# Exercices #3

## Variables de classe et méthodes de classe et constantes

Ces exercices s'appuient sur les classes **`Vaisseau`** et **`VolEnFormation`** développées à l'issue du TP2. Vous pouvez télécharger l'archive seance2_fin.tgz contenant ces classes à l'adresse suivante :

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

Chaque vaisseau est doté d'un numéro de châssis. Celui-ci est un identifiant unique qui ne peut être modifié. Nous nous proposons d'implémenter cet élément.

**Question 1 :**
Définissez une constante d'instance **`NUMÉRO_DE_CHÂSSIS`** qui définira le numéro du châssis et ses accesseurs si elle doit en avoir

*Correction :*
*Pas de méthode d'accès en écriture (setter) puisqu'il s'agit d'une constante. Par contre il faut pouvoir consulter le numéro de châssis : il faut donc une méthode publique d'accès en lecture.*

```
public class Vaisseau
{
     ...
     // -----------------------------------------------------
     // Constantes d'instance
     private final int NUMÉRO_DE_CHÂSSIS;

     /**
      * Retourne le nombre du châssis du Vaisseau
      * @return le numéro de châssis
      */
     public int getNuméroDeChâssis()
     {
          return this.NUMÉRO_DE_CHÂSSIS ;
     }
}
```

Pour définir le numéro de châssis de chaque instance nous allons introduire une variable de classe **`prochainNuméroDeChâssis`** de **`Vaisseau`** et ses accesseurs.
Lors de chaque instanciation, cette variable de classe sera :

1/ lue. La valeur lue sera attribuée comme numéro de châssis à la nouvelle instance

2/ incrémentée de façon à ce qu'a l'instanciation suivante, le numéro de châssis soit différent et augmenté de +1.

**Question 2 :**
Définissez la variable de classe **`prochainNuméroDeChâssis`** et ses accesseurs.

*Correction :*

```
// ------------------------------------------------
// Variable de classe
private static int prochainNuméroDeChâssis = 0 ;
```

**Question 3 :**
Définissez la méthode d'accès à la variable de classe **`getProchainNuméroDeChâssis()`**.

*Correction :*
*Les méthodes interagissant avec des variables de classe sont des méthodes de classe.*

```
// ------------------------------------------------
// Variable de classe
private static int prochainNuméroDeChâssis = 0 ;

// ------------------------------------------------
// Méthode de classe

// Méthode d'accès privée (utilisée seulement dans le
// constructeur) d'accès au prochain numéro de châssis
private static int getProchainNuméroDeChâssis()
{
     return Vaisseau.prochainNuméroDeChâssis;
}
```

**Question 4 :**
Définissez la méthode d'accès à la variable de classe **`incrémenteProchainNuméroDeChâssis()`** qui permet d'incrémenter la variable de classe.

*Correction :*
*Attention, ces méthodes n'ont pas a être publique. Elles ne doivent surtout pas être accessibles aux utilisateurs de la classe.*

```
// ------------------------------------------------
// Méthode de classe

// Méthode privée (sutilisée seulement dans le constructeur
// d'incrémentation du prochain numéro de châssis
```

```
    private static void incrémenteProchainNuméroDeChâssis()
    {
        Vaisseau.prochainNuméroDeChâssis ++;
    }
```

**Question 5 :**
Modifiez le code des constructeurs pour que l'attribution des numéros de châssis soit prise en compte lors de l'instanciation.

*Correction :*
*Si l'ensemble des constructeur appels le constructeurs champs à champs, directement ou indirectement il n'es pas nécessaire de tous les modifier. Il suffit de changer le constructeur champs à champs :.*

```
public Vaisseau( String cat, int nbPass, double alt)
{
    this.nbMaxPassagers = nbPass ;
    this.catégorie = cat ;
    this.altitude = alt ;
    this.NUMÉRO_DE_CHÂSSIS = Vaisseau.getProchainNuméroDeChâssis();
    Vaisseau.incrémenteProchainNuméroDeChâssis();
}

public Vaisseau( Vaisseau v ) {
    this.nbMaxPassagers = v.nbMaxPassagers ;
    this.catégorie = v.catégorie ;
    this.altitude = v.altitude ;
    this.NUMÉRO_DE_CHÂSSIS = v.NUMÉRO_DE_CHÂSSIS ;
    // on n'incrémente surtout pas le prochain numéro de châssis.
}
```

