

# Remise à niveau Programmation

---

*Programmation Objet avec Java*

Valeur(s) de retour de méthode

Chaîne(s) de caractères

Objet composé

Constructeurs par Copie

Les tableaux

**#2**

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



# Valeurs de retour de méthode

## Syntaxe

Les méthodes, comme les fonctions dans d'autres langages, peuvent, avoir une valeur de retour. Ceci est précisé lors de la déclaration de la méthode (dans son entête) et lors de sa définition par la présence d'une instruction **return** dans son corps d'instruction.

```
public type_retour nom_de_la_methode(type_param par1, ...) {  
    // Corps d'instruction  
    ... ;  
    ... ;  
    return ... ;  
}
```

Les types possibles pour la déclaration des valeurs de retour sont les mêmes que pour la déclaration des types des variables avec le type **void** en plus :

type_retour	Valeur retournée
<b>void</b>	rien
<b>char, byte, short, int, long, double, double, boolean</b>	Une valeur du type primitif
<b>String, PointPlan, Droid, ...</b>	Une référence vers une instance du type déclaré

## Cas de conception

Plusieurs situations conduisent à définir des méthodes renvoyant une valeur. Nous en présenterons 3 :

- **Situation 1** : La méthode réalise un calcul et retourne au programme appelant une *valeur calculée d'un type atomique*.
- **Situation 2** : La méthode crée ou sélectionne un objet et retourne au programme appelant *la référence de l'objet*.
- **Situation 3** : La méthode effectue plusieurs opérations et retourne au programme appelant *plusieurs éléments* qui peuvent être des valeurs atomiques et/ou des références.

# Retour d'un type atomique

On développe une classe **PointPlan** permettant de décrire et de manipuler les points définis dans un espace 2D. On souhaite doter cette classe d'une méthode (service) qui renvoie la distance à l'origine du point correspondant à l'instance considérée.

## Implémentation

Nous définissons une méthode **distanceAOrigine()** dont le type de la valeur de retour est **double** car la distance d'un point à l'origine du repère s'exprime comme un nombre réel. Le corps d'instruction de la méthode réalise le calcul et l'instruction **return** renvoie au programme appelant le résultat de ce calcul.

```
public class PointPlan
{
    . . .
    // Distance entre le point et l'origine du repère
    public double distanceAOrigine()
    {
        double dist ;
        dist =      Math.pow( this.getAbscisse(), 2) +
                  Math.pow( this.getOrdonnée(), 2) ;
        return Math.sqrt( dist) ;
    }
}
```



## Appel/Utilisation

L'appel de la méthode **distanceAOrigine()** pourra être effectué sur toute instance de **PointPlan** dans une expression admettant pour paramètre un **double**.

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

        // Définition d'un point
        PointPlan pOri = new PointPlan( 7., 5.) ;

        // Déclaration d'une variable pour stocker la distance
        double dist;

        // Appel de la méthode et affectation de la valeur
        dist = pOri.distanceAOrigine() ;
    }
}
```



Schéma :

# Retour d'un objet (référence)

On veut définir à partir d'un point, un nouveau point qui soit son symétrique par rapport à l'origine du repère. Pour cela on va créer une méthode dans la classe **PointPlan**,

- dont la fonction sera de renvoyer une nouvelle instance de **PointPlan**
- dont les coordonnées seront les symétriques de l'instance sur laquelle la méthode aura été appelé.

## Implémentation

Nous définissons une méthode **symétrique()** dont le type de la valeur de retour est **PointPlan** car la méthode renvoie un point symétrique. Le corps d'instruction instancie un nouveau **PointPlan** avec les coordonnées symétriques du point courant au moyen de l'opérateur **new**. L'opérateur **new** retourne la référence de la nouvelle instance de **PointPlan** qui est à son tour retournée au programme appelant la méthode par l'instruction **return**. La méthode renvoie donc une référence vers un **PointPlan** qui a été créé par la méthode.

