L'API Collections de java
(package java.util)

Philippe Genoud
dernière mise à jour : 10/02/2024 17:19

L'API des Collections dans java.util

- Collection :
  - *objet qui regroupe de multiples objets (éléments) dans une seule entité.*
- Utilisation de collections pour
  - *stocker, retrouver et manipuler des données*
  - *transmettre des données d'une méthode à une autre*
- Exemples :
  - *un dossier de courrier: collection de mails,*
  - *un répertoire téléphonique: collection d'associations noms numéros de téléphone*
  - *une file d'attente d'événements*
  - *…*

L'API des Collections dans java.util

- Dans les premières versions de Java quelques implémentations pour différents types de collections :
  - *tableaux*
  - `Vector` *: tableaux dynamiques (listes)*
  - `HashTable` *: tables associatives*

- Une **modification majeure** de Java 2 a été d'inclure un véritable « **framework** » pour la gestion des collections (package `java.util`)
  - *architecture unifiée pour représenter et manipuler les collections*
  - *composée de trois parties*
    - *une hiérarchie d'interfaces permettant de représenter les collections sous forme de types abstraits*
    - *des implémentations de ces interfaces*
    - *des algorithmes réalisant des opérations fréquentes sur les collections (recherche, tri…)*
  - *similaire à STL (Standard Template Library) en C++ et au collection framework de Smalltalk.*

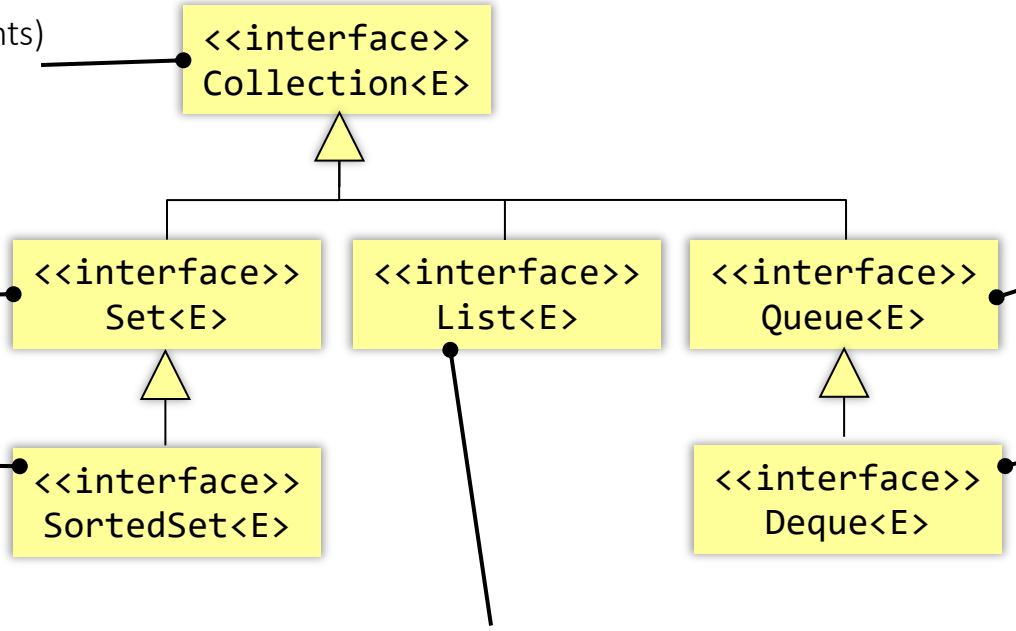
- Depuis Java 5, l'API des collections (`java.util`) s'appuie sur la généricité

- Avantages d'un « framewok » pour les collections :
  - *réduction de l'effort de programmation*
  - *amélioration de la qualité et de la performance des programmes*
  - *permet interopérabilité entre des API non directement liées*
  - *facilite l'apprentissage et l'utilisation de nouvelles API*
  - *réduit l'effort de conception de nouvelles API*
  - *encourage la réutilisation du logiciel*

- Peut être perçu à premier abord comme étant plutôt complexe
  - *le framework proposé pour java ne l'est pas tant que cela (dixit les auteurs du « Java Tutorial », on n'est pas obligés d 'être d'accord…)*
  - ***le retour sur investissement est important***

Les principaux types

Racine de la hiérarchie
- représente un groupe d'objets (les éléments)
- pas d'implémentation directe dans le jdk



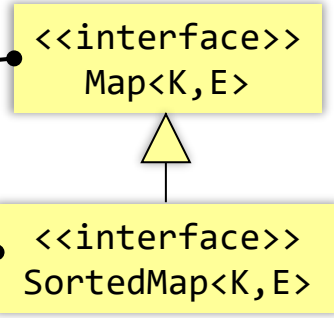
Collection dont les éléments ne peuvent être dupliqués (abstraction de la notion d'ensemble en mathématiques)

Ensemble trié

file d'attente avec une notion de tête et queue (FIFO first-in first-out)

file d'attente où les éléments peuvent être inséré en tête et en queue (FIFO first-in first-out, LIFO last-in, first-out)

Collection dont les éléments sont ordonnés (et peuvent être accédés via un index (0..n-1). Les éléments dans une liste peuvent être dupliqués.



Associe un objet à une clé (à une clé ne peut être associé qu'un seul objet)

« Map » qui maintient ses clés triées selon un ordre ascendant (par exemple une dictionnaire, un annuaire téléphonique)

Collections — Interface Collection

Interface `Collection<E>`

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
boolean	**add(E e)** Ensures that this collection contains the specified element (optional operation).
boolean	**addAll(Collection<? extends E> c)** Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear()** Removes all of the elements from this collection (optional operation).
boolean	**contains(Object o)** Returns true if this collection contains the specified element.
boolean	**containsAll(Collection<?> c)** Returns true if this collection contains all of the elements in the specified collection.
boolean	**equals(Object o)** Compares the specified object with this collection for equality.
int	**hashCode()** Returns the hash code value for this collection.
boolean	**isEmpty()** Returns true if this collection contains no elements.
Iterator<E>	**iterator()** Returns an iterator over the elements in this collection.
boolean	**remove(Object o)** Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll(Collection<?> c)** Removes all of this collection's elements that are also c
boolean	**retainAll(Collection<?> c)** Retains only the elements in this collection that are con
int	**size()** Returns the number of elements in this collection.

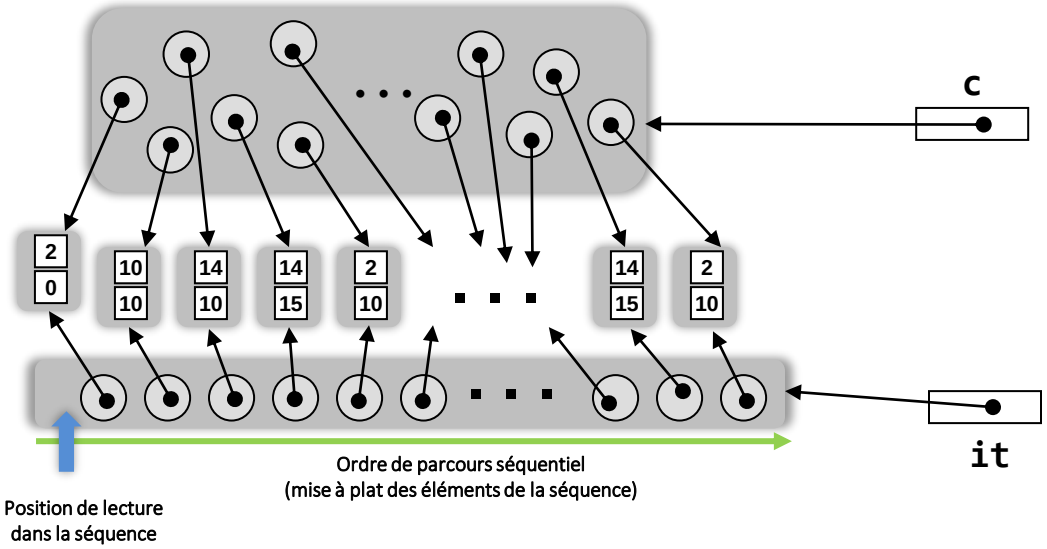
Object[]	**toArray()** Returns an array containing all of the elements in this collection.
<T> T[]	**toArray(T[] a)** Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Méthode optionnelle : une implémentation donnée de cette interface n'est pas obligée de supporter cette opération. Si il n'y a pas de support pour une méthode optionnelle, une `UnsupportedOperationException` sera lancée. Permet de réduire le nombre d'interface tout en répondant à un nombre maximum de cas

`true` si l'opération a modifié la collection

basée sur `equals(Object o)`

renvoie un objet «iterateur» qui permet le parcours séquentiel des éléments d'une collection.
Pas de garantie concernant l'ordre dans lequel les éléments sont retournés.




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

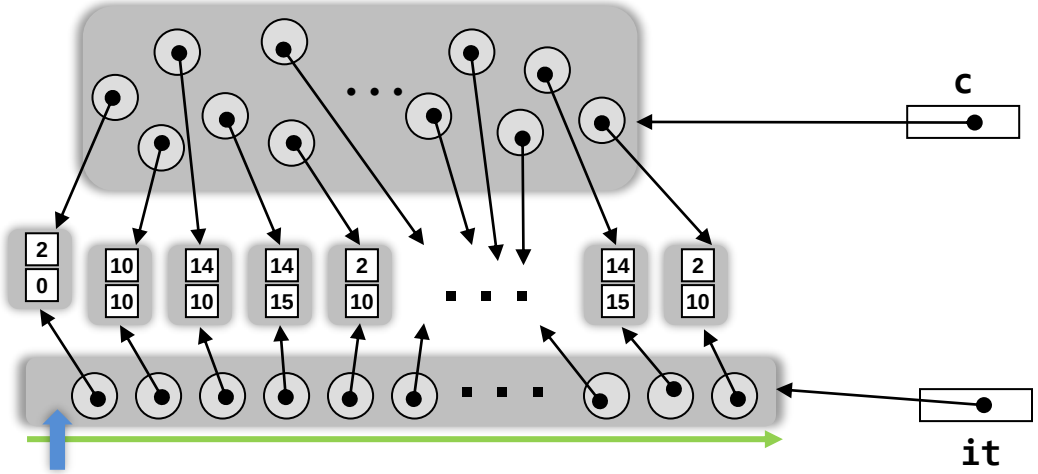
```
Iterator<Point> it = c.iterator();
```

Fourni un objet *itérateur* sur la collection

- Un ordre de parcours (séquence) des éléments
- Une position de parcours (avant le premier élément de la séquence)

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

L'objet *itérateur* est instance d'une classe qui implémente l'interface `Iterator` définie dans le package `java.util`

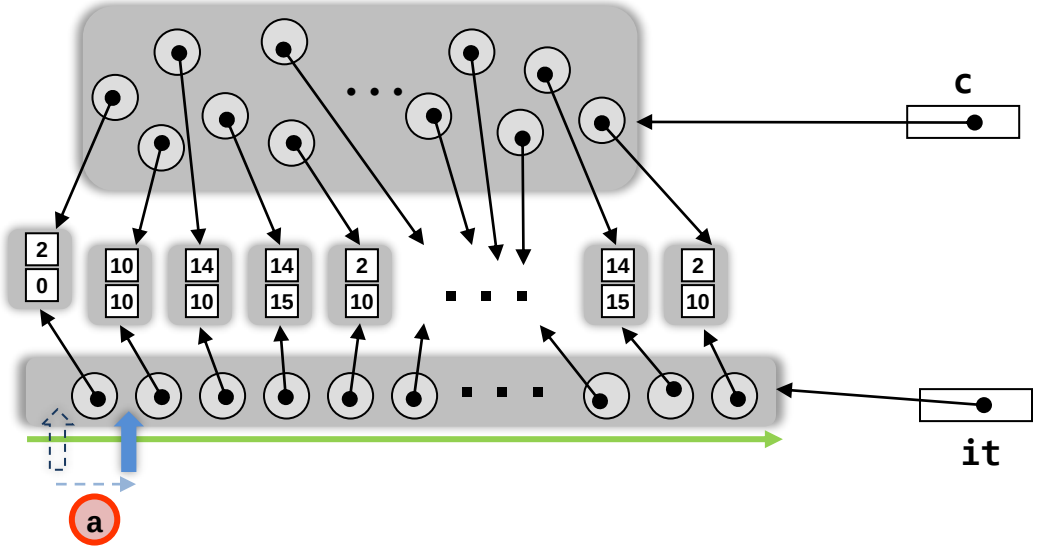



```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Permet d'effectuer une lecture séquentielle :



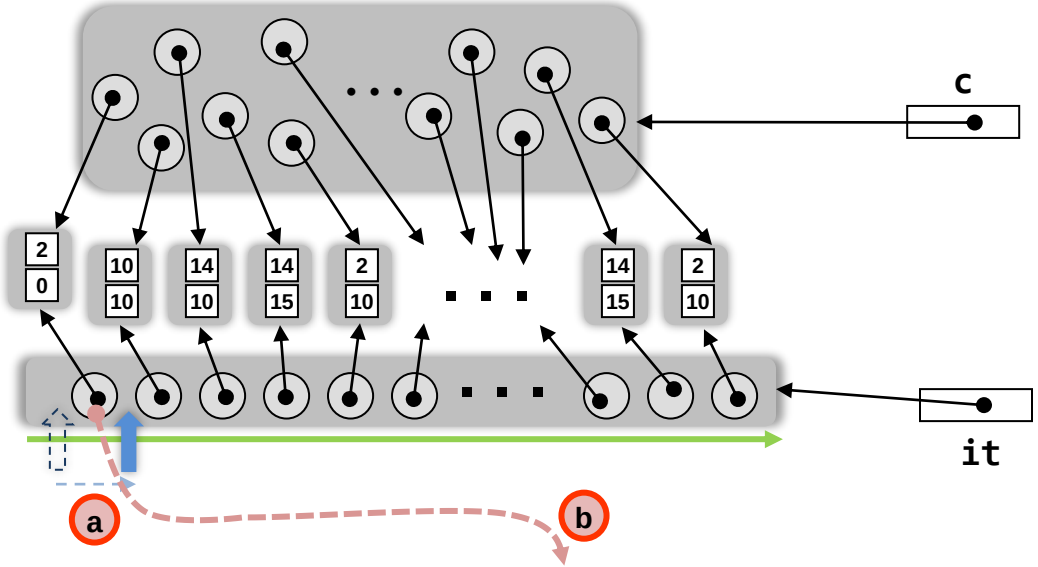

