Les collections dans Java

Les collections

Collection :

- Modèle de structures de données
  - Abstraction d'un regroupement des données élémentaires
    - Réduit à quelques cas
  - Protocole de gestion
  - protocole d'accès
- Un cas classique d'abstraction générique et de mise en œuvre des concepts objets

Utilisation de collections pour

- Stocker, retrouver des données élémentaires dans un regroupement de données de nature similaire.

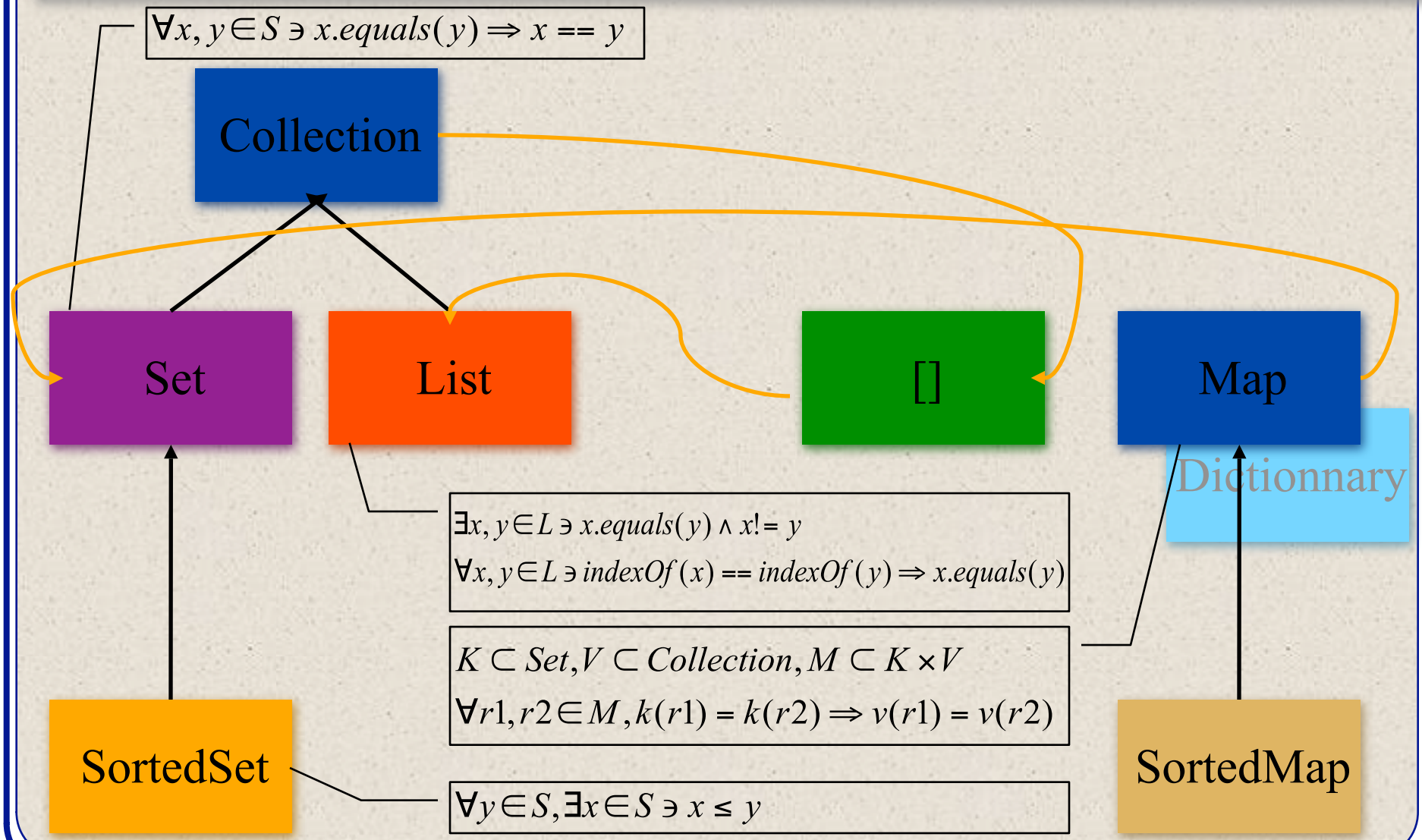
Exemples :

