Remise à niveau Programmation

Programmation Objet avec Java

Types enveloppes

Généricité

Collections

#6

Licence Pro
Métiers de l'Informatique : Conception, Développement,
Test de Logiciels, Parcours "Génie Logiciel, Système d'Information"
2020-2021

Types enveloppe

Java propose des types enveloppes (wrapper) pour chacun de ses types atomiques.
Ces types encapsulés sont des classes qui permettent

- de manipuler des valeurs atomiques comme des instances de classe (ce qui sera nécessaire plus tard dans ce chapitre)
- d'introduire un certain nombre de méthodes et de constantes de classe très pratiques.

L'utilisation des ces classes enveloppes est très intuitive.

Exemple

Afin d'illustrer cela, nous allons présenter rapidement la classe enveloppe **`Integer`**.
Toutes les classes enveloppes sont proposées avec un constructeur prenant en paramètre le type atomique correspondant :

```
{
     Integer i = new Integer( 3) ;
}
```

Les développeurs ont porté la plus grande attention à assurer un maximum de flexibilité entre le type atomique et le type enveloppe. Il existe donc une grande souplesse permettant de passer du type enveloppé au type atomique. Toutes ces expressions sont admises :

```
public class TestInteger {
     public static int plus1( int x){ return x+1 ;}

     public static void main( String [] args)
     {
          // Soit un entier
          int i1 = 3 ;

          // Les deux instructions suivantes font la même chose
          // grace aux procédures de boxing/unboxing
          Integer i2 = new Integer( i1) ;
          Integer i3 = i1 ;

          // Il est toujours possible de proposer un objet enveloppé là
          // où est attendu une valeur atomique
          int i4 = i3 ;
          int i5 = TestInteger.plus1(i3) ;
     }
}
```

Consultez les JavaDoc de ces classes. Pour la classe **`Integer`** :
https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html

Vous noterez les variables de classes et les méthodes de classes qui peuvent se révéler très utiles.
Vous pourrez également noter que les classes **`Integer`**, **`Long`**, **`Float`**, **`Double`** et d'autres sont des classes filles de la classe **`Number`**.

Problème de modularité

Afin d'illustrer le propos de ce début de chapitre nous allons poser un problème et voir comment le résoudre efficacement avec les outils de Java.

On souhaite manipuler des points (ce que nous avons largement fait depuis le début. Seulement, nous souhaitons pouvoir, selon nos nécessités, définir des points dont les coordonnées pourraient alternativement être des entiers, des flottants simple ou double précision. Voici une implémentation naïve.

```
public class PointInteger
{
     private int abscisse ;
     private int ordonnée;

     public PointInteger(int x,int y){ this.abscisse=x ;
this.ordonnée=y ;}
     public int getAbscisse(){return this.abscisse ;}
     public void translate(int dx, int dy){ this.abscisse+=dx ;
this.ordonnée+=dy ;}
     ...}
```

```
public class PointFloat
{
     private float abscisse ;
     private float ordonnée;

     public PointFloat(float x, float y){ this.abscisse=x ;
this.ordonnée=y ;}
     public float getAbscisse(){return this.abscisse ;}
     public void translate(float dx, float dy){ this.abscisse+=dx ;
this.ordonnée+=dy ;}
     ...}
```

```
public class PointDouble
{
     private double abscisse ;
     private double ordonnée;

     public PointDouble(double x, double y){ this.abscisse=x ;
this.ordonnée=y ;}
     public double getAbscisse(){return this.abscisse ;}
     public void translate(double dx, double dy){ this.abscisse+=dx ;
this.ordonnée+=dy ;}
     ...}
```

Si vous faites bien attention au code précédent vous vous apercevrez que seul le type change d'une classe à l'autre. Mis à part ce type, le reste du code est **IDENTIQUE**.
Les programmeurs, par soucis de modularité n'aiment pas répéter le même code. Il cherche à le *« factoriser »*.
Comment faire cela ?

Problème de modularité

Une solution pourrait être cherchée du côté de l'héritage en définissant une classe mère :

```
public class Point
{
     private Number abscisse ;
     private Number ordonnée;

     public Point (Number x, Number y){ this.abscisse=x ;
this.ordonnée=y ;}
     public Number getAbscisse(){return this.abscisse ;}
     //public void translate(Number dx, Number dy){ this.abscisse+=dx ;
this.ordonnée+=dy ;}
     ...
}
```

Le souci, c'est que maintenant on est beaucoup trop général. Par exemple la classe **`Number`** n'implémente pas les opérations aussi simples que celles de l'arithmétique !.

