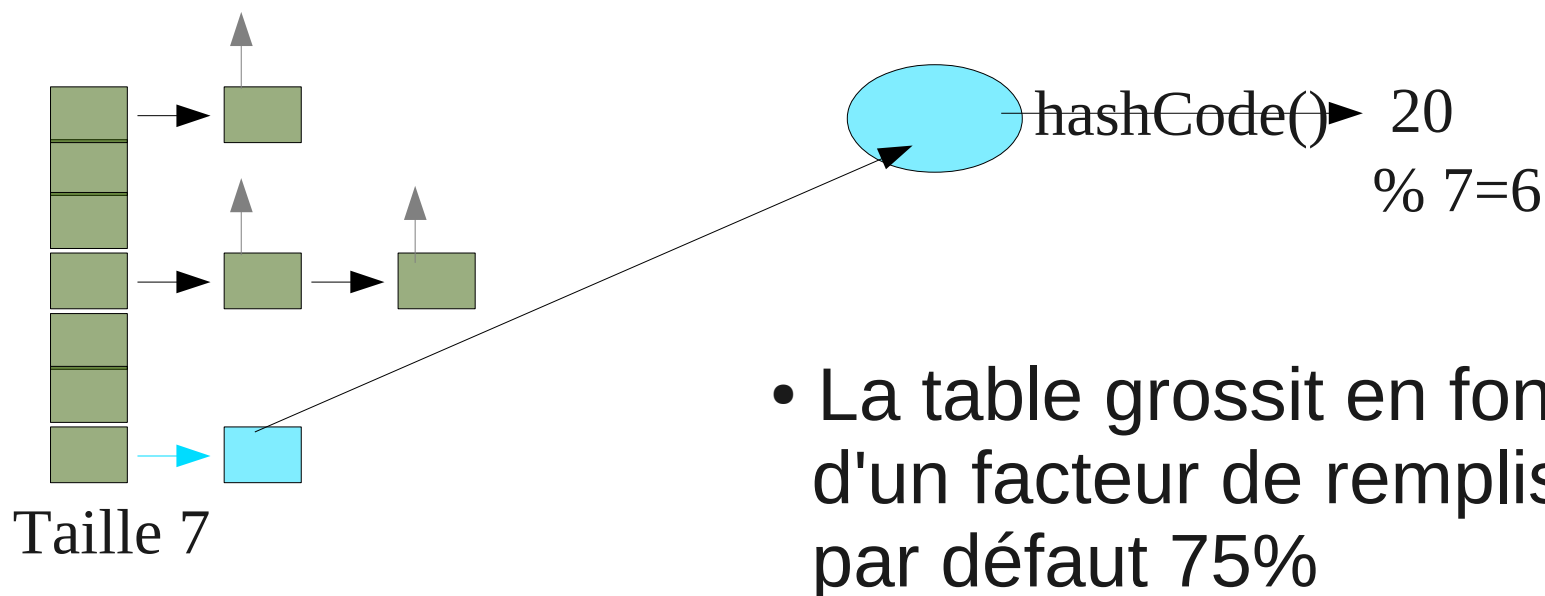
# AbstractSet

---

- Hérite de **AbstractCollection**
- A part l'implantation de **removeAll()** qui est optimisée, pas de différence avec une **AbstractCollection** si ce n'est le type
- Sert de classe de base pour toutes les implantations des ensembles

# HashSet

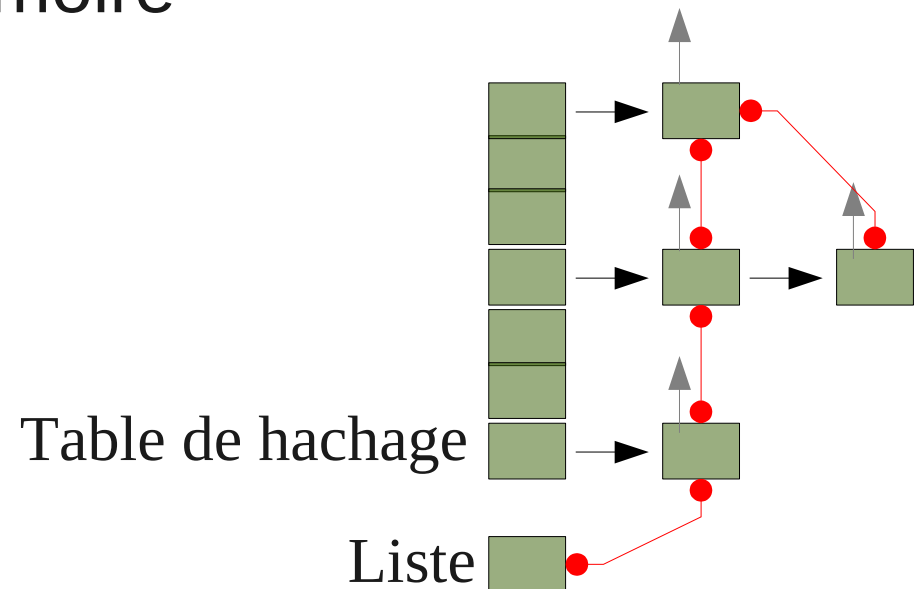
- Implantation sous forme de table de hachage
- Utilise les méthodes **hashCode()** et **equals()** d'un élément



- La table grossit en fonction d'un facteur de remplissage, par défaut 75%

# LinkedHashSet

- Table de hachage + liste doublement chaînée
- Parcours + efficace que **HashSet** mais consomme + de mémoire
- Ordre dans la liste par insertion
- L'iterateur utilise la liste



---

# CopyOnWriteArraySet

---

- Tableau dynamique se dupliquant à chaque modification
- Permet les accès/modification concurrents
- l'itérateur est un *snapshot iterator*, *thread-safe* et *read-only*
- A utiliser si :
  - peu d'éléments
  - Beaucoup d'accès par rapport aux modifications

---

# EnumSet

---

- Stocke uniquement les valeurs d'une énumération, 2 implantations différentes :
  - Regular (utilise un long)
  - Jumbo (utilise un tableau de long)
- Création par des méthodes factories :
  - **allOf**(Class<E>), **noneOf**(Class<E>)
  - **range**(E from, E to)
  - **of**(E), **of**(E,E), ..., **of**(E,E...)



---

# SortedSet

---

- Ensemble ordonné par un comparateur
  - Définit les méthodes :
    - `Comparator<? super E> comparator()`
      - Renvoie le comparateur utiliser pour définir l'ordre
    - `E first(), last()`
      - Renvoie le premier/dernier élément
    - `SortedSet<E> headSet(E),subSet(E,E),tailSet(E)`
      - Sous-ensemble restreint de l'ensemble courant
  - Très peu utiliser car **SortedSet** ne définit pas assez de méthodes et ne possédait qu'une implantation

---

# headSet,subSet,tailSet

---

- Sous-ensemble qui sont des vues de l'ensemble de départ
  - **SortedSet<E> headSet(E e)**  
ensemble entre premier élément inclu et e exclu
  - **SortedSet<E>subSet(E e1,E e1),**  
ensemble entre le1 inclu et e2 exclu
  - **SortedSet<E>tailSet(E e)**  
ensemble entre e inclu et le dernier inclu
- Pas facile de créer un headSet() avec l'élément e inclu

```
String key=...
SortedSet<String> head = s.headSet(key+"\0");
```

---

# NavigableSet

---

- Set ordonné par un comparateur permettant de navigué entre les éléments (étend SortedSet)
  - **E lower(E e), floor(E), higher(E e), ceiling(E e)**
    - Resp. renvoie l'élément  $<$ ,  $<=$ ,  $>$  et  $>=$  à  $e$ .
  - **E pollFirst(), pollLast()**
    - Renvoie en le supprimant le premier/dernier élément
- Parcourir ou utiliser l'ensemble en ordre descendant
  - NavigableSet<E> **descendingSet()**
  - Iterator<E> **descendingIterator()**

---

# NavigableSet (suite)

---

- Sous-ensemble avec possibilité d'inclure ou non le ou les éléments
  - NavigableSet<E> **headSet**(E toElement, boolean inclusive)
  - NavigableSet<E> **subSet**(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)
  - NavigableSet<E> **tailSet**(E fromElement, boolean inclusive)