- un dossier de courrier : collection de mails
- un répertoire téléphonique : collection d'associations noms numéros de téléphone.
- …

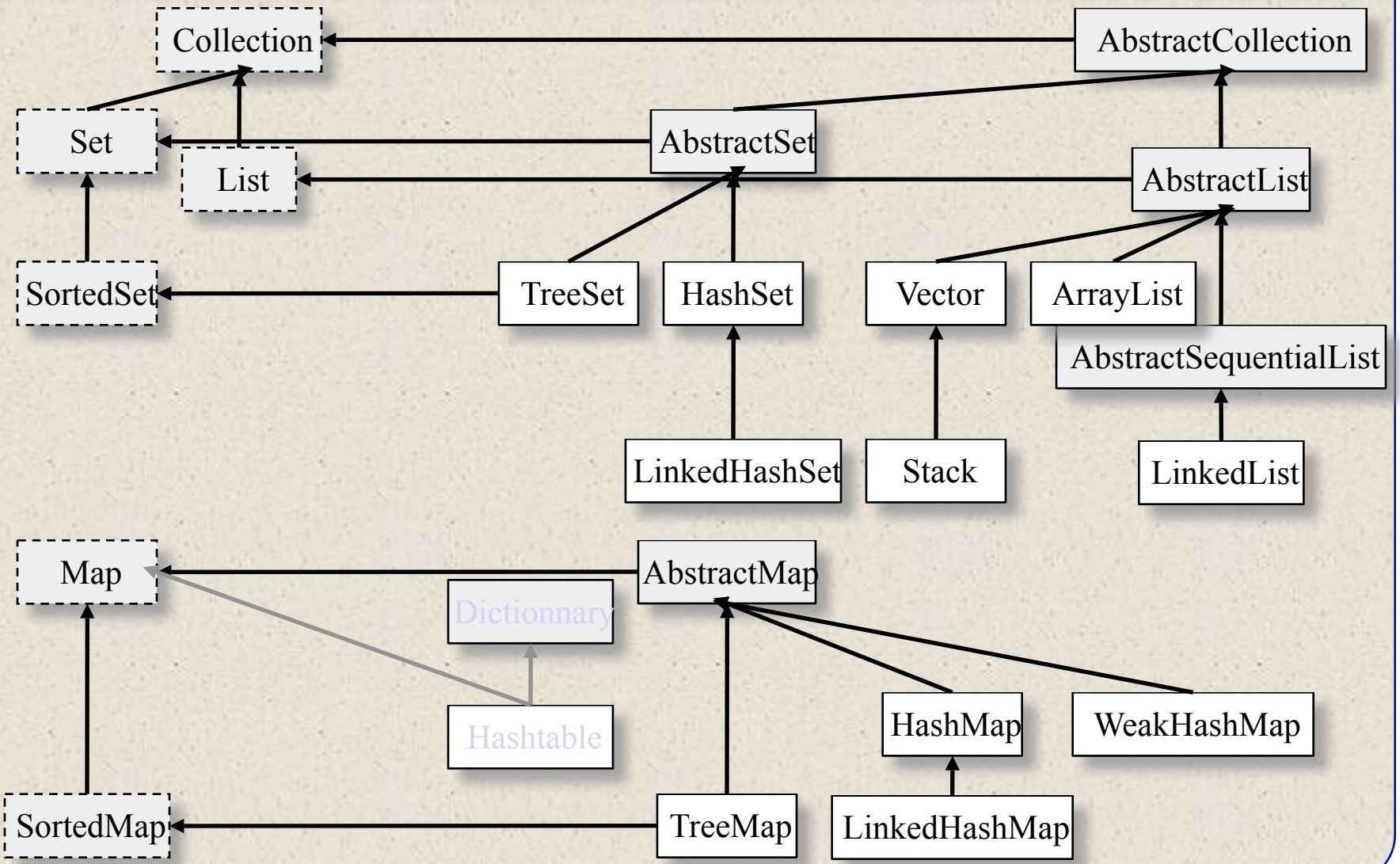
Les collections dans Java

- Types préexistants
  - Tableau
    - Intégré dans le langage sous une forme spécifique []
    - Opérateurs de conversion depuis ou vers cette représentation.
    - Généricité prise en compte
  - Vector
    - Tableau à taille variable, abstraction du tableau dans le modèle objet.
  - Hashtable
    - Premier modèle de table associative, est devenu obsolète depuis l'existence de Map
- Conversions
  - Depuis le type []
    - Dans la classe Arrays, la méthode statique asList de profil : [] ->List
  - Vers le type []
    - Dans la classe Collection, la méthode statique toArray de profil : -><Object>[]
    - Dans la classe Collection, la méthode statique toArray de profil : <T>[] -> <T>[]