**Question 6 :**
Modifiez la méthode **`toString()`** pour qu'elle intègre le numéro de châssis.

*Correction :*
```
public String toString()
{
    String desc = "\n\n" ;
    desc += "        /- Vaisseau - Châssis : " + getNuméroDeChâssis()
+ "\n" ;
    desc += "     -------------\n" ;
    desc += "  /  Capacité  = " + this.getNbMaxPassagers() + "
passagers\n" ;
    desc += "<    Altitude  = " + this.getAltitude() + " mètres\n" ;
    desc += "  \\  Catégorie = " + this.getCatégorie() + "\n" ;
    desc += "     -------------\n" ;
    desc += "        \\- Vaisseau - Châssis : " +
getNuméroDeChâssis() + "\n" ;
    return desc ;
```

```
}
```

**Question 7 :**
Si ce n'est pas déjà fait définissez un programme de test pour vérifier le bon fonctionnement de ces nouvelles fonctionnalités

*Correction :*

```
public class TestChassisVaisseau
{
     public static void main(String [] args)
     {
          Vaisseau xwingT65 = new Vaisseau("Chasseur Léger", 2, 10000)
;
          Vaisseau stalker = new Vaisseau("Vaisseau Lourd", 46785,
1800000) ;
          Vaisseau stalkerBIS = new Vaisseau(stalker) ;

          System.out.println( xwingT65);
          System.out.println( stalker);
          System.out.println( stalkerBIS);

          System.out.println( xwingT65.getNuméroDeChâssis());
          System.out.println( stalker.getNuméroDeChâssis());
          System.out.println( stalkerBIS.getNuméroDeChâssis());

          for ( int i = 2; i < 1000; i++)
          {
               stalker = new Vaisseau();
          }
          System.out.println( stalker.getNuméroDeChâssis());
     }
}
```

# Héritage

On souhaite définir une classe **`CoordonnéesSpatiale`** décrivant un point dans un espace à 3 dimensions qui utiliserait la classe **`PointPlan`** que l'on a l'habitude de manipuler:
Pour définir la classe **`CoordonnéesSpatiale`** on s'appuie sur le principe d'héritage. Un premier essai conduit à l'implémentation d'une classe **`CoordonnéesSpatiale`** suivante.

```
public class CoordonnéesSpatiale extends PointPlan
{
      private double cote  ;

      public Point3D() {
            super() ;
      }
```

```
	public Point3D(double x, double y, double z) {
		this.abscisse = x ;
		this.ordonnée = y ;
		this.cote     = z ;
	}

	public Point3D(Point3D p) {
		this.abscisse = p.abscisse ;
		this.ordonnée = p.ordonnée ;
		this.cote     = p.cote ;
	}

	public double getAbscisse() {
		return this.abscisse ;
	}

	public double getOrdonnée() {
		return this.ordonnée ;
	}

	public double getCote() {
		return this.cote ;
	}
}
```

**Question 8 :** Énumérez toutes les anomalies de conception de la classe **`CoordonnéesSpatiale`** ainsi que les instructions rejetées par le compilateur. Justifier chaque réponse.

*Correction :*
Les 2 constructeurs (champ à champ et par copie) ne peuvent affecter une valeur aux variables d'instances privées de la classe mère par le code :

```
	this.abscisse = x ;
	this.ordonnée = y ;
```

Il y a rejet du compilateur. Pour faire cela il faut utiliser le constructeur de la classe mère **`super()`**

Par exemple, une définition correcte du constructeur champ a champ serait

```
public CoordonnéesSpatiale ()
{
	super( x, y) ;
	this.cote = z ;
}
```

De plus il est parfaitement INUTILE de redéfinir les méthodes **`getAbscisse()`** et **`getOrdonnée()`** dans la classe **`CoordonnéesSpatiale`**, puisque cette dernière en hérite de la classe mère **`PointPlan`**.

