

Langage Java Chapitre 6

Les modificateurs d'accès, le mot réservé **this**.

Les classes, l'héritage, le polymorphisme, le mot réservé **super**.

Les attributs/méthodes d'instance et de classe.

Les classes abstraites, les interfaces, les classes anonymes.

Les JavaBeans. Les pattern "template method" et "bridge".

1. Les modificateurs d'accès

La classe Individu

```
package agenda;

public class Individu {

    public String nom;
    public String prenom;
    public int age ;

    public Individu(String n,String p, int x ){
        nom = n;
        prenom = p;
        age = x ;
    }
    public void afficheIndividu(){
        System.out.println("Nom : "+ nom + " Prénom : "+ prenom);
    }
}
```

La classe Individu est déclarée avec le modificateur **public**, elle est ainsi visible par toutes les autres classes. Si le modificateur est omis, la classe est visible uniquement dans le package où elle est placée.

Les champs String *nom* et *prenom* sont précisés **public**.

Le constructeur et la méthode *afficheIndividu()* sont précisés **public**.

La classe de test test00 suivante est à l'extérieur du package agenda:

```
import agenda.Individu;
public class test00 {
    public static void main( String[] args) {
        Individu pers1 = new Individu("Aimar","Jean",25);
        System.out.println("Nom : " + pers1.nom+ " Prénom : " + pers1.prenom + "Age : " +
            pers1.age);
    }
}
```

Q1 Tester le fonctionnement de la classe Individu ci-dessus. Justifier les affichages obtenus.

Modifions la classe de test:

```
import agenda.Individu;
public class test00 {
    public static void main( String[] args) {
        Individu pers1 = new Individu("Aimar","Jean",25);
        System.out.println("Nom : " + pers1.nom+ " Prénom : " + pers1.prenom + "Age : " +
            pers1.age) ;
        pers1.nom="Dupond" ;
        System.out.println("Nom : " + pers1.nom+ " Prénom : " + pers1.prenom + "Age : " +
            pers1.age) ;
    }
}
```

Q2 Tester le fonctionnement de la classe Individu ci-dessus. Justifier les affichages obtenus.

Cet exemple met en évidence ce qui peut être un dysfonctionnement de la classe Individu. L'utilisateur de la classe **Individu** crée un individu avec ses attributs (nom, prénom, âge) mais il peut modifier comme il le veut n'importe lequel de ces attributs.

Les attributs ne sont pas protégés, ils sont modifiables de l'extérieur de la classe (par une classe externe), il peut en découler une utilisation fantaisiste de ces attributs.

Les modificateurs d'accès sont prévus pour restreindre l'accès aux membres -attributs et méthodes- d'une classe par une autre classe.

Les modificateurs d'accès sont les mots réservés **public**, **protected** et **private**.

- Les champs précisés **public** sont accessibles à tous les autres objets qui manipulent un objet instance de la classe concernée. On dit qu'ils sont accessibles à l'extérieur de la classe.
- Les champs **private** sont accessibles uniquement par les méthodes de la classe.
- Les champs **protected** sont accessibles par les méthodes de la classe, par les méthodes des classes dérivées ainsi que par les méthodes des classes placées dans le même package.

L'accessibilité d'un élément est appelée la **portée**. On qualifie un élément de **public** lorsqu'on veut qu'il soit accessible à l'extérieur de la classe, pour tout objet qui peut accéder à un objet instance de cette classe, on dit qu'il a la même **portée** que sa classe.

Un élément **private** a sa **portée** limitée à sa classe.

Un élément **protected** a sa **portée** limitée à sa classe, aux classes dérivées et aux classes du même package.

Le tableau suivant montre l'accessibilité des membres d'une classe en fonction du modificateur utilisé.

Niveau d'accès

Modificateurs	Classe	Package	Sous-classe	Extérieur
public	OUI	OUI	OUI	OUI
protected	OUI	OUI	OUI	NON
<i>Sans modificateur</i>	OUI	OUI	NON	NON

private	OUI	NON	NON	NON
---------	-----	-----	-----	-----

Q3

- Tester le fonctionnement des classe Individu et test00 (compiler le fichier Individu.java avec : `javac -d lib -classpath lib Individu.java`).
- Remplacer pour les 3 attributs le modificateur d'accès public par private :

```
private String nom;
private String prenom;
private int age ;
```

Recompiler les 2 classes.

Quels sont les messages du compilateur javac à la suite de la compilation de test00.java ?

Quelle est la solution pour afficher les attributs nom, prenom et age ?

2. Les accesseurs et l'encapsulation

L'encapsulation consiste à protéger les données d'un objet afin de les sécuriser. La solution la plus efficace est d'utiliser le modificateur **private** pour tous les attributs à protéger.

Les attributs **private** ne sont alors accessibles que par les méthodes de la classe elle-même.

On peut avoir besoin de connaître leurs valeurs à l'extérieur de la classe, il faut pour cela définir des méthodes **public** qui retournent les valeurs de ces attributs, ces méthodes sont appelées des **accesseurs**.

Les méthodes qui permettent d'accéder en lecture comme en écriture à des attributs privés sont appelées des **accesseurs**.

Exemple :

Accesseurs de type get – qui retournent la valeur d'un attribut privé - appelés **getter**

```
public String getNom() {
    return nom ;
}
public String getPrenom() {
    return prenom ;
}
```

Accesseurs de type set – qui initialisent un attribut privé – appelés **setter** (ou **mutateurs**)

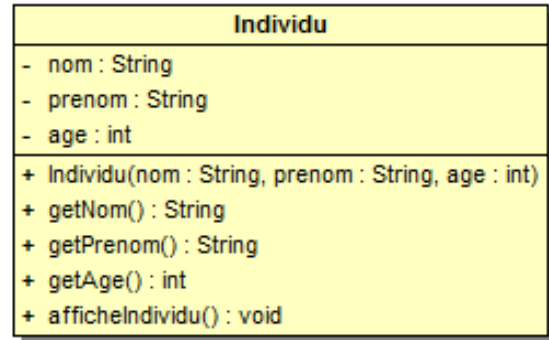
```
public void setNom(String nouveaunom) {
    nom = nouveaunom ;
}
public void setPrenom(String nouveauprenom) {
    prenom = nouveauprenom ;
}
```

Remarque : il peut être inutile de déclarer des attributs privés si on crée des accesseurs public de type **set**.

Modèle UML de la classe Individu : signe - pour private, + pour public

Q4

Coder la classe **Individu** ci-contre et une classe de test **TestIndividu**. La classe de test créera un **individu** puis vérifiera le fonctionnement des diverses méthodes de **Individu**.



3. Révision

3.1 Qu'est-ce qu'une classe ?

Il n'existe pas de réelle définition d'une classe en programmation objet.

Une classe sert à fabriquer des objets, on peut alors imaginer qu'une classe est un moule.

Un objet est caractérisé par un certain nombre de propriétés.

Par exemple, pour un objet rectangle on aurait les propriétés suivantes :

- Largeur
- Longueur
- Position du centre (X,Y) dans le plan.
- Angle de rotation (si le rectangle n'est pas en position horizontale).

Cet objet rectangle peut être inanimé (il ne bouge plus dans le plan) mais, en informatique, on peut lui faire subir des opérations comme :

- tradater
- tourner
- agrandir
- diminuer
- afficher

On définit ainsi une classe Java pour tous nos objets de type Rectangle à créer.

Ecriture simplifiée en Java de la classe **Rectangle**.

```
public class Rectangle {
    /* Les propriétés = attributs = données membres */
    public float largeur ;
    public float longueur ;
    //Position du centre du rectangle (X,Y) dans le plan.
    public int X ;
    public int Y ;
    public float angle ;
    /* Les opérations = méthodes = fonctions membres */
    public void tradater(int x, int y) { /* A coder ici */ }
    public void tourner(float a) { /* A coder ici */ }
```

```

public void agrandir(float a) { /* A coder ici */ }
public void diminuer(float a) { /* A coder ici */ }
public void afficher() { /* A coder ici */ }
}

```

Les attributs sont des variables particulières car ils sont déclarés dans la classe mais à l'extérieur de toutes les méthodes. C'est pour cela qu'on dit qu'ils sont les propriétés de la classe.

Les attributs sont accessibles par toutes les méthodes de la classe.

On ne code ici que les méthodes **translater()** et **afficher()**.

```

void translater(int x, int y) {
    X = X + x ;
    Y = Y + y ;
}
void afficher() {
    System.out.println("Ma longueur= "+longueur+ " et ma largeur= "+largeur) ;
    System.out.println("Je suis en X = "+X+ " et en Y = "+Y) ;
}

```

Les méthodes afficher() et translater() accèdent aux attributs X et Y.

3.2 Créer (instancier) les objets

La classe **Rectangle** est le moule utilisé pour fabriquer des objets de type rectangle. Ces objets sont des objets informatiques.

Comment créer les objets ?

Il faut maintenant créer des objets de type Rectangle à partir de la classe précédente. Cela peut se faire en 2 étapes :

1) Déclarer une variable de type Rectangle : c'est notre objet, il s'appelle ici monrec.

```
Rectangle monrec;
```

2) Créer l'objet avec l'opérateur **new** :

```
monrec = new Rectangle() ;
```

Où créer les objets ?