---

# TreeSet

---

- Implantation de NavigableSet à base d'un arbre rouge/noir qui classe les éléments en utilisant :
  - Un comparateur
  - Ou l'ordre naturel (Comparable.compareTo())
- L'implantation garantit  **$O(\ln n)$**  pour l'insertion, la recherche ou la suppression

---

# ConcurrentSkipListSet

---

- Implantation de NavigableSet à base d'un arbre fils gauche/frère droit à deux niveaux
  - Un comparateur
  - Ou l'ordre naturel (Comparable.compareTo())
- L'implantation garantit  **$O(\ln n)$**  pour l'insertion, la recherche ou la suppression
- **size()** n'est pas une opération en temps constant
- L'iterateur est *weakly* consistant

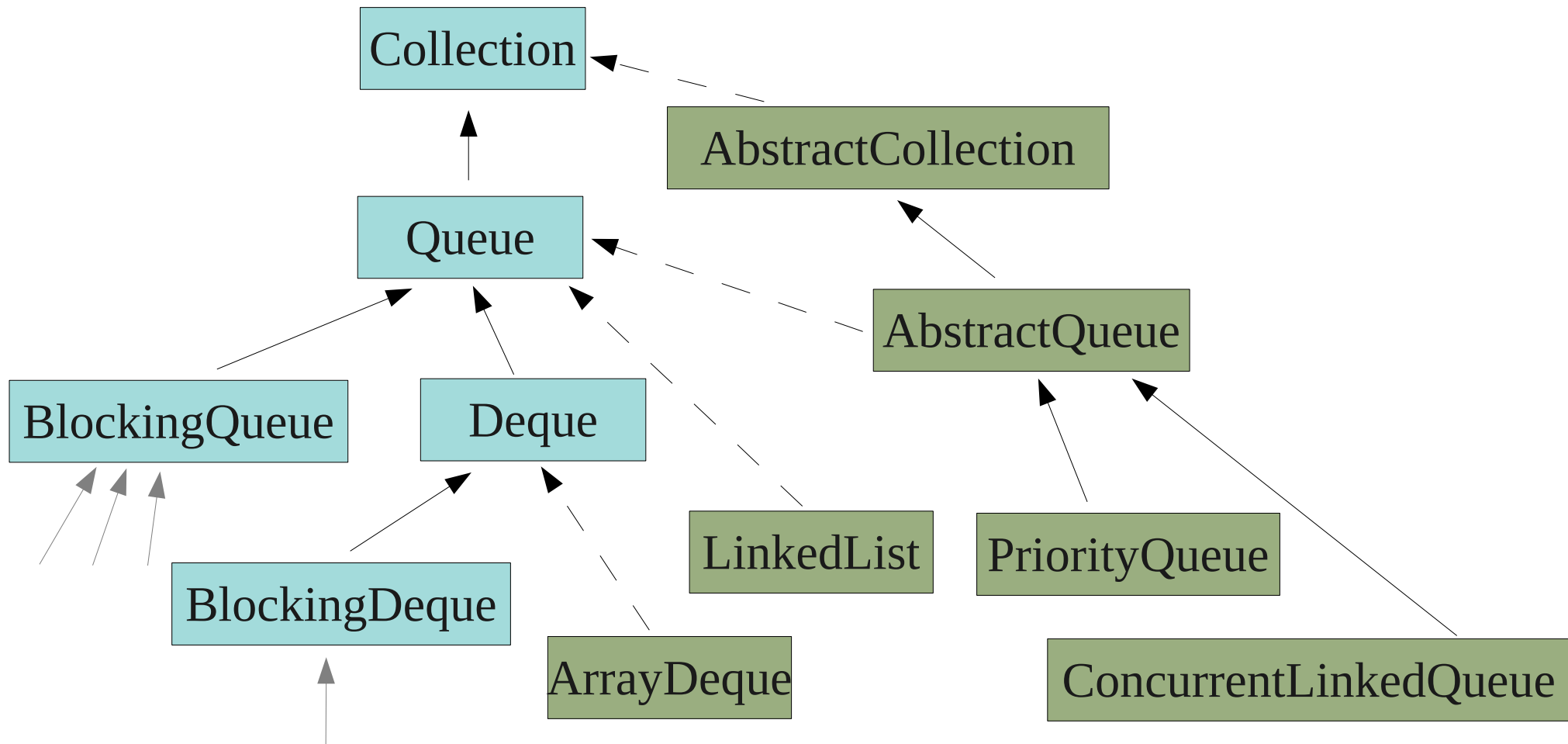
# Complexités

- Complexité des opérations en fonction de l'implantation

	size()	clear()	add	remove	contains()	Parcours
HashSet	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	7
LinkedHashSet	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	4
COWArraySet	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	3
EnumSet(regular)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	1
EnumSet(jumbo)	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	2
TreeSet	$O(1)$	$O(1)$	$O(\ln n)$	$O(\ln n)$	$O(\ln n)$	5
ConcSkipListSet	$O(n)$	$O(1)$	$O(\ln n)$	$O(\ln n)$	$O(\ln n)$	6

- Les parcours (pour beaucoup d'éléments) sont classées par ordre de vitesse (1 le + rapide)

# Queue & Deque





---

# Queue et Deque

---

- Il existe deux interfaces différentes :
  - **Queue** (1.5), interface des files d'objet (FIFO)
    - Insertion en queue
    - Suppression en tête
  - **Deque** (1.6), interface des files/piles d'objets (FIFO/LIFO) accessibles par la tête ou la queue
    - Insertion en tête ou en queue
    - Suppression en tête ou en queue

**Deque** hérite de **Queue**

---

# API de Queue

---

- Interface décrivant des Files FIFO (First in, first out)
  - E **peek()**, E **element()**  
demande un élément sans l'enlever
  - boolean **offer(E)**, boolean **add(E)**  
ajoute un élément en queue
  - E **poll()**, E **remove()**  
retire un élément en tête
- Contrairement aux Collections, **null** n'est pas un élément permis

---

## 2 sémantiques

---

- Runtime exception si erreur
  - **element()**, **remove()** lèvent une **NoSuchElementException** si la file est vide
  - **add()** lève **IllegalStateException** si la file est pleine
- Valeur de retour si erreur
  - **peek()**, **poll()** retourne **null** si la file est vide
  - **offer()** return **faux** si la file est pleine

---

# API de Deque

---

- Interface décrivant des Piles et Files
  - E **peekFirst/Last()**, E **getFirst/Last()**  
demande un élément sans l'enlever
  - boolean **offerFirst/Last(E)**, boolean **addFirst/Last(E)**  
ajoute un élément en queue
  - E **pollFirst/Last()**, E **removeFirst/last()**  
retire un élément en tête

---

# Deque et parcours

---

- A part l'iterateur (iterator()) classique, Deque possède un itérateur d'escendant
  - Iterator<E> **descendingIterator()**
- Recherche et suppression d'élément
  - boolean **removeFirstOccurrence**(Object o)
  - boolean **removeLastOccurrence**(Object o)

---

# Méthodes de Queue vers Deque

---

- Equivalence entre les méthodes de **Queue** et celle de **Deque**
  - Demander un élément (sans le retirer)  
`Queue.element()/peek()` -> `Deque.getFirst()/peekFirst()`
  - Ajouter un élément en queue  
`Queue.add(e)/offer(e)` ->  
`Deque.addLast(e)/offerLast(e)`
  - Retirer un élément en tête  
`Queue.remove()/poll()` -> `Deque.removeFirst()/pollFirst()`

---

# BlockingQueue/BlockingDeque

---

- Ce sont des queues dont les opérations peuvent bloquer jusqu'à ce que la ressource soit disponible
  - Implantations de BlockingQueue:
    - LinkedBlockingQueue
    - ArrayBlockingQueue
    - PriorityBlockingQueue
    - SynchronousQueue
    - DelayQueue
  - Implantation de BlockingDeque
    - LinkedBlockingDeque