```
public class PointPlan {  
    . . .  
    // retourne un point symétrique à l'instance courante  
    public PointPlan symétrique() {  
        return new PointPlan(-this.getAbscisse(), -  
this.getOrdonnée());  
    }  
}
```



## Appel/Utilisation

```
public class TestPointPlan {  
    public static void main (String [] args) {  
  
        // Définition d'un premier point  
        PointPlan pOri = new PointPlan( 7., 5.) ;  
  
        // Déclaration d'une variable pour stocker la référence  
        // d'un point symétrique au premier  
        PointPlan pSym;  
  
        // Appel de la méthode sur le point et affectation  
        // de la référence renvoyée du nouveau point à la variable  
        pSym = pOri.symétrique() ;  
    }  
}
```



Schéma :

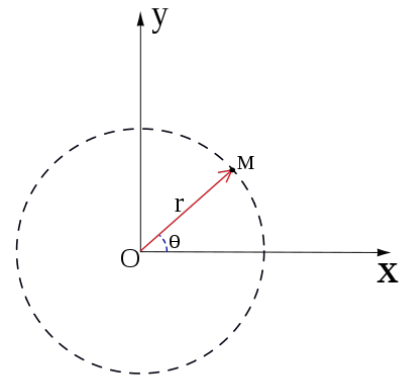
# Retour de plusieurs valeurs

Dans le cas où une méthode doit renvoyer plusieurs valeurs (atomiques et/ou des références), le concepteur de la méthode peut créer une classe permettant *d'envelopper les résultats dans une unique instance*.

Par exemple, si on veut retourner au programme appelant

- la distance à l'origine d'une instance de **PointPlan**, ainsi que
- l'angle que forme ce point avec l'axe des abscisses.

On crée une classe qui disposera de 2 variables d'instances dont le but sera de stocker chacun des 2 résultats (qui correspondent en fait aux coordonnées polaires du point).



## Implémentation

Définition de la classe « résultat » :

```
public class PointPolaire
{
    // Stocke la distance à l'origine
    private double module ;
    // Stocke l'angle avec l'axe des abscisses
    private double argument ;

    public PointPolaire( double mod, double arg){ . . . }
    public void setModule(double mod) { . . . }
    public double getModule(){ . . . }
    public void setArgument(double arg) { . . . }
    public double getArgument(){ . . . }
}
```



Définition de la méthode de calcul des résultats (dans la classe **PointPlan**)

```
public class PointPlan
{
    . . .
    // retourne les 2 coord polaires dans un objet PointPolaire
    public PointPolaire coorPolaires() {

        // calcul du module
        double module = Math.sqrt(
            Math.pow( this.getAbscisse() , 2) +
            Math.pow( this.getOrdonnée() , 2) ) ;

        // calcul de l'argument
        double argument = Math.atan(
            this.getOrdonnée() / this.getAbscisse() ) ;

        // Retour des résultats stockés dans un objet PointPlolaire
        return new PointPolaire( module, argument) ;
    }
}
```



# Retour de plusieurs valeurs

## Appel/Utilisation

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

        // Définition d'un premier point
        PointPlan pOri = new PointPlan( 7., 5.) ;

        // Déclaration d'une variable pour stocker la référence
        // de l'objet contenant les résultats qui sera renvoyée
        // par la méthode.
        PointPolaire pol;

        // Appel de la méthode de calcul sur le premier point et
        // affectation de la référence de l'objet contenant les
        // résultats des calculs à une variable
        pol = pOri.coorPolaires() ;

        // Extraction/accès aux résultats des différents calculs :

        // accès à l'angle:
        double angle = pol.getArgument() ;

        // accès à la distance:
        double distance = pol.getModule() ;

    }
}
```



Schéma :

# Les Chaînes de caractères

Dans un programme Java, un littéral *chaîne de caractères* donne lieu à la création implicite un objet de type **String**. La variable à laquelle est affecté le littéral contient en fait une référence vers l'objet de type **String** créé.