```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Permet d'effectuer une lecture séquentielle :

a) Avance d'un élément dans la séquence




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

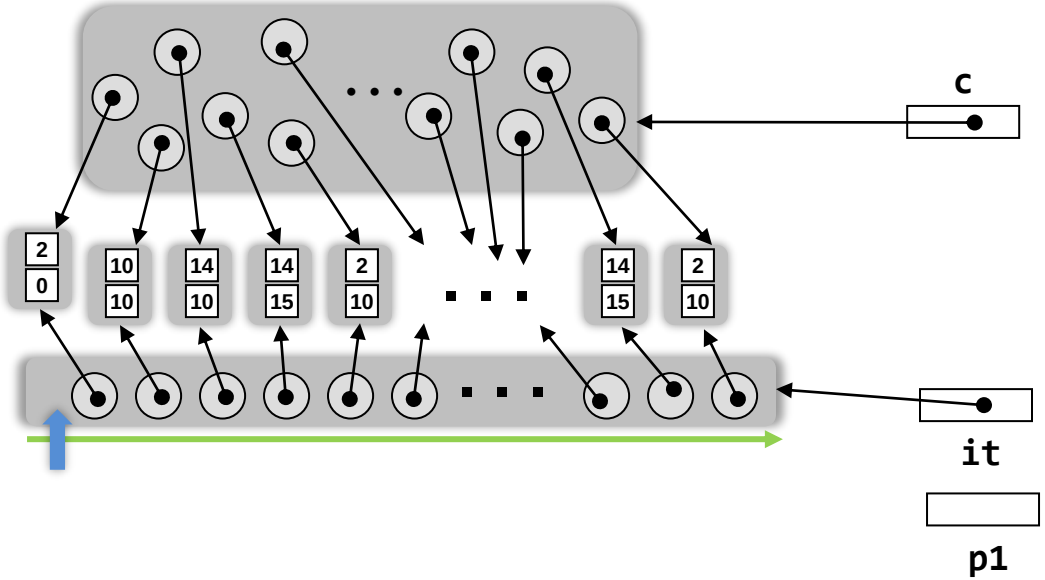
```
Iterator<Point> it = c.iterator();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Permet d'effectuer une lecture séquentielle :

a) Avance d'un élément dans la séquence

b) Retourne la référence de l'élément lu




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

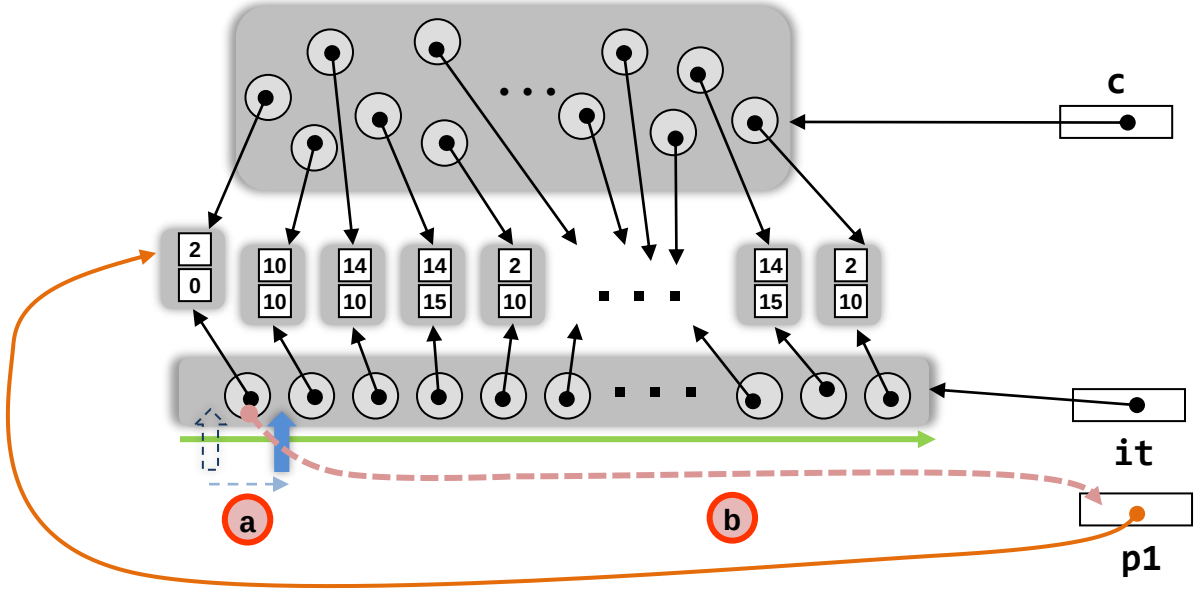
```
Point p1 = it.next();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Permet d'effectuer une lecture séquentielle :

a) Avance d'un élément dans la séquence

b) Retourne la référence de l'élément lu




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

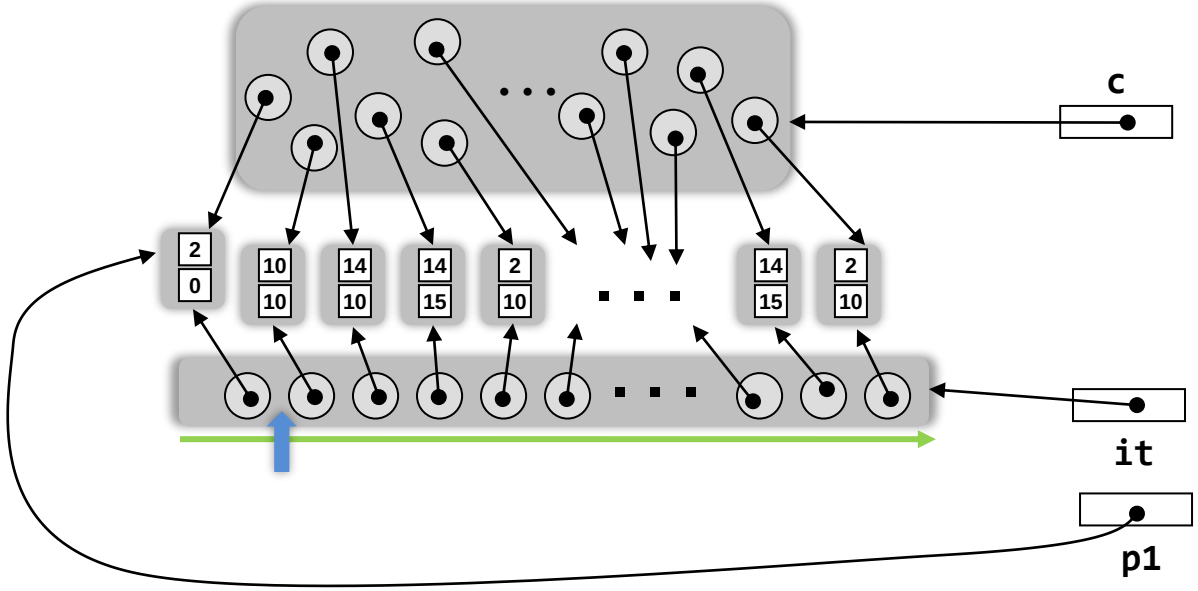
```
Point p1 = it.next();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Permet d'effectuer une lecture séquentielle :

a) Avance d'un élément dans la séquence

b) Retourne la référence de l'élément lu




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

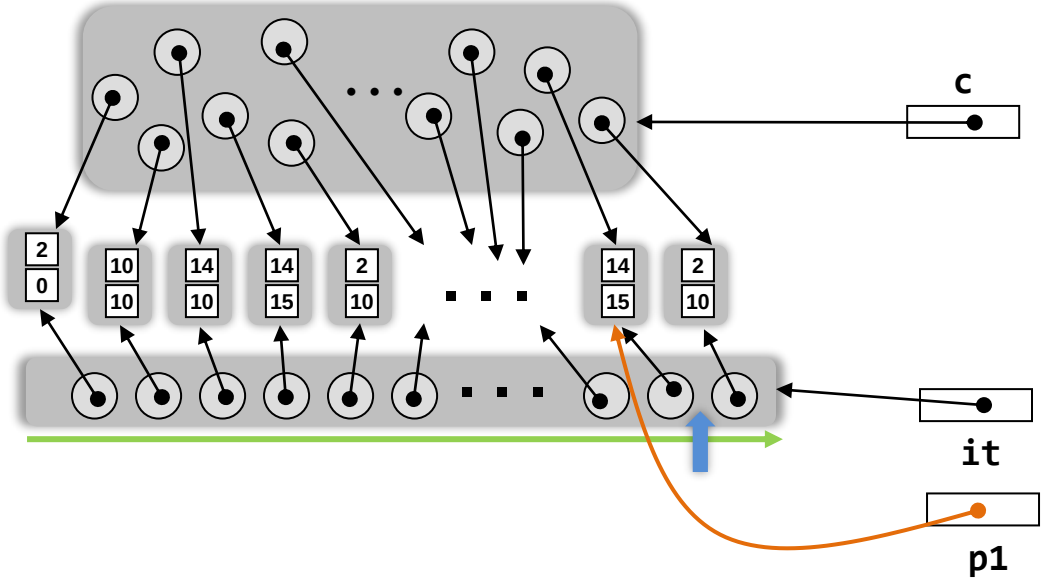
```
Point p1 = it.next();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Test de fin de parcours

true si il y a encore au moins un élément suivant

false sinon




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

```
Point p1 = it.next();
 •••
p1 = it.next();
```

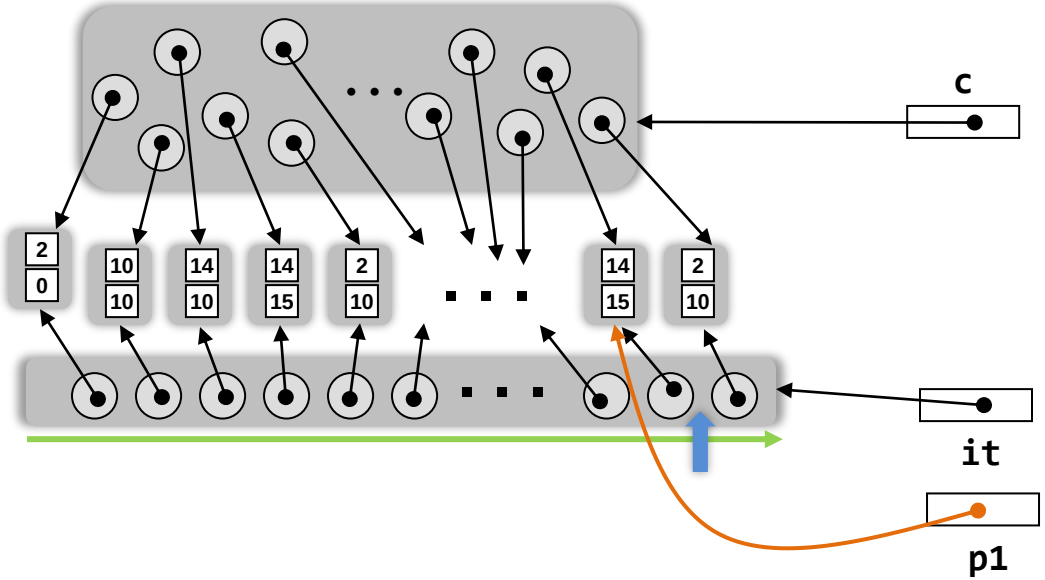
```
it.hasNext();  → true
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Test de fin de parcours

true si il y a encore au moins un élément suivant

false sinon




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

```
Point p1 = it.next();
```

•••

```
p1 = it.next();
```

```
it.hasNext();  → true
```

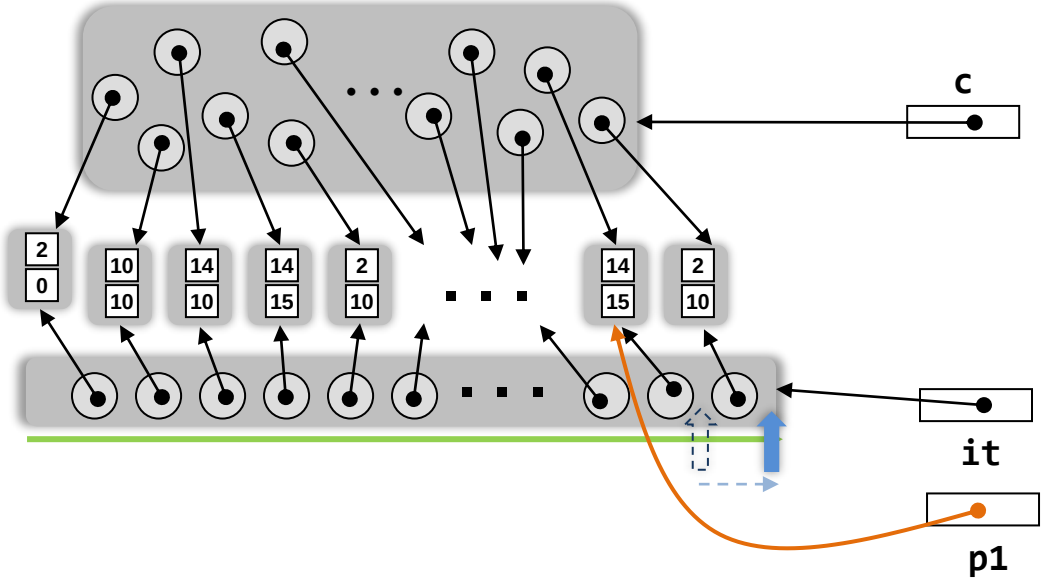
```
it.next();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Test de fin de parcours

true si il y a encore au moins un élément suivant

false sinon




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

```
Point p1 = it.next();
•••
p1 = it.next();
```

```
it.hasNext();  → true
```

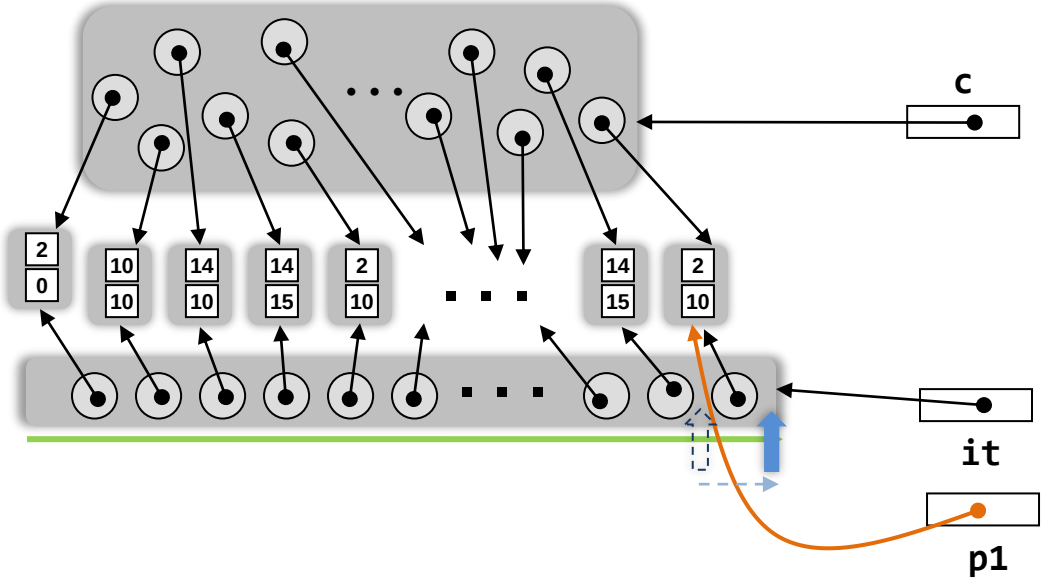
```
it.next();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Test de fin de parcours

true si il y a encore au moins un élément suivant

false sinon




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