---

# AbstractQueue

---

- Classe de base des implantations de l'interface **Queue**
- Il faut définir les méthodes : **peek**, **offer**, **poll**, **size** et **iterator** (avec **remove**)
- Les autres méthodes de l'interface **Queue** **element**, **add**, **remove** sont définies à partir des méthodes précédentes



# AbstractQueue - Example

```
public class SingletonQueue<E> extends AbstractQueue<E> {
    public boolean offer(E e) {
        element=e;
        return true;
    }
    public E poll() {
        E e=element;
        element=null;
        return e;
    }
    public E peek() {
        return element;
    }
    public int size() {
        return (element==null)?0:1;
    }
    ...
    E element;

    public static void main(String[] args) {
        Queue<String> queue=new
        SingletonQueue<String>();
        queue.offer("toto");
        queue.peek();    // toto
        queue.poll();    // toto
        queue.remove();  // NoSuchElementException
    }
}
```

---

# AbstractQueue - Exemple

---

- Et l'itérateur :

```
public Iterator<E> iterator() {  
    return new Iterator<E>() {  
        public boolean hasNext() {  
            return element!=null;  
        }  
        public E next() {  
            E e=element;  
            if (e==null)  
                throw new NoSuchElementException();  
            return e;  
        }  
        public void remove() {  
            if (hasNext())  
                SingletonQueue.this.remove();  
            else  
                throw new NoSuchElementException();  
        }  
    };  
}
```

---

# Implantations

---

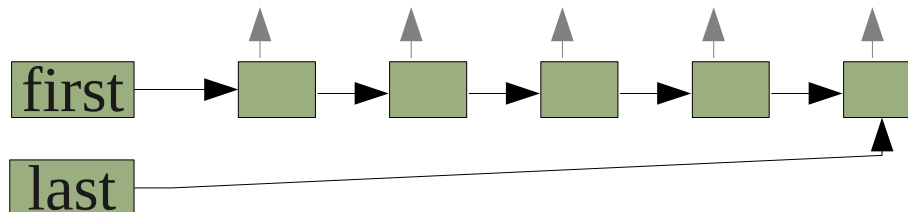
- Queue et Deque possède quatres implantations différentes :
  - PriorityQueue (Queue)
  - ConcurrentLinkedQueue (Queue)
  - LinkedList (Queue & Deque)
  - ArrayDeque (Deque)

---

# ConcurrentLinkedQueue

---

- Queue permettant les accès concurrents (mais n'utilise pas de synchronized !!)
- Liste simplement chaînée maintenant la tête et la queue
- Iterator weakly consistant, `iterator.remove()` change l'élément d'un noeud à **null**, il est ensuite ejecté par **poll()** ou **offer()**



---

# PriorityQueue

---

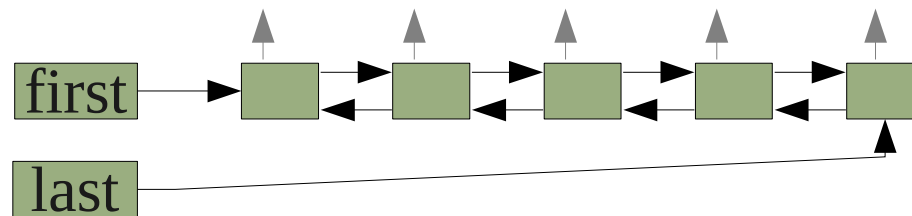
- Arbre binaire de recherche codé dans un tableau.
- L'ordre sur les éléments est donné par :
  - un comparateur
  - ou l'ordre naturel (`Comparable.compareTo()`)
- L'implantation garantit  **$O(\ln n)$**  pour **`poll()`** et **`offer()`**
- Attention l'iterateur traverse la queue dans n'importe quel ordre !!

---

# LinkedList

---

- LinkedList implante les interfaces Queue, Deque et List
- C'est une liste **double chaînée** qui maintient la tête et la queue de la liste

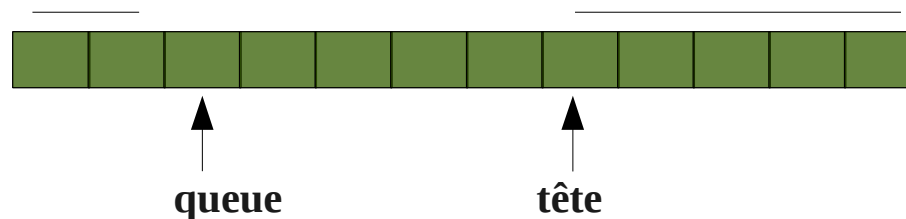


---

# ArrayDeque

---

- ArrayDeque implante les interfaces Deque.
- Buffer circulaire qui se redimensionne automatiquement



- Cette classe a été conçu pour remplacer
  - `java.util.Stack`
  - `java.util.LinkedList` vue comme une queue

# Complexités

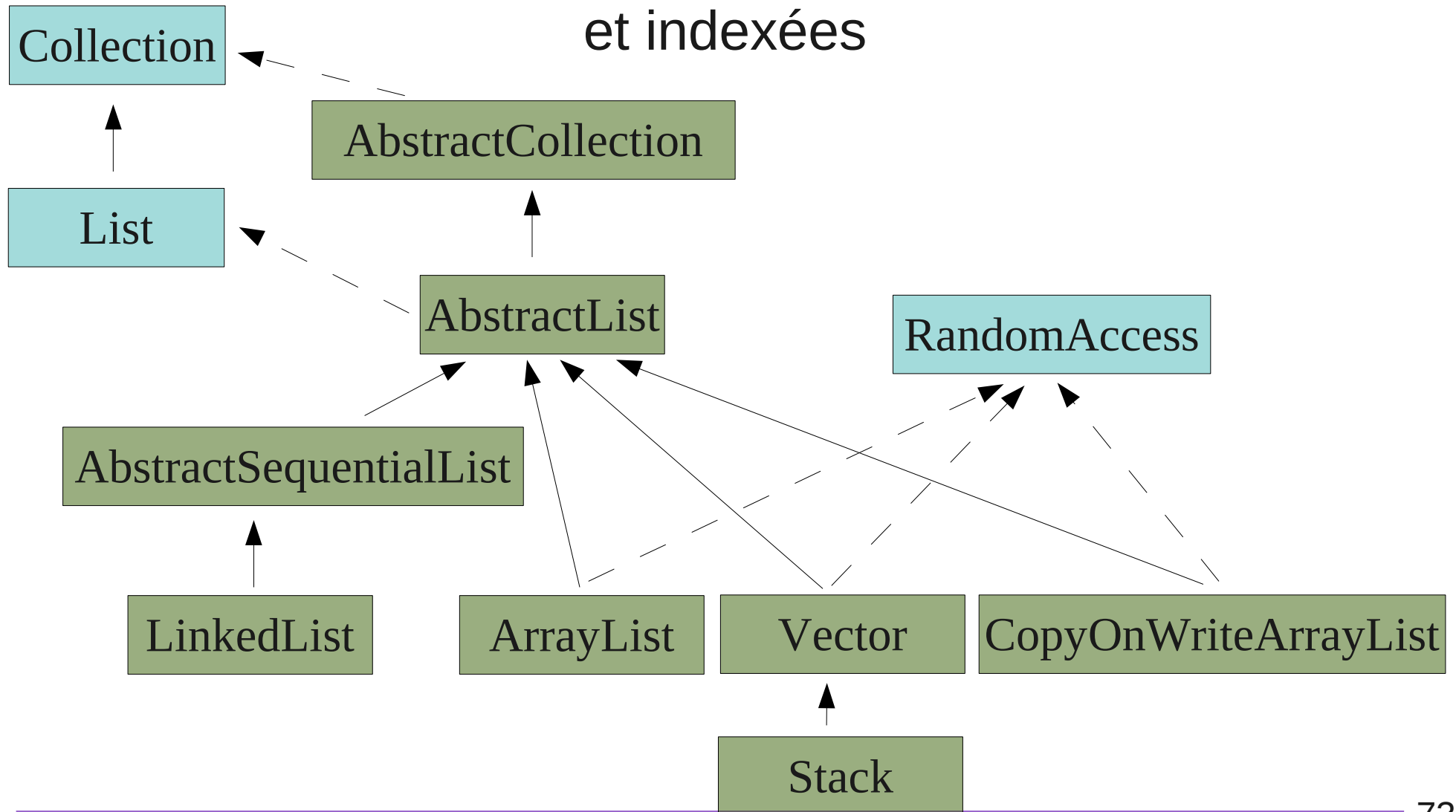
- Complexités des implantations de queue/deque

	size()	peek/element	poll/remove	offer/add
CrtLinkedQueue	<b>O(n)</b>	<b>O(1)</b>	<b>O(1)</b>	<b>O(1)</b>
PriorityQueue	<b>O(1)</b>	<b>O(1)</b>	<b>O(1)</b>	<b>O(ln n)</b>
LinkedList	<b>O(1)</b>	<b>O(1)</b>	<b>O(1)</b>	<b>O(1)</b>
ArrayDeque	<b>O(1)</b>	<b>O(1)</b>	<b>O(1)</b>	<b>O(1)</b>