Pour la suite du TP vous devez récupérer les fichiers définissant les classes **`Passager`** **`Mécanicien`** et **`Pilote`**. Ces fichiers sont disponibles en téléchargement dans l'archive seance3_debut.tgz sur le site :

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

**Question 9 :**

Écrivez une classe **`TestPersonne`** pour :

- Créer deux passagers, changer le numéro de billet de l'un d'eux et changer l'âge de l'autre
- Créer deux mécaniciens, transférer l'un d'eux dans un nouveau service et changer l'âge de l'autre
- Créer deux pilotes, ajouter 2 heures de vol à l'un d'eux et changer l'âge de l'autre.

*Correction :*

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

          // ------------------------------------------------------
          // Les passagers

          Passager passa1 = new Passager("John", 48, 45736 ) ;
          Passager passa2 = new Passager("Doe",  47, 45743 ) ;

          System.out.println( passa1 );
          System.out.println( passa2 );

          passa1.setNuméroDeBillet( 45744 ) ;
          passa2.setAge( 43 );

          System.out.println( passa1 );
          System.out.println( passa2 );

          // ------------------------------------------------------
          // Les mécaniciens

          Mécanicien meca1 = new Mécanicien("Newton", 48, 1100187736,
"Maintenance", "Fluide") ;
          Mécanicien meca2 = new Mécanicien("Isaac", 29, 110675423,
"Maintenance", "Façonage" ) ;

          System.out.println( meca1 );
          System.out.println( meca2 );

          meca1.setService( "Pont d'envol" ) ;
          meca2.setAge( 43 );

          System.out.println( meca1 );
```

```
          System.out.println( meca2 );

          // ------------------------------------------------------
          // Les pilotes

          Pilote pilote1 = new Pilote("Isaac", 67, 110675423,
"Flibuste", 12567 ) ;
          Pilote pilote2 = new Pilote("Newton", 22, 1100187736,
"Flotte", 554) ;

          System.out.println( pilote1 );
          System.out.println( pilote2 );

          pilote2.setNombreHeuresDeVol( 2 +
pilote2.getNombreHeuresDeVol() );
          pilote1.setAge( 29 ) ;

          System.out.println( pilote1 );
          System.out.println( pilote2 );
     }
}
```

**Question 10 :**
Identifiez ce que les classes **`Mécanicien`** et **`Pilote`** ont en commun.

*Correction :*

```
private int niveauGris ;
private String nom ;
private int    age ;
private int    numéroProfessionnel ;
private String service ;

public void setNom( String n ) { ... }
public void setAge(    int a ) { ... }
public void setNuméroProfessionnel ( int n ) { ... }
public void setService( String s ) { ... }

public String getNom( ) { ... }
public    int getAge( ) { ... }
public    int getNuméroProfessionnel ( ) { ... }
public String getService( ) { ... }
```

*Ainsi qu'une partie du code de toString()*

**Question 11 :**
Une classe **`Personnel`** permettrai de regrouper (ou factoriser) des propriétés de ces deux classes. Est-ce que ça a un sens de le faire ?

*Correction :*
*Définir une classe mère de* **`Personne`** *et* **`Voiture`** *au prétexte que les deux auraient un nom,*

*n'a aucun sens et doit être évité : on n'utilise l'héritage que parce que c'est utile, d'une part, **et** que ça a un sens, d'autre part.*

**Question 12 :**
Écrire une classe **`Personnel`** pour généraliser **`Mécanicien`** et **`Pilote`**.

*Correction :*