```
Point p1 = it.next();
•••
```

```
p1 = it.next();
```

```
it.hasNext();  → true
```

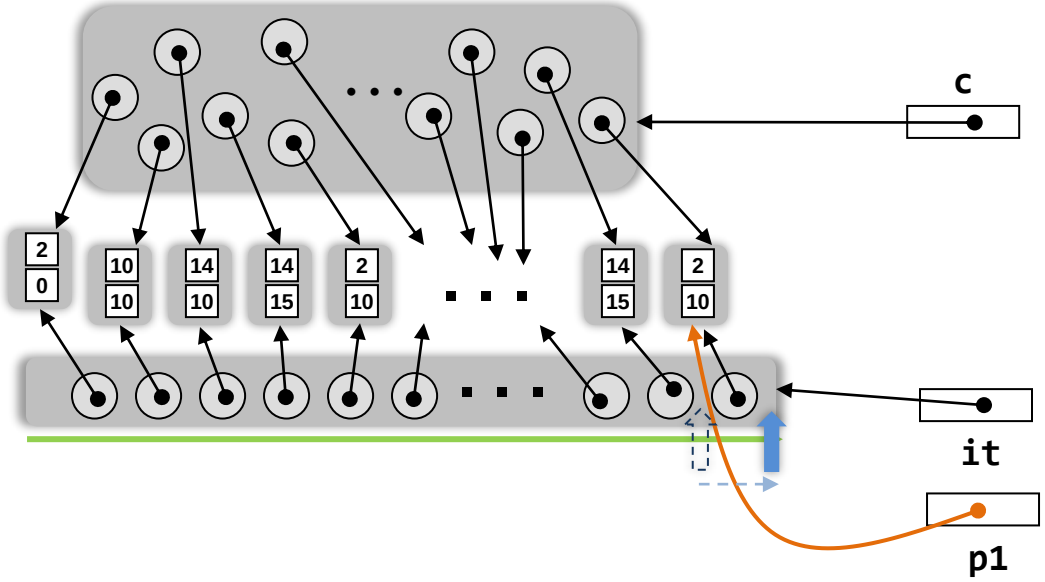
```
it.next();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Test de fin de parcours

true si il y a encore au moins un élément suivant

false sinon




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

```
Point p1 = it.next();
```

•••

```
p1 = it.next();
```

```
it.hasNext();  → true
```

```
it.next();
```

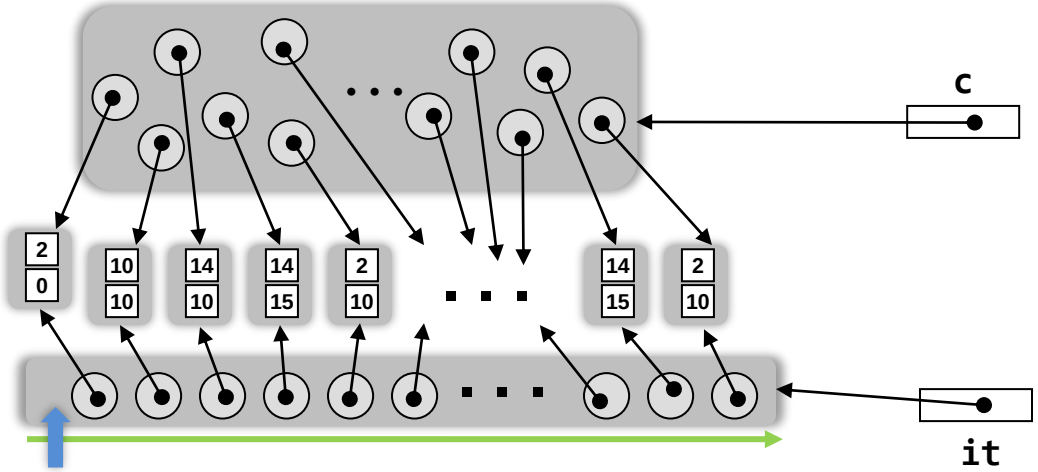
```
it.hasNext();  → false
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Test de fin de parcours

true si il y a encore au moins un élément suivant

false sinon



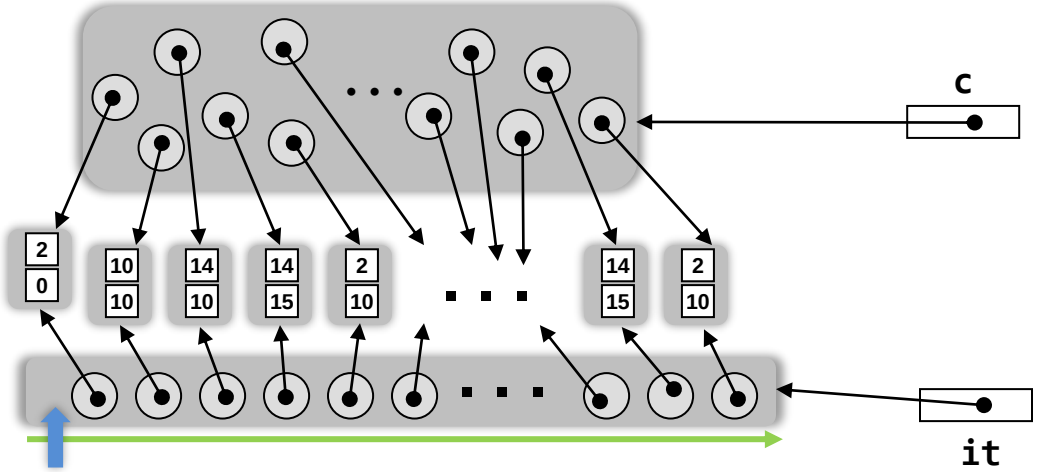

```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Parcours séquentiel de tous les éléments de la collection

```
Iterator<E> it = c.iterator(); //récupérer itérateur
while (it.hasNext()) {  // tant qu'il y a un elt
  E p = it.next();  // lire élément suivant
  ...               // traiter l'élément
}
```




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

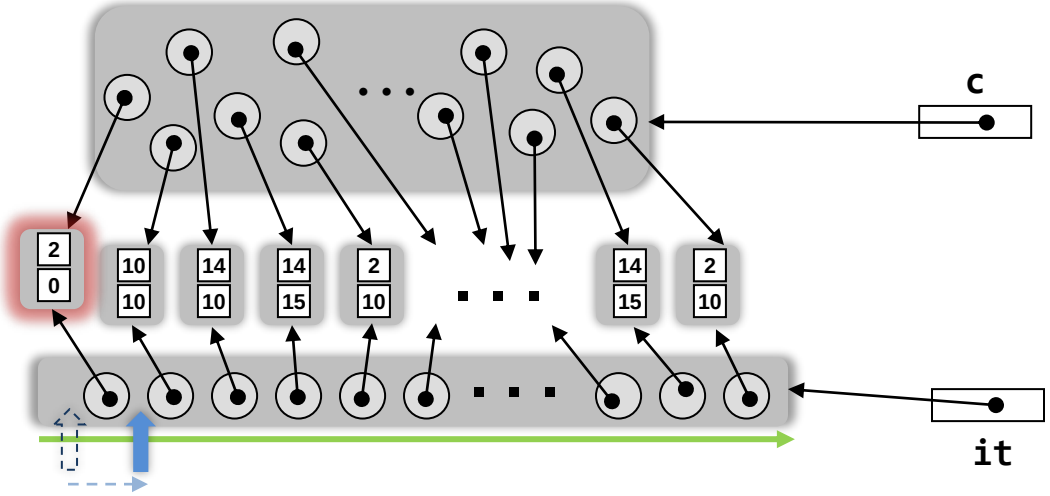
```
Iterator<Point> it = c.iterator();
```

```
while (it.hasNext()) {
  Point p = it.next();
  System.out.println(p.distance());
}
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Parcours séquentiel de tous les éléments de la collection

```
Iterator<E> it = c.iterator(); //récupérer itérateur
while (it.hasNext()) {  // tant qu'il y a un elt
  E p = it.next();  // lire élément suivant
  ...               // traiter l'élément
}
```




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

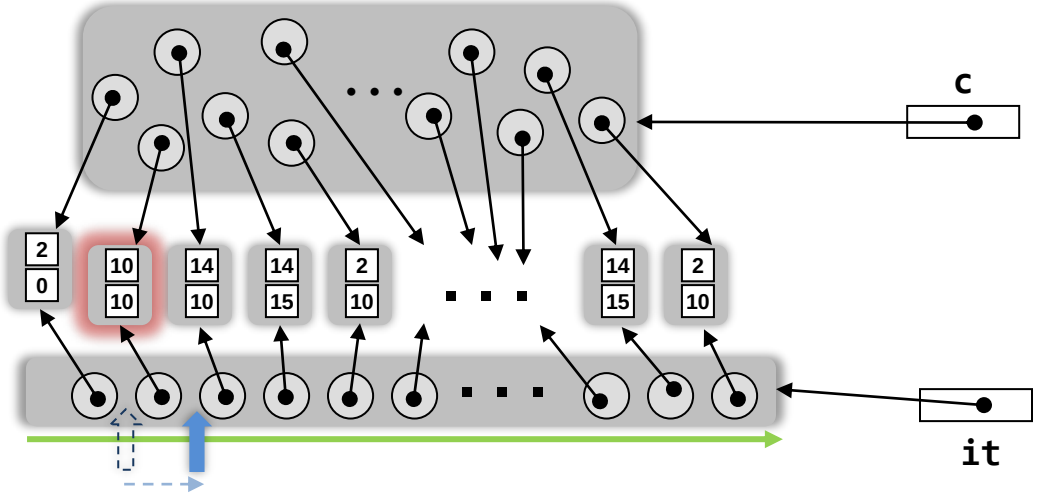
```
while (it.hasNext()) {
  Point p = it.next();
  System.out.println(p.distance());
}
```

→ 2

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Parcours séquentiel de tous les éléments de la collection

```
Iterator<E> it = c.iterator(); //récupérer itérateur
while (it.hasNext()) {  // tant qu'il y a un elt
  E p = it.next();  // lire élément suivant
  ...               // traiter l'élément
}
```




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

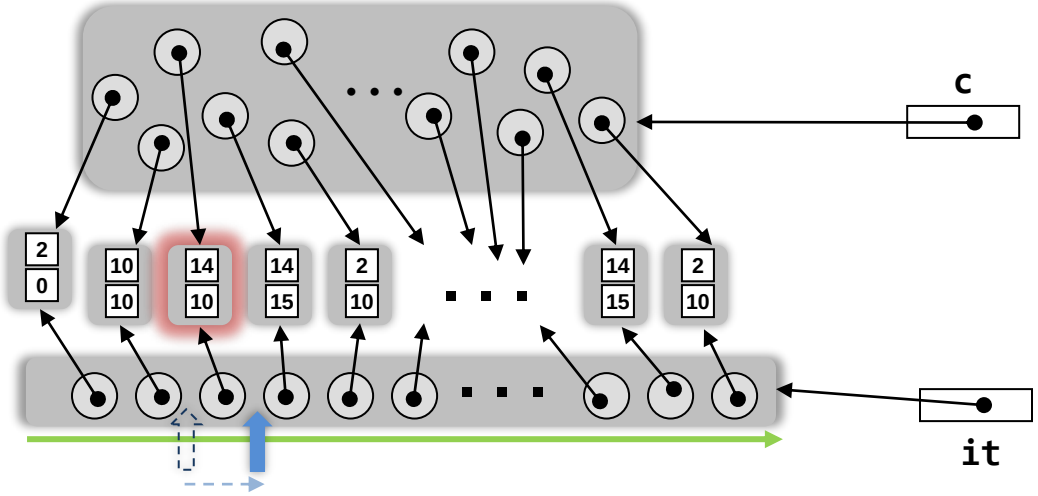
```
while (it.hasNext()) {
  Point p = it.next();
  System.out.println(p.distance());
}
```

```
2
14,1421
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Parcours séquentiel de tous les éléments de la collection

```
Iterator<E> it = c.iterator(); //récupérer itérateur
while (it.hasNext()) {  // tant qu'il y a un elt
  E p = it.next();  // lire élément suivant
  ...               // traiter l'élément
}
```




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

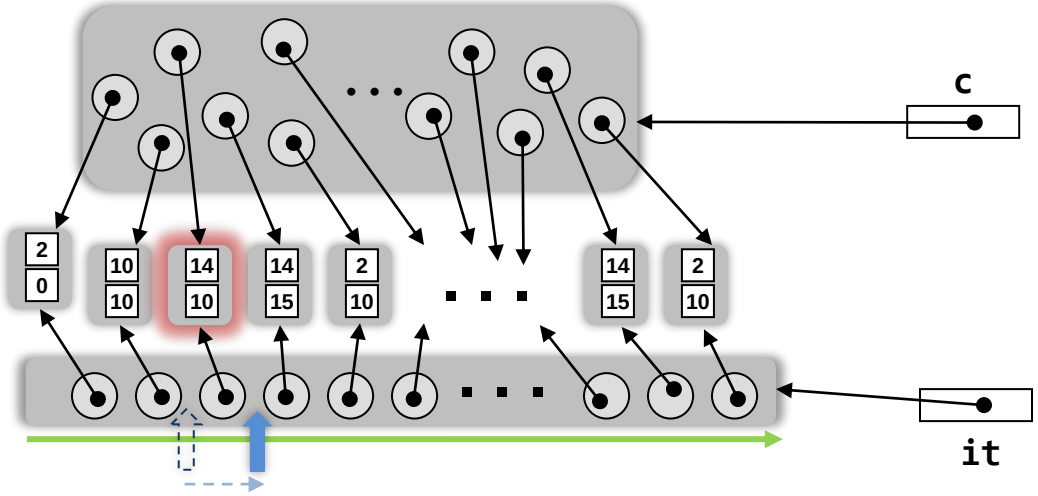
```
while (it.hasNext()) {
  Point p = it.next();
  System.out.println(p.distance());
}
```

```
2
14,1421
...
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Parcours séquentiel de tous les éléments de la collection