# List

- Gère les listes séquentielles et indexées



---

# L'interface List

---

- En plus des méthodes de Collection, définie des méthodes travaillant sur les index :
  - E **get**(int), boolean **set**(int,E)
  - boolean **add**(int,E), E **remove**(int), boolean **addAll**(index,Collection)
  - int **indexOf**, **lastIndexOf**
  - ListIterator<E> **listIterator**()
  - List<E> **subList**()

---

# Accès indexés

---

- Une liste est représenté par une suite d'éléments numéroté de 0 à `size()-1`
  - Les méthodes sont :
    - E **get**(int index) renvoie l'élément situé à l'index
    - boolean **set**(int index, E e) remplace l'élément à l'index par e, true si la liste a changé
    - boolean **add**(int index,E e) insère un élément, true si la liste a changé
    - boolean **remove**(int index) supprime un élément, true si la liste à changée

---

# List indexée et temps constant

---

- Toutes les implantations de List supporte la recherche indexée d'élément, mais toute ne le font pas en temps constant
  - Liste à acces indexé en temps constant  
**ArrayList, COWArrayList**
  - Liste à accès indexé en temps linéaire  
**LinkedList**
- Les listes à accès indexé en temps linéaire hérite de **AbstractSequentialList** et pas d'**AbstractList**.

---

# RandomAccess

---

- **RandomAccess** permet d'indiquer que les accès get/set se font en temps constant

```
public interface Callable<V> {  
    void call(V v);  
}  
public static <T> void map(List<T> list, Callable<? super T> c) {  
    if (list instanceof RandomAccess)  
        for(int i=0; i<list.size(); i++) // par index  
            c.call(list.get(i));  
    else  
        for(T t: list) // avec un itérateur  
            c.call(c);  
}
```

- L'accès par index est plus rapide que par itérateur dans ce cas

---

# Recherche par index

---

- Les méthodes **indexOf()** et **lastIndexOf()** recherchent un élément en utilisant `equals()`
- **indexOf(Object)** renvoie l'index du premier objet trouvé, -1 sinon
- **lastIndexOf(Object)** renvoie l'index du dernier objet trouvé, -1 sinon

---

# L'itérateur de liste (ListIterator)

---

- Les listes possèdent un itérateur spécifique héritant de **Iterator**
- **ListIterator** permet, en plus
  - de parcourir une liste dans les deux sens (gauche<->droite) **hasPrevious()**, **previous()**
  - de remplacer/ajouter des éléments dans la liste **set(item)**, **add(item)**
  - d'obtenir les index correspondant à la position courante **nextIndex()**, **previousIndex()**

# ListIterator (2)

- Exemple de parcourt inverse

```
public static void descendingPrint(List<?> list) {  
    ListIterator it=list.listIterator(list.size()); // à la fin  
    for(;it.hasPrevious();)  
        System.out.println(it.previous());  
}
```

- Remplacer un élément à la liste par l'itérateur

```
public static <T> void replace(List<T> list, T o1, T o2) {  
    ListIterator it=list.listIterator();  
    for(;it.hasNext()) {  
        if (o1.equals(it.next()))  
            it.set(o2);  
    }  
}
```



# Sous-liste

- `List<E> subList(int start,int end)` est une liste partielle (start inclus, end exclus) de la liste sous-forme de vue

```
public static <T extends Comparable<T>> T turtleMax(List<T> list) {  
    int size=list.size();  
    if (size==1)  
        return list.get(0);  
    T v1=turtleMax(list.subList(0,size/2));  
    T v2=turtleMax(list.subList(size/2,size));  
    return (v1.compareTo(v2)<0)?v2:v1;  
}
```

---

# Implantations

---

- L'interface de liste possède 3 implantations différentes :
  - ArrayList
  - Vector (même implantation que ArrayList)
  - Stack (même implantation que ArrayList, ArrayDeque)
  - CopyOnWriteArrayList
  - LinkedList

# AbstractList

- Classe de base des listes indexées
- Manque les méthodes **get(index)** et **size()**

```
public class Integers extends AbstractList<Integer> {
    public Integers(int maxInt) {
        this.maxInt=maxInt;
    }
    public int size() {
        return maxInt;
    }

    public Integer get(int index) {
        return index;
    }
    private final int maxInt;
}

public static void main(String[] args) {
    int sum=0;
    for(int i:new Integers(1000))
        sum+=i;
    System.out.println(sum);
}
```

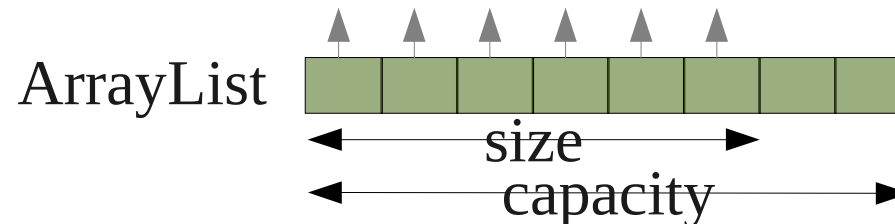
# AbstractList (2)

- Liste mutable si on ajoute set(index,E)

```
public class ArrayView<T> extends AbstractList<T> {  
    public ArrayView(T... array) {  
        this.array=array;  
    }  
    public int size() {  
        return array.length;  
    }  
    public T get(int index) {  
        return array[index];  
    }  
    public T set(int index, T t) {  
        T old=array[index];  
        array[index]=t;  
        return old;  
    }  
    private final T[] array;  
}
```

# ArrayList

- Les éléments sont stockés dans un tableau redimensionnable
- Le facteur de redimensionnement est 1,5
- Pour éviter les allocations inutiles, il est intéressant de passer un taille par défaut (**new ArrayList(1000)**)



---

# Vector

---

- Ancienne version de ArrayList (pre-1.2), a été modifiée pour implanter List
- Synchronisé par défaut (4x plus lent que ArrayList)
- Le facteur de redimensionnement est 2
- A n'utiliser que par compatibilité avec des APIs existantes

---

# Stack

---

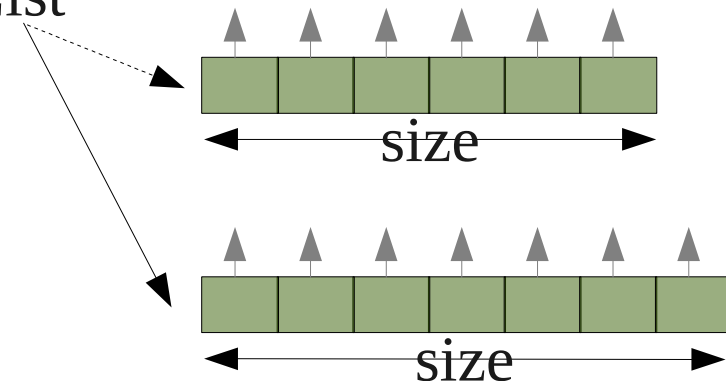
- Hérite de `java.util.Vector`
- Synchronisé par défaut (argh!)
- Utiliser `java.util.ArrayDeque` à la place :
  - `Stack.peek()`  $\Leftrightarrow$  `ArrayDeque.peekLast()`
  - `Stack.push()`  $\Leftrightarrow$  `ArrayDeque.offerLast()`
  - `Stack.pop()`  $\Leftrightarrow$  `ArrayDeque.pollLast()`
- Est très peu utilisé