Quand le compilateur rencontre le littéral "bonjour" :

- 1) il crée un objet de type **String** contenant la valeur "bonjour"
- 2) il associe la référence de cet objet au littéral "bonjour"

La concaténation de deux chaînes par l'opérateur + donne lieu à une 3<sup>ème</sup> instance :

Schéma :

Cette instance de la classe **String** possède :

- une variable d'instance privée (**final**: c'est une constante)
- un constructeur par copie **public String( String s )**
- une méthode **public int length()** qui permet de connaître la longueur de la chaîne.
- une méthode **public boolean equals( String s )** qui permet de comparer le contenu de 2 chaînes.

Soit le code suivant :

```
{  
    String s1 = « bonjour » ;  
    String s2 = new String( s1 ) ;  
}
```

Compare les références

```
if( s1 == s2 )  
    System.out.println( 'identiques' ) ;  
else  
    System.out.println( 'différentes' ) ;
```

Affichage

« différentes »

Compare le contenu des chaînes

```
if( s1.equals(s2) )  
    System.out.println( 'identiques' ) ;  
else  
    System.out.println( 'différentes' ) ;
```

Affichage

« identiques »



# Les Chaînes de caractères

La méthode **toString()** est la méthode générique retournant une chaîne de caractères représentant le contenu d'un objet.

Elle doit être définie dans chaque classe.

```
public class PointPlan
{
    // Les attributs sont privés
    private double abscisse ;
    private double ordonnée ;

    // Affichage du contenu de l'objet
    public String toString() {
        return( ''x = '' + this.getAbscisse() + ', y = '' +
this.getOrdonnée()) ;
    }
}
```



Cette méthode peut-être appelée explicitement, mais elle est aussi appelée implicitement à chaque fois qu'une variable référence (de n'importe quelle classe) apparaît

- en argument de **System.out.println()** ;
- comme terme dans une expression de concaténation

## Appel explicite

```
{
    PointPlan p ;
    p = new PointPlan() ;
    System.out.println(p.toString()) ;
}
```

```
{
    PointPlan p ;
    p = new PointPlan() ;
    String s = ''Pt:'' + p.toString();
}
```

## Appel implicite (usuel)

```
{
    PointPlan p ;
    p = new PointPlan() ;
    System.out.println(p) ;
}
```

```
{
    PointPlan p ;
    p = new PointPlan() ;
    String s = ''Pt:'' + p;
}
```

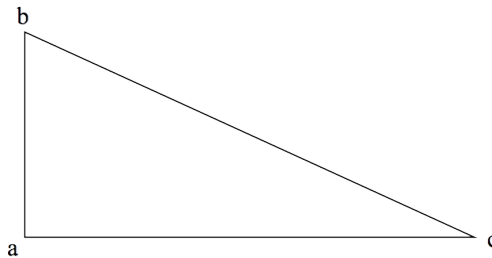
# Les objets composés

La programmation objet a pour intérêt d'organiser les données dans des instances, de composer de nouveaux objets à partir de l'union d'autres objets plus simples.

Il est donc plus que fréquent que les variables d'instance d'un objet fassent référence à des instances d'autres objets. On parle alors d'objets composés (associés à d'autres objets).

## Exemple :

Il est naturel de considérer un triangle comme étant un objet composé de 3 sommets.



Si l'on dispose d'une classe **PointPlan**, alors le triangle sera défini à partir de 3 instances de **PointPlan** qui joueront respectivement le rôle de chacun des sommets.