```
Iterator<E> it = c.iterator(); //récupérer itérateur
while (it.hasNext()) {  // tant qu'il y a un elt
  E p = it.next();  // lire élément suivant
  ...               // traiter l'élément
}
```




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
```

```
for (Point p : c) {
  System.out.println(p.distance());
}
```

```
2
14,1421
...
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Parcours séquentiel de tous les éléments de la collection

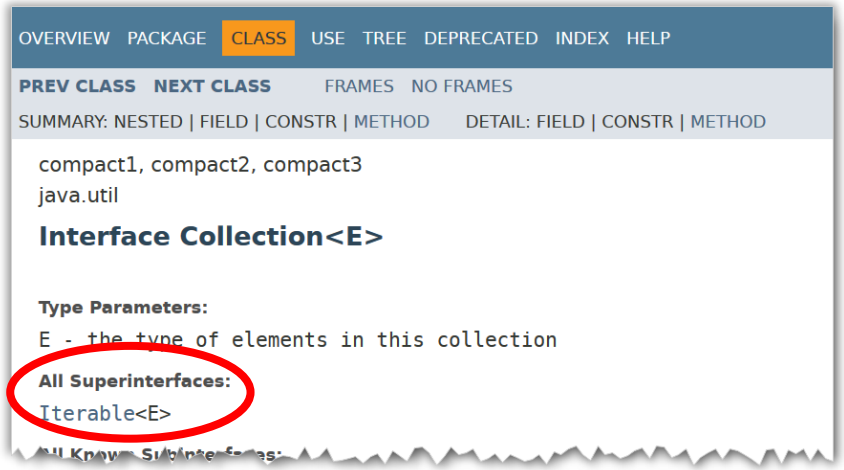
```
Iterator<E> it = c.iterator(); //récupérer itérateur
while (it.hasNext()) {  // tant qu'il y a un elt
  E p = it.next();  // lire élément suivant
  ...               // traiter l'élément
}
```

Écriture simplifiée pour parcours itératif : boucle **for-each**

```
for (E e: c) {
   traiter E
}
```

Interface Iterator

Interface `Collection<E>` étend l'interface `Iterable<E>` définie dans le package `java.lang`



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

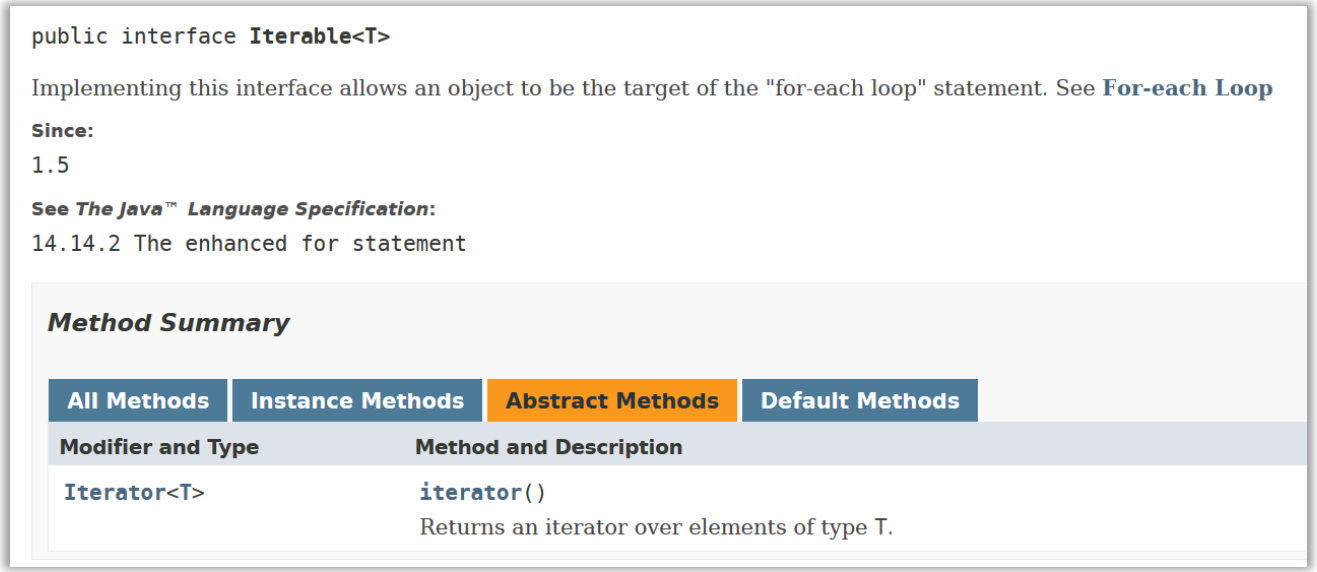
Interface Collection<E>

Type Parameters:

E - the type of elements in this collection

All Superinterfaces:

Iterable<E>



`public interface Iterable<T>`

Implementing this interface allows an object to be the target of the "for-each loop" statement. See **For-each Loop**

Since:

1.5

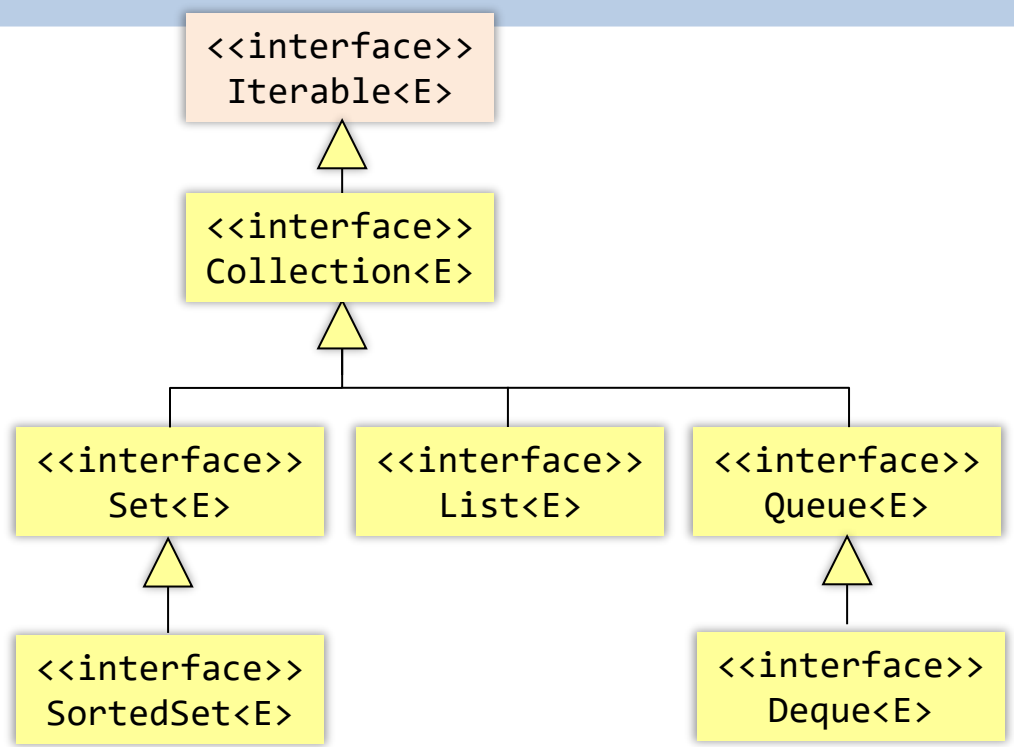
See *The Java™ Language Specification*:

14.14.2 The enhanced for statement

Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
`Iterator<T>`	`iterator()` Returns an iterator over elements of type T.

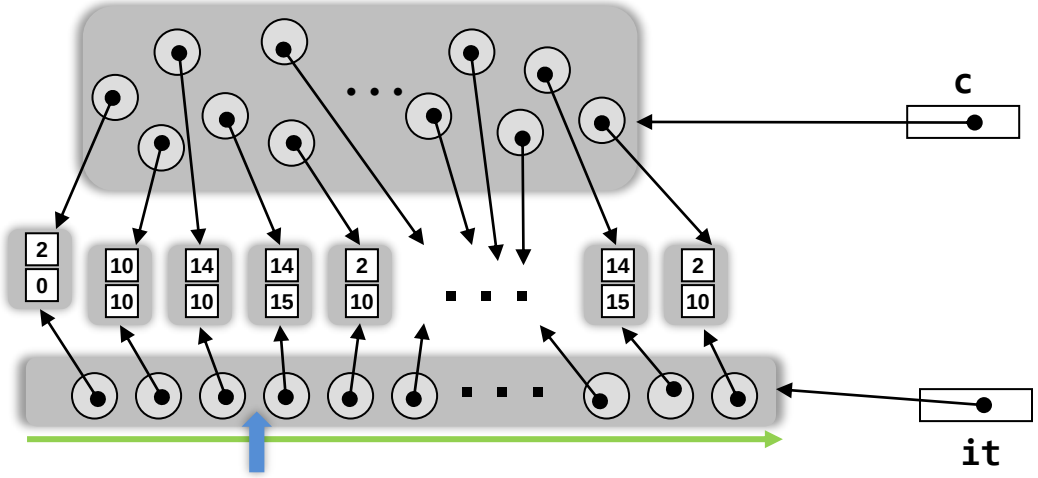


Parcours séquentiel de tous les éléments de la collection

```
Iterator<E> it = c.iterator(); //récupérer itérateur
while (it.hasNext()) {  // tant qu'il y a un elt
  E p = it.next();  // lire élément suivant
  ...               // traiter l'élément
}
```

Écriture simplifiée pour parcours itératif : boucle **for-each**

```
for (E e: c) {
   traiter E
}
```

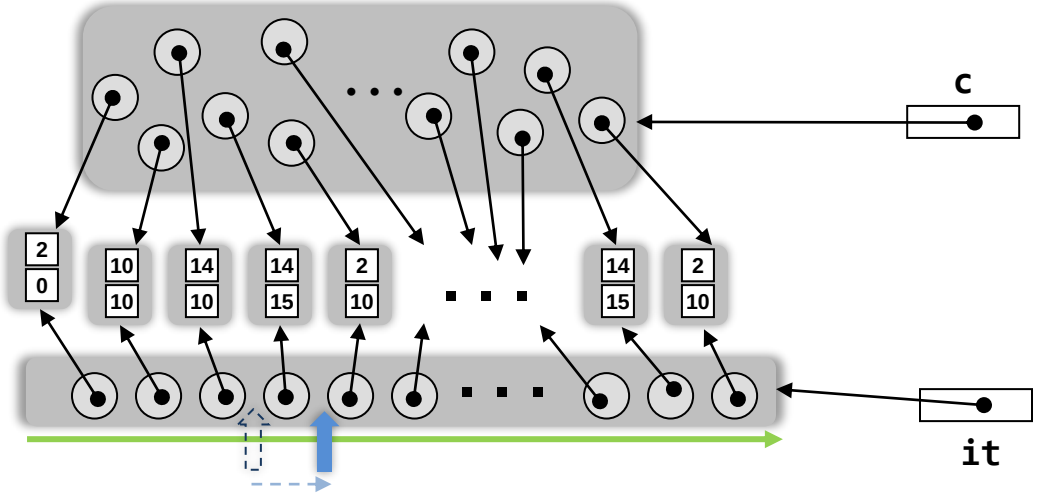



```
Collection<Point> c = new ...
c.add(new Point(2,0);
...

Iterator<Point> it = c.iterator();
...
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();      // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Permet de supprimer le dernier élément parcouru:

L'élément retiré de la collection est l'élément renvoyé par le dernier appel à `next()`




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

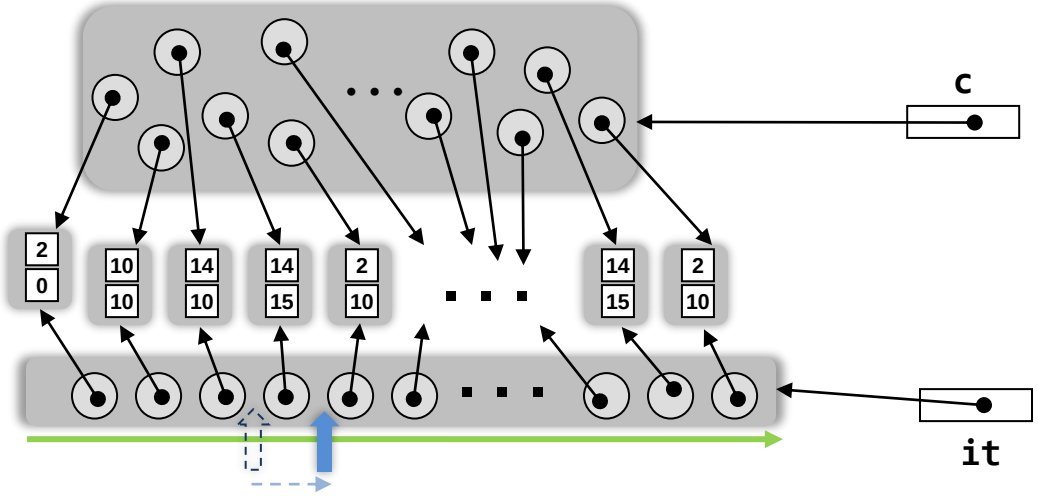
```
Iterator<Point> it = c.iterator();
 ...
```

```
it.next();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();      // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Permet de supprimer le dernier élément parcouru:

L'élément retiré de la collection est l'élément renvoyé par le dernier appel à `next()`




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

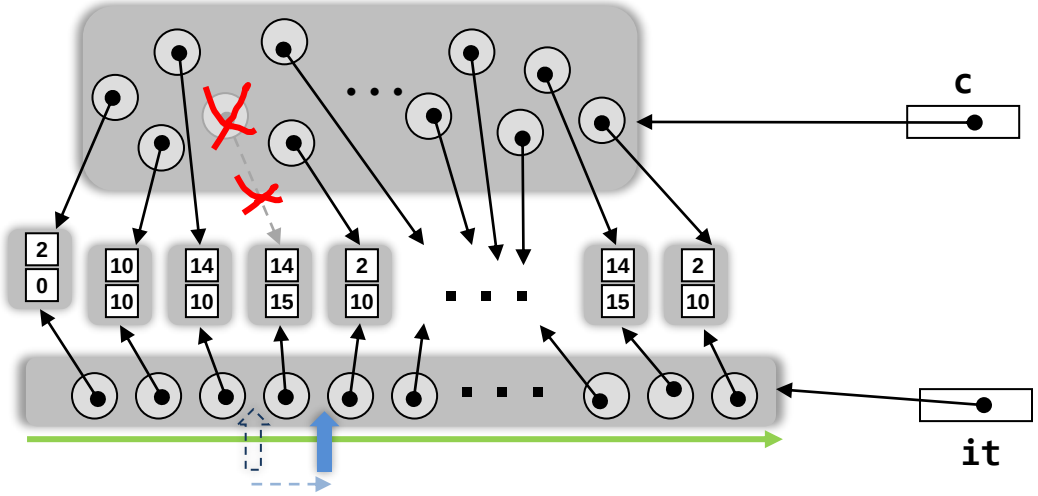
```
Iterator<Point> it = c.iterator();
 ...
```

```
it.next();
it.remove();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Permet de supprimer le dernier élément parcouru:

L'élément retiré de la collection est l'élément renvoyé par le dernier appel à `next()`




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

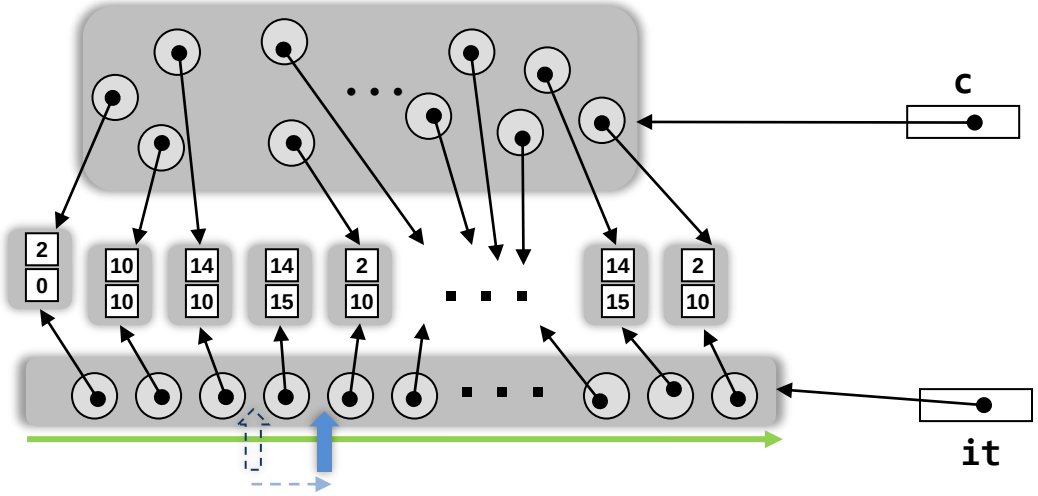
```
Iterator<Point> it = c.iterator();
 ...
```

```
it.next();
it.remove();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();    // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Permet de supprimer le dernier élément parcouru:

L'élément retiré de la collection est l'élément renvoyé par le dernier appel à `next()`




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
 ...
```

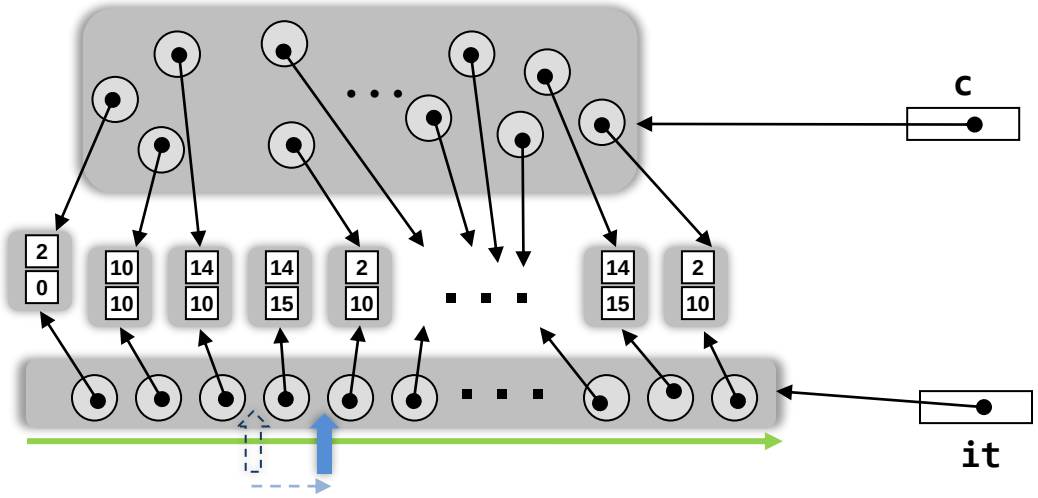
```
it.next();
it.remove();
```

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();      // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Permet de supprimer le dernier élément parcouru:

L'élément retiré de la collection est l'élément renvoyé par le dernier appel à `next()`

Impossible d'appeler remove() sans un appel correspondant à `next()`




```
Collection<Point> c = new ...
c.add(new Point(2,0);
...
```

```
Iterator<Point> it = c.iterator();
...
```

```
it.next();
it.remove();
it.remove();
```

IllegalStateException

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();      // Optional
    default void forEachRemaining(Consumer<? super E> action); // 1.8+
}
```

Permet de supprimer le dernier élément parcouru:

L'élément retiré de la collection est l'élément renvoyé par le dernier appel à `next()`

Impossible d'appeler remove() sans un appel correspondant à `next()`

Collections Interface List

collection simple et ordonnée d'éléments qui autorise les doublons. La liste étant ordonnée, un élément peut être accédé à partir de son **index**.

Interface `List<E>`

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**add**(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns true if this list contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this list contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this list for equality.
E	**get**(int index) Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns true if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	**remove**(int index) Removes the element at the specified position in this list (optional operation).
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	**size**() Returns the number of elements in this list.
List<E>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

Collections Interface List

Interface `List<E>`

collection simple et ordonnée d'éléments qui autorise les doublons. La liste étant ordonnée, un élément peut être accédé à partir de son **index**.

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**add**(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns true if this list contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this list contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this list for equality.
E	**get**(int index) Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns true if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	**remove**(int index) Removes the element at the specified position in this list (optional operation).
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	**size**() Returns the number of elements in this list.
List<E>	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

Méthodes spécifiques aux listes

- parcours d'une liste

public interface **ListIterator<E>**
extends Iterator<E>

An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list. A ListIterator has no current element; its *cursor position* always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next(). An iterator for a list of length n has n+1 possible cursor positions, as illustrated by the carets (^) below:

```
                      Element(0)   Element(1)   Element(2)   ... Element(n-1)
cursor positions:  ^           ^            ^            ^                  ^
```

Note that the remove() and set(Object) methods are *not* defined in terms of the cursor position; they are defined to operate on the last element returned by a call to next() or previous().

This interface is a member of the Java Collections Framework.

Since:
1.2

See Also:
Collection, List, Iterator, Enumeration, List.listIterator()

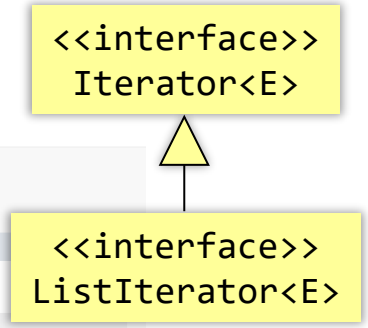
Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
void	**add**(E e) Inserts the specified element into the list (optional operation).
boolean	**hasNext**() Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean	**hasPrevious**() Returns true if this list iterator has more elements when traversing the list in the reverse direction.
E	**next**() Returns the next element in the list and advances the cursor position.
int	**nextIndex**() Returns the index of the element that would be returned by a subsequent call to **next()**.
E	**previous**() Returns the previous element in the list and moves the cursor position backwards.
int	**previousIndex**() Returns the index of the element that would be returned by a subsequent call to **previous()**.
void	**remove**() Removes from the list the last element that was returned by **next()** or **previous()** (optional operation).
void	**set**(E e) Replaces the last element returned by **next()** or **previous()** with the specified element (optional operation).

Methods inherited from interface java.util.Iterator

forEachRemaining




```java
package fr.im2ag.m2cci.listes;

import fr.im2ag.m2cci.geom.Point;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;


public class TestListes {

    public static void main(String[] args) {

        List<Point> listeDePoints = new ArrayList<>();

        for (int i = 0; i < 10; i++) {
            listeDePoints.add(new Point(Math.round(Math.random() * 28),
                    Math.round(Math.random() * 14)));
        }

        Iterator<Point> it = listeDePoints.iterator();

        while (it.hasNext()) {
            Point pt = it.next();
            System.out.println(pt);
        }

        for (Point pt : listeDePoints) {
            System.out.println(pt);
        }

        listeDePoints.forEach(pt -> {
            System.out.println(pt);
        });

        for (int i = 0; i < listeDePoints.size(); i++) {
            Point pt = listeDePoints.get(i);
            System.out.println(pt);
        }

        ListIterator<Point> listIt = listeDePoints.listIterator();
        while (listIt.hasNext()) {
            Point pt = listIt.next();
            System.out.println(listIt.previousIndex() + " : " + pt);
        }
    }

}
```

Interface `Set<E>`

compact1, compact2, compact3
java.util

Interface Set<E>

ne définit pas de nouvelles méthodes,
Ajoute simplement des restrictions sur les définitions des opérations en interdisant la duplication des éléments

Type Parameters:

E - the type of elements maintain[illegible] this set

All SuperInterfaces:

Collection<E>, Iterable<E>

All Known Subinterfaces:

[illegible]

All Methods	Instance Methods	Abstract Methods	Default Methods

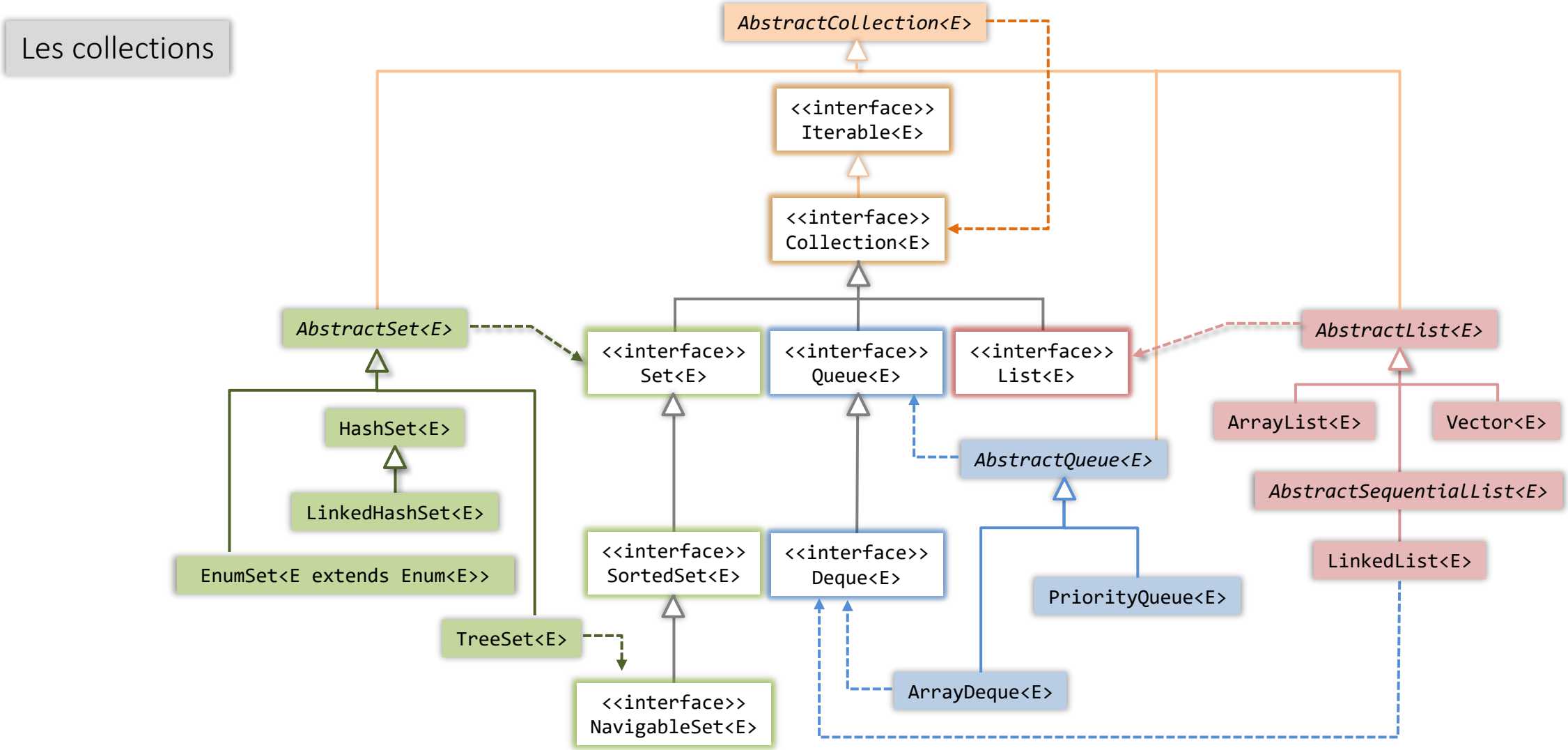
Modifier and Type	Method and Description
boolean	**add(E e)** Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll(Collection<? extends E> c)** Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear()** Removes all of the elements from this set (optional operation).
boolean	**contains(Object o)** Returns true if this set contains the specified element.
boolean	**containsAll(Collection<?> c)** Returns true if this set contains all of the elements of the specified collection.
boolean	**equals(Object o)** Compares the specified object with this set for equality.

[illegible]

int	**hashCode()** Returns the hash code value for this set.
boolean	**isEmpty()** Returns true if this set contains no elements.
Iterator<E>	**iterator()** Returns an iterator over the elements in this set.
boolean	**remove(Object o)** Removes the specified element from this set if it is present (optional operation).
boolean	**removeAll(Collection<?> c)** Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll(Collection<?> c)** Retains only the elements in this set that are contained in the specified collection (optional operation).
int	**size()** Returns the number of elements in this set (its cardinality).
default Spliterator<E>	**spliterator()** Creates a Spliterator over the elements in this set.
Object[]	**toArray()** Returns an array containing all of the elements in this set.
<T> T[]	**toArray(T[] a)** Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

Collections : implémentations

- `java.util` fourni des classes d'implémentation des types abstraits
  - *ces classes se situent dans une hiérarchie de classes abstraites : factorisation de code, réutilisation pour des implémentations spécifiques*



Collections : implémentations de List<E>

- `ArrayList<E>`
  - implémente toutes les méthodes de l'interface List
  - autorise l'ajout d'éléments null
  - utilise un tableau pour stocker ses éléments
  - adaptation automatique de taille du tableau au nombre d'éléments de la collection → lorsque le tableau est plein nécessité d'instancier un nouveau tableau et de recopier les éléments de la collection dans celui-ci.

`ArrayList()` Crée une instance vide de la collection avec une capacité initiale de 10

`ArrayList(Collection<? Extends E> c)` Crée une instance contenant les éléments de la collection fournie en paramètre dans l'ordre obtenu en utilisant son iterator

`ArrayList(int initialCapacity)` Crée une instance vide de la collection avec la capacité initiale fournie en paramètre

`public void ensureCapacity(int)` Augmente capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre

`public void trimToSize()` Ajuste capacité du tableau sur sa taille actuelle

Opérations d'accès rapides

```
public E get(int index)
public E set(int index, E element)
```

Opérations d'insertion/suppression peuvent être coûteuses : réallocation de tableau, décalages d'éléments