# CopyOnWriteArrayList

- N'hérite pas de AbstractList
- Gère la concurrence en évitant les synchros
- Effectue une copie à chaque modification
  - Permet d'utiliser des itérateurs sans synchro
  - Itérateur read-only

```
CopyOnWriteArrayList<String> cow=  
    new CopyOnWriteArrayList<String>();  
cow.add("toto");  
Iterator<String> it=cow.iterator();  
cow.add("tutu");  
for(;it.hasNext();)  
    System.out.println(it.next()); // toto
```

COWArrayList





---

# CopyOnWriteArrayList (2)

---

- Exemple avec plusieurs threads

```
final CopyOnWriteArrayList<String> list=new CopyOnWriteArrayList<String>();
final ScheduledExecutorService service=Executors.newScheduledThreadPool(3);
service.scheduleAtFixedRate(new Runnable() {
    public void run() {
        for(String s:list)    // ou System.out.println(Arrays.toString(list));
            System.out.print(s);
            System.out.println();
    }
},0,333,TimeUnit.MILLISECONDS);
for(String arg:args) {
    Thread.sleep(1000);
    list.add(arg);
}
service.shutdown();
```

---

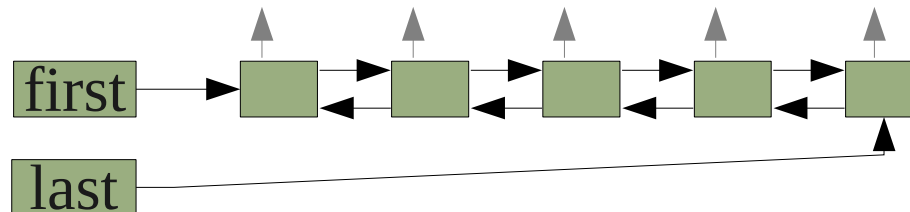
# AbstractSequentialList

---

- Classe de base pour les listes séquentielles
- Bizarrement, hérite de **AbstractList**
- On doit redéfinir les méthodes **size()** et **listIterator()**
- Existe une seule sous-classe : **LinkedList**

# LinkedList

- Liste toujours doublement chaînée
- Maintient une référence sur le début et la fin de la liste
- La taille est conservée pour éviter le parcours



---

# Tableau vu comme une List

---

- Implantation non nommée qui permet de voir un tableau comme une liste
- Tout changement effectué sur le tableau est repercuté sur la liste et vice-versa
- `<T> List<T> Arrays.asList(T[] array)`

```
public static void main(String[] args) {  
    List<String> list=Arrays.asList(args);  
    System.out.println(args[0]);        // hello  
    System.out.println(list.get(0));    // hello  
    list.set(0,"tutu");  
    System.out.println(args[0]);        // tutu  
    list.add("world");                  // UnsupportedOperationException  
}
```

# Complexités

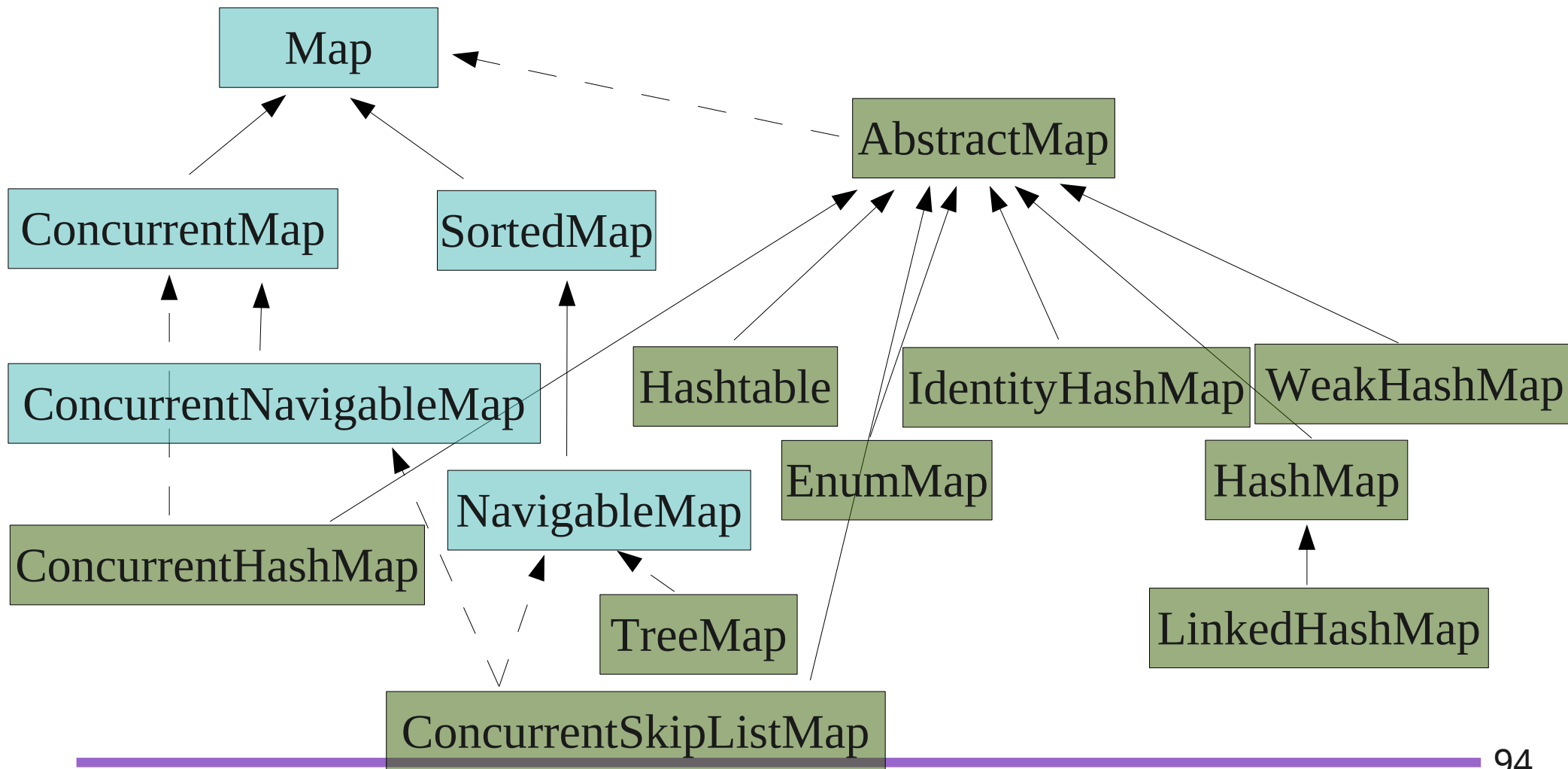
- Complexité des implantations de listes

	size()	clear()	get/set	add(début)	add()fin	remove(int)
ArrayList	<b>O(1)</b>	<b>O(n)</b>	<b>O(1)</b>	<b>O(n)</b>	<b>O(1)</b>	<b>O(n)</b>
COWArrayList	<b>O(1)</b>	<b>O(1)</b>	<b>O(1)</b>	<b>O(n)</b>	<b>O(n)</b>	<b>O(n)</b>
LinkedList	<b>O(1)</b>	<b>O(1)</b>	<b>O(n)</b>	<b>O(1)</b>	<b>O(1)</b>	<b>O(n)</b>

	iterateur	lstdxOf()	Parcours indexé	Parcours itérateur
	add()	indexOf()		
	remove()	contains()		
ArrayList	<b>O(n)</b>	<b>O(n)</b>	<b>O(n) / 2</b>	<b>O(n) / 3</b>
COWArrayList	<b>O(n)</b>	<b>O(n)</b>	<b>O(n) / 1</b>	<b>O(n) / 2</b>
LinkedList	<b>O(1)</b>	<b>O(n)</b>	<b>O(n<sup>2</sup>)</b>	<b>O(n) / 1</b>