```
public class Personnel {

     private String nom ;
     private int    age ;
     private int    numéroProfessionnel ;
     private String service ;

     public void setNom( String n ) { this.nom = n ; }
     public void setAge(    int a ) { this.age = a ; }
     public void setNuméroProfessionnel ( int n ) {
this.numéroProfessionnel = n ; }
     public void setService( String s ) { this.service = s ; }


     public String getNom( ) { return this.nom ; }
     public    int getAge( ) { return this.age ; }
     public    int getNuméroProfessionnel ( ) { return
this.numéroProfessionnel ; }
     public String getService( ) { return this.service ; }

     public Personnel( String nom, int age, int numéroProfessionnel,
String service) {
          this.setNom( nom) ;
          this.setAge( age) ;
          this.setNuméroProfessionnel( numéroProfessionnel) ;
          this.setService( service );
     }

     public String toString() {
          String res = "Personnel[ " ;
          res += "nom : " + this.getNom() + ", ";
          res += "age : " + this.getAge() + ", ";
          res += "n° de professionnel : " +
this.getNuméroProfessionnel() + ", ";
          res += "service : " + this.getService() + "]" ;
          return res ;
     }
}
```

**Question 13 :**
Proposez le code modifié des classes **`Mécanicien`** et **`Pilote`** pour qu'elles tirent partie de l'héritage de la classe **`Personnel`**.

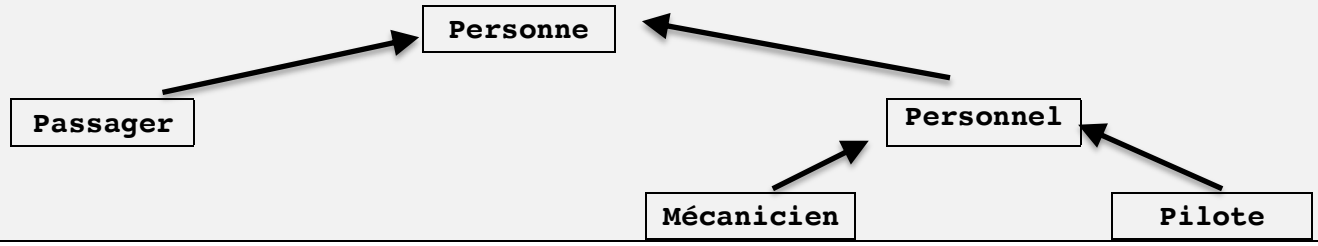
*Correction :*

```
public class Pilote extends Personnel {

     private int    nombreHeuresDeVol ;

     public void setNombreHeuresDeVol( int n ) {
this.nombreHeuresDeVol = n ; }

     public    int getNombreHeuresDeVol( ) { return
this.nombreHeuresDeVol ; }

     public Pilote( String nom, int age, int numéroProfessionnel,
String service , int nombreHeuresDeVol) {
          super( nom, age, numéroProfessionnel, service );
          this.setNombreHeuresDeVol( nombreHeuresDeVol) ;
     }

     public String toString() {
          String res = "Pilote[ " ;
          res += super.toString() + ", " ;
          res += "# heures de vol : " + this.getNombreHeuresDeVol() +
"]" ;
          return res ;
     }
}
```

```
public class Mécanicien
{
     // sur le même modèle que Pilote

}
```

**Question 14 :**
Proposez une hiérarchie de classe pour prendre en compte la classe **`Passager`**.

*Correction :*




**Question 15 :**
Définissez la classe **`Personne`** et modifiez le code des classes **`Passager`**, et **`Personnel`** pour tenir compte de cette nouvelle hiérarchie.

# Polymorphisme

On rappelle ici que l'ensemble des classes hérite de la classe **`Object`** certaines méthodes génériques. Notamment, parmi celles-ci, elles héritent de la méthode **`equals(Object )`** qui renvoie **`true`** si les deux instances ont la même référence et **`false`** sinon.

**Question 16 :**
Quels sont les éléments qui définissent la signature d'une méthode.

*Correction :*

La signature d'une méthode est constituée
- du nom de la classe dans laquelle elle est définie
- du nom de la méthode
- du nombre/type/ordre de ses paramètres

Le type de la valeur de retour ne fait pas partie de la signature.

Signature : **NomClasse ::nomMethode( type param, …)**

Considérons la classe **`Personne`** suivante :