```
public boolean add(E e)
public void add(int index, E element)
public boolean remove(Object o)
public E remove(int index)
protected void removeRange(int fromIndex, int toIndex)
```

Collections : implémentations de List<E>

- `LinkedList<E>`
  - implémente toutes les méthodes de l'interface List
  - autorise l'ajout d'éléments null
  - utilise une liste doublement chaînée dans laquelle les éléments de la collection sont reliés par des pointeurs
    - pas besoin d 'être redimensionnée quelque soit le nombre d'éléments qu'elle contient

`LinkedList()` Crée une instance vide de la collection

`LinkedList(Collection<? Extends E> c)` Crée une instance contenant les éléments de la collection fournie en paramètre dans l'ordre obtenu en utilisant son iterator

Opérations utilisant index nécessitent parcours de la liste

Opérations d'insertion/suppression plus efficaces que pour `ArrayList`

Opérations d'accès direct au premier et dernier élément permettent de gerer la liste comme une pile ou une file

```
public E getFirst()
public E getLast()
public void addFirst(E element)
public void addLast(E element)
public E removeFirst()
public E removeLast()
```

Les classes « enveloppes »

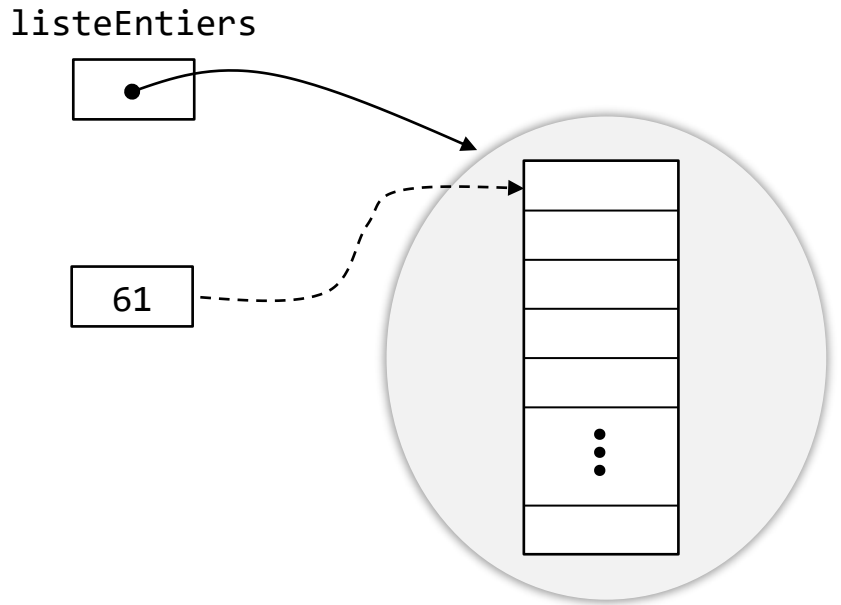
- Le type pour les listes est un type générique (paramétré) `List<E>`
  - *E un type objet : classe ou interface*
- Ajout d'un élément à une liste: `public boolean add(E o)`
  - *o est une référence vers un objet de type E*
- Quid des types de primitifs `byte, short, int, long, double, float, char, boolean` ?


```
error: unexpected type
          List<int> listeEntiers = new ArrayList<>();
  required: reference
  found:    int
```

erreur de compilation

```
List<int> listeEntiers = new ArrayList<>();
listeEntiers.add(61);
```

Les types primitifs ne sont pas des objets !



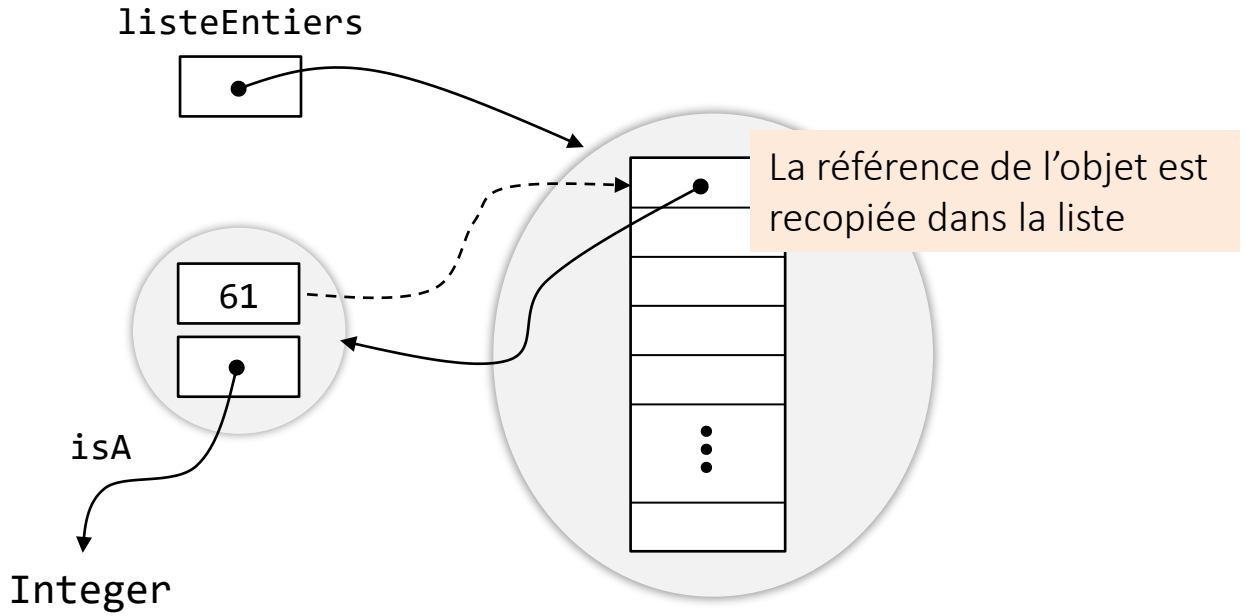
Les classes « enveloppes »

- Pour permettre l'utilisation de valeurs de types primitif en tant qu'objet on «encapsule» la valeur dans un objet à l'aide d'une classe « enveloppe » (« *wrapper class* »)

Type objet

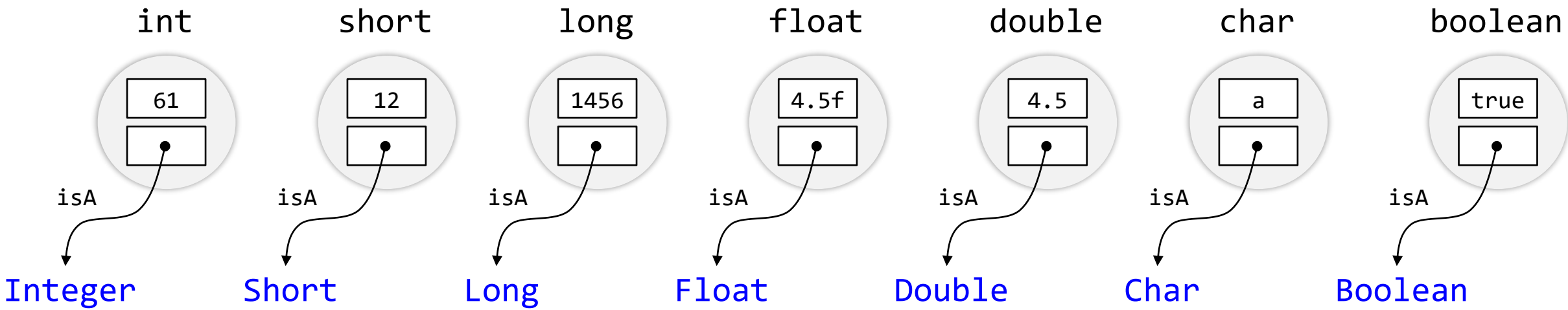
```
List<Integer> listeEntiers = new ArrayList<>();

listeEntiers.add(new Integer(61));
```

Encapsule la valeur dans une instance de sa classe « enveloppe »



Les classes « enveloppes »

- Pour chacun des types de base il existe une classe enveloppe dont le nom correspond au nom du type de base (sauf pour int) mais avec une MAJUSCULE



- Principaux services des classe enveloppes

Classe enveloppe	*Création d'une instance (constructeur)*	*Accès à la valeur*	*Méthodes statiques pour conversions de chaînes vers les valeurs du type de base*

`Integer`	`Integer(int value)`
	`Integer intObj = new Integer(45);`
		`int integerValue()`
		`int i = intObj.integerValue();`
			`Integer valueOf(String s)`
			`int parseInt(String s)`
			`Integer intObj = Integer.valueOf("45");`

`Double`	`Double(double value)`	`double doubleValue()`	`Double valueOf(String s)` `double parseDouble(String s)`
`Boolean`	`Boolean(boolean value)`	`boolean booleanValue()`	`Boolean valueOf(String s)`
…			

Les objets définis par les classes enveloppes sont immutables (comme les **`String`**) :
leur valeur ne peut être modifiée (pas de méthodes **`set`…**)

Wrappers objects / valeurs de type simple

- Avantages et inconvénients des « wrapper » objects

Avantages

- C’est un objet Java
- Peut être mis à `null`
- Peut être référencé plusieurs fois
- Peut être stocké dans une collection

Inconvénients

- Espace mémoire supérieur au type simple
- Consomme du temps processeur lors d’une instanciation mémoire par le constructeur
- Peu pratique pour le calcul
- Ne peut changer de valeur sans ré-allocation (pas de set)

- Un autre ajout à java dans la version 1.5 : l'autoboxing
- La possibilité de passer automatiquement d'un type primitif (`int`, `float`, `double`, `char`, …) vers le type objet correspondant (`Integer`, `Float`, `Double`, `Char`, …) et inversement.
  - *Une technique déjà employée dans C# (bien fait, ils n'avaient qu'à pas copier !)*

```
Double d = 3.14; // autoboxing                    Double d = new Double(3.14);
Integer i = 14;                                   Integer i = new Integer(14);

double d1 = d;   // unboxing                       double d1 = d.doubleValue();
double d2 = i;                                    double d2 = i.intValue();

List<Double> listeDoubles = new ArrayList<>();    List<Double> listeDoubles = new ArrayList<>();
listeDoubles.add(3.14);  // autoboxing            listeDoubles.add(new Double(3.14));
double d3 = listeDoubles.get(i); // unboxing      double d3 = listeDoubles.get(i).doubleValue();
```

Interface Map (tables associatives)

Interface `Map<K,V>`

Tables associatives ou dictionnaires (implémentée sous la forme d'associations de paires de type **clés (Key)/valeurs (Values)**). La **clé** doit être unique. Par contre une même **valeur** peut être associée à plusieurs clés différentes.

Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
void	**clear()** Removes all of the mappings from this map (optional operation).
boolean	**containsKey(Object key)** Returns `true` if this map contains a mapping for the specified key.
boolean	**containsValue(Object value)** Returns `true` if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	**entrySet()** Returns a **Set** view of the mappings contained in this map.
boolean	Compares the specified object with this map for equality.
V	**get(Object key)** Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
int	**hashCode()** Returns the hash code value for this map.
boolean	**isEmpty()** Returns `true` if this map contains no key-value mappings.
Set<K>	**keySet()** Returns a **Set** view of the keys contained in this map.
V	**put(K key, V value)** Associates the specified value with the specified key in this map (optional operation).
void	**putAll(Map<? extends K,? extends V> m)** Copies all of the mappings from the specified map to this map (optional operation).
V	**remove(Object key)** Removes the mapping for a key from this map if it is present (optional operation).
int	**size()** Returns the number of key-value mappings in this map.
Collection<V>	**values()** Returns a **Collection** view of the values contained in this map.

`Map.Entry` définit un couple clé valeur

On accède directement à la valeur à partir de la clé

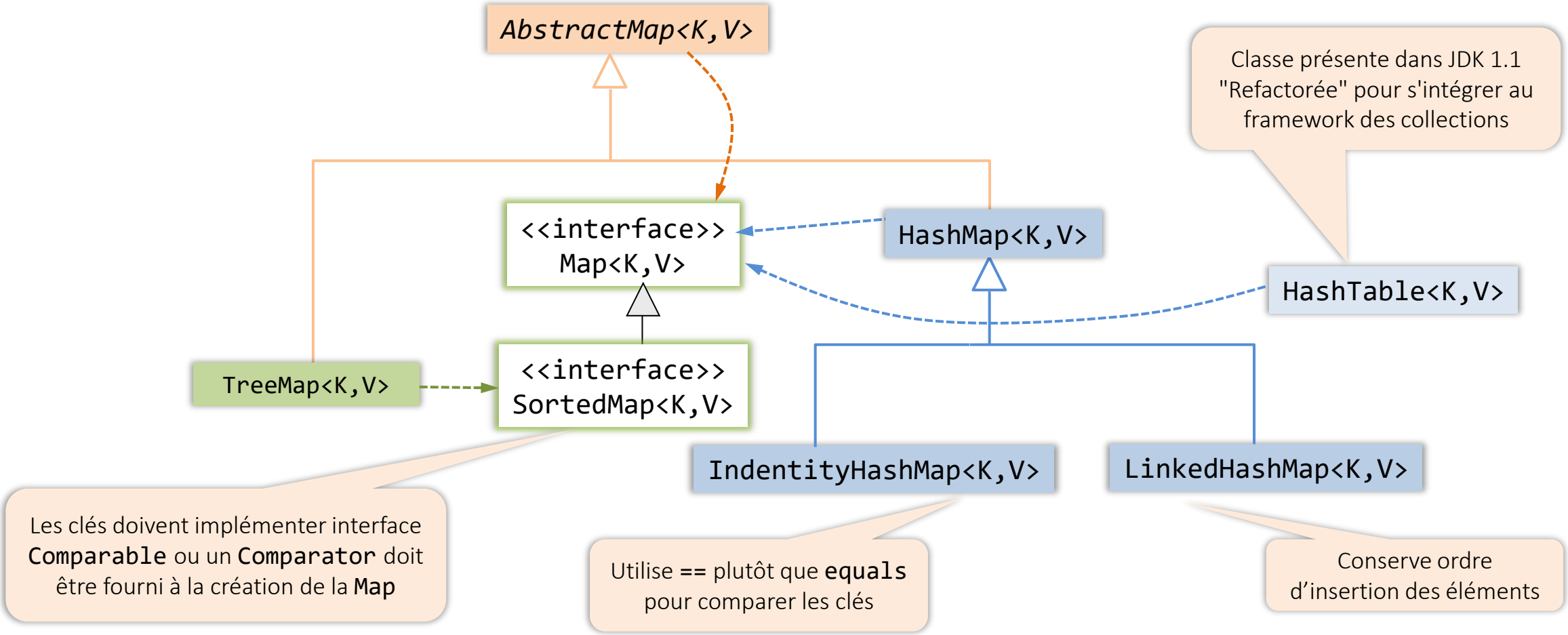
Les clés forment un ensemble (chaque clé est unique)

Associe une valeur à une clé

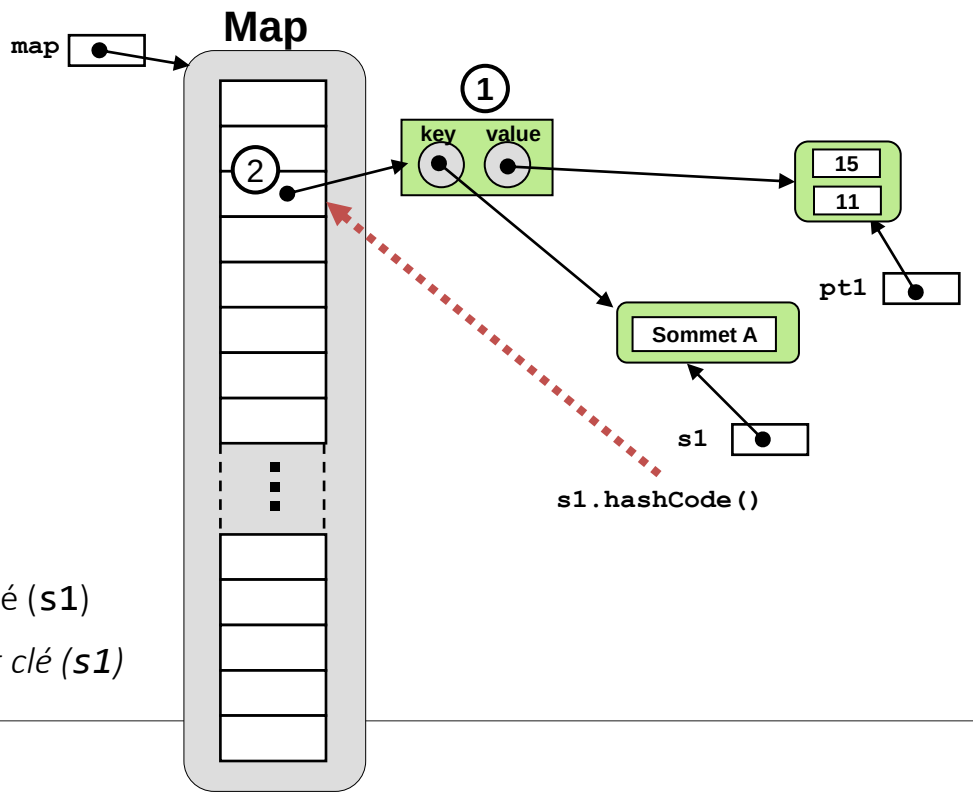
Les valeurs constituent une collection

Maps (Tables associatives) : implémentations

Les Maps



Map : abstraction de structure de données avec adressage dispersé (hashCoding)



La Map peut être vue comme une table de couples clé/valeur (`Map.Entry`)