# Map

- Association entre une clé et une valeur



---

# Map<K,V>

---

- Les associations sont des types paramétrés par le type des clés (**K**) et le type des valeurs associées (**V**)
- Toutes les associations sauf exception :
  - acceptent **null** comme clé ou valeur
  - ne permettent pas les accès concurrents
  - Testent si un objet existe ou non par rapport à la méthode **equals()** de l'objet

- L'interface Map définit les méthodes
  - **isEmpty()**, **size()**, **clear\***()
  - **put\***(), **get()**, **remove\***()
  - Opération groupée : **putAll\***()
  - Tests : **containsKey()**, **containsValue()**
  - Vues : **keySet()**, **values()**, **entrySet()**



---

# Méthode d'accès

---

- L'interface Map définit les méthodes d'accès
  - V **put**(K clé, V valeur)  
associe une clé à une valeur, renvoie l'ancienne valeur ou null
  - V **get**(Object clé)  
demande la valeur pour une clé, null si il n'y a pas d'association
  - boolean **remove**(Object clé)  
supprime le couple clé/valeur à partir d'une clé, vrai si l'association a été modifiée

---

# Opération groupée et Test

---

- L'interface Map définit une seule opération groupée
  - **putAll**(Map<? extends K,? extends V> map)  
permet d'insérer des couples dans une association
- Il existe deux opération permettant de tester :
  - **containsKey**(Object clé) qui teste si une clé est présente
  - **containsValue**(Object value) qui teste si une valeur existe
- **containsKey** n'est pas obligatoire si l'on doit faire un `get()` car celui-ci renvoie null

---

# Map.Entry<K,V>

---

- Interface (interne) définissant un couple clé/valeur.
- Elle est paramétrée par le type de la clé (**K**) et le type de la valeur (**V**)
- Map.Entry possède les méthodes :
  - K **getKey()** : renvoie la clé
  - V **getValue()** : renvoie la valeur
  - V **setValue(V value)** : change la valeur, renvoie l'ancienne

---

# Les Vues

---

- L'interface Map définit trois vues :
  - Set<K> **keySet()** :  
l'ensemble des clés
  - Collection<V> **values()** :  
la collection des valeurs
  - Set<Map.Entry<K,V>> **entrySet()** :  
l'ensemble des couples clé/valeur
- Toute modification sur la vue est reflétée sur l'association et vice-versa

---

# Exemple d'utilisation de vue

---

- Calcule la fréquence d'apparition des mots passés en argument

```
public static <T> Map<T,Integer> frequency(List<T> values) {
    HashMap<T,Integer> map=new HashMap<T,Integer>();
    for(T v:values) {
        Integer i=map.get(v);
        map.put((i==null)?1:i+1);
    }
    return map;
}
public static void main(String[] args) {
    Map<String,Integer> map=frequency(Arrays.asList(args));
    for(Map.Entry<String,Integer> entry:map.entrySet())
        System.out.printf("%s: %d\n",entry.getKey(),entry.getValue());
}
```

---

# Implantations

---

- L'interface Map possède 8 implantations :
  - HashMap
  - HashTable (même implantation que HashMap)
  - LinkedHashMap
  - IdentityHashMap
  - WeakHashMap
  - EnumMap
  - ConcurrentHashMap (ConcurrentMap)
  - ConcurrentSkipListMap (ConcurrentNavigableMap)
  - TreeMap (NavigableMap)

---

# AbstractMap

---

- Classe de base des Map, manque **entrySet()**

```
public static <T> Map<Integer,T> asMap(final List<T> list) {  
    return new AbstractMap<Integer,T>() {  
        public T get(Object key) {  
            if (!(key instanceof Integer))  
                return null;  
            return list.get(((Integer)key).intValue());  
        }  
        public Set<Map.Entry<Integer,T>> entrySet() {  
            return new AbstractSet<Map.Entry<Integer,T>>() {  
                public int size() {  
                    return list.size();  
                }  
                public Iterator<Map.Entry<Integer,T>> iterator() {  
                    final ListIterator<T> it=list.listIterator();  
                    return new Iterator<Map.Entry<Integer,T>>() {  
                        public boolean hasNext() {  
                            return it.hasNext();  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

# AbstractMap (2)

- Suite de l'exemple :

```
public Map.Entry<Integer,T> next() {  
    return new SimpleImmutableEntry<Integer,T>(it.nextIndex(),it.next());  
}  
public void remove() {  
    it.remove();  
}  
};  
};  
};  
};  
};
```

```
// java ListAsMap hello world
public static void main(String[] args) {
    Map<Integer,String> map=asMap(Arrays.asList(args));
    System.out.println(map); // {0=hello, 1=world}
}
```



---

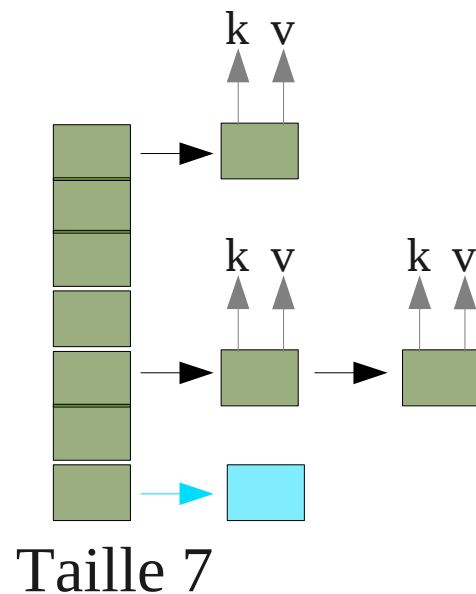
# Map.Entry

---

- **AbstractMap** possède deux classe implémentant Map.Entry :
  - AbstractMap.**SimpleEntry**<K,V> pour les couples clés/valeurs mutable
  - AbstractMap.**SimpleImmutableEntry**<K,V> pour les couples clés/valeurs non mutable
- Ces classes fournissent une implémentation par défaut pour les Map.Entry

# HashMap

- Table de hachage avec liste chaînée pour les collisions se redimensionnant au besoin (\*2 si facteur de remplissage atteint)



```
map.put(k,v)  
k.hashCode()= 20  
% 7=6
```

- La table grossit en fonction d'un facteur de remplissage, par défaut 75%

---

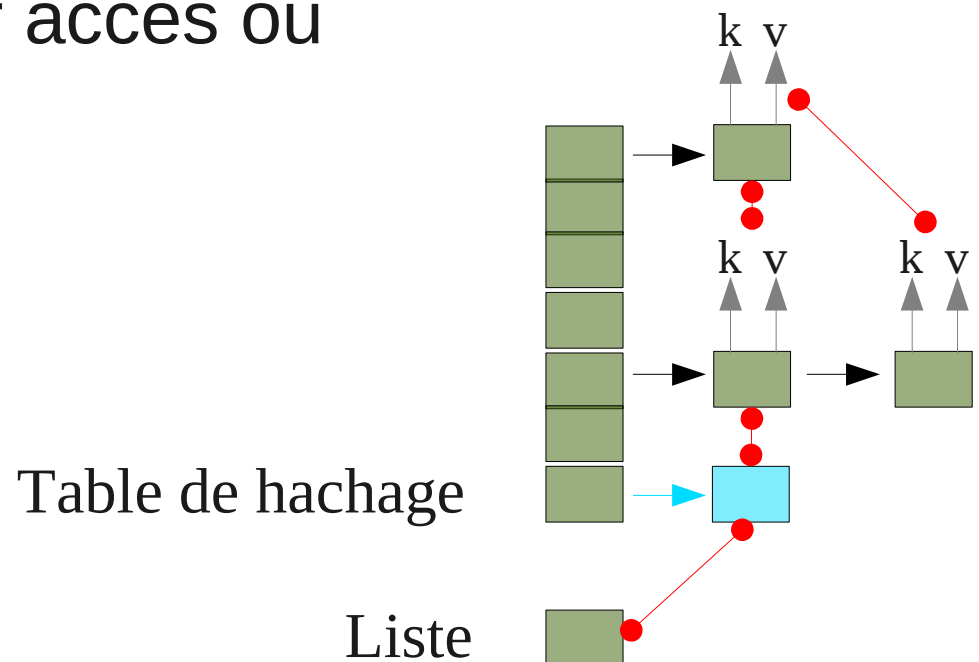
# Hashtable

---

- Ancienne version de HashMap (pre-1.2), a été modifiée pour implanter Map
- N'accepte pas null comme clé
- Synchronisé par défaut (4x plus lent que HashMap)
- Le facteur de redimensionnement est 2
- A n'utiliser que par compatibilité avec des APIs existantes

# LinkedHashMap

- Table de hachage + liste doublement chaînée
- Parcours + efficace que **HashMap** mais consomme + de mémoire
- Ordre dans la liste par accès ou par insertion
- L'itérateur utilise la liste



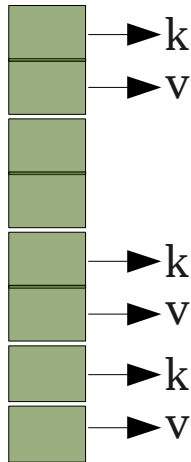
# LinkedHashMap (2)

- Il est possible de transformer facilement une LinkedHashMap en cache