La méthode main vue dans les chapitres précédents doit être utilisée pour créer (instancier) des objets de type Rectangle.

On crée pour cela une classe d'application **MainRectangle** qui est ici très simple car elle ne contient que la méthode main().

On suppose travailler dans un même dossier, c'est à dire que les fichiers Rectangle.java et MainRectangle.java sont placés dans un même dossier ainsi que les fichiers .class correspondants. Sinon il faut faire un package.

```

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

```

```

    Rectangle monrec;
    monrec = new Rectangle() ;
}

```

Comment utiliser les méthodes (fonctions membres) de la classe Rectangle ?

Par exemple, le nom de l'objet (monrec) suivi d'un point (.) suivi du nom de la méthode (afficher). On ajoute alors les 3 instructions écrites en gras.

```

public class MainRectangle {
public static void main(String[] args)
{
    Rectangle monrec;
    monrec = new Rectangle() ;
    monrec.afficher() ;
    monrec.translater(10,15) ;
    monrec.afficher() ;
}
}

```

Q5

Ecrire les classes Rectangle.java et MainRectangle.java dans un même dossier.

Compiler les 2 classes.

Exécuter MainRectangle.

Que valent les attributs de l'objet Rectangle créé?

Réponse : l'objet monrec est bien créé mais tous ses attributs sont mis à 0...

3.3 Le constructeur de la classe Rectangle

La JVM crée un constructeur par défaut, l'exemple précédent montre que l'objet monrec est bien créé mais que tous ses attributs sont mis à 0...

Cet exemple montre bien l'utilité d'un constructeur. On pourra alors choisir des dimensions, une place dans le plan et une orientation par défaut. On propose ici un constructeur qui initialise les attributs avec des valeurs par défaut.

```

public class Rectangle {
    public float largeur ;
    public float longueur ;
    //Position du centre (X,Y) dans le plan.
    public int X ;
    public int Y ;
    public float angle ;

    public Rectangle() {
        largeur = 10 ;
        longueur = 20 ;
        X= 10 ;
        Y = 5 ;
        angle = 0 ;
    }
}

```

```

    }
    .....
}

```

Q6

Modifier la classe Rectangle en ajoutant le constructeur ci-dessus.

Compiler Rectangle.java.

Exécuter MainRectangle. Relever les valeurs affichées. Conclure

3.4 Danger des attributs public

Modifier la classe MainRectangle comme il est indiqué.

```

public class MainRectangle {
public static void main(String[] args)
{
    Rectangle monrec;
    monrec = new Rectangle() ;
    monrec.afficher() ;
    monrec.largeur = -10 ;
    monrec.longueur = -20 ;
    monrec.afficher() ;
    monrec.largeur = 30 ;
    monrec.longueur = 20 ;
    monrec.afficher() ;
}
}

```

Q7

Tester le programme précédent. L'observation des résultats affichés montrent plusieurs dysfonctionnements de la classe Rectangle.

La longueur et la largeur doivent être des entiers positifs ou nuls.

Modifier la classe Rectangle afin de corriger les dysfonctionnements.

Tester le fonctionnement.

3.5 Surcharge du constructeur

On demande de surcharger le constructeur Rectangle en écrivant 2 nouveaux constructeurs dont les signatures sont données :

```

//Construit un rectangle horizontal centré sur l'origine
public Rectangle(float largeur, float longueur)
//Construit un rectangle horizontal dont le centre est donné
public Rectangle(float largeur, float longueur, int X, int Y)

```

Ces constructeurs devront vérifier la validité de leurs arguments.

Voir le paragraphe 3.8 si nécessaire pour l'utilisation de this..

Q8

Coder les constructeurs demandés. Tester.

3.5 Rappel: Le mot clé **this**

On choisit l'exemple de la classe `Individu` pour expliquer l'opérateur **this**.

Q9

Réécrire le constructeur de la classe `Individu` de la manière suivante :

```
public Individu( String nom, String prenom, int age ){
    this.nom = nom;
    this.prenom = prenom;
    this.age = age;
}
```

Les arguments du constructeur ont les mêmes noms que les attributs de la classe: il y a un risque de confusion.

Dans l'instruction **this.nom = nom ;**

il est nécessaire de préciser `this.nom = nom` pour distinguer à gauche de l'égalité l'attribut `Nom` de la variable `Nom` placée à droite de l'égalité.

Au cours de l'exécution du programme **this** est une variable qui **référence l'objet en cours d'exécution**, on peut aussi dire que **this** identifie l'objet dont la méthode est en cours d'exécution.

this.nom peut se définir comme l'attribut `nom` de l'objet courant, alors que l'identifiant `nom` se rapporte à l'argument passé au constructeur.

Q10 Tester le fonctionnement des classes avec cette variante du constructeur.

Q11

Soit la classe `Individu` à compléter

```
package agenda;
import java.io.*;

public class Individu {

    private String nom;
    private String prenom;
    private age ;

    // Constructeur à compléter
    public Individu(String nom,String prenom, int age){
        ... = nom;
        ... = prenom;
        ... = age;
    }
    public void afficheIndividu(){
        // A compléter pour afficher les attributs
    }
    // A compléter avec 3 accesseurs de type get pour les 3 attributs
} //fin de la classe
```

Soit la classe `Test01` incomplète pour mettre en œuvre la classe `Individu`.

```
import agenda.Individu ;
```



```

import java.io.*;

public class Test01{
public static String lireClavier() throws IOException
{
BufferedReader clavier = new BufferedReader( new InputStreamReader(System.in));
String texte=clavier.readLine();
return texte ;
}
public static void main( String[] args) throws IOException
{
    String unNom;
    String unPrenom;
    int unAge ;
    System.out.println("Entrer le nom");

// A compléter pour la saisie du nom, du prénom et de l'âge
// A compléter pour créer un objet pers1 de type Individu

    pers1.afficheIndividu();
}
}

```

Travail

Compléter et compiler le fichier Individu.java.

Compléter le fichier Test01.java

Compiler le fichier Test01.java, puis exécuter l'application.

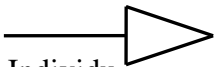
4. Les classes, l'héritage

On veut créer une classe Abonné. Un abonné a un nom, un prénom , un âge comme un individu. Un abonné est un individu qui possède en plus un abonnement avec un N° de téléphone. Un abonne est un individu avec une fonctionnalité supplémentaire: c'est un individu "spécialisé".

Il est possible de créer une classe Abonné à partir de la classe Individu sans avoir besoin de tout réécrire. Cela fait appel au mécanisme de l'héritage.

La classe Abonne héritera de la classe Individu

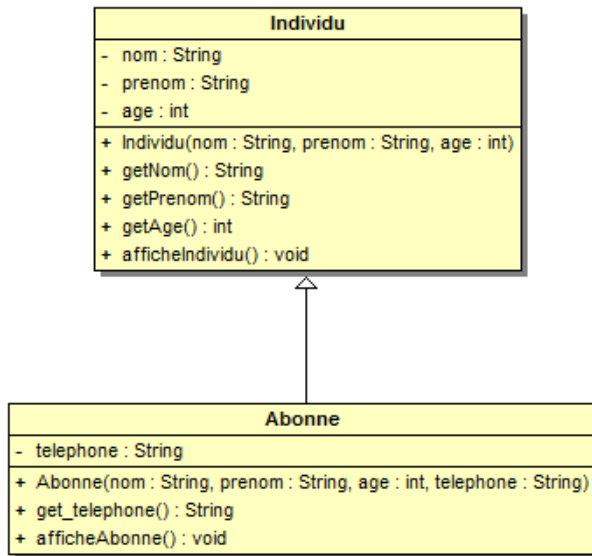
Diagramme des classes (UML)

La flèche fermée  précise une relation d'héritage: la classe Abonne hérite (dérive) de la classe Individu.

Le signe - indique un membre **private**.

Le signe + indique un membre **public**.

Le signe # indique un membre **protected**.



Il faut écrire un fichier par classe, chaque fichier portant le nom de la classe avec l'extension .java.

On peut néanmoins écrire plusieurs classes dans un même fichier, il faut dans ce cas donner comme nom au fichier le **nom de la seule classe qui doit être public**.

La solution habituelle consiste à écrire la classe Individu dans un fichier Individu.java et la classe Abonne dans un fichier séparé Abonne.java

package agenda;

```

public class Individu {
    private String nom;
    private String prenom;
    private int age ;
    public Individu(String nom,String prenom, int age ){
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
    }
    public void afficheIndividu(){
        System.out.println("Nom : "+nom+ " Prénom : "+prenom);
    }
    public String getNom() {
        return nom ;
    }
    public String getPrenom()  {
        return prenom ;
    }
    public int getAge()  {
        return age ;
    }
}

```

package agenda;

```

public class Abonne extends Individu {

```

```

private String telephone;
public Abonne(String nom, String prenom, int age, String telephone){
    super(nom, prenom, age); // constructeur de la super classe
    this.telephone= telephone;
}
public void afficheAbonne(){
    afficheIndividu(); //appel de la méthode afficheIndividu() de la classe mère
    System.out.println("Téléphone : "+telephone);
}
}

```

Dans l'exemple ci-dessus, la classe Abonne **dérive** de la classe Individu. On dit également que la classe Abonne **hérite** de la classe Individu ou **étend** la classe Individu.