```
Map<String,Point> map = new HashMap<>();
```

Ajouter une entrée à la Map

```
String s1 = new String("Sommet A");
```

```
Point pt1 = new Point(15,11);
```

```
map.put(s1,pt1);
```

1) Créer un objet `Map.Entry` référençant s1 et pt1

2) Ranger cette `Map.Entry` dans la table à une position qui dépend de la clé (`s1`)

*Cette position est calculée en envoyant le message **`hashCode()`** à l'objet clé (**`s1`**)*

Méthode **`hashCode`** de la classe **`String`**

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html#hashCode()

Tout objet possède une méthode **`hashCode()`**

public int hashCode()

Returns a hash code for this string. The hash code for a String object is computed as

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

using int arithmetic, where s[i] is the *i*th character of the string, n is the length of the string, and ^ indicates exponentiation. (The hash value of the empty string is zero.)

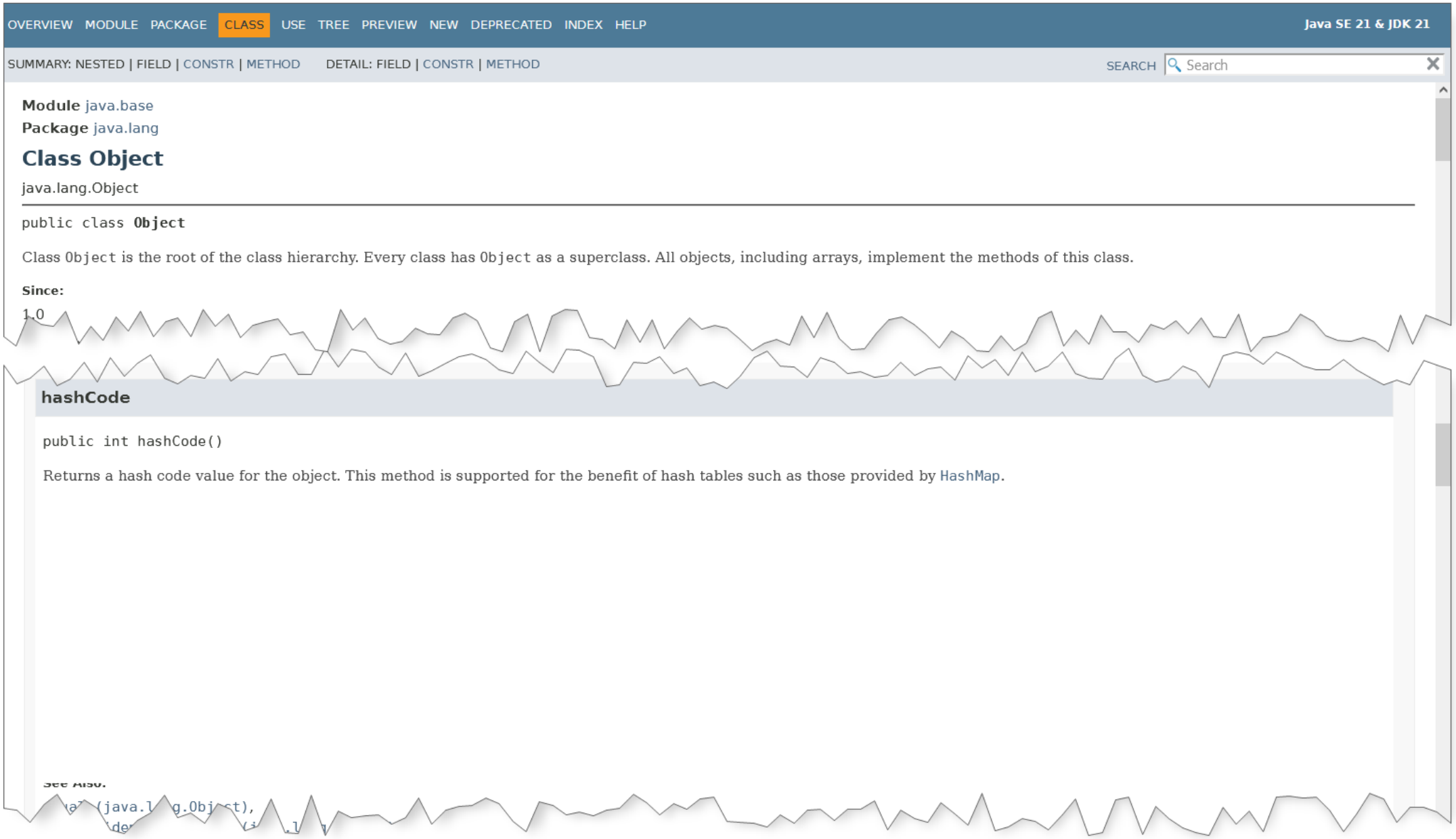
Overrides:

hashCode in class Object

Ret

Map : insertion (à propos de hashCode)

Contrat de la méthode hashCode



OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP Java SE 21 & JDK 21

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEARCH Search

Module java.base
Package java.lang

Class Object

java.lang.Object

```
public class Object
```

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Since:

1.0

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.

See Also:

(java.lang.Object),

Map : insertion (à propos de hashCode)

Contrat de la méthode hashCode

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP Java SE 21 & JDK 21

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEARCH Search

Module java.base
Package java.lang

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Object.html#hashCode()

Class Object

java.lang.Object

```
public class Object
```

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:

1.0

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` 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` 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.

Implementation Requirements:

As far as is reasonably practical, the `hashCode` method defined by class `Object` returns distinct integers for distinct objects.

Returns:

a hash code value for this object.

See Also:

(java.l g.Obj ct),

Map : insertion (à propos de hashCode)

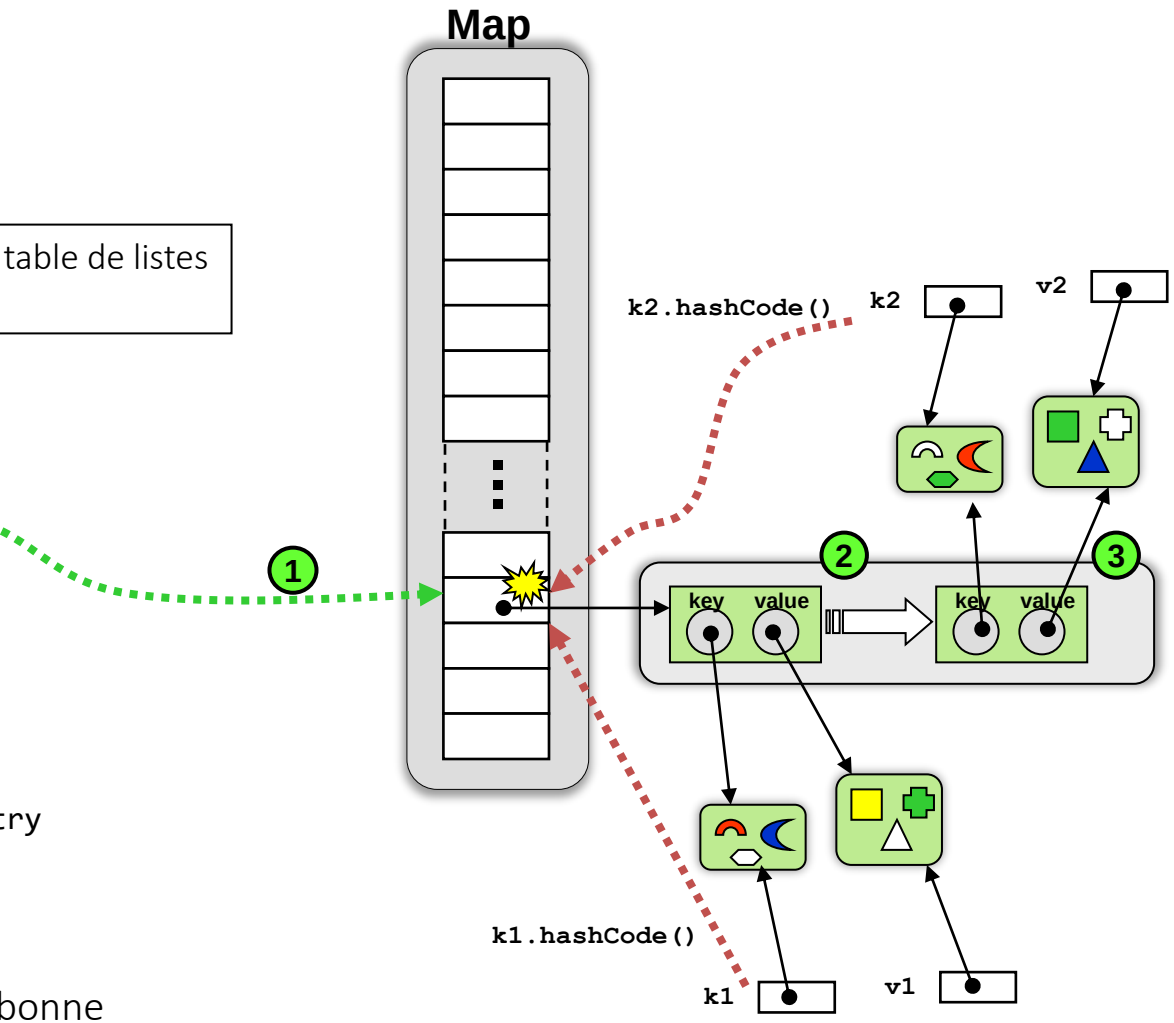
```
map.put(k1,v1);
map.put(k2,v2);
```

Collision
`k2.hashCode() == k1.hashCode()`

En fait la `Map` n'est pas une table de `Map.Entry` mais une table de listes de `Map.Entry`

Retrouver une valeur stockée dans la `Map`

```
map.get(k2);
```



1. `k2.hashCode()` → position de la clé dans la `Map`
2. Parcours séquentiel de la liste `Map.Entry` à la recherche d'une clé `k` telle `que k.equals(k2)`
3. Si `k` est trouvée, retour de la valeur associée dans la `Map.entry`
   Sinon retour de `null`

Une bonne fonction de hashCode doit assurer une bonne dispersion des clés en minimisant les collisions

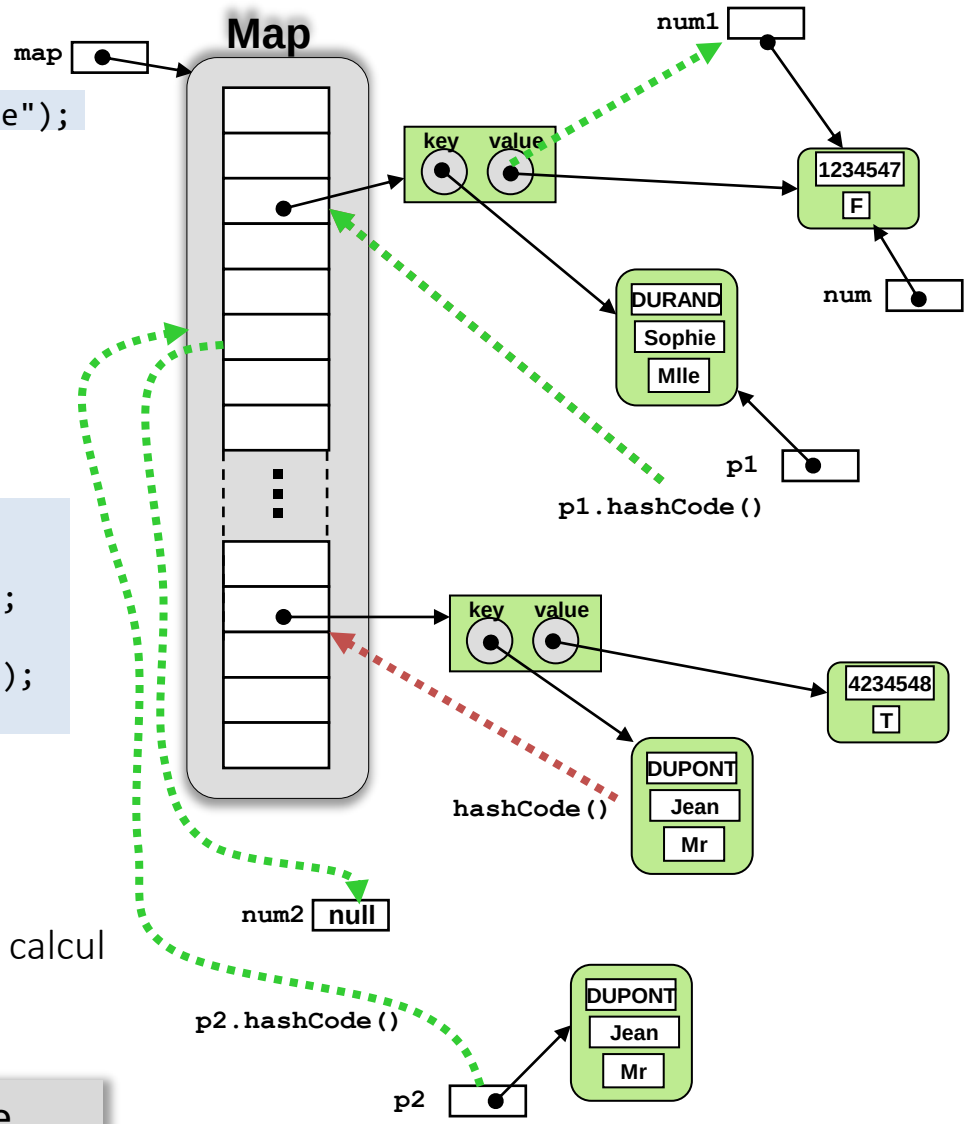
Map : insertion (à propos de hashCode)