```
public class Cache<K,V> extends LinkedHashMap<K,V> {
    public Cache(int maxCacheSize) {
        super(11,0.5f,true); // (capacity,load factor,access/insertion)
        this.maxCacheSize=maxCacheSize;
    }
    protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
        return size() > maxCacheSize;
    }
    private final int maxCacheSize;
    public static void main(String[] args) {
        Cache<String,String> cache=new Cache<String,String>(3);
        cache.put("annie","dog");
        cache.put("virgil","snake");
        cache.put("jordan","cat");
        cache.get("annie"); // dog
        cache.put("melanie","fish");
        System.out.println(cache); // {jordan=cat, annie=dog, melanie=fish}
    } }
```

# IdentityHashMap

- Table de hachage “fermée”, un seul tableau
- Si il y a collision, on utilise une deuxième fonction de hachage



Taille 4

- Utilise `System.identityHashCode()` au lieu de `hashCode()`
- Utilise `==` pour tester l'égalité au lieu de `equals()`

# Exemple d'IdentityHashMap

```
public class Node {
    public Node(int number) {
        this.number=number;
    }
    public void add(Node node) {
        nodes.add(node);
    }
    public String toString() {
        Map<Node,?> visited=new IdentityHashMap<Node, Object>();
        return traverse(visited,this,new StringBuilder()).toString();
    }
    private StringBuilder traverse(Map<Node,?> visited,Node node,
                                   StringBuilder builder) {
        if (visited.containsKey(node))
            return builder.append("... ");
        visited.put(node,null); // ok
        builder.append(node.number).append(' ');
        for(Node sub:node.nodes)
            traverse(visited,sub,builder);
        return builder;
    }
    private final int number;
    private final ArrayList<Node> nodes=new ArrayList<Node>();
}

public static void main(String[] args) {
    Node root=new Node(1);
    Node node2=new Node(2);
    Node node3=new Node(3);
    root.add(node2); root.add(node3);
    node2.add(root);
    System.out.println(root);
} // 1 2 -.- 3
```

---

# WeakHashMap

---

- Table de hachage semblable à HashMap mais les clés sont stockées en utilisant des références faibles (*weak*)
- Si la clé n'est pas référencée autre part dans le programme, le couple clé/valeur est supprimé de la table



---

# EnumMap

---

- Map spécialisée permettant d'associer une valeur à une clé qui est une valeur d'une même énumération
- Implanté sous forme d'un tableau non redimensionnable (pas besoin)
- Toutes les opérations possèdent la même complexité que HashMap mais elles sont plus rapides

---

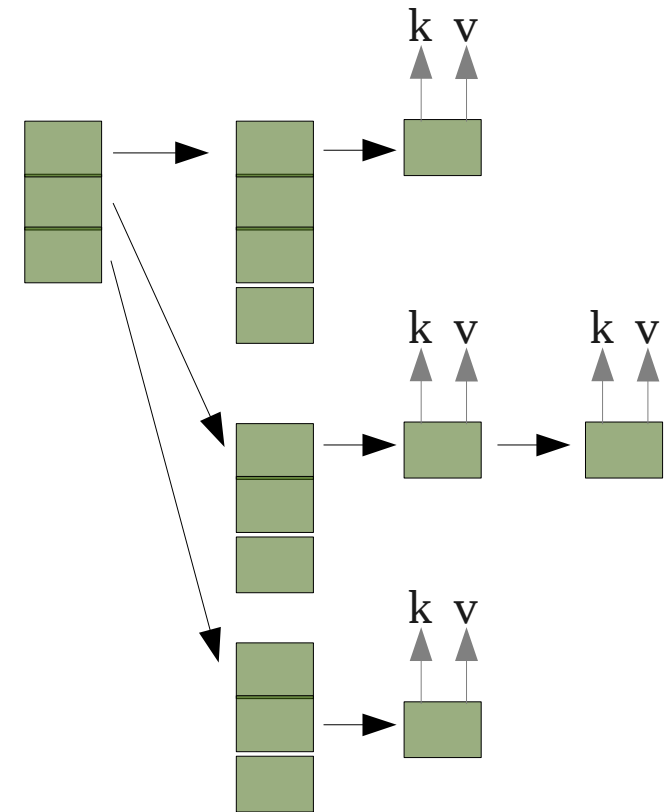
# ConcurrentMap

---

- Interface des tables de hachage à accès concurrent
  - Ajoute les méthodes :
    - V **putIfAbsent**(K clé, V valeur)  
associe la valeur à la clé si la clé n'est pas déjà associée
    - boolean **remove**(Object clé, Object valeur)  
supprime un couple clé/valeur
    - V **replace**(K key, V value)  
remplace la valeur d'une clé si celle-ci existe
    - boolean **replace**(K key, V ov, V nv)  
remplace le couple key/ov par key/nv

# ConcurrentHashMap

- Map permettant les accès concurrents (n'utilise pas de synchronized !!)
- Table de Table de hachage
- Chaque sous table possède un verrou ce qui évite de bloquer toute la table lors de deux accès en écriture
- Les accès en lecture ne bloquent pas les sous-tables



---

# SortedMap

---

- Interface indiquant que l'ensemble des clés est ordonné (par un comparateur)
- Définit les méthodes :
  - `Comparator<? super K> comparator()`
  - `K firstKey(), lastKey()`
  - `SortedMap<K,V> headMap(K),subMap(K,K),tailMap(K)`
- Les sous-ensembles créés sont des vues de l'ensemble de départ

---

# NavigableMap

---

- Interface dont l'ensemble des clés est ordonné par un comparateur et navigable (**NavigableSet**)
  - **E** **lowerKey**(K k), **floorKey**(K), **higherKey**(K), **ceilingKey**(K)
  - **E** **lowerEntry**(K k), **floorEntry**(K), **higherEntry**(K), **ceilingEntry**(K)
    - Resp. renvoie la clé ou l'entry <, <=, > et >= à k.
  - **Map.Entry** **firstEntry**(), **lastEntry**()
    - Renvoie le premier/dernier entry
  - **Map.Entry** **pollFirstEntry**(), **pollLastEntry**()
    - Renvoie en le supprimant le premier/dernier entry

---

# NavigableMap (2)

---

- Vues en ordre descendant :
  - NavigableSet<K> **descendingKeySet()**
  - NavigableMap<K,V> **descendingMap()**
- Vue de la NavigableMap :
  - NavigableMap<K,V> **headMap**(K toElement, boolean inclusive)
  - NavigableMap<K,V> **subMap**(K fromElement, boolean fromInclusive, K toElement, boolean toInclusive)
  - NavigableMap<K,V> **tailMap**(K fromElement, boolean inclusive)

---

# TreeMap

---

- Utilise un arbre rouge/noir et classe les clés en utilisant :
  - Un comparateur
  - Ou l'ordre naturel (`Comparable.compareTo()`)
- L'implantation garantit  **$O(\ln n)$**  pour l'insertion, la suppression ou la recherche sur l'ensemble des clés

---

# ConcurrentSkipListMap

---

- Implantation de NavigableMap à base d'un arbre fils gauche/frère droit à deux niveaux
- L'implantation garantit  $O(\ln n)$  pour l'insertion, la recherche ou la suppression
- **size()** n'est pas une opération en temps constant
- Pour **entrySet()**, les Entry produites sont *read-only* (**SimpleImmutableEntry**)



# Complexités

- Complexité des implantations de Map

	size()	put/get	vues/iterator
HashMap	<b>O(1)</b>	O(1) / 2	O(n) / 5
LinkedHashMap	<b>O(1)</b>	O(1) / 2	O(n) / 2
IdentityHashMap	<b>O(1)</b>	O(1) / 4	O(n) / 5
WeakHashMap	<b>O(1)</b>	O(1) / 3	O(n) / 6
EnumMap	<b>O(1)</b>	O(1) / 1	O(n) / 1
CrtHashMap	<b>O(n)</b>	O(1) / 5	O(n) / 6
TreeMap	<b>O(1)</b>	O(ln n)	O(n) / 3
ConcSkipListMap	<b>O(n)</b>	O(ln n)	O(n) / 4

- Les valeurs (/ n) spécifient un ordre de vitesse (sur mon ancienne machine !!)

---

# Collections

---

- La classe `java.util.Collections` contient un ensemble de méthodes statiques permettant :
  - de créer des collections immutables
  - de créer des vues de collections (immutable, checked, synchronisée)
  - de voir une collection comme une autre
  - d'utiliser des algorithmes sur
    - Les collections
    - Les listes

---

# Les Collections Immutables

---

- Collections sans élément
  - `<T> List<T> emptyList()`
  - `<K,V> Map<K,V> emptyMap()`
  - `<T> Set<T> emptySet()`
- Collections Singleton (1 seul élément/couple)
  - `<T> Set<T> singleton(T élément)`
  - `<T> List<T> singletonList(T élément)`
  - `<K,V> Map<K,V> singletonMap(K clé, V valeur)`

# Les Collections Immutables (2)

- Listes d'éléments identiques
  - `<T> List<T> nCopies(int taille, T élément)`