De plus, quand bien même la classe **`Number`** permettrait de coder l'ensemble des méthodes nécessaires (exemple : *supra* en retirant la méthode **`translate()`**), nous aurions mis en place une classe beaucoup trop générale pour être facilement utilisable. En effet il n'aurait par exemple pas été possible d'écrire

```
public class TestPoint
{
     public static void main( String [] args)
     {
     Point p = new Point(Integer x, Integer y) ;
     Integer i = p.getAbscisse() ;
}
```

Car la méthode **`getAbscisse()`** renvoie un **`Number`** qui est plus général qu'un entier. L'instruction serait donc rejetée à la compilation !

Il existe une façon de s'en sortir élégamment en Java. La solution de ce type de problème est la généricité.

Généricité

En pratique, jusqu'ici, à chaque fois que l'on a dû déclarer une variable d'un programme ou un paramètre d'une méthode ou bien la valeur de retour d'une méthode, nous avons dû spécifier dans cette déclaration le type de la valeur ou de la variable.
Ce type pouvait être un des types atomiques (**`double`**, **`float`**, **`int`**, **`boolean`**, …) ou être le nom d'une classe.

Définition

La généricité permet de rendre le type déclaré paramétrable et donc variable : *i.e.* c'est un mécanisme qui permet de définir des classes et des méthodes sans que l'on fixe a priori les types des données qui sont manipulées. Cela permet de

- Réutiliser des classes et des méthodes avec différents types de données (pour plusieurs types différents de données, le même code sera utilisé).
- Augmenter la sécurité du code en captant des erreurs à la compilation et non à l'exécution.

Pour cela il s'agit de définir des classes et des méthodes possédant un ou plusieurs paramètres représentant des types indéterminés à la compilation qui seront remplacés par des types déterminés à l'exécution.

Paramètres génériques

Ces paramètres de type sont appelés *paramètres génériques*. Par exemple, une classe possédant un ou plusieurs paramètres génériques sera appelée *classe générique*.

Exemple

```
public class UneClasse <T,U>
```

Ici, les paramètres **`T`** et **`U`** sont des paramètres génériques associés à la définition de la classe **`UneClasse.`**
Il est d'usage que le nom des paramètres génériques soit très court (une seule lettre majuscule).

Définition

- Un paramètre générique est une variable **représentant** un type non primitif (une classe, par exemple **`Personne`**, **`Point`**, **`Float`** ...)
- Chaque paramètre générique peut être utilisé partout où l'on attend un type (non primitif)
- Chaque paramètre générique est initialisé à l'exécution

Paramètres génériques

Les paramètres génériques sont donnés entre les symboles **<** et **>** (les chevrons ouvrants et fermants) dans les phases déclaratives des noms de classe, et sont utilisés comme n'importe quel type/classe ensuite.

Exemple

Nous définissons ici une classe générique **Couple**, composées de deux membres. Le type **T** de ses membres est générique. Il sera définit à l'exécution. **T** pourra alors prendre n'importe quelle valeur parmi les noms de classes définis.

```
public class Couple <T>
{
     private T premier ;
     private T second ;

     public Couple( T v1, T v2)
     {
          this.premier = v1 ;
          this.second = v2 ;
     }

     public T getPremier()
     {
          return this.premier ;
     }

     public T getSecond()
     {
          return this.second ;
     }
}
```

Par exemple, on peut de la même façon définir un **Couple** de **Double** ou un **Couple** de **Point**:

```
public class Test {
     public static void main(String [] args){

          Double d1 = new Double ( 1.2) ;
          Double d2 = new Double ( 0.7) ;
          // T est un Double (type encapsulé). Nous définissons ainsi
          // un couple de Double.
          Couple <Double> cd = new Couple<Double>( d1, d2);

          Point p1 = new Point( 1.2, 2.3) ;
          Point p2 = new Point( 0.7, -13.) ;
          // T est un Point (type encapsulé). Nous définissons ainsi
          // un couple de Point.
          Couple <Point> cp = new Couple<Point>( p1, p2);
     }
}}
```

A l'exécution le paramètre générique **T** sera instancié par un type : Le type **Double** pour le **Couple cd**, **Point** pour le **Couple cp**

Paramètres génériques

A l'exécution le type de l'objet retourné par les méthodes **`getPremier()`** et **`getSecond`** () sera du type *effectif* avec lequel a été appelé le constructeur de la classe générique.
Ainsi :

```
public class Test {
     public static void main(String [] args){

          Double d1 = new Double ( 1.2) ;
          Double d2 = new Double ( 0.7) ;
          // Le constructeur est appelé avec Float comme paramètre
          // effectif du paramètre générique.
          Couple <Double> cd = new Couple<Double>( d1, d2);

          // La méthode renvoie donc un élément du type Double (valeur
          // effective de T
          Double d = cf.getPremier();
     }
}}
```