Nous aurons donc 4 objets :

- 3 objets **PointPlan** (un pour chaque sommet)
- 1 objet **Triangle** qui unira les 3 **PointPlan** dans un triangle.

```
public class Triangle
{
    // Variable d'instance faisant référence à des instances
    // d'une autre classe

    private PointPlan a ;
    private PointPlan b ;
    private PointPlan c ;

    // Constructeurs
    ...

    // getters, setters et autres méthodes
    ...
}
```




Schéma :

# Les objets composés


## La classe Triangle crée les sommets

```
public class Triangle
{
    // Variable d'instance
    private PointPlan a ;           // sommet a
    private PointPlan b ;           // sommet b
    private PointPlan c ;           // sommet c

    // Constructeurs
    public Triangle(    double x_a, double y_a, double x_b, double y_b,
                        double x_c, double y_c) {
        this.setA( new PointPlan( x_a, y_a)) ;
        this.setB( new PointPlan( x_b, y_b)) ;
        this.setC( new PointPlan( x_c, y_c)) ;
    }
}
```



```
public class TestTriangle
{
    public static void main (String [] args)
    {
        // Définition d'un premier point
        Triangle t = new Triangle( 0., 0., 0., 3., 6., 0.) ;
    }
}
```



C'est alors la classe **Triangle** qui a la *responsabilité* des **PointPlan** (définition, destruction, ...). A priori, on ne peut accéder à ces points qu'en passant par les services (vue publique) de la classe **Triangle**.

Schéma :

# Les objets composés

## Les sommets sont créés en dehors de la classe Triangle

```
public class Triangle
{
    // Variable d'instance
    private PointPlan a ;           // sommet a
    private PointPlan b ;           // sommet b
    private PointPlan c ;           // sommet c

    // Constructeurs
    public Triangle( PointPlan p_a, PointPlan p_b, PointPlan p_c)
    {
        this.setA( p_a );
        this.setB( p_b );
        this.setC( p_c );
    }
}
```



```
public class TestTriangle
{
    public static void main (String [] args)
    {
        PointPlan p1 = new PointPlan( 0., 0.) ;
        PointPlan p2 = new PointPlan( 0., 3.) ;
        PointPlan p3 = new PointPlan(6., 0.) ;
        Triangle t = new Triangle( p1, p2, p3) ;

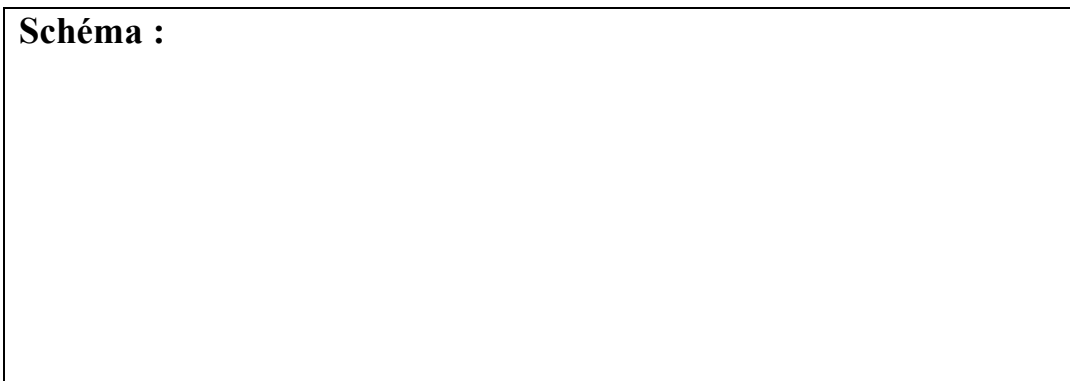
        p3.setAbscisse( 10.) ;      // modifie le triangle
    }
}
```



Dans ce cas c'est la classe qui instancie les **PointPlan** qui en a la **responsabilité**.

Potentiellement ces points peuvent être utilisés pour définir d'autres objets, et il est possible d'y accéder sans passer par les services (vue publique) de la classe triangle.

Schéma :



# Constructeur par copie

Nous avons déjà introduit la notion de constructeur. Cette méthode particulière qui porte le nom de la classe et renvoie une référence vers une nouvelle instance créée par l'appel à l'opérateur **new**.

Nous avons détaillé 2 types de constructeurs.