```
public class Personne {
	private String nom ;
	private int age ;

	public Personne(String unNom, int unAge) {
		this.nom = unNom;
		this.age = unAge ;
	}
}
```

Et sa classe de test :

```
public class TestPersonne {
	public static void main(String[] args) {
		Personne p1 = new Personne("Toto", 30) ;
		Personne p2 = new Personne("Toto", 30) ;
		if(p1.equals(p2))
			System.out.println("EGALES") ;
		else
			System.out.println("DIFFERENTES") ;
	}
}
```

**Question 17 :**
Qu'affiche le programme de test ? Pourquoi ? Dessinez les références et les instances.

*Correction :*

**DIFFERENTES**

C'est **`Object::equals(Object)`** qui est appelé. **`equals()`** n'étant pas redéfinie, ce **`equals()`** de **`Object`** ne compare que les références

**Question 18 :**
Qu'affiche le programme de test si la méthode suivante est rajoutée dans la classe **`Personne`**. Justifiez votre réponse.

```
public boolean equals(Personne p)
{
	return (this.nom.equals(p.nom) && this.age==p.age) ;
}
```

*Correction :*

**EGALES**

Ici, on surcharge `equals` et comme **`p1`** et **`p2`** sont des **`Personne`**, c'est **`Personne::equals(Personne)`** qui est appelée.
*Attention :* pas de redéfinition ici car la signature est différente Il aurait fallu écrire **`public boolean equals(Object p)`** pour parler de redéfinition

**Question 19 :**
Qu'affiche le programme de test si ce dernier est modifié de la façon suivante.

```
public static void main(String[] args) {
	Personne p1 = new Personne("Toto", 30) ;
	Object p2 = new Personne("Toto", 30) ;
	if(p1.equals(p2))
		System.out.println("EGALES") ;
	else
		System.out.println("DIFFERENTES") ;
}
```

*Correction :*

**DIFFERENTES**

**`p2`** étant une référence à un **`Object`**, c'est une méthode avec pour signature **`equals(Object)`** qui est appelée. Cette méthode correspond à **`Object::equals(Object)`** héritée de **`Object`**

**Question 20 :**
Qu'affiche le programme de test si ce dernier est modifié de la façon suivante.

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

```
        Personne p1 = new Personne("Toto", 30) ;
        Object p2 = new Personne("Toto", 30) ;
        if(p1.equals((Personne)p2))
                System.out.println("EGALES") ;
        else
                System.out.println("DIFFERENTES") ;
}
```

*Correction :*

**EGALES**

Ici, avec le cast, c'est une méthode avec pour signature **`equals(Personne)`** qui est appelée. La méthode appelée est donc **`Personne::equals(Personne)`**, c'est à dire la surcharge de **`Object::equals(Object).`**

**Question 21 :**
Qu'affiche le programme de test si ce dernier est modifié de la façon suivante.

```
public static void main(String[] args)
{
        Object p1 = new Personne("Toto", 30) ;
        Personne p2 = new Personne("Toto", 30) ;
        if(p1.equals(p2))
                System.out.println("EGALES") ;
        else
                System.out.println("DIFFERENTES") ;
        }
}
```

*Correction :*

**DIFFERENTES**

Comme **`Personne::equals(Personne)`** n'est pas une redéfinition de **`Object::equals(Object)`**, elle ne masque pas cette dernière. En conséquence, **`p1`** étant un **`Object`** et aucun polymorphisme n'étant à l'oeuvre, c'est **`Object::equals(Object)`** qui est appelée.

**Question 23 :**
Proposez une implémentation correcte de la méthode **`equals(...)`** pour la classe **`Personne`**.

*Correction :*

```
public boolean equals(Object o)
{
        if ( ! ( o instanceof Personne ) )
                return false;
        Personne p = (Personne) o ;
        return ( this.getNom().equals( p.getNom() ) &&
                        this.getAge() == p.getAge() ) ;
```

```
}
```

## Exercices optionnels

Soient les 3 classes suivantes (données dans l'archive de départ) :

```
public class Vaisseau
{
	public void départ()
	{
		System.out.println("Je parts.");
	}
}
```