De la même façon :

```
public class Test {
     public static void main(String [] args){

          Point p1 = new Point( 1.2, 2.3) ;
          Point p2 = new Point( 0.7, -13.) ;
          // Le constructeur est appelé avec Point comme paramètre
          // effectif du paramètre générique.
          Couple <Point> cp = new Couple<Point>( p1, p2);

          // La méthode renvoie donc un élément du type Point (valeur
          // effective de T
          Point p = cp.getPremier();
     }
}}
```

Remarque

Le constructeur générique de la classe **`Couple`**, ne fait référence au paramètre générique **`T`** que dans le typage de ses paramètres :

```
public Couple( T v1, T v2) { ... }
```

Pourtant, lors de l'appel du constructeur, il faut préciser à nouveau le type de **`T`** entre chevron juste derrière le nom du Constructeur.

```
{
     // Comme attendu la valeur de T est donnée lors de la déclaration
     Couple<Double> c ;
     // la valeur de T est ÉGALEMENT rappelée lors de l'appel au
constructeur !
     c = new Couple <Double>( 1.2, 2.3) ;
}
```

Interface générique

De la même façon qu'il est possible de définir une classe générique, il est possible de définir une interface générique.
Une interface est dite générique si elle possède un ou plusieurs paramètres génériques.

Exemple

```
public interface InterfaceCouple <T>
{
          public T getPermier() ;
          public T getSecond() ;
}
```

Comme toute interface, elle ne peut être instanciée. Elle ne fait que définir un contrat abstrait. Elle a vocation à être réalisée par une implémentation dans une classe concrète :

```
public class Point <T> implements InterfaceCouple <T>
{
     private T abscisse ;
     private T ordonnée ;

     public Point( T x, T y) { this.abscisse = x ; this.ordonnée = y ; }
     public T getAbscisse(){ return this.abscisse ; };
     public getOrdonnée(){ return this.ordonnée ; };

     public T getPermier(){
          return this.getAbscisse() ;
     };

     public T getSecond(){
          return this.getOrdonnée() ;
     };
}
```

On peut ainsi définir un **Point** alternativement avec des coordonnées flottantes simple précision, double précision ou entières :

```
{
     Point <Float> pF ;
     pF = new Point <Float>(1.2f, 4.7f) ;

     Point <Double> pD ;
     pD = new Point <Double>(1.2, 4.7) ;

     Point <Integer> pI ;
     pI = new Point < Integer >(1, 2) ;

     ...
}
```

Interface générique - suite

Grace à la généricité du contrat abstrait qui se traduit par une généricité dans le contrat des classes concrètes qui implémentent l'interface, il est possible de récupérer respectivement un flottant simple précision, double précision ou un entier à l'appel des méthodes du contrat abstrait de l'interface **`InterfaceCouple`** (comme à l'appel des méthodes des contrats génériques concrets de la classe **`Point`**) :

```
...

Float f1 = pF.getPremier() ;
Float fA = pF.getAbscisse() ;

Double d1 = pD.getPremier() ;
Double dA = pD.getAbscisse() ;

Integer i1 = pI.getPremier() ;
Integer iA = pI.getAbscisse() ;

System.out.println("f1 = " + f1 + " instanceof Float = " + (f1 instanceof Float));
System.out.println("fA = " + fA + " instanceof Float = " + (fA instanceof Float));

System.out.println("d1 = " + d1 + " instanceof Double = " + (d1 instanceof Double));
System.out.println("dA = " + dA + " instanceof Double = " + (dA instanceof Double));

System.out.println("i1 = " + i1 + " instanceof Integer = " + (i1 instanceof Integer));
System.out.println("iA = " + iA + " instanceof Integer = " + (iA instanceof Integer));
```

Affiche **`true`** à chaque fois.

Interface Comparable :

Les dernières versions de Java proposent une version générique de l'interface comparable qu'il faut utiliser. Par exemple :

```
public class Polygone implements Comparable <Polygone>
{
    ...

    public int compareTo( Polygone p)
    {
    ...
    }
}
```

Méthodes génériques

La généricité peut se circonscrire à une méthode (statique ou non).
Une méthode générique possède un ou plusieurs *paramètres génériques*
(Afin de faciliter la présentation de cette possibilité en Java nous allons la détailler au moyen d'une méthode statique (de classe).)