Abstraction des structures de données



Hiérarchies des structures de données



Interface Collection

```
public interface Collection {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);    // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);    // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear();                    // Optional

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

Interface Collection

```
/**
*@ensure this.contains(o)
*@return !_this.contains(o)
*/
boolean add(Object o)
/**
*@ensure !this.contains(o)
*@return _this.contains(o)
*/
boolean remove(Object o)
/**
*@return
*/
boolean contains(Object o)
```

$\exists x \in C \ni x.equals(o)$

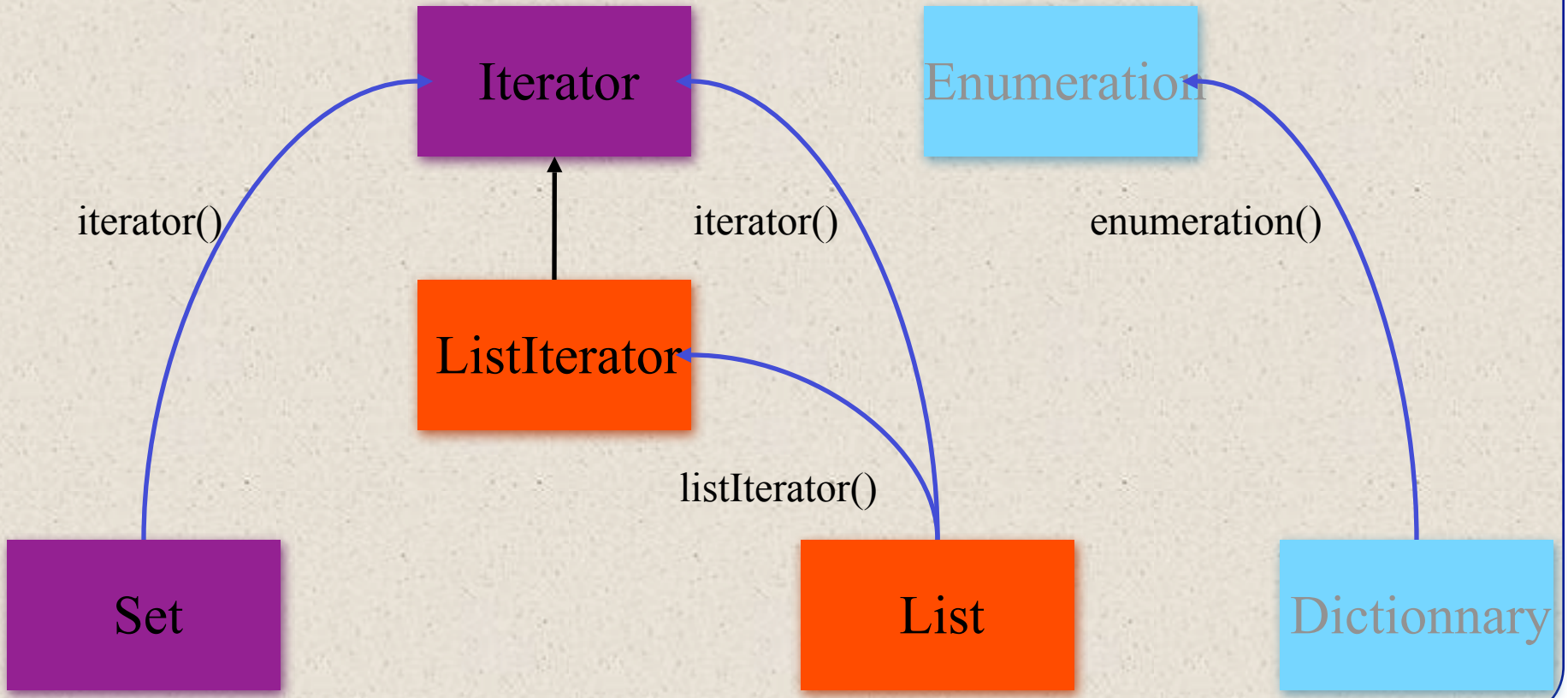
```
/**
*@require extensibleWith(o)
*@ensure contains(o)
*/
void add (Object o)
      throw  ExtensibleException
boolean extensibleWith(object o)
```


Interface List