```
public class VaisseauAVoile extends Vaisseau
{
	public void départ()
	{
		System.out.println("Je mets les voiles.");
	}

	public void percute() {
		System.out.println("Je coule.");
	}
}
```

```
public class VaisseauDeLEspace extends Vaisseau
{
	public void départ()
	{
		System.out.println("Je lance les moteurs a fusion.");
	}

	public void percute()
	{
		System.out.println("Je dépressurise.");
	}
}
```

Soit le programme principal suivant :

```
1  public class TestVaisseau2 {
2      public static void main(String[] args) {
3
4              Vaisseau v1 ;
5              v1 = new Vaisseau() ;
6              v1.départ() ;
7              v1.percute() ;
8              v1 = new VaisseauDeLEspace() ;
9              v1.départ() ;
10             v1.percute() ;
```

```
11
12          VaisseauDeLEspace v2 = new VaisseauDeLEspace() ;
13          v2.départ() ;
14          v2.percute() ;
15          v2 = new Vaisseau() ;
16          v2.départ() ;
17          v2.percute() ;
18
19          v1 = (Vaisseau)( v2 ) ;
20          v1.départ() ;
21          v1.percute() ;
22
23          v2 = (VaisseauDeLEspace)( v1 ) ;
24          v2.départ() ;
25          v2.percute() ;
26
27          Vaisseau [] flotte=new Vaisseau [3];
28          flotte[0] = new VaisseauDeLEspace();
29          flotte[1] = new Vaisseau();
30          flotte[2] = new VaisseauAVoile();
31
32          for (int i = 0 ; i < flotte.length; i++)
33                flotte[i].départ();
34
35    }
36 }
```

**Question 24 :**
Dans le programme de test ci-dessus, quelles instructions posent problème ?

*Correction :*
**Les lignes 7, 10, 15, 21 ne passent pas la compilation**

**Question 25 :**
Sans les instructions problématiques, qu'afficherait le programme ?

*Correction :*

```
Je parts.
Je lance les moteurs a fusion.
Je lance les moteurs a fusion.
Je dépressurise.
Je lance les moteurs a fusion.
Je dépressurise.
Je lance les moteurs a fusion.
Je lance les moteurs a fusion.
Je dépressurise.
Je lance les moteurs a fusion.
Je parts.
Je mets les voiles.
```

**Question 26 :**
Dessinez l'arbre d'héritage complet des 3 classes.

*Correction :*


**Question 27 :**
Redéfinissez les méthodes de la classe **`Object`** pour chacune des 3 classes de Vaisseau.

*Correction :*
Pour la classe Vaisseau

```
Public class Vaisseau
{
   ...

   public boolean equals( Object o)
   {
      if ( ! ( o instanceof Vaisseau ))
           return false ;
      Vaisseau v = (Vaisseau) o ;
      return (  this.getCatégorie().equals(v.getCatégorie()) &&
                this.getAltitude() == v.getAltitude() &&
                this.getNbMaxPassagers() == v.getNbMaxPassagers()) ;
   }
}
```

Pour la classe VaisseauDeLEspace

```
public class VaisseauDeLEspace
{
   ...

   public boolean equals( Object o)
   {
      if ( ! ( o instanceof VaisseauDeLEspace))
           return false
      VaisseauDeLEspace v = (VaisseauDeLEspace) o ;
      return (  super.equals(v));
   }
}
```

Idem pour la classe VaisseauAVoile

**Question 28 :**
Modifiez la classe **`VolEnFormation`** pour qu'elle admette un nombre de Vaisseau compris entre 2 et 20 inclus.

*Correction :*
Pour la classe VolEnFormation

```
   // -------------------------------------------------
   // Constante de classe
   public static final int TAILLE_MIN = 2;
   public static final int TAILLE_MAX = 20;