- *Le constructeur champs à champs*, qui présente autant de paramètres qu'il y a de variables d'instance à initialiser,
- *Le constructeur par défaut*, qui ne prend aucun paramètre et attribut des valeurs par défaut à chaque variable d'instance.
- 

Nous allons ici introduire un troisième constructeur, *le constructeur par copie*. Ce dernier admet pour paramètre une instance de la même classe que l'instance à créer. L'instance passée en paramètre sert d'exemple/modèle. La nouvelle instance créée sera une copie (valeurs identiques) de l'instance passée en paramètre :

```
public class PointPlan
{
    // Variable d'instance -ie attributs
    private double abscisse ;
    private double ordonnée ;

    // Constructeur Champ à Champ : Initialise un nouveau
    // point avec les coordonnées passées en paramètre
    public PointPlan( double x, double y) {
        this.setAbscisse( x) ;
        this.setOrdonnée( y) ;
    }

    // Constructeur par copie: Initialise un nouveau point avec des
    // coordonnées identique à celles du point passé en paramètre
    public PointPlan(PointPlan p) {
        abscisse = p.abscisse ;
        ordonnée = p.ordonnée ;
    }
}
```



Dans le corps d'instruction du constructeur par copie on évite d'appeler les accesseurs ou les autres constructeur () qui peuvent avoir en charge d'autre traitement.

On préfère utiliser des affectations en travaillant directement avec les variables d'instance.

```
public PointPlan(){
    this(0,0);
}

public PointPlan( PointPlan p ){
    this.setAbscisse(p.getAbscisse());
    this.setOrdonnée(p.getOrdonnée());
}
```

# Objets Composés et Constructeur par copie

Dans le cas où une classe est composée d'autres instances (certaines variables sont des variables référence vers un objet), **il est nécessaire de propager la copie aux instances référencées**

```
public class Triangle
{
    private PointPlan a ;           // sommet a
    private PointPlan b ;           // sommet b
    private PointPlan c ;           // sommet c

    public Triangle( PointPlan p_a, PointPlan p_b, PointPlan p_c) {
        this.setA( p_a) ;
        this.setB( p_b) ;
        this.setC( p_c) ;
    }

    // !!!!! A NE PAS FAIRE – INCORRECT !!!!!
    // public Triangle( Triangle t ) {
    //     a = t.a ;
    //     b = t.b ;
    //     c = t.c ;
    // }

    public Triangle( Triangle t ) {
        a = new PointPlan( t.a) ;
        b = new PointPlan( t.b) ;
        c = new PointPlan( t.c) ;
    }
}
```



Schéma :

# Les tableaux

## Définition

Un tableau est

- une structure de données contenant un ensemble d'éléments de *même type*,
- *de taille constante* définie lors de l'instanciation dynamique du tableau,
- dont chaque élément est associé à un *indice de position* permettant d'y accéder.

Comme dans beaucoup de langage la numérotation des positions commence à 0 et va jusqu'à la *taille\_du\_tableau-1*

<b>Indice :</b>	0	1	2	...	n-2	n-1
<b>Contenu :</b>	5.	-3.7	10.5	...	-8.0	3.

Le contenu d'un tableau peut être constitué

- de valeurs d'un des types primitifs (toujours du même type),
- de références vers des instances (toujours de même classe ou sous classe)

## Syntaxe

La création d'un tableau se déroule en 3 étapes principales :

```
1 {
2   // 1- Déclaration/création d'une variable référence sur le tableau
3   int [] tab;
4
5   // 2- Instanciation/création du tableau
6   tab = new int [4] ;
7
8   // 3- Initialisation des valeurs du tableau
9   for ( int i = 0 ; i < tab.length ; i++)
10  {
11      tab[i] = 2 * i + 1;
12  }
13 }
```



- ligne 6 : L'opérateur **new** est appelé sans faire référence à un constructeur. Lors de cette instanciation, *le tableau est initialisé avec un contenu par défaut* : la valeur 0 si il s'agit d'un tableau d'entiers.
- ligne 9 : **length** est une variable d'instance !! **publique** !! contenant le nombre d'éléments du tableau qui est établi une fois pour toute (constant **final**) à la création du tableau par l'appel à **new**.