```
public interface List extends Collection {
    // Positional Access
    Object get(int index);
    Object set(int index, Object element);          // Optional
    void add(int index, Object element);            // Optional
    Object remove(int index);                       // Optional
    boolean addAll(int index, Collection c);        // Optional

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

    // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

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

Iterator

Un iterator est un objet permettant d'énumérer les éléments constituant une collection dans ordre et d'une manière qui dépend de la nature de la collection.



Iterator

`Iterator iterator()`

- cette méthode renvoie un objet « iterator » qui permet le parcours séquentiel des éléments d'une collection.

`Iterator` **est une interface définie dans le package** java.util

Iterator

`Iterator iterator()`

– cette méthode renvoie un objet « iterator » qui permet le parcours séquentiel des éléments d'une collection.

`Iterator` **est une interface définie dans le package** java.util

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

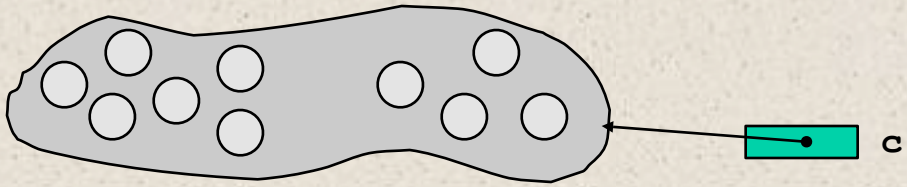

Iterator

`Iterator iterator()`

– cette méthode renvoie un objet « iterator » qui permet le parcours séquentiel des éléments d'une collection.

`Iterator` **est une interface définie dans le package** java.util

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




```
Collection c = new ...
```