Soit la classe suivante **`TableauUtile`** définissant des méthodes de classes (sorte de fonction) permettant de rendre des services génériques sur les tableaux quels que soient le type d'objet qu'ils contiennent comme :

- renvoyer le premier élément du tableau passé en paramètre
- afficher le premier élément du tableau passé en paramètre

```
public class TableauUtile
{
      public static <T> void afficherPremierÉlément( T [] tab)
      {
            System.out.println( tab[0]) ;
      }

      public static <T> T getPremierÉlément( T [] tab)
      {
            return tab[0] ;
      }
}
```

Remarque

Le paramètre générique de la méthode est donné entre chevrons (**< >**) juste à la suite de la définition de la porté de la méthode et juste avant la déclaration du type de la valeur de retour de la méthode.

Ensuite, partout où cela est nécessaire, il est possible de faire référence au paramètre numérique par son nom (nom qui sera comme toujours utilisé en lieu et place d'un type).
L'appel de la méthode se fait en précisant le type du paramètre générique :

```
public class Test
{
      public static void main(String [] args)
      {
            Point p1 = new Point(4, 5) ;
            Point p2 = new Point(6,7) ;
            Point[] t1 = {p1,p2} ;

            Point p = TableauUtile.<Point>getPremierÉlément(t1);
      }
}
```

Contraindre la généricité

Il est possible de poser des contraintes sur les valeurs que peut prendre un paramètre générique lors de l'exécution.

Exemple

Il est possible d'indiquer que le paramètre générique **`T`** sera obligatoirement instancié par la classe **`Point`** ou une des ses classes dérivées.
L'appel de la méthode se fait en précisant le type du paramètre générique :

```
<T extends Point >
```

Exemple

Il est également possible d'imposer que le paramètre générique **`T`** soit obligatoirement instancié par une classe qui implémente les interfaces **`Comparable`** *et* **`Serializable`**

```
<T extends Comparable & Serializable >
```

Attention

On utilise **`extends`** au lieu de **`implements`**

Différents types de Collection

Une collection est une classe permettant de stocker et d'organiser des objets
Java distingue (principalement) 3 catégories de collections :

- les **`List`**:
  - liste = groupe ordonné d'éléments (contenant d'éventuelles duplications)
  - classes : **`ArrayList`**, **`LinkedList`**, **`Vector`**

- les **`Set`** :
  - ensemble = groupe d'éléments (donc non ordonnés) sans duplication
  - classes : **`HashSet`**, **`TreeSet`**

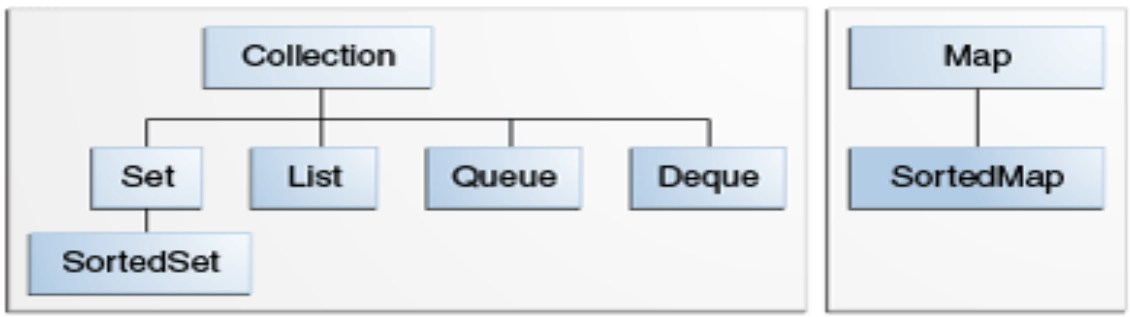
- les **`Map`**:
  - table d'association = ensemble de couples (clef/valeur). Les clefs sont uniques (pas de doublons), les valeurs sont libres (doublons autorisés)
  - classes : **`HashTable`**, **`HashMap`**, **`TreeMap`**

Ces conteneurs de données offrent des services que n'offrent pas les tableaux statiques :