Schéma :

# Les tableaux

## Initialisation lors de la déclaration

- Il est possible de donner directement la liste des valeurs du tableau lors de sa déclaration

```
1 {  
2     // 1- Déclaration/initialisation  
3     int [] tab = {1, 3, 9, -4};  
4 }
```

- Le **new** est implicite,
- La longueur (**length**) est fixée par le nombre d'éléments dans la liste.

## Paramétrage de la taille

- Les tableaux sont créés *dynamiquement*. Leur taille n'a donc pas à être connue lors de la programmation. Elle doit seulement être connue (et fixée définitivement) lors de la création (instanciation) du tableau.
- Ainsi on peut avoir :

```
{  
    Scanner sc = new Scanner( System.in) ;  
    int taille = sc.nextInt() ;  
    int [] tab = new int [taille];  
  
    // ensuite on procède normalement à l'initialisation  
    // des valeurs du tableau  
    ...  
}
```

## Syntaxe

Très intuitivement, pour faire appel à la *i*ème valeur du tableau ou pour modifier cette valeur on utilise la notation indicée entre crochet :

```
{  
    int [] tab = {1, 3, 9, -4};  
    System.out.println( tab[2]) ;  
    tab[2] = 15 ;  
    System.out.println( tab[2]) ;  
}
```



Affiche successivement les valeurs **9** puis **15**.




# Les tableaux de références

Il permet de maintenir, gérer, accéder à un ensemble d'objets du même type.  
Il se crée de façon sensiblement similaire aux tableaux de valeurs primitives :

## Syntaxe

Tableau de **PointPlan**.

```
1 {  
2     // 1- Déclaration/création d'une variable référence sur le tableau  
3     PointPlan [] t;  
4  
5     // 2- Instanciation/création du tableau  
6     t = new PointPlan [3] ;  
7  
8     // 3- Initialisation des valeurs du tableau  
9     for ( int i = 0 ; i < t.length ; i++)  
10    {  
11        t[i] = new PointPlan(i, 2*i);  
12    }  
13 }
```



- ligne 6 : le tableau est créé avec des valeurs par défaut : la référence **null**. Le tableau ne contient à ce stade aucun objet **PointPlan**. Il fait les instancier et placer leurs références dans le tableau.
- ligne 9 à 12 : Le contenu des cases du tableau est initialisé avec les références des objets **PointPlan** instanciés par l'appel itératif à l'opérateur **new**.

## Interaction avec les instances du tableau

Très intuitivement, on accède à l'instance par la référence stockée dans le tableau et on peut appliquer une méthode sur cette instance :

```
{  
    t[2].setOrdonnée(3.);  
    System.out.println( t[1].getAbscisse() );  
}
```




Schéma :

# Passage par référence

Comme dans beaucoup de langages, le passage d'un tableau en paramètre d'une méthode se fait par référence. Si un tableau est passé en paramètre d'une méthode, les modifications apportées au contenu du tableau dans la méthode sont répercutées dans le tableau du programme appelant.

```
public class Test
{
    public static void reset( PointPlan [] tab) {
        for( int i = 0 ; i < tab.length ; i++)
        {
            tab[i].setAbscisse(0.) ;
            tab[i].setOrdonnée(0.) ;
        }
    }

    public static void main( String [] args)
    {
        PointPlan [] tab = new PointPlan [3] ;

        System.out.println( '' Avant : '' ) ;
        for ( int i = 0 ; i < tab.length ; i++ ){
            tab[i] = new PointPlan( i, 2*i ) ;
            System.out.println( tab[i] ) ;
        }

        Test.reset( tab ) ;
        System.out.println( '' Après: '' ) ;
        for ( int i = 0 ; i < tab.length ; i++ ){
            System.out.println( tab[i] ) ;
        }
    }
}
```



Schéma :