On dit également que la classe Individu est la **super classe** de la classe Abonne qui est alors la **sous-classe**.

Le mot réservé **extends** indique qu'une classe hérite d'une autre.

L'héritage sert à personnaliser (spécialiser) des classes existantes pour fabriquer des classes plus spécifiques. La classe Individu est une classe plus générale qui gère notamment les attributs *nom* et *prenom*. La classe **Abonne** ajoute une fonctionnalité à la classe **Individu** en ajoutant un attribut *telephone* et l'accessor *getTelephone()*. Un objet **Abonne** est construit à partir d'un objet **Individu** en ajoutant la propriété *telephone*.

La sous-classe (classe dérivée ou classe fille) dispose directement des attributs et des méthodes de la super classe (classe mère ou super classe) à l'exception des éléments qualifiés *private* (symbole - sur le diagramme des classe).

Le constructeur de la classe **Abonne** appelle le constructeur de sa super classe, par la ligne de code **super(nom,prenom,age) ;**

☛ Il est obligatoire de déclarer un constructeur dans une classe dérivée dès lors qu'il y a un constructeur avec arguments dans la super classe (classe mère). Il faut alors appeler l'exécution du constructeur de la classe mère par **super(...)** dans le code du constructeur de la classe dérivée, **super(...)** étant la **1^{ère} instruction du constructeur de la classe dérivée**. L'appel de **super(arg1, arg2...)** doit respecter la signature du constructeur de la classe mère: nombre des arguments, type, ordre.

Si la classe mère n'a pas de constructeur ou simplement un constructeur sans argument, l'appel de **super()** dans le constructeur de la sous-classe n'est pas obligatoire, la JVM exécutera automatiquement le constructeur par défaut de la super classe lors de la construction de l'objet instance de la classe dérivée.

Règles d'utilisation de super(...) dans le constructeur de la sous classe

Super classe (classe mère)	Pas de constructeur	1 seul constructeur sans argument	1 ou plusieurs constructeur mais <u>pas de constructeur sans arguments</u>
Constructeur de la sous-classe	Pas obligatoire	Pas obligatoire	Obligatoire
Appel de super dans le constructeur de la sous-classe	Pas obligatoire	Pas obligatoire	Obligatoire

S'il est nécessaire, l'appel du constructeur de la super classe par l'instruction `super(param1...)` ; doit impérativement être la 1^{ère} instruction du code du constructeur de la classe dérivée.

Les attributs `nom`, `prenom` et `age` sont privés dans la classe mère `Individu`, ils ne sont pas accessibles dans la classe dérivée. Il faut dans ce cas utiliser les accesseurs `getNom()`, `getPrenom()` et `getAge()`.

Les attributs de la classe mère sont private: réécriture de la méthode `afficheAbonne()` en utilisant les accesseurs

On peut remplacer

```
public void afficheAbonne(){
    afficheIndividu() ;           //appel de la méthode afficheIndividu() de la classe mère
    System.out.println("Téléphone : "+telephone);
}
```

par

```
public void afficheAbonne(){
    System.out.println("Nom : "+getNom()+ " Prenom : "+getPrenom()+ " Téléphone : "+telephone);
}
```

en utilisant les accesseurs.

5. La redéfinition des méthodes

Réécrivons la classe `Individu` en changeant seulement le nom de la méthode `afficheIndividu()` par un nom plus neutre comme `affiche()`.

```
public class Individu {
    ....
    public void affiche(){
        System.out.println("Nom : "+nom+ " Prénom : "+prenom);
    }
}
```

De même, réécrivons la classe `Abonne` en changeant seulement le nom de la méthode `afficheAbonne()` par `affiche()`.

```
public class Abonne extends Individu {
    .....
    public void affiche(){
        System.out.println("Nom : "+getNom()+ " Prenom : "+getPrenom+
```

```

    " Téléphone : "+telephone);
}
}

```

Voici un exemple de **redéfinition** d'une méthode : la méthode **affiche()** de la classe Individu est **redéfinie** (ou **surdéfinie**) dans la classe Abonne.

Créons un objet de type Individu :

```

Individu martel = new Individu("Martel", "Charles",45) ;
// Exécution de la méthode affiche de la classe Individu
martel.affiche()

```

Créons un objet de type Abonne:

```

Abonne colomb = new Abonne("Colomb", "Christophe",40,"0601020304") ;
// Exécution de la méthode affiche de la classe Abonne
colomb.affiche()

```

Il est possible de faire exécuter la méthode **affiche()** de la classe Individu dans la méthode **affiche()** de la classe Abonne, on utilise pour cela le mot réservé super : **super.affiche()** ;

```

public class Abonne extends Individu {
.....
    public void affiche(){
        super.affiche() ;           // Exécution de la méthode affiche de la classe Individu
        System.out.println(" Téléphone : "+telephone);
    }
}

```

Ce cas n'est qu'un exemple, car en général, rien n'oblige une méthode redéfinie à faire exécuter la méthode correspondante de sa super classe.

Attention : il ne faut pas confondre surcharge et redéfinition.

Une méthode d'une classe est redéfinie quand elle est déclarée dans une de ses sous-classes avec le même identificateur, le même type et les mêmes arguments (même prototype).

Une méthode d'une classe est surchargée quand elle est déclarée plusieurs fois dans une classe avec le même identificateur mais avec des arguments différents.

La **surcharge** et la **redéfinition** des méthodes sont des exemples de **polymorphisme**.

Le mot réservé **final** doit être utilisé lorsqu'on ne veut pas qu'une méthode soit redéfinie dans une classe dérivée: **public final int calculCRC() {---}**

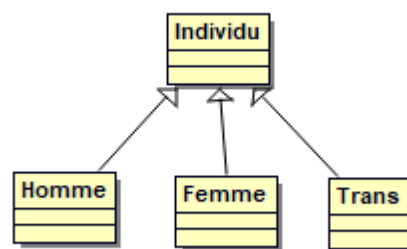
Notion sur le polymorphisme

Soit le diagramme de classes ci-contre :

On peut affirmer :

- **Un homme est un individu.**
- **Une femme est un individu.**
- **Un trans est un individu.**

Donc on peut affecter à un objet Individu un objet instance d'une des classes dérivées de Individu.



C'est-à-dire :

```
unObjetIndividu ← unObjetHomme
unObjetIndividu ← unObjetFemme
unObjetIndividu ← unObjetTrans
```

Donc en programmation :

```
public class Individu {...}
public class Homme extends Individu {}
public class Femme extends Individu {}
```

On peut donc écrire, par exemple :

```
Individu i , j ;
Homme h ;
h = new Homme (...) ;
i = h ; // un homme est un individu
j = new Femme(...) ; // une femme est un individu
```

Mais **un individu n'est pas forcément un homme**, et de la même manière, **un individu n'est pas forcément une femme**, à cause de la relation d'héritage donc l'affectation suivante est incorrecte :

```
Individu i , j ;
Homme h ;
i = new Individu (...) ; //OK
h = i ; // incorrect : un individu n'est pas forcément un homme
```

L'affectation `unObjetHomme ← unObjetIndividu` n'est pas correcte car un individu peut être un homme ou une femme ou un trans. On ne possède pas assez d'informations pour valider cette affirmation.

Ajoutons une méthode affiche dans la classe Individu :

```
public class Individu {
    public void affiche() {
        System.out.println("Individu");
    }
}
```

La classe d'application :

```
public class Humanite {

    public static void main(String[] args) {
        Individu i1 , i2 , i3;
        Homme h1 = new Homme() ;
        i1 = h1 ;
        i1.affiche();
        Femme f1 = new Femme() ;
        i2 = f1 ;
        i2.affiche();
    }
}
```

Affichage obtenu

Individu Individu

La méthode **affiche()** de la classe **Individu** est exécutée.

Redéfinissons la méthode **affiche()** uniquement dans les classes **Homme** et **Femme** :

```
public class Homme extends Individu {
    public void affiche() {
        System.out.println("Homme");
    }
}
public class Femme extends Individu {
    public void affiche() {
        System.out.println("Femme");
    }
}
public class Trans extends Individu {
}
```

La classe d'application :

```
public class Humanite {

    public static void main(String[] args) {
        Individu i1 , i2 , i3;
        Homme h1 = new Homme() ;
        i1 = h1 ;
        i1.affiche();                //1
        Femme f1 = new Femme() ;
        i2 = f1 ;
        i2.affiche();                //2
        i3 = new Individu() ;
        i3 = new Trans() ;
        i3.affiche();                //3
        System.out.println();
    }
}
```

Affichage

Homme
Femme
Individu

- 1 La méthode **affiche()** de la classe **Homme** est exécutée.
- 2 La méthode **affiche()** de la classe **Femme** est exécutée
- 3 La méthode **affiche()** de la classe **Individu** est exécutée

La méthode redéfinie appelée dépend du type instancié et pas du type déclaré. Cette propriété du polymorphisme est très utilisée en POO. Elle est mise en œuvre dans l'exercice 6.5 du sac postal.