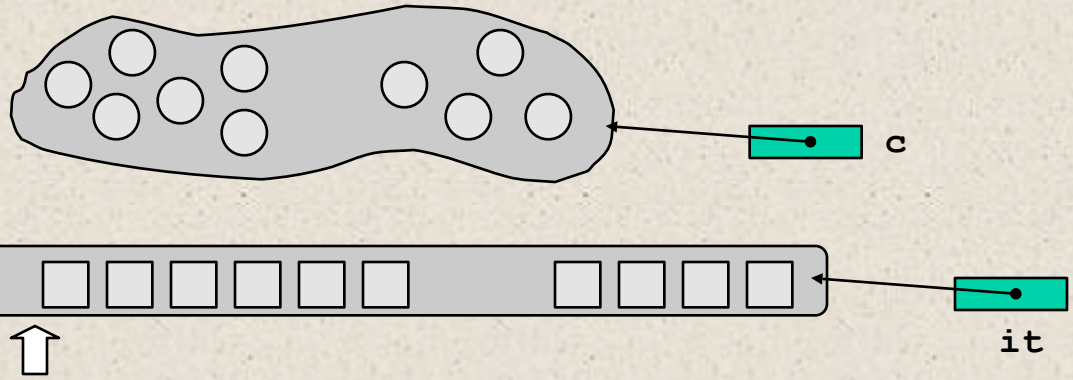
Iterator

`Iterator iterator()`

– cette méthode renvoie un objet « iterator » qui permet le parcours séquentiel des éléments d'une collection.

`Iterator` **est une interface définie dans le package** java.util

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




```
Collection c = new ...

Iterator it = c.iterator();
```

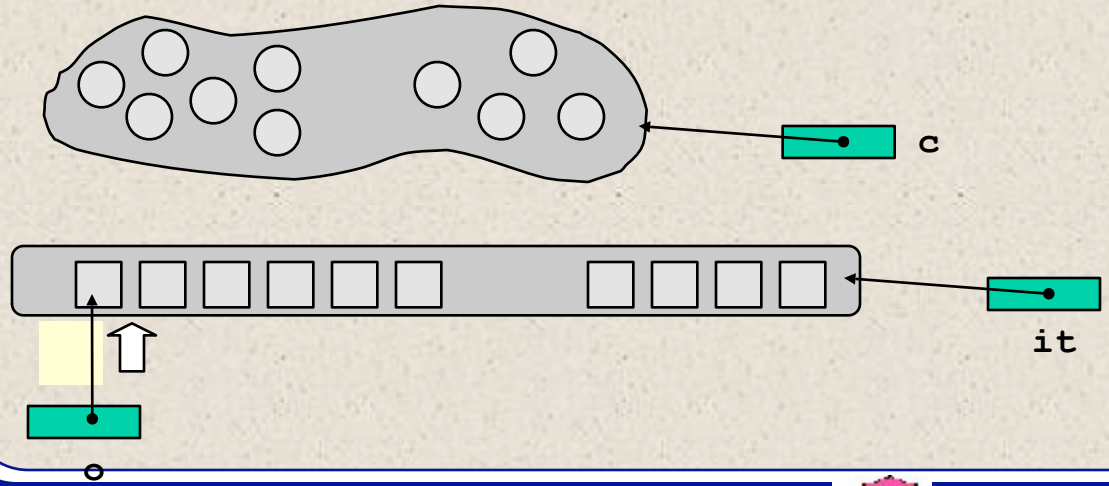
Iterator

`Iterator iterator()`

– cette méthode renvoie un objet « iterator » qui permet le parcours séquentiel des éléments d'une collection.

`Iterator` **est une interface définie dans le package** java.util

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




```
Collection c = new ...

Iterator it = c.iterator();

Object o = it.next();
```