# Tableau - variable d'instance

De la même façon qu'il existe des variables référençant des tableaux dans le corps des programmes, il est possible et fréquent de définir des variables d'instance de type tableau. Ces tableaux en variable d'instance peuvent être des tableaux de type primitif comme des tableaux de référence

```
public class Triangle {  
  
    // Déclaration d'un variable d'instance qui est tableau  
    private PointPlan [] sommets ;  
  
    public Triangle( PointPlan [] sommets ) {  
        this.setSommets( sommets )  
    }  
  
    public void setSommets( PointPlan [] sommets ) {  
        this.sommets = sommets ;  
    }  
}
```



```
public class Test {  
    public static void main( String [] args) {  
        PointPlan [] som = new PointPlan [3] ;  
  
        for( int i = 0 ; i < som.length ; i++ )  
            som[i] = new PointPlan( i, 2 * i +1) ;  
  
        Triangle t = new Triangle( som ) ;  
    }  
}
```




Schéma :

# Tableau - valeur de retour

De la même façon, une méthode peut renvoyer un tableau comme valeur de retour. Cela revient alors à renvoyer la référence du tableau au programme appelant.

```
public class Triangle {  
    [ . . . ]  
  
    // Retourne le tableau de références des objets PointPlan  
    // constituant les sommets du triangle  
    public PointPlan [] getSommets() {  
        return this.sommets ;  
    }  
    // Retourne un tableau des valeurs des abscisses des sommets  
    // du triangle  
    public PointPlan [] getAbscissesSommets() {  
        double [] ab = new double [ this.getSommets().length ] ;  
        for( int i = 0 ; i < this.getSommets().length ; i++ )  
            ab[i] = this.getSommet(i).getAbscisse() ;  
        return ab ;  
    }  
}
```



**Attention :** Si l'on modifie les valeurs de ce tableau dans le programme appelant, c'est tout le **Triangle** qui peut être modifié !!

```
public class Test {  
    public static void main( String [] args) {  
        PointPlan [] som = new PointPlan [3] ;  
        for( int i = 0 ; i < 3; i++ )  
            som[i] = new PointPlan( i, 2 * i +1) ;  
        Triangle t = new Triangle(som) ;  
  
        PointPlan [] tab = t.setSommets() ;  
        tab[0].translate( 10.,10.);  
  
        double [] abscisses = t.getAbscissesSommets() ;  
    }  
}
```




Schéma :

# Bonnes Pratiques

```
public class Triangle {  
  
    private PointPlan [] sommets ;  
  
    public Triangle( PointPlan [] sommets ) {  
        this.setSommets( sommets ) ;  
    }  
  
    // Appel au constructeur par défaut de PointPlan  
    public Triangle() {  
        this( new PointPlan [3] ) ;  
        for( int i = 0 ; i < this.getSommets().length; i++ )  
            this.setSommet( i, new PointPlan() ) ;  
    }  
  
    // Copie profonde et appel au constructeur par copie de PointPlan  
    public Triangle( Triangle t ) {  
        this.sommets = new PointPlan [3] ;  
        for( int i = 0 ; i < this.sommets.length; i++ )  
            this.sommets[i] = new PointPlan( t.sommets[i] ) ;  
    }  
  
    public void setSommets( PointPlan [] sommets ) {  
        this.sommets = sommets ;  
    }  
  
    public PointPlan [] getSommets() {  
        return this.sommets ;  
    }  
  
    public void setSommet(int i, PointPlan p) {  
        this.sommets[i] = p ;  
    }  
  
    public PointPlan getSommet(int i) {  
        return this.sommets[i] ;  
    }  
  
    // Appel implicite de toString() de PointPlan  
    public String toString() {  
        String s = ''Triangle('' ;  
        for( int i = 0 ; i < this.getSommets().length ; i++ )  
            s = s + this.getSommet(i);  
        // s = s + this.getSommet(i).toString() ;  
        s = s + ''')'' ;  
        return s ;  
    }  
}
```



# Exercices #2

---

On suppose que vous disposez de la classe **Vaisseau** précédemment définie et de la classe développée dans le cours.