```
Map<Personne,NumTel> map = new HashMap<>();
Personne p1 = new Personne("DURAND","Sophie","Mlle");
NumTel num = new NumTel("1234547",'F');
map.put(p1,num);
NumTel num1 = map.get(p1);
System.out.println(num1);
```

→ `1234547 (F)`

```
map.put(new Personne("DUPONT","Jean","Mr"),
        new NumTel("4234548",'D'));
String nom = LectureClavier.lireChaine("Nom : ");
String prenom =
          LectureClavier.lireChaine("Prénom : ");
String civilite =
         LectureClavier.lireChaine("Civilité : ");
Personne p2 = new Personne(nom,prenom,civilite);
NumTel num2 = map.get(p2);
System.out.println(num2);
```

→ `null` ???



`p2` est un objet distinct de la clé (même si il représente la même personne) et la fonction `hashCode()` héritée de `Object` base son calcul sur la référence (adresse mémoire de cet objet)

→ il faut redéfinir `hashCode()` dans `Personne`

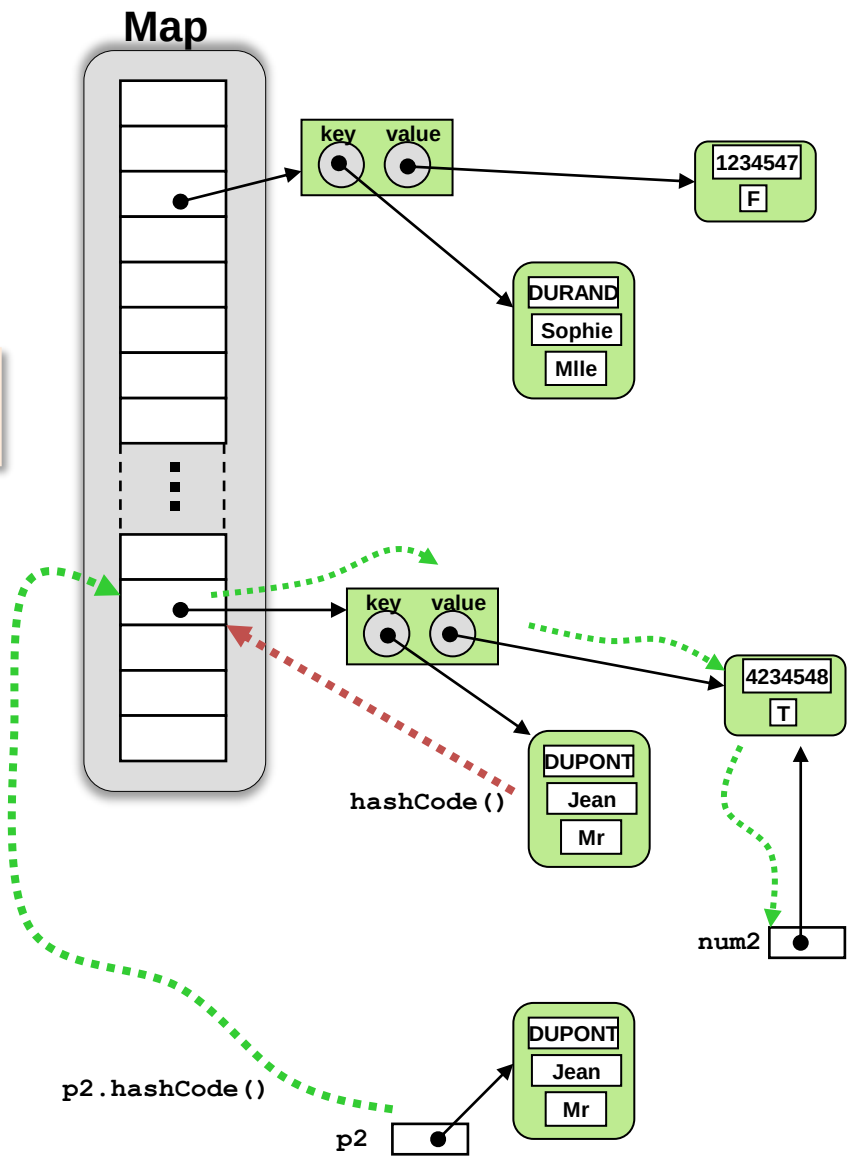
Map : insertion (à propos de hashCode)

```
public class Personne {
  ...
  public boolean equals(Object o) {
     if (! (o instanceof Personne) )
       return false;

     Personne p = (Personne) o;
     return this.civilite == p.civilite &&
            this.nom.equals(p.nom) &&
            this.prenom.equals(p.prenom);
  }


  public int hashCode() {
      String nomPlusPrenom = nom + prenom;
      return nomPlusPrenom.hashCode() + civilite_;
  }

}
```

ATTENTION : les **String** sont des objets, bien penser à utiliser **equals** et non pas == pour comparer des chaînes

```
Personne p2 = new Personne("DUPONT","Jean","Mr");
NumTel num2 = map.get(p2);
System.out.println(num2);    →  4234548 (T)
```

hashCode() doit TOUJOURS être redéfinie en cohérence avec **equals()**



Parcours des éléments d'une Map

- Parcours de clés

`Set<K>`	**`keySet()`** Returns a `Set` view of the keys contained in this map.

- Parcours de valeurs

`Collection<V>`	**`values()`** Returns a `Collection` view of the values contained in this map.

- Parcours des couples clé/valeur

`Set<Map.Entry<K,V>>`	**`entrySet()`** Returns a `Set` view of the mappings contained in this map.

`void`	**`forEach(BiConsumer<? super K,? super V> action)`** Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.

- Ordre de tri défini par deux interfaces

java.lang

Interface Comparable<T>

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
int	`compareTo(T o)` Compares this object with the specified object for order.

- Doit être implémentée par les classes d'objets pour définir un ordre naturel utilisé par le tri d'une collection.

java.util

Interface Comparator<T>

Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
int	`compare(T o1, T o2)` Compares its two arguments for order.
boolean	`equals(Object obj)` Indicates whether some other object is "equal to" this comparator.

- Utilisée pour permettre le tri d'objets qui n'implémentent pas l'interface `Comparable` ou pour définir un ordre de tri différent de celui défini avec `Comparable`

- Utilisation de `Comparable<T>` et `Comparator<T>`
  - *La classe* `Collections` *propose des implémentations d'algorithme de tri pour les listes*

`static <T extends Comparable<? super T>> void`	**`sort(List<T> list)`** Sorts the specified list into ascending order, according to the **natural ordering** of its elements.
`static <T> void`	**`sort(List<T> list, Comparator<? super T> c)`** Sorts the specified list according to the order induced by the specified comparator.

```
class Person {

    private int age;
    private String firstName;
    private String lastName;

    ...
}
```

```
class Student extends Person implements Comparable<Student> {

    private int studID;

    ...

    @Override public int compareTo(Student a) {
        return a.studId < b. studId;
    }
}
```

```
List<Student> students = getSomeStudents();

// Trier la liste par numéro d'étudiants
Collections.sort(students);

// Trier la liste par age
Collections.sort(students, new ByAgeAscending())
```

```
class ByAgeAscending implements Comparator<Person> {
    @Override public int compare(Person a, Person b) {
        return a.getAge() < b.getAge();
    }
}
```

- Tri
- Recherche min / max

static <T extends **Object** & **Comparable**<? super T>> T	**min**(**Collection**<? extends T> coll) Returns the minimum element of the given collection, according to the *natural ordering* of its elements.
static <T> T	**min**(**Collection**<? extends T> coll, **Comparator**<? super T> comp) Returns the minimum element of the given collection, according to the order induced by the specified comparator.

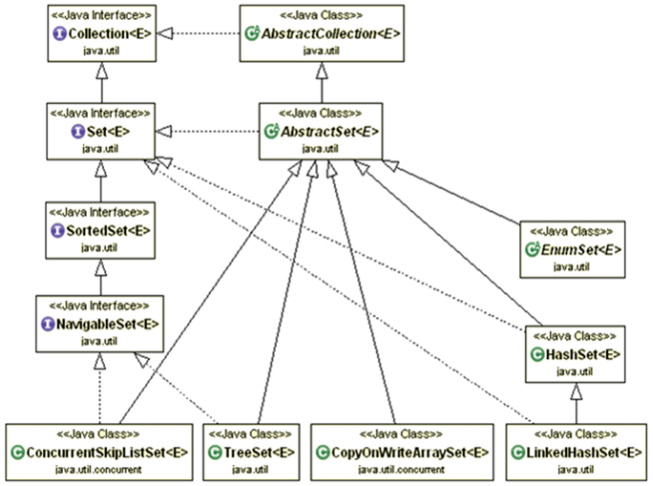
- Mélanger

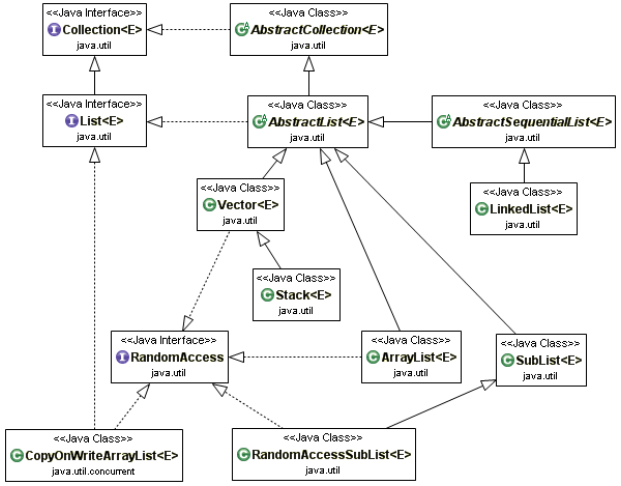
static void	**shuffle**(**List**<?> list) Randomly permutes the specified list using a default source of randomness.
static void	**shuffle**(**List**<?> list, **Random** rnd) Randomly permute the specified list using the specified source of randomness.

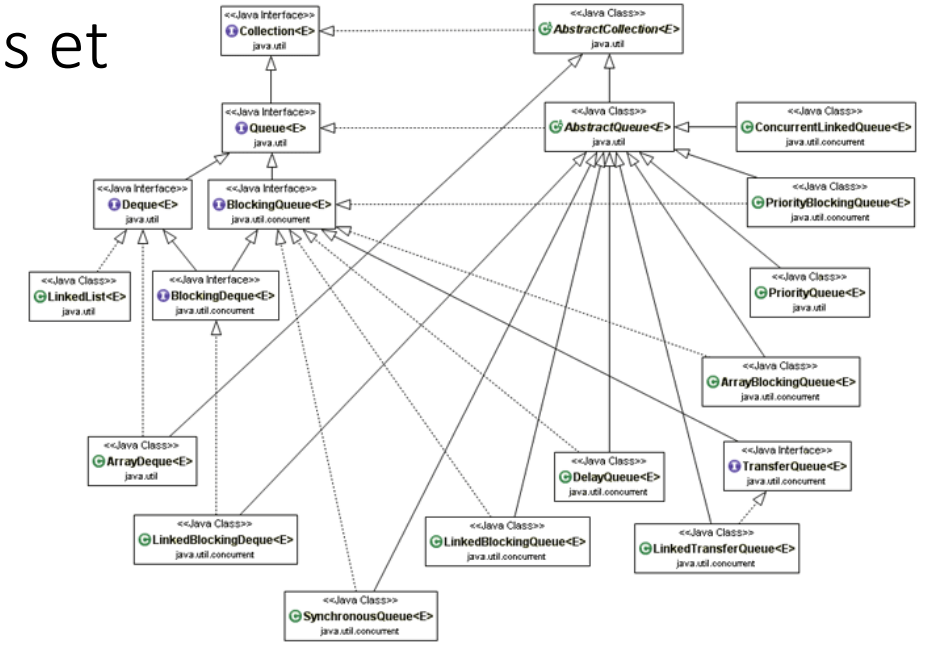
- …

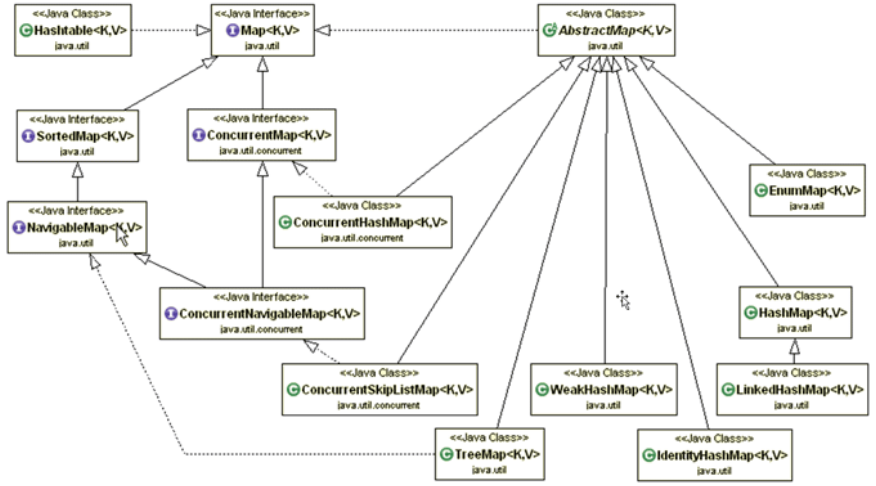
https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

- Ce cours a présenté l'essentiel pour les collections et les types de base
- Beaucoup d'autres fonctionnalités









- Pour en savoir plus et aller plus loin :

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