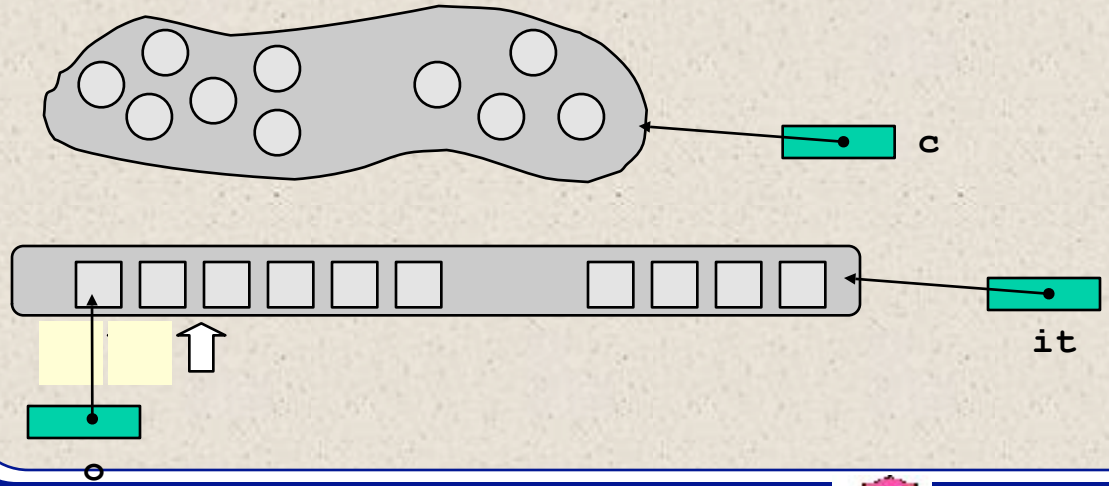
Iterator

`Iterator iterator()`

– cette méthode renvoie un objet « iterator » qui permet le parcours séquentiel des éléments d'une collection.

`Iterator` **est une interface définie dans le package** java.util

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




```
Collection c = new ...

Iterator it = c.iterator();

Object o = it.next();
it.next();
```

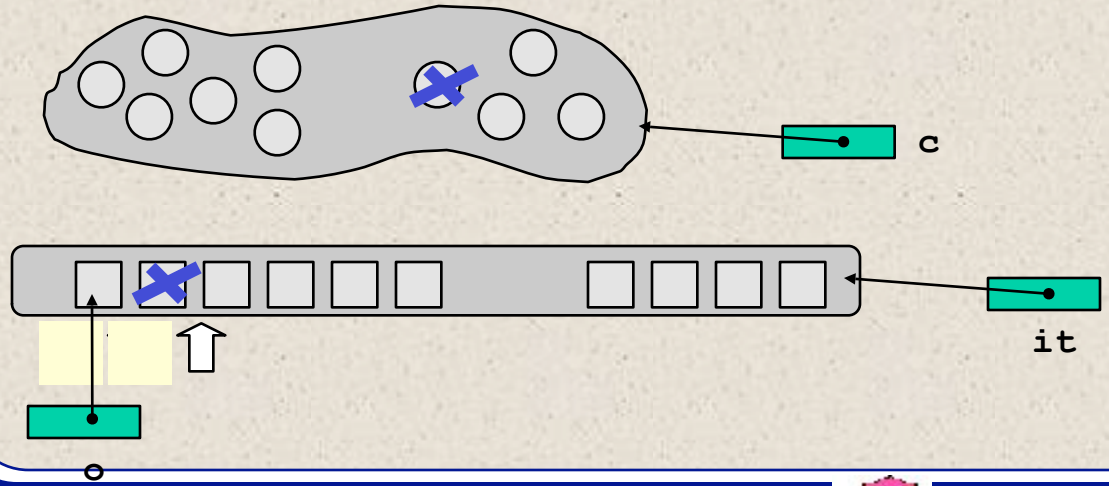
Iterator

`Iterator iterator()`

– cette méthode renvoie un objet « iterator » qui permet le parcours séquentiel des éléments d'une collection.

`Iterator` **est une interface définie dans le package** java.util

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




```
Collection c = new ...

Iterator it = c.iterator();

Object o = it.next();
it.next();
it.remove();
```