6. Exercices

6.1 Soit le code suivant de 2 classes:

```
public class A {
    private int i ;
    public A(int i) { this.i = i ; }
}
```

```
public class B extends A { }
```

Est-ce que ce code compile ? s'exécute ?

6.2 Ecrire les fichiers Individu.java et Abonne.java, et les compiler dans le package agenda..

6.3 On surchargera le constructeur Individu() avec un 2^{ème} constructeur pour saisir depuis le clavier le nom, le prénom et l'âge. Tester le fonctionnement.

6.4 On surchargera le constructeur Abonne avec un 2^{ème} constructeur pour saisir depuis le clavier le téléphone. Tester le fonctionnement.

6.5 Modifier les droits d'accès des membres nom et prenom de la classe Individu de private en protected. Tester le fonctionnement en créant des abonnés. Quel peut-être l'intérêt de protected ?

6.6 Le sac postal, version 1, voir le diagramme UML page suivante

On souhaite représenter la tournée d'un postier. Ce postier est muni d'un sac (d'une capacité maximale limitée à 10 unités de volume!) dans lequel il place les courriers qu'il récupère lors de sa tournée.

Ces courriers sont de deux types :

- des lettres, dont le tarif d'affranchissement dépend du caractère urgent ou non de chaque lettre. Plus précisément, on affranchit les lettres à 50 cents, plus 30 cents si elles sont urgentes. Une lettre occupera une unité de volume dans le sac postal.
- des colis (paquet), dont l'affranchissement est fonction du volume (1 € par unité de volume).

La classe **Courrier** :

- **timbre** représente la valeur de l'affranchissement du courrier, on a choisi ici un float, on aurait pu choisir un String (attribut **protected**, car #).
- **vitesse** est un booléen, il précise l'urgence (à true) du courrier (attribut **protected**, car #).
- **volume** est un entier indiquant le volume occupé par le courrier, 1 pour une lettre, 1 ou plus pour un colis (attribut **protected**, car #). Dans un but de simplification, on convient que l'unité de volume de base est égale au volume d'une lettre.

Le constructeur **Courrier()** initialise les attributs à des valeurs par défaut : timbre=50, vitesse=false, volume=1.

Les méthodes **afficher()** et **affranchir()** peuvent rester vides car elles sont redéfinies dans les classes dérivées.

Les classes **Lettre** et **Paquet**

Elles dérivent de la classe **Courrier**.

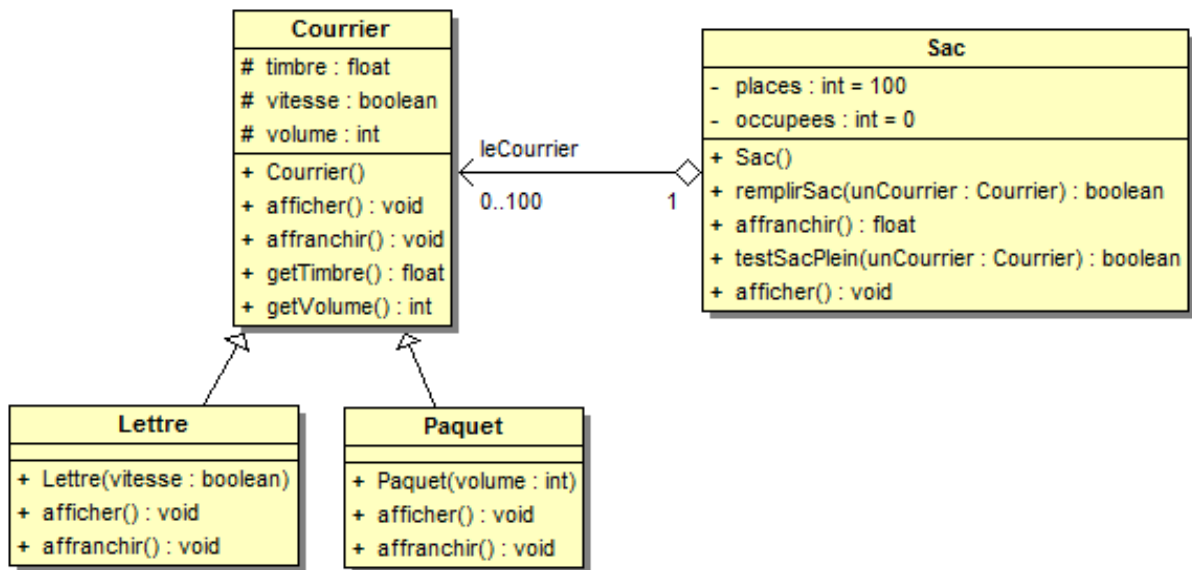
Les constructeurs sont spécialisés pour chacun de ces courriers, l'un reçoit la vitesse comme argument (pour une Lettre), l'autre le volume (pour un Paquet).

La méthode **affranchir()** calcule l'affranchissement en fonction de la spécification précédente.

La méthode **afficher()** affiche un message propre à chaque type d'objet créé.

On propose ici un diagramme UML des classes nécessaires (sans la classe d'application contenant la méthode main, ici la classe de test).

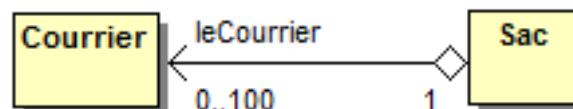
Les accesseurs ne sont pas tous volontairement représentés.



La flèche fermée ————> précise une relation d'héritage : les classes **Lettre** et **Paquet** héritent (dérivent) de la classe **Courrier**.

a) Coder les classes **Courrier**, **Lettre** et **Paquet**. Coder une classe de test qui crée des objets **Lettre** et **Paquet** pour vérifier le fonctionnement.

La classe **Sac**



La flèche avec un losange signifie une association (lien) de type **agrégation** entre les 2 classes : **1 objet « Sac » peut contenir de 0 à 100 objets « Courier »**.

Les chiffres 1 et l'intervalle [0..100] sont appelés la cardinalité.

Cette association est implémentée dans la classe Sac par un attribut de type tableau de Courier:

```
private Courier [] leCourrier ;
```

Lorsque le postier a terminé sa tournée, il revient au bureau de poste et pose son sac dans une machine qui affranchit automatiquement l'ensemble des courriers contenus dans le sac. On suppose que le sac postal peut contenir 100 unités de volume au maximum, on choisit un tableau de 100 pour simuler ce stockage : **leCourrier** est un tableau de 100 objets Courier.

La méthode **boolean remplirSac(unCourier :Courier)** de la classe Sac remplit le tableau de Courier (**leCourrier[]**) avec des objets Lettre ou des objets Paquet. Attention, une Lettre occupe une case du tableau leCourrier mais un Paquet peut occuper plusieurs cases. Elle retourne true si le courrier a bien été mis dans le sac.

Cet exercice met en application le polymorphisme: on déclare un tableau de Courier dans la classe Sac, et on remplit ce tableau par des objets de type Lettre ou Paquet, qui dérivent de la classe Courier. Ce polymorphisme est appelé polymorphisme par sous typage car on peut remplacer un objet d'une classe mère par un objet d'une classe dérivée.

La méthode `affranchir()` doit balayer le sac (le tableau **leCourrier**) et appeler à chaque fois la méthode `affranchir()` du Courrier contenu. Ainsi, chaque courrier sera affranchi correctement, en fonction de son type précis, bien que le sac n'ait pas "connaissance réelle" de son type...

b) Coder la classe `Sac`. Ne pas oublier d'ajouter l'attribut **`private Courrier [] leCourrier`** qui implémente l'agrégation vue ci-dessus dans la classe `Sac`

c) Coder un programme de test qui effectuera les traitements suivants :

1. créer un sac et quelques lettres et colis
2. placer les courriers dans le sac
3. affranchir le sac
4. afficher le tarif des différents courriers contenus dans le sac
5. modifier la méthode `affranchir` de la classe `Sac` afin de donner le total de l'affranchissement du `Sac`.

7. La classe `Object` du package `java.lang`

La classe `Object` du package `java.lang` est la classe mère de plus haut niveau. Toutes les classes sont des descendantes directes ou indirectes de la classe `Object`.

La méthode `toString()` est souvent redéfinie dans les classes dérivées pour retourner une description textuelle de l'objet

Exercices

- 7.1 Etudier les méthodes `equals()` et `toString()` de la classe **`java.lang.Object`** en utilisant la documentation Oracle. Préciser avec soin le fonctionnement de la méthode `equals()`.
- 7.2 Afficher le message retourné par la méthode `toString()` de la classe **`Individu`** puis de la classe **`Abonne`**.
- 7.3 Redéfinir la méthode `toString()` des classes **`Individu`** et **`Abonne`** afin retourner un message significatif.

8. Le mot clé `static`

Les attributs et méthodes des classes `Individu` et `Abonne` sont appelés attributs ou méthodes **d'instance**.

Prenons le cas des attributs de la classe `Individu`, chaque **objet** (instance) `Individu` créé a son propre nom, son propre prénom et son propre âge.

Il est possible de créer des attributs communs à tous les objets instances d'une même classe : ces attributs sont appelés **attributs de classe**.