```
public class SmartNode implements Iterable<SmartNode> {
    public SmartNode add(SmartNode node) {
        switch(nodes.size()) {
            case 0:
                nodes=Collections.<SmartNode>singletonList(node);
                break;
            case 1:
                nodes=new ArrayList<SmartNode>(nodes);
            default:
                nodes.add(node);
        }
        return this;
    }
    ...
    private List<SmartNode> nodes=Collections.<SmartNode>emptyList();
}
```

---

# Vues vérifiées de collections

---

- Vues vérifiées (checked) (1.5)
  - *Collection*<E> *checkedCollection*(*Collection*<E> c, *Class*<E> type)
    - Avec les Interfaces *Collection*/*Set*/*SortedSet*/*List*
  - *Map*<K,V> *checkedMap*(*Map*<K,V> m, *Class*<K> keyType, *Class*<V> valueType)
    - Avec les Interfaces *Map*/*SortedMap*
- Pratique surtout pour déboguer

# Exemple de vue vérifiée

- Exemple avec ou sans la vue vérifiée

```
public static void main(String[] args) {
    List<String> list=new ArrayList<String>();
    // scénario 2, ajout de cette ligne
    list=Collections.checkedList(list,String.class);

    Collections.addAll(list,args);

    List<? super Integer> iList=
        (List<? super Integer>)(List<?>)list; // unsafe

    iList.add(3);
    // 2. ClassCastException: Attempt to insert class Integer element
    // 2. into collection with element type class String

    for(String s:list) // 1. ClassCastException
        System.out.println(s.length());
}
```

---

# Vues immutables de collections

---

- Vues immutables
  - `<T> Collection<T> unmodifiableCollection(Collection<? extends T> c)`
  - `<K,V> Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> m)`

# Exemple de vue immutable

- Permet de renvoyer des collections immutables

```
public class MyNode {
    public void add(MyNode node) {
        nodes.add(node);
    }
    public List<MyNode> children() {
        return unmodifiableNodes;
    }
    private List<MyNode> nodes=new LinkedList<MyNode>();
    private final List<MyNode> unmodifiableNodes=
        Collections.<MyNode>unmodifiableList(nodes);
    public static void main(String[] args) {
        MyNode root=new MyNode();
        ...
        for(MyNode node:root.children())
            System.out.println(node);
        root.children().set(0,new MyNode()); //UnsupportedOperationException
    }
}
```



---

# Vues synchronisées

---

- Vues synchronisées
  - `<T> Collection<T> synchronizedCollection(Collection<? extends T> c)`
  - `<K,V> Map<K,V> synchronizedMap(Map<? extends K, ? extends V> m)`
- Pas très utile, utiliser plutôt :
  - Des blocs `synchronized` là où il faut
  - Des collections concurrentes

# Exemple de vue synchronisée

- Attention à synchronizer l'iterateur !!

```
final List<Integer> list=Collections.synchronizedList(
    new ArrayList<Integer>());
final Random random=new Random();
final ScheduledExecutorService service=Executors.newScheduledThreadPool(2);
service.scheduleAtFixedRate(new Runnable() {
    public void run() {
        list.add(random.nextInt(1000));
    }
},0,3,TimeUnit.MILLISECONDS);
service.scheduleAtFixedRate(new Runnable() {
    public void run() {
        synchronized(list) { // obligatoire !!
            for(int value:list)
                if (value==100) {
                    System.out.println(list.size());
                    service.shutdownNow();
                    return;
                }
        }
    }
},0,1000,TimeUnit.MILLISECONDS); }
```

---

# Voir une ... comme une

---

- Créer un Set à partir d'une Map (1.6):
  - `<E> Set<E> newSetFromMap(Map<E, Boolean> map)`  
la Map doit être vide et ne plus être accédée

Permet de créer un Set pour les implantation de Map

```
Set<String> weakHashSet = Collections.newSetFromMap(  
    new WeakHashMap<String, Boolean>());
```

- Voir un **Deque** comme une **Queue** (1.6)
  - `<T> Queue<T> asLifoQueue(Deque<T> deque)`  
l'insertion s'effectue en tête et la suppression en queue

---

# Algorithmes sur les collections

---

- Collections contient 3 algorithmes sur les collections :
  - boolean **disjoint**(Collection<?> c1, Collection<?> c2)  
test si l'intersection est vide
  - int **frequency**(Collection<?> c, Object o)  
renvoie le nombre d'occurrence d'un objet
  - T **max/min**(Collection<? extends T> coll, Comparator<? super T> comp)  
renvoie le maximum/minimum d'une collection

---

# Algorithmes sur les listes

---

- Et 10 algorithmes sur les listes
  - Recherche dichotomique
    - `<T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)`
  - Trie
    - `<T> void sort(List<T> list, Comparator<? super T> c)`
  - Remplissage de liste
    - `<T> void fill(List<? super T> list, T obj)`
  - Copie de liste
    - `<T> void copy(List<? super T> dest, List<? extends T> src)`

---

# Algorithmes sur les listes

---

- Inverser un liste
  - void **reverse**(List<?> list)
- Recherche de sous-liste
  - int **indexOfSubList**(List<?> source, List<?> target)
  - int **lastIndexOfSubList**(List<?> source, List<?> target)
- Remplacement d'un élément par un autre
  - <T> boolean **replaceAll**(List<T> list, T oldVal, T newVal)
- Déclage à droite ou à gauche
  - void **rotate**(List<?> list, int distance)
- Mélange aléatoire
  - void **shuffle**(List<?> list, Random rnd)

---

# En résumé

---

- Choisir pour l'algorithme que vous voulez implémenté
  - 1) l'interface à utiliser (List, Set, Queue, Deque, Map)
  - 2) Puis l'implantation
  - 3) Si vous hésitez entre deux implantations :
    - faire un test en vrai grandeur (pas un micro-test)