**Question 1 :** Définissez une classe **VolEnFormation** composée d'un tableau d'objets **Vaisseau**, d'une position définie par une abscisse et une ordonnée et une altitude de vol souhaitée. Vous définirez les variables d'instance.

- La responsabilité des **Vaisseau** revient à la classe qui instancie le **VolEnFormation**

Vous pourrez choisir d'implémenter la position au moyen d'un **PointPlan**.

**Question 2 :** Définissez des accesseurs publics aux variables d'instance dont la classe **VolEnFormation** à la responsabilité. Il devra en outre être possible d'accéder directement à n'importe quel **Vaisseau** du **VolEnFormation**.

**Question 3 :** Définissez le constructeur champ à champ de la classe **VolEnFormation**.

**Question 4 :** Définissez une méthode publique **nbDeVaisseaux()** qui retourne le nombre de constitutifs de la formation.

**Question 5 :** Augmentez chacune des classes **VolEnFormation**, **Vaisseau** et **PointPlan** d'une méthode **toString()**. Celles-ci vous serviront désormais lors des tests de vos classes. La méthode **toString()** de la classe **VolEnFormation** appellera évidemment la méthode **toString()** des instances de **Vaisseau** qui la compose.

**Question 6 :** Définissez une méthode privée **altitudeValide()**, qui renvoie un booléen : **True** si l'altitude est strictement supérieur à 0m et **False** sinon. Cette méthode sera appelée dans le corps de la méthode d'accès en écriture (setter) de la variable définissant l'altitude. Si l'altitude transmise au setter n'est pas valide, alors elle est fixée arbitrairement à une valeur seuil de 1000m.

**Question 7 :** Définissez une méthode publique **appliqueAltitudeDeVol()** qui lorsqu'elle est appelée fixe l'altitude effective de vol des vaisseaux de la formation à l'altitude souhaitée pour le vol en formation.

**Question 8 :** Définissez une méthode **altitudesDesVaisseaux()** de la classe **VolEnFormation** qui retourne la liste des altitudes des vaisseaux qui composent la formation.

**Question 9 :** Définissez une classe **TestVolEnFormation** dans laquelle vous définirez complètement une formation en vol composées de 3 **Vaisseau** ayant des altitudes de vol différentes. Vous fixerez ensuite l'altitude souhaitée de vol pour la formation à 167000m au

moyen de l'accessor approprié. Le programme récupérera les altitudes des **Vaisseau** au moyen de la méthode **altitudesDesVaisseaux** et les comparera à l'altitude fixée pour la formation. Si l'une des altitudes des **Vaisseau** est différentes de l'altitude fixée pour le vol en formation alors cette dernière sera transmise et appliquée à chaque **Vaisseau**.

Au moyen d'affichages, vous procéderez à tous les contrôles nécessaires pour vérifier le bon comportement de votre programme. Notamment, vous modifierez la catégorie d'un des vaisseaux depuis le programme principal et vous vérifierez qu'elle est aussi modifiée dans le vol en formation.

**Question 10 :** Définissez les constructeurs par copie des classes **Vaisseau** et **VolEnFormation**.

**Question 11 :** Dans la classe **TestVolEnFormation** proposez un jeu d'instruction permettant de tester les constructeurs par copie

### *Exercices complémentaires*

**Question 12 :** Dessinez les relations entre les objets en présence et leur évolution. Dans la classe **VolEnFormation** :

**Question 13 :** Définissez une méthode **déplacement()** qui permet de modifier la position du vol en formation en passant en paramètre de combien il faut modifier l'abscisse et l'ordonnée de la position.

**Question 14 :** Définissez une méthode **destruction()** qui prend en paramètre un entier. Si l'entier saisi est 1 (resp. 2), le premier (resp. second) **Vaisseau** est détruit (la référence dans le vol en formation doit être la référence **null**).

**Question 15 :** Définissez la Javadoc de toutes ces classes