Le mot réservé **static** permet de déclarer des **méthodes/attributs de classe**, par opposition aux **méthodes/attributs d'instance**.

```
package agenda;
public class Abonne extends Individu {
    private String telephone;
    public static int nombreAbonnes = 0 ;

    public Abonne(String nom, String prenom, int age, String telephone){
        super(nom, prenom, age); // constructeur de la super classe
        telephone= this.telephone;
        System.out.println("Je suis le constructeur de Abonne");
        nombreAbonnes++ ;
    }
    public void afficheAbonne(){
        afficheIndividu() ;           //appel de la méthode afficheIndividu() de la classe mère
        System.out.println("Téléphone : "+telephone);
    }
}
```

L'attribut **nombreAbonnes** est commun à tous les objets instances de la classe Abonne. Il est initialisé à 0 puis incrémenté par le constructeur, chaque nouvel abonné créé augmente de 1 NombreAbonnes.

Un membre **static** de classe est accessible par **NomClasse.NomMembreStatiqueClasse**, par exemple : **Abonne.nombreAbonnes** permet d'accéder à **nombreAbonnes** sans instancier d'objet sur la classe Abonne; il n'est même pas nécessaire d'instancier la classe pour accéder à ses attributs static.

```
public class MainTestAbonne{
    public static void main(String[] args) {
        Abonne a1, a2 , a3, a4 ;
        System.out.println("Nombre d'abonnés: "+Abonne.nombreAbonnes);
        a1 = new Abonne("Topar","Jean",50,"0303020101");
        System.out.println("Nombre d'abonnés: "+Abonne.nombreAbonnes);
        a2 = new Abonne("Zouzou","Rachid",23,"06050401");
        System.out.println("Nombre d'abonnés: "+Abonne.nombreAbonnes);
        a3 = new Abonne("Boudamba","Roger",25,"0609080704");
        System.out.println("Nombre d'abonnés: "+Abonne.nombreAbonnes);
        a4 = new Abonne("Suchet","Malika",42,"0603020104");
        System.out.println("Nombre d'abonnés: "+Abonne.nombreAbonnes);
    }
}
```

Affichage produit:

```
Nombre d'abonnés: 0
Je suis le constructeur de individu
Je suis le constructeur de Abonne
Nombre d'abonnés: 1
Je suis le constructeur de individu
Je suis le constructeur de Abonne
Nombre d'abonnés: 2
```

Je suis le constructeur de individu
 Je suis le constructeur de Abonne
 Nombre d'abonnés: 3
 Je suis le constructeur de individu
 Je suis le constructeur de Abonne
 Nombre d'abonnés: 4

On dispose ainsi d'un attribut contenant le nombre d'abonnés créés.

On peut améliorer le programme en déclarant **nombreAbonnes** private et en créant une méthode accesseur pour **nombreAbonnes**. Il faut que cette méthode soit **static** elle aussi.

```
public class Abonne extends Individu {
    private String telephone;
    private static int nombreAbonnes = 0 ;

    public Abonne(String nom, String prenom, int age, String telephone){
        super(nom, prenom, age); // constructeur de la super classe
        Telephone= telephone;
        nombreAbonnes++ ;
    }

    public static int getNombreAbonnes() {
        return nombreAbonnes ;
    }
    ....
}
```

La méthode **getNombreAbonnes()** est appelée **méthode de classe** car elle est déclarée **static**.

Utilisation :

```
public static void main(String[] args) {
    Abonne a1, a2 , a3, a4 ;
    System.out.println("Nombre d'abonnés: "+Abonne.getNombreAbonnes());
    a1 = new Abonne("Topar","Jean",50,"0303020101");
    System.out.println("Nombre d'abonnés: "+Abonne.getNombreAbonnes());
    ....
}
```

Règles d'utilisation :

- Les méthodes d'instance peuvent accéder directement aux attributs d'instance.
- Les méthodes d'instance peuvent accéder directement aux attributs de classe.
- Les méthodes de classe peuvent accéder directement aux attributs de classe.
- Les méthodes de classe ne peuvent pas accéder directement aux attributs d'instance. Elles doivent passer par un objet instance de la classe, elles ne peuvent pas utiliser le mot réservé this.

[Q12](#) Etudier et tester l'exemple précédent.

Intérêt:

Attributs de classe : l'exemple précédent montre que l'attribut de classe **nombreAbonnes** est commun à tous les abonnés créés, il sert à compter le nombre d'abonnés créés en étant

augmenté de 1 à chaque nouvel abonné créé.

Méthodes de classes :

L'exemple précédent montre qu'un attribut de classe **private** nécessite une méthode accesseur de classe pour y accéder de l'extérieur.

Les méthodes de classes sont très utilisées pour fabriquer des classes « librairie ». La classe Math en est l'exemple le plus frappant. La classe Math ne contient que des méthodes de classe (static), ces méthodes peuvent être utilisées simplement sans avoir besoin de créer des objets de type Math, il suffit de préfixer le nom de la méthode par Math. :

Extrait de la classe Math

static double	log (double a) Returns the natural logarithm (base <i>e</i>) of a double value.
static double	log10 (double a) Returns the base 10 logarithm of a double value.
static double	log1p (double x) Returns the natural logarithm of the sum of the argument and 1.
static double	max (double a, double b) Returns the greater of two double values.
static float	max (float a, float b) Returns the greater of two float values.
static int	max (int a, int b)

Exemple :

```
int a, b ;
....
int max = Math.max(a,b) ;
```

9. Les classes abstraites

Pour déclarer une classe abstraite on utilise le mot réservé **abstract**.

Une classe abstraite peut contenir des méthodes **abstract** qui n'ont pas d'implémentation. Les classes dérivées doivent alors définir ses méthodes afin de pouvoir être instanciées. Une classe qui contient une méthode **abstract** doit être elle aussi impérativement **abstract**.

On ne peut pas instancier d'objet sur une classe abstraite, mais cela est possible sur une classe "concrète" dérivée de la classe abstraite.

Une classe abstraite est suffisamment générale pour ne pas avoir d'objets intéressants à instancier à son niveau. Elle prépare des données et des méthodes (elle sert de modèle de base) pour des classes dérivées qui seront adaptées à l'instanciation d'objet.

Exemple :

```
public abstract class GraphicObject {
    // declare fields
    // declare non-abstract methods
    public abstract void draw();
```

```
}
```

La méthode **draw()** est ici abstraite, on ne voit que son **prototype**.

```
class Circle extends GraphicObject {
    void draw() {        /* codage de draw */    ...    }
    ....
}
class Rectangle extends GraphicObject {
    void draw() {        /* codage de draw */    ...    }
    ....
}
```

La méthode draw() est implémentée dans chaque classe dérivée.

Exercices

9.1 Une classe FormeGeometrique abstraite et une classe dérivée

```
public abstract class FormeGeometrique {
    double posX, posY;
    void deplacer(double x,double y) {
        posX=x;
        posY=y;
    }
    void afficherPosition() {
        System.out.println("position : (" +posX+", "+posY+"");
    }
    public abstract double surface() ;
    public abstract double perimetre() ;
}
```

Un exemple de classe dérivée de FormeGeometrique :

```
public class Cercle extends FormeGeometrique {
    double rayon;
    public Cercle(double x, double y, double r) {
        posX=x; posY=y; rayon=r;
    }
    public double surface() {
        return Math.PI*Math.pow(rayon, 2.);
    }
    public double perimetre() {
        return 2*rayon*Math.PI;
    }
}
```

Etudier cet exemple.

Coder une application qui crée des objets Cercle.

Compléter cette application en créant une classe Rectangle qui hérite de FormeGeometrique et définit les 2 méthodes abstraites.

Compléter cette application en créant une classe TriangleRectangle qui hérite de FormeGeometrique et définit les 2 méthodes abstraites.

9.2 Ajouter la méthode `estPlusGrandeQue()` dans la classe `FormeGeometrique`.

```
public abstract class FormeGeometrique {
    .....
    public int estPlusGrandeQue(FormeGeometrique s) {
        if (surface() > s.surface())
            return 1;
        else if (surface() < s.surface())
            return -1;
        else return 0;
    }
}
```

La méthode `estPlusGrandeQue()` :

- Retourne 1 si la surface passée en argument est plus petite que la surface qui exécute la méthode.
- Retourne -1 si la surface passée en argument est plus grande que la surface qui exécute la méthode.
- Retourne 0 si les surfaces sont identiques.

Montrer et vérifier dans le `main()` qu'on peut comparer les surfaces de figures différentes.

On voit également ici un exemple de polymorphisme, la méthode `estPlusGrandeQue(FormeGeometrique s)` reçoit comme argument un objet instance d'une classe dérivée de `FormeGeometrique`.

9.3 Le sac postal, version 2

Modifier l'exercice du sac postal en déclarant abstraite la classe `Courrier` ainsi que les méthodes `Afficher()` et `Affranchir()` de cette classe.