   // -------------------------------------------------
   // Constructeurs

   public VolEnFormation(Vaisseau [] form, double posX, double posY,
double alt)
   {
      if ( form.length > VolEnFormation.TAILLE_MAX )
      {
           System.out.println( "Taille du tableau trop longue");
           System.out.println( "Seules les " + this.TAILLE_MAX + "
premières vaisseaux.");
      }
      else if ( form.length < VolEnFormation.TAILLE_MIN )
      {
           System.out.println( "Taille du tableau trop courte");
           System.out.println( "La fomration sera complétrée par des
Vaisseaux par default.");
           this.formation = new Vaisseau [VolEnFormation.TAILLE_MIN]; ;
           for (int i = 0; i < VolEnFormation.TAILLE_MIN; i++)
           {
                if ( i < form.length )
                     this.setVaisseau( i, form[i]);
                else
                     this.setVaisseau( i, new Vaisseau());
           }
      }
      else
      {
           this.formation = form;
      }
      this.position = new PointPlan(posX, posY, "Position");
      this.setAltitudeFixée( alt);
   }

   /**
    * Constructeur par copie
    * @param le model de vol en formation
```

```
    */
   public VolEnFormation( VolEnFormation v)
   {
      this.position = new PointPlan( v.position);
      this.altitudeFixée = v.altitudeFixée;
      this.formation = new Vaisseau [ v.formation.length ] ;
      for (int i = 0; i < this.formation.length; i++ )
           this.formation[i] = new Vaisseau( v.formation[ i ] ) ;
   }

   // ------------------------------------------------------------
   // Redéfinition de boolean equals( Object o )

   public boolean equals( Object o)
   {
      if ( ! ( o instanceof VolEnFormation ) )
           return false;

      VolEnFormation v = (VolEnFormation) o;

      if (    this.formation.length != v.formation.length ||
              ( ! this.position.equals(v.position)) ||
              this.getAltitudeFixée() != v.getAltitudeFixée() )
           return false;
      boolean all = true ;
      for (int i = 0; i < this.formation.length; i++)
           all = all && ( this.getVaisseau(i).equals(v.getVaisseau(i)))
;
      return all;

   }
}
```

**Question 29 :**
Il est possible de définir un vol en formation avec des **`VaisseauAVoile`**. Vous le testerez en proposant un programme de test définissant :

- une formation de 4 **`Vaisseau`**
- avec 2 **`Vaisseau`**génériques
- un **`VaisseauDeLEspace`**
- un **`VaisseauAVoile`**

Dans ce programme vous appliquerez les méthodes **`départ()`** et **`percute()`** à tous les vaisseaux de la formation. Est-ce possible ?

*Correction :*
La classe Vaisseau dispose bien d'une méthode départ() mais pas de méthode percute(). Seules ses fille l'on. Le programme ne compile donc pas (liaison statique) si on cherche à percuter l'enesemble des Vaisseaux de la formation qui sont déclarés comme des instance de la classe Vaisseau

```
public class TestVolEnFormation2
{
```

```
     public static void main( String [] args)
     {
          Vaisseau [] formation = new Vaisseau [4];
          formation[0] = new Vaisseau() ;
          formation[1] = new Vaisseau() ;
          formation[2] = new VaisseauDeLEspace();
          formation[3] = new VaisseauAVoile();

          VolEnFormation vol = new VolEnFormation( formation, 0., 0.,
100000.) ;

          for ( int i = 0; i < vol.getFormation().length; i++)
          {
               vol.getVaisseau(i).départ();
          }
     /*
          for ( int i = 0; i < vol.getFormation().length; i++)
          {
               vol.getVaisseau(i).percute();
          }
     */
     }
}
```

**Question 29 :**
Vérifiez que le constructeur par copie de la classe **`VolEnFormation`** fonctionne correctement.

*Correction :*

```
public class TestVolEnFormation2
{
     public static void main( String [] args)
     {
          ...
          VolEnFormation vol2 = new VolEnFormation( vol ) ;
          vol2.setVaisseau( 3, new VaisseauAVoile() ) ;

          System.out.println( vol  ) ;
          System.out.println( vol2 ) ;

          System.out.println( vol.equals( vol2 ) ) ;
          System.out.println( vol.getVaisseau(3).equals(
vol2.getVaisseau(3) ) ) ;
     }
}
```