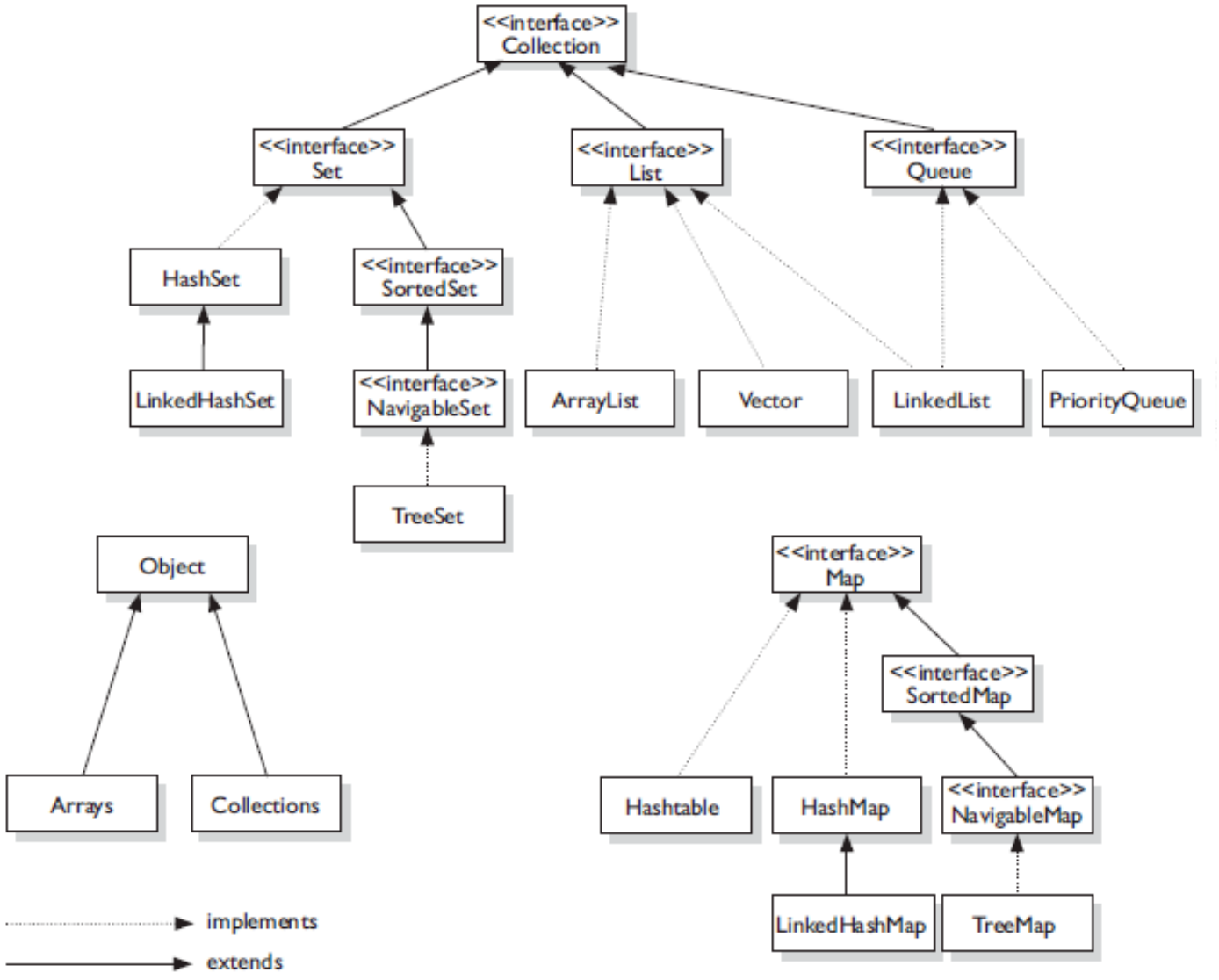
- Ils sont de taille dynamique : contrairement aux tableaux (statiques), lorsque leur capacité atteint ses limites, ce conteneur de données se redimensionne automatiquement (augmentation de taille dynamique automatique).
- Ils proposent des outils de parcours tels que les itérateurs,
- Il proposent un certains nombre de méthodes facilitant la gestion du contenu
- Ils implémentent des algorithmes qui optimisent l'accès aux données.

Selon les contraintes que l'on souhaite placer sur la collection comme l'unicité des éléments, l'ordonnancement des données, la navigabilité dans la collection l'efficacité des parcours mono ou bi-directionnels ou encore l'efficacité des opérations d'ajout ou de suppression de donnée, le programmeur choisira l'une ou l'autre des différentes collections proposées par l'API de Java.

Les collections vont implémenter des interfaces génériques. Selon les collections, les interfaces implémentées pourront être différentes.