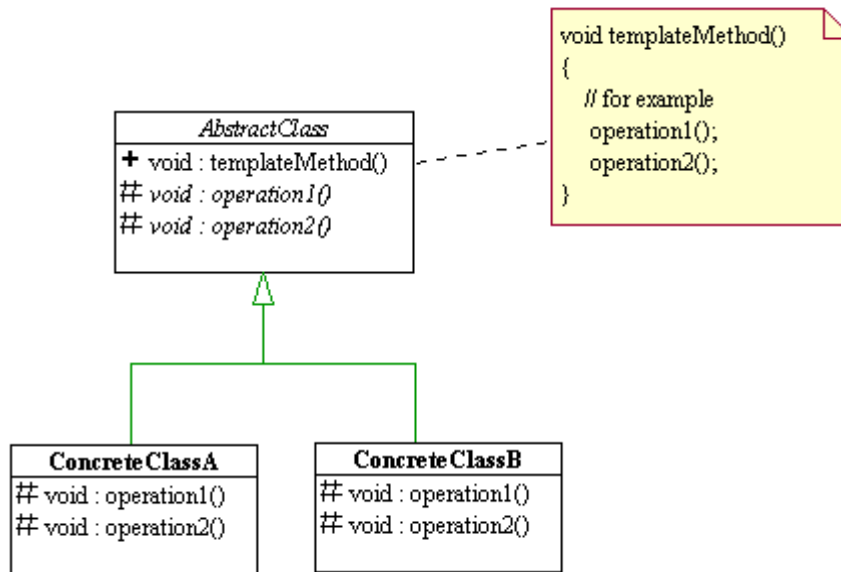
Représentation UML d'une classe abstraite : le nom de la classe et les noms des méthodes abstraites sont écrits en *italique*.

<i>Courrier</i>
#timbre : String #vitesse : boolean #volume : int
+Courrier() + <i>afficher()</i> : void + <i>affranchir()</i> : void

9.4 Dessiner le diagramme UML des classes étudiées dans les exercices 9.1 et 9.2.

10. Le pattern "Template Method"

Un pattern est un modèle de conception que les programmeurs peuvent utiliser pour répondre à des situations.

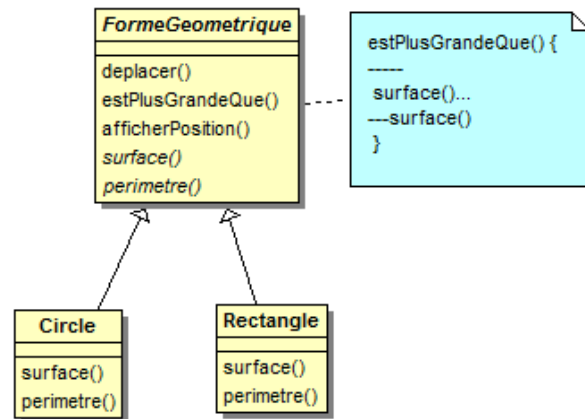


On définit une classe abstraite avec une méthode dite "template" qui appelle l'exécution d'une ou de plusieurs méthodes abstraites. Ces méthodes abstraites sont alors implémentées dans des classes concrètes.

Ce pattern s'applique à l'exemple précédent des formes géométriques.

La méthode `estPlusGrandeQue()` appelle l'exécution de la méthode abstraite `surface()` définie dans les classes concrètes.

La méthode `estPlusGrandeQue()` est la méthode template.



11. Les interfaces

Une interface ne contient que des déclarations (prototypes, signatures) de méthodes, elle peut également contenir des définitions de constantes.

Une interface ressemble à une classe abstraite. La différence est qu'une classe abstraite peut déclarer des méthodes implémentées (contenant du code) et des attributs, alors qu'une interface ne contient que des déclarations de méthodes (prototypes de méthodes, méthodes non codées) et éventuellement des attributs de classe constants.

Les méthodes d'une interface sont toutes implicitement public et abstract.

La classe qui implémente une interface doit coder toutes les méthodes de l'interface implémentée.

Une classe pourra implémenter une ou plusieurs interfaces en utilisant le mot réservé **implements**. C'est dans la classe qui implémente les interfaces qu'on écrit le code des méthodes des interfaces.

- Créons une interface simple avec deux méthodes.

```
package animaux;

public interface Animal {
    void parler() ;
    void manger(Animal unAnimal);
}
```

L'intérêt de l'interface, dans cet exemple est d'avoir 2 méthodes utilisables par une application qui utilise des objets animaux divers.

- Ecrivons une classe **Lion** qui implémente l'interface **Animal**.

```
package animaux;

public class Lion implements Animal {
    @Override
    public void parler() {
        System.out.println("Wrouaah Rouaah Raaaa");
    }

    @Override
    public void manger(Animal unAnimal) {
        // TODO Auto-generated method stub
        String nomcomplet = unAnimal.getClass().getName();
        String nom = nomcomplet;
        if (nomcomplet.contains(".") == true){
            String []t = nomcomplet.split("\\.");
            nom = t[t.length-1];
        }
        System.out.println("Je dévore un(e) "+nom);
    }
}
```

- Ecrivons une classe **Gazelle** qui implémente l'interface **Animal**.

```
package animaux;

public class Gazelle implements Animal {
    @Override
    public void parler() {
        // TODO Auto-generated method stub
        System.out.println("Hak hak hak");
    }

    @Override
    public void manger(Animal unAnimal) {
        System.out.println("Je suis un herbivore ruminant");
    }
}
```

- Ecrivons la classe **UneJungle** qui utilise les classes **Gazelle** et **Lion**.

```
package animaux;
```

```

public class UneJungle {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Lion lion = new Lion() ;
        Gazelle gazelle = new Gazelle() ;
        lion.parler();
        gazelle.parler();
        gazelle.manger(null);
        lion.manger(gazelle);
    }
}

```

L'interface **Animal** contient les méthodes offertes aux programmeurs que les objets **Lion** et **Gazelle** fournissent. Le programmeur utilisant ces classes **Lion** et **Gazelle** emploie les mêmes méthodes **parler()** et **manger()**, il doit juste être attentif aux éventuels arguments donnés.

Règles

En JAVA, une classe peut implémenter plusieurs interfaces.

```
public class A implements Int1, Int2 { .... }
```

En JAVA, un objet est du type de sa propre classe mais aussi du type des interfaces que sa classe implémente.

Un objet peut être déclaré avec soit le type de sa propre classe, soit le type de chacune des interfaces que sa propre classe a implémentées.

```

A a1 = new A() ;           // type de sa propre classe
Int1 a2 = new A() ;        // type de la 1ère interface implémentée
Int2 a3 = new A() ;        // type de la 2ème interface implémentée

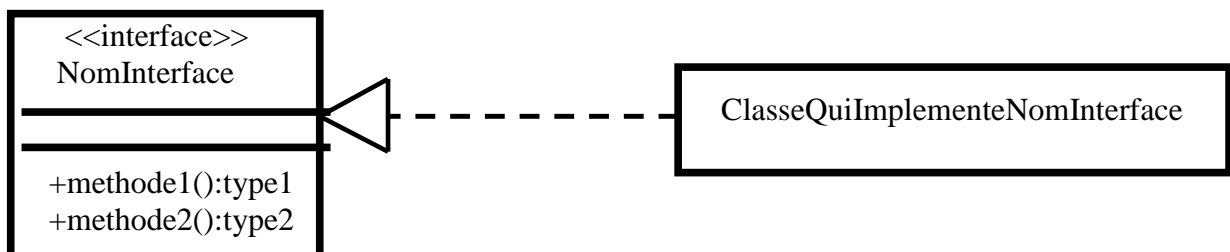
```

Cette propriété est utilisée dans le programme où un objet de type **Animal** est passé comme argument dans la définition de la méthode **manger(Animal unAnimal)**. Cette méthode est appelée dans la méthode **main()** **lion.manger(gazelle)**; avec un objet **Gazelle** comme argument.

Représentation UML d'une interface

Une interface est précisée par le stéréotype <<interface>>

On peut également préciser la classe qui **réalise – implémente** – l'interface.



On remarquera la même flèche que l'héritage mais en traits pointillés.

Exercices

10.1 Mettre en œuvre l'application utilisant l'interface donnée en exemple:

1. Ajouter une méthode **chasser(Animal unAnimal)** dans l'interface **Animal**.
2. Implémenter cette méthode dans les classes **Lion** et **Gazelle**.
3. Tester le fonctionnement dans la classe **UneJungle**.
4. Ajouter les classes **Serpent** (son =kss kss kss) et **Grenouille** (son=.coa coa croa).
5. Tester le fonctionnement de ces classes dans la classe **UneJungle**.

10.2 Reprendre/Refaire l'exercice 5 page 11 du chapitre 3 (interface Comparable).

12. Les classes anonymes

Il existe de nombreux cas où une classe doit uniquement implémenter une interface qui ne contient qu'une seule méthode.

Une classe anonyme est une classe qui n'a pas de nom et dont on définit seulement le corps (l'implémentation de la classe).

On montre ici l'exemple d'une classe anonyme qui est également une **classe interne**, c'est à dire définit dans une autre classe, ce qui est son utilisation normale.

Exemple simple de classes internes anonymes créées à partir d'une interface.