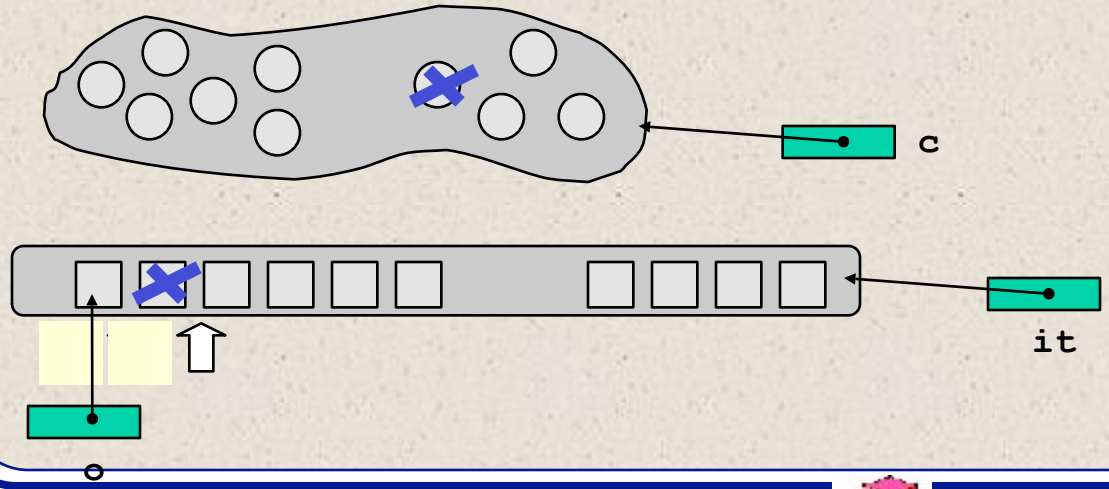
Iterator

`Iterator iterator()`

– cette méthode renvoie un objet « iterator » qui permet le parcours séquentiel des éléments d'une collection.

`Iterator` **est une interface définie dans le package** java.util

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




```
Collection c = new ...

Iterator it = c.iterator();

Object o = it.next();
it.next();
it.remove();
it.remove();
```

Iterator

L'absence de généricité impose de recaster l'objet obtenu dans le type approprié.

Iterator

Modèle général d'utilisation d'un iterator qui ne constitue pas un modèle d'ordre supérieur car le opération à effectuer n'est pas paramétrée.

L'absence de généricité impose de recaster l'objet obtenu dans le type approprié.

Iterator

Modèle général d'utilisation d'un iterator qui ne constitue pas un modèle d'ordre supérieur car le opération à effectuer n'est pas paramétrée.

L'absence de généricité impose de recaster l'objet obtenu dans le type approprié.

Un Iterator ne permet le parcours d'une collection qu'une fois et une seule. Un ListIterator autorise des retours en arrière.

Iterator

Modèle général d'utilisation d'un iterator qui ne constitue pas un modèle d'ordre supérieur car le opération à effectuer n'est pas paramétrée.

```
Collection c; // c est une collection de <T>
<T> e;
for(Iterator i = c.iterator(); i.hasNext();){
        e= (<T>)i.next();
        //Traiter e
}
```

L'absence de généricité impose de recaster l'objet obtenu dans le type approprié.

Un Iterator ne permet le parcours d'une collection qu'une fois et une seule. Un ListIterator autorise des retours en arrière.

Modèle d'itérateur

```
class Collection {
  …
  public static void forAll(Collection c; Action a) {
        for(Iterator i = c.iterator();i.hasNext();) {
              a.execute(i.next());
        }
  }
  public static Object forAll(Collection c; Fonction f) {
        Object r = f.execute(i.next());
        for(Iterator i = c.iterator();i.hasNext();) {
              Object e = i.next();
              f.combine(r,f.execute(i.next()));
        }
  }
…
}
Collection.forAll(c,new Action() { public void execute(Object o) { <T> e = (<T>)o;
  ….
}});
```


Les collections particulières

– Les collections non modifiables
  - L'objectif est d'assurer l'invariance de l'objet que l'on peut caractériser par la formule : _this.equals(this)
  - Assurée par l'émission systématique de l'exception UnsupportedOperationException par chaque opération sensée violée cette propriété.
  - Exemple de la méthode add :

```
public boolean add (Object o){
        throws new UnsupportedOperationException();
}
```

– Les collections synchronisées
  - L'objectif est d'assurer l'accès concurrent à la collection par divers threads
  - Exemple de la méthode add :

```
Collection c;
Object mutex;
public boolean add (Object o){
        synchronized(mutex){return c.add(o)};
}
```