Interfaces & Implémentations



Source : http://www.corejavatutorial.com/collections/

interface Collection<T>

L'interface **Collection<T>** est implémentée par l'ensemble des classes que l'on appelle Collections. Le contrat public que passe cette interface et qu'implémentent les classes concrètes s'appuie sur les méthodes et fonctionnalités suivantes :

```
public interface Collection <T>
{
     // le nombre d'éléments contenus
     int size() ;

     // Vrai si il n'y a aucun élément contenu et Faux sinon.
     boolean isEmpty() ;

     // Vrai si element est contenu dans la Collection et et Faux sinon.
     boolean contains(Object element) ;

     // Tente l'ajout de element à la collection et renvoie vrai si elle
     // a réussi à l'ajouter et false si elle n'a pas réussi.
     boolean add(E element) ;

     // Tente de retirer element de la collection et renvoie vrai si
     // elle a réussi à le retirer et false si elle n'a pas réussi.
     boolean remove(Object element) ;

     // Renvoie un itérateur sur la collection
     Iterator<E> iterator() ;

     // Vrai si la collection courante contient tous les éléments de c.
     boolean containsAll(Collection<?> c) ;

     // Ajoute à la collection courante l'ensemble des éléméents de c
     boolean addAll(Collection<? extends E> c) ;

     // supprime de la collection courante l'ensemble des éléments de c.
     boolean removeAll(Collection<?> c) ;

     // conserve dans la collection courante uniquement les éléments
     // également présents dans c.
     boolean retainAll(Collection<?> c) ;

     // vide la collection courante de l'ensemble de ses éléments.
     void clear() ;
}
```

interface Collection<T>

Redéfinition de la méthode equals()

Pour assurer leur contrat, les implémentations concrètes des méthodes **`remove()`**, **`contains()`** et parfois **`add()`** font appel à la méthode **`equals()`**sur les objets **`T`** de la collection.

Par exemple, **`remove(element)`** recherche si la **`Collection <T>`** contient un objet identique à **`element`** au sens de **`equals()`**et le supprime.

De la même façon, les collections qui implémentent l'interface **`Set`** (collections dans laquelle les éléments sont uniques – *ie* pas de doublons) ont absolument besoin que l'égalité entre deux éléments soit définie pour assurer l'unicité de ceux-ci : il n'y aura pas deux éléments identiques au sens de **`equals()`** dans une collection **`Set`**.

Si la méthode **`equals()`** n'a pas été redéfinie pour la classe alors c'est celle hérité de sa classe mère ou par défaut de celle de la classe **`Object`** qui sera utilisé !!!

For-each

Comme mentionné précédemment, les collections offrent des fonctionnalités nouvelles de parcours.

Pour rappel, dans les tableaux statiques, le seul moyen de parcourir les éléments du tableau était de définir une variable entière dont l'incrémentation permettait de jouer le rôle d'un indice de position variable dont la valeur était comprise entre 0 et la taille du tableau.

Avec les collections apparait une nouvelle structure de contrôle itérative la boucle **for-each** :

```
{
     ArrayList <Point> collection = new ArrayList<Point>() ;

     ...

     for ( Point elem : collection)
     {
          System.out.println( elem) ;
     }
}
```

Cette boucle fonctionne pour toutes les collections et passe en revue l'ensemble des éléments contenus dans la collection. Cependant, il sera nécessaire d'utiliser un itérateur (de la classe Iterator ou ListIterator) dès que l'on aura à supprimer des éléments de la collection lors du parcours, ou lorsque l'on voudra, au moyen d'une seule boucle, parcourir en parallèle plusieurs collections.

ArrayList <T>

Cette structure remplace le plus souvent les tableaux statiques.

Elle implémente un conteneur de données sous forme d'une liste ordonnée où chaque élément (seules les références sont stockées dans la collection) est repéré par l'indice **`int i`** de sa position dans la liste.

En plus des méthodes implémentées de l'interface **`Collection`**, les **`ArrayList`** proposent des méthodes spécifiques utilisant cet indice :

```
{
     // ajoute elem en dernière position de la liste courante (retourne
     // true si l'ajout est effectué, sinon false)
     public boolean add(E elem){ ... }

     // insère la référence elem au rang i de la liste courante (les
     // références de rang supérieur ou égal à i sont décalées d'un
     // cran vers la droite)
     void add(int i, E elem) { ... }

     // retire de la liste la référence de rang i et la retourne
     // (les références de rang supérieur ou égal à i sont décalées
     // d'un cran vers la gauche)
     E remove(int i ) { ... }

     // retourne la référence de rang i de la liste courante
     E get(int i) { ... }

     // remplace la référence de rang i (et la retourne) par la
     // référence elem
     E set(int i, E elem) { ... }

     // cherche dans la liste courante une référence désignant un objet
     // égal (au sens de equals) à l'objet référencé par elem et retour-
     // ne le rang de cette référence si elle est trouvée, sinon -1
     int indexOf(E elem) { ... }
}
```

Exemple de création de liste :

```
public static void main( String [] args)
{
     // Déclaration et instanciation
     ArrayList <Point> sommets = new ArrayList<Point>() ;
     Point p = new Point( 2.3, -7.5) ;

     // Remplissage cumulatif
     sommmets.add(p) ;
     sommets.add( new Point( 7.4, 3.1) ) ;
}
```