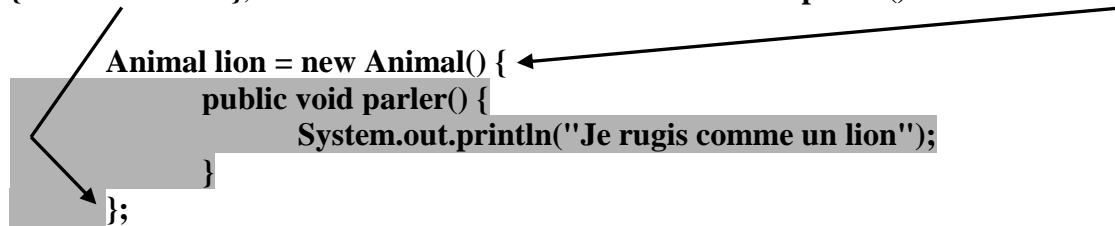
```
interface Animal {
    public void parler() ;
}

public class TestAnonyme {
    //1ère classe interne anonyme, objet lion
    Animal lion = new Animal() {
        public void parler() {
            System.out.println("Je rugis comme un lion");
        }
    };
    //2ème classe interne anonyme, objet chien
    Animal chien = new Animal() {
        public void parler() {
            System.out.println("J'aboie comme un chien");
        }
    };

    public static void main(String[] args) {
        TestAnonyme t = new TestAnonyme();
        t.lion.parler();
        t.chien.parler();
    }
}
```

L'objet est créé avec l'opérateur **new**, le nom de l'interface puis les parenthèses classiques ()

d'un constructeur. L'implémentation de la classe anonyme est placée entre l'accolade ouvrante { et la fermante }, c'est à dire ici la définition de la méthode **parler()**.



```
Animal lion = new Animal() {
    public void parler() {
        System.out.println("Je rugis comme un lion");
    }
};
```

Q13 Tester l'exemple ci-dessus. Justifier les affichages obtenus.

Q14 Ajouter une classe anonyme pour un chat. Tester.

13. L'égalité des objets, retour sur la classe Object

Les méthode equals() et hashCode() de la classe Object

boolean equals(Object object)

Elle compare les références de 2 objets.

Elle retourne true si ces références sont identiques et si l'argument object n'est pas nul.

```
package application
public class A {
    private int lea ;
    String noma;
    public A(int leA, String nomA) {
        lea = leA ;
        noma = nomA ;
    }
}
```

```
package application
public class TestEquals {
    public static void main(String []args){
        A a1 = new A(5,"boum") ;
        A a2 = new A(5,"boum") ;
        A a3 = a1;
        A a4 = new A(6,"boum");

        System.out.println(a1.equals(a1)+" "+a1.hashCode()+" "+a1);
        System.out.println(a1.equals(a2)+" "+a2.hashCode()+" "+a2);
        System.out.println(a1.equals(a3)+" "+a3.hashCode()+" "+a3);
        System.out.println(a1.equals(a4)+" "+a4.hashCode()+" "+a4);
    }
}
```

Résultats

true 705927765 application.A@2a139a55	1)
false 366712642 application.A@15db9742	2)
true 705927765 application.A@2a139a55	3)
false 1829164700 application.A@6d06d69c	4)

- 1) L'objet a1 est comparé à lui même, la méthode equals retourne true.
- 2) Les objets a1 et a2 ont des attributs de même valeur mais la méthode equals retourne false.
- 3) L'objet a3 est créé par copie de a1, la méthode equals retourne true.
- 4) Les objets a1 et a4 ont un attribut différent, la méthode equals retourne false.


Explications

Ces résultats de comparaison montrent le fonctionnement de la méthode equal():

La méthode equal() retourne false quand elle compare 2 objets créés séparément avec des attributs identiques, c'est le cas des objets a1 et a2, ligne 2).

La méthode **toString()** retourne un String égal au :

nom de la classe + "@" + le hashcode en hexadécimal.

application.A@2a139a55

La méthode **hashCode()** retourne le hascode de l'objet en base 10.

On peut vérifier l'égalité entre le hashcode en hexadécimal et le hashcode en décimal, c'est à dire : $0x2a139a55 = 705927765_{10}$

Les objets a1, a2 et a4 ont des hashcodes différents, a1 et a3 ont des hashcodes identiques.

Le nom de la classe est obtenu par **getClass().getName()**

La redéfinition des méthodes equals() et hashCode()

La redéfinition de la méthode equals() est impérative si on choisit que des objets ayant des attributs identiques soient égaux.

La spécification Java Oracle demande alors de redéfinir également la méthode **hashCode()** car 2 objets égaux doivent avoir le même hashcode.

La méthode **equals()** redéfinie doit donc porter sur les attributs.

► Eclipse propose une solution pour redéfinir automatiquement ces 2 méthodes :

1) Sélectionner la classe | clic droit | Source | Generate hashCode() and equals()...

2) La fenêtre suivante demande de choisir/valider les attributs utilisés pour la génération de ces 2 méthodes. On peut également choisir le point d'insertion, c.a.d l'endroit où ce code sera placé par Eclipse. Faire éventuellement ce choix.

3) Validez maintenant par **OK**.

Le code généré est le suivant:

```
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + lea;
    result = prime * result + ((noma == null) ? 0 : noma.hashCode());
    return result;
}
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    A other = (A) obj;
    if (lea != other.lea)
        return false;
    if (noma == null) {
        if (other.noma != null)
            return false;
    } else if (!noma.equals(other.noma))
        return false;
    return true;
}
```

Travail

1. Exécuter l'exemple du cours donné page 1. Observer les résultats.
2. Générer avec Eclipse les méthodes hashCode() et equals().
3. Comparer avec les exemples précédents.
4. Etudier avec soin le code de la méthode equals() générée par Eclipse.
5. Etudier le code de la méthode hashCode().

14. Le mot réservé final

Sur une variable : il permet de définir une constante.

```
public class Harley {
    final int vitesseMax = 250 ;
    ---
}
```

Un attribut final peut être initialisé une seule fois dans le programme, soit lors de sa déclaration, soit dans le constructeur.

Une variable final peut être déclarée et initialisée dans une méthode.

Sur une méthode : la méthode ne peut pas être surchargée.

```
public class Harley {
    final void turnLeft(float degree) {...}
    ---
}
```

Sur une classe : la classe ne peut pas être dérivée (étendue) .

```
final class Harley {
    ---
}
```

15. Le polymorphisme

Le **polymorphisme** caractérise la possibilité des fonctions/méthodes ou des objets de prendre plusieurs formes. Le polymorphisme traite donc de la conversion de type.

Un polymorphisme est dit :

- **Ad hoc**
ou
- **Non ad hoc** (ou **universel**)

Ad Hoc : Il en existe 2 types.

- **Le transtypage implicite ou coercition (coercion).**

Exemple : une fonction/méthode est définie pour recevoir un argument de type **double** et elle est appelée avec un argument de type **int**.

```
class A {
    public void f(double x) {
        System.out.println("Valeur = "+x);
    }
    ---
}
//dans une méthode (main ou autre)
A a = new A();
a.f(2.56);
a.f(5); //casting implicite du type int vers le type double
```

- **La surcharge ou redéfinition des méthodes.**

Ce type de polymorphisme est largement détaillé dans les précédents chapitres.

Non ad hoc ou universel : Il en existe 2 types :

- **Le polymorphisme d'inclusion**: Il consiste essentiellement à un polymorphisme par **sous typage** en programmation objet. Dans le cas le plus courant, les instances d'une classe peuvent référencer les instances de ses sous-classes.

Si **B** extends **A** alors tout **B** est un **A**.

L'exercice du sac postal utilise le polymorphisme universel par sous-typage: on déclare un tableau de 100 objets Courrier qu'on remplit avec des objets Lettre ou Colis qui sont des instances des classes dérivées de Courrier.

Autre exemple

```
class Vehicule {
    public void f(){ System.out.println("Je suis le véhicule");}
}
class Voiture extends Vehicule{
    public void f(){ System.out.println("Je suis la voiture"); }
}

public class PolyMain {
    public static void main(String[] args) {
        Vehicule V1 ;
        Voiture voit1 ;

        voit1 = new Voiture() ;
        voit1.f();

        V1 = voit1 ;
        V1.f() ;
    }
}
```

Affichage :

```
Je suis la voiture
Je suis la voiture
```

La méthode redéfinie appelée dépend du type instancié et non pas du type déclaré. Cette propriété du polymorphisme est très utilisée en POO. Elle est mise en œuvre dans l'exercice 6.5 du sac postal.

Le changement de type (sur typage) suivant génère des erreurs :

```
Vehicule V2 = new Vehicule();
Voiture voit2 ;
Voiture voit3 ;
voit2 = V2; // Impossible de convertir Vehicule en Voiture,
// erreur de compilation
voit3 = (Voiture)V2 ; //ça compile mais erreur d'exécution
```

```
Exception in thread "main" java.lang.ClassCastException: Vehicule
cannot be cast to Voiture at PolyMain.main(PolyMain.java:32)
```

- **Le polymorphisme paramétré**, on passe un paramètre pour choisir le type. Cela correspond à la **généricité** dans la programmation. La **généricité** désigne la possibilité de créer des classes dites **template**.

Exemple :

```

public class Solo<T> {
    private T valeur;

    public Solo(){
        this.valeur = null;
    }

    public Solo(T valeur){
        this.valeur = valeur;
    }

    public void setValeur(T valeur){
        this.valeur = valeur;
    }

    public T getValeur(){
        return this.valeur;
    }
}
// La classe solo peut être utilisée avec tous les types
public class TestGenericite {
    public static void main(String[] args) {
        Solo<Integer> val1 = new Solo<Integer>(12);
        int nbre = val1.getValeur();
        System.out.println("valeur = "+nbre );
        Solo<String> val2 = new Solo<String>("yes");
        String s = val2.getValeur();
        System.out.println("valeur = "+s );
    }
}

```

Le langage Java définit un ensemble de classes appelé Collection. Une collection est un objet de stockage qui peut être vu comme un tableau dynamique.