Iterateur

Il existe essentiellement deux types d'itérateurs associés aux collections :

- Les itérateurs mono-directionnels qui implémentent l'interface **`Iterator`**. Toutes les **`Collections<T>`** proposent une méthode **`iterator()`** qui renvoie une instance d'itérateur.
  Ces itérateurs permettent de parcourir l'ensemble des éléments de la collection.
  - Si la collection est ordonnée, les éléments sont parcourus dans l'ordre, depuis le premier jusqu'au dernier : à chaque appel de la méthode **`next()`** sur l'instance d'itérateur, l'instance suivante de la collection est retournée.
    Il n'y a pas de moyen de revenir en arrière (parcours rétrograde)
  - Si la collection ne propose pas d'ordre naturel ou défini (**`Set, Map`**), alors les éléments sont parcourus d'après un ordre fixé inconnu.
  -
- Les itérateurs bi-directionnel : Ils sont implémentés à partir de l'interface **`ListIterator`** dans le cas de certaines collections et permettent de parcourir la collection d'après l'ordre naturel et d'après un ordre rétrograde.

Les itérateurs sont instanciés par les collections

Les **`ArrayList`** proposent les deux types d'itérateur ;

- la méthode **`iterator()`** fournit une instance d'itérateur monodirectionnel
- la méthode **`listIterator()`** fournit une instance d'itérateur bi-directionnel

```
public class Test
{
     public static void affiche( ArrayList<Point> sommets)
     {
          ListIterator it = sommets.listIterator() ;

     }
}
```

Les services des itérateurs

```
// ----------------------------------------------------------
// Méthodes communes aux instances d'Iterator et de ListIterator


// retourne true s'il y a un élément derrière le curseur,
// false si l'on est en fin de collection
public boolean hasNext() ;

// retourne l'élément situé derrière le curseur et avance le
// curseur d'un cran
public E next() ;

// retire la référence retournée au dernier appel next( )
//(il faut obligatoirement avoir appelé next( ) pour utiliser
// remove( ) )
public void remove() ;

// ----------------------------------------------------------
// Méthodes des instances de ListIterator

// retourne true s'il y a un élément avant le curseur,
// false si l'on est en début de collection
public boolean hasPrevious() ;

// retourne l'élément situé derrière le curseur et recule le
// curseur d'un cran
public E previous() ;

// Insère l'élément derrière le curseur et avance le curseur
// d'un cran
void add (E elem) ;

// Remplace le dernier élément renvoyé par next() ou previous
// elem
void set( E elem) ;
```

Utilisation des iterateurs

```
import java.util.ArrayList;
import java.util.ListIterator;

public class TestArray
{
     public static void main( String [] args)
     {
          String sbuf;

          ArrayList <String> maListe = new ArrayList<String>( );

          maListe.add( "s0");
          maListe.add( "s1");
          maListe.add( "s2");
          maListe.add( "s3");
          System.out.println( maListe);      //[s0, s1, s2, s3]

          ListIterator<String> it = maListe.<String>listIterator();
                                             //  ↓
                                             //[s0, s1, s2, s3]
          it.add( "i0");      // ---------------------------------------
                                             //    ↓
                                             //[i0, s0, s1, s2, s3]
          sbuf = it.next();   // sbub = "s0" ---------------------------
                                             //        ↓
                                             // [i0, s0, s1, s2, s3]
          it.add( "i1") ;     // ---------------------------------------
                                             //             ↓
                                             // [i0, s0, i1, s1, s2, s3]
          sbuf = it.next();   // sbub = "s1" ---------------------------
                                             //                  ↓
                                             // [i0, s0, i1, s1, s2, s3]
          it.remove();        // ---------------------------------------
                                             //             ↓
                                             // [i0, s0, i1, s2, s3]
          it.add( "i2") ;     // ---------------------------------------
                                             //                  ↓
                                             // [i0, s0, i1, i2, s2, s3]
          sbuf = it.previous(); // sbub = "i2" ------------------------
                                             //             ↓
                                             // [i0, s0, i1, i2, s2, s3]
          while (it.hasNext())
               it.next();                    //                       ↓
                                             // [i0, s0, i1, i2, s2, s3]
          it.set( "iF");      // ---------------------------------------
                                             //                       ↓
                                             // [i0, s0, i1, i2, s2, iF]
     }
}
```

Exercices #6

Pour plus de commodité dans les tests, nous avons rajouté à **`PointPlan`** une méthode pour initialiser un point avec une abscisse et une ordonnée aléatoire (valeurs décimales à une virgule, entre 0 et 10).
La classe **`Polygone`** est implémentée avec un tableau, ce qui rend malaisé l'ajout de nouveaux points, les tableaux étant de taille fixe.
Vous pouvez trouver les codes sources dans l'archive seance6_debut.tgz en suivant le lien :

https://www.lipn.univ-paris13.fr/~santini/RN3/

Écrire une classe **`TestPolygone`** qui permette de :

Question 1 :
Instancier un polygone nommé **`poly`**.

Question 2 :
Ajouter huit **`PointPlan`** aléatoires.

Question 3 :
Insérer un nouveau point de coordonnées *(20,20)* en quatrième position.

Question 4 :
Insérer un nouveau point de coordonnées *(25,25)* en treizième position. Avant même de tester le programme, quel résultat attendez-vous ?

Question 5 :
Insérer un nouveau point de coordonnées *(30,30)* en dixième position.

Question 6 :
Ajouter un nouveau point de coordonnées *(40,40)*. Avant même de tester le programme, quel résultat attendez-vous ?

Question 7 :
Ajouter un nouveau point de coordonnées *(50,50)*. Avant même de tester le programme, quel résultat attendez-vous ?

Vous devriez obtenir une sortie ressemblant à :

```
poly ; 8 ; (5.9,7.8)(5.4,9.5)(2.3,3.4)(4.8,1.2)(4.4,4.6)(3.2,2.8)(2.1,4.5)(1.3,8.7)
poly ; 9 ;
(5.9,7.8)(5.4,9.5)(2.3,3.4)(20.0,20.0)(4.8,1.2)(4.8,1.2)(4.8,1.2)(4.8,1.2)(4.8,1.2)
Erreur dans this.inserer() poly ; 9 ;
(5.9,7.8)(5.4,9.5)(2.3,3.4)(20.0,20.0)(4.8,1.2)(4.8,1.2)(4.8,1.2)(4.8,1.2)(4.8,1.2)
poly ; 10 ;
(5.9,7.8)(5.4,9.5)(2.3,3.4)(20.0,20.0)(4.8,1.2)(4.8,1.2)(4.8,1.2)(4.8,1.2)(4.8,1.2)(30.
0,30.0) Erreur dans this.ajouter() poly ; 10 ;
(5.9,7.8)(5.4,9.5)(2.3,3.4)(20.0,20.0)(4.8,1.2)(4.8,1.2)(4.8,1.2)(4.8,1.2)(4.8,1.2)(30.
0,30.0) Erreur dans this.inserer() poly ; 10 ;
(5.9,7.8)(5.4,9.5)(2.3,3.4)(20.0,20.0)(4.8,1.2)(4.8,1.2)(4.8,1.2)(4.8,1.2)(4.8,1.2)(30.
```

```
0,30.0)
```

Nouvelle implémentation de Polygone

Dans l'exercice précédent, vous avez vu quelques limitations des tableaux. Les questions suivantes visent à produire une nouvelle implantation pour **`Polygone`**, de manière à ce que tous les ajouts et insertions de l'exercice précédent soient couronnés de succès (sauf l'ajout en treizième position). Cette nouvelle implantation pourra utiliser les **`ArrayList`** vus en cours.
L'interface publique de la classe **`Polygone`** ne doit pas changer, de manière à assurer une compatibilité avec toutes les classes qui utiliseraient déjà la version *tableaux* de la classe **`Polygone`**. Vous utiliserez donc tel quel le **`TestPolygone`** précedent. Vérifier qu'un code modifié continue de rendre au moins les mêmes services que la version précédente, c'est faire ce que l'on appelle un *test de non régression*.

Question 8 :
Donnez les nouvelles variables d'instance de la classe **`Polygone`**.

Question 9 :
Réécrire les deux premiers constructeurs.

Question 10 :
Réécrire tous les accesseurs (getters, setters etc)

Question 11 :
Réécrire **`inserer(...)`**. **Attention :** ce n'est pas parce qu'un **`ArrayList`** augmente sa taille en fonction des besoins qu'il est possible d'insérer des éléments après une case qui n'existe pas.

Question 12 :
Est-ce que **`translater(...)`** a besoin d'être réécrite ? Si oui, faites-le d'une manière qui reste également compatible avec la première version du programme. C'est ainsi que la méthode **`translater(...)`** aurait dû être codée dès le début.

Question 13 :
Même question avec **`creerTranslater(...)`**, **`toString(...)`** et **`ajouterRandom(...)`**.

Question 14 :
Réécrire les deux derniers constructeurs. Pour vous aider, la méthode **`Arrays.asList(tab)`** retourne un **`ArrayList`**, transformation du tableau **`tab`** en **`ArrayList`**.

Documentation d'une classe

Question 15 :
Réalisez une javadoc complète de la classe **`Polygone`**