Une collection en Java permet de stocker des objets d'un même type. Le programmeur peut choisir et déclarer une collection du type d'objet qu'il souhaite stocker dans sa collection.

Les collections utilisent pour cela la généricité.

Le fonctionnement d'une collection est le même quelque soit le type d'objet pour laquelle elle a été déclarée par le programmeur.

Exemple de la déclaration par Java d'une Collection :

class ArrayList<E> : la collection **ArrayList** peut être utilisée par le programmeur pour y stocker des objets de type E, type (classe) qu'il peut lui-même créer.

Exemple:

ArrayList<Abonne > lesAbonnes ;

Q15 Tester les exemples donnés.

16. Compléments sur la généricité

Le caractère générique (joker) <?>

Ce caractère indique un type inconnu.

Exemple avec une méthode (static pour l'exemple proposé):

```
public static void afficheElements(List<?> elements) {
    for (Object o : elements)
        System.out.println(o );
}
```

Une classe quelconque :

```
public class A {
    public void affiche() {
        System.out.println("Je suis A");
    }
    public String toString() {
        return "HaHa";
    }
}
```

La classe de test avec la méthode:

```
import java.util.ArrayList;
import java.util.List;

public class TestABC {
    public static void afficheElements(List<?> elements) {
        for (Object o : elements)
            System.out.println(o );
    }
    public static void main(String[] args) {
        List<A> l = new ArrayList<>();
        l.add(new A());
        l.add(new A());
        l.add(new A());
        afficheElements(l);
    }
}
```

L'exécution de ce programme affiche 3 fois « HaHa ».

Les caractères génériques d'extension <? extends T>

Ces caractères désignent un type T et toutes les classes qui étendent T.

La méthode **void afficheLesElements(List <? extends A> e)** reçoit comme paramètre un objet de type List qui lui-même est paramétré avec tout objet de type A ou instance d'une classe dérivée de A.

La méthode **void afficheElement(Tampon <? extends A> e)** reçoit comme paramètre un objet de type Tampon qui lui-même est paramétré avec tout objet de type A ou instance d'une classe dérivée de A.

Les classes :

```
public class A {
    public void affiche() {
        System.out.println("Je suis A");
    }
    public String toString() {
        return "HaHa";
    }
}
```

```
public class B extends A {
    public void affiche() {
        System.out.println("Je suis B");
    }
    public String toString() {
        return "BeBe";
    }
}
```

```
public class C extends A {
    public void affiche() {
        System.out.println("Je suis C");
    }
    public String toString() {
        return "CeCe";
    }
}
```

La classe Tampon est une classe paramétrée, elle accepte comme paramètre tout objet de type A ou instance d'une classe dérivée de A.

```
public class Tampon <X extends A> {
    private X x ;
    public Tampon(X x) {
        this.x =x ;
    }
    public void affiche() {
        System.out.println("Je suis Tampon");
        x.affiche();
    }
}
```

```
import java.util.ArrayList;
import java.util.List;
```

```
public class TestABC {
    public static void afficheLesElements(List <? extends A> e) {
```

```

        for (Object o : e)
            System.out.println(o );
    }
    public static void afficheElement(Tampon <? extends A> e) {
        e.affiche();
    }
    public static void main(String[] args) {
        List<A> l = new ArrayList<>();
        l.add(new A());
        l.add(new B());
        l.add(new C());

        afficheLesElements(l) ;

        Tampon<B> var = new Tampon<B>(new B()) ;
        Tampon<C> var1 = new Tampon<C>(new C()) ;
        Tampon<A> var2 = new Tampon<A>(new A()) ;
        afficheElement(var);
        afficheElement(var1);
        afficheElement(var2);
    }
}

```

Affichage obtenu

HaHa

BeBe

CeCe

Je suis Tampon

Je suis B

Je suis Tampon

Je suis C

Je suis Tampon

Je suis A

Les caractères génériques super < ? super T>

Ces caractères désignent un type T et toutes les super classes de T.

Prenons l'exemple d'une méthode :

```

public void afficheElement(Tampon <? super A> e) {
    e.affiche();
}

```

Tester les exemples précédents.

17. Liste d'arguments variables

La liste variable d'arguments est précisée par les 3 points ...

void demomethod(String... args)

1^{er} exemple:

```

public class Sample{

```

```

void demomethod(String... args) {
    for (String s : args) {
        System.out.println(s);
    }
}

public static void main(String args[] ){
    new Sample().demomethod("Ram", "Rahim", "Robert");
    new Sample().demomethod("Krishna", "Kasyap");
    new Sample().demomethod();
}
}

```

Cela donne :

```

Ram
Rahim
Robert
Krishna

```

2^{ème} exemple:

```

public class VarargsExample{
    void demoMethod(String name, int age, int... marks) {
        System.out.println();
        System.out.println("Name: "+name);
        System.out.println("Age: "+age);
        System.out.print("Marks: ");
        for (int m: marks) {
            System.out.print(m+" ");
        }
    }
}

public static void main(String args[] ){
    VarargsExample obj = new VarargsExample();
    obj.demoMethod("Krishna", 23, 90, 95, 80, 69 );
    obj.demoMethod("Vishnu", 22, 91, 75, 94 );
    obj.demoMethod("Kasyap", 25, 85, 82);
    obj.demoMethod("Vani", 25, 93);
}
}

```

Cela donne :

```

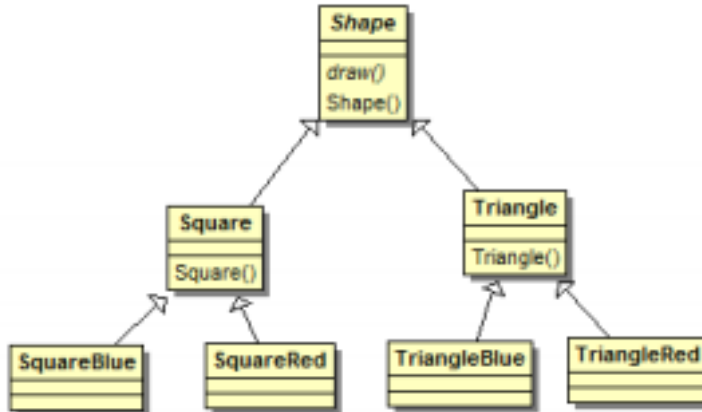
Name: Krishna
Age: 23
Marks: 90 95 80 69
Name: Vishnu
Age: 22
Marks: 91 75 94
Name: Kasyap
Age: 25
Marks: 85 82
Name: Vani
Age: 25
Marks: 93

```

Tester ces exemples.

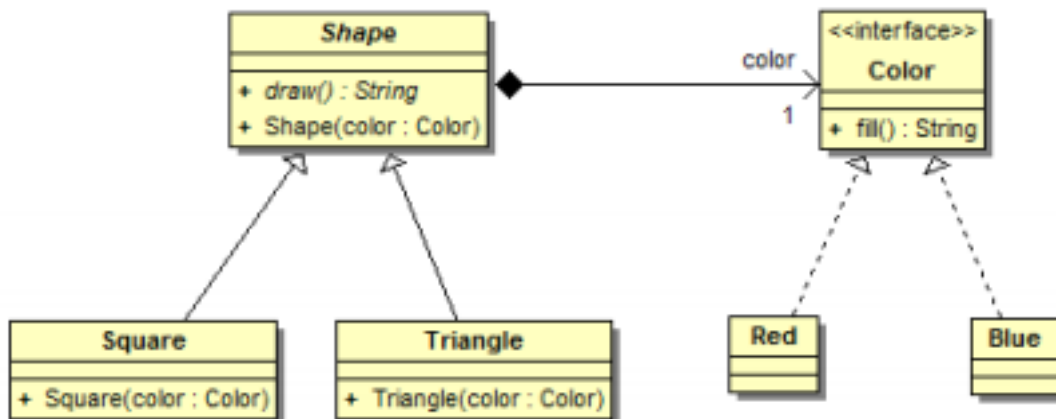
18. Le pattern « Bridge »

Imaginons qu'on souhaite modéliser la réalisation de figures géométriques quelconques colorées, par exemple un triangle rouge, un triangle vert, un carré rouge... Une modélisation possible serait la suivante:



Il faut créer de nouvelles classes pour réaliser un carré vert, un carré orange, un triangle vert...

La couleur est une propriété commune à toutes les formes, on peut alors ajouter cette propriété à la classe abstraite sous la forme d'une **composition**.



On peut donc maintenant:

- créer de nouvelles formes indépendamment de la couleur,
- créer de nouvelles couleurs indépendamment des formes.

On a séparé les 2 abstractions (Shape et Color) qui peuvent être modifiées indépendamment l'une de l'autre, en tenant compte uniquement des besoins de l'application.

Le code Java:

```

public interface Color {
    String fill() ;
}

////////////////////////////////////
public class Blue implements Color {
    @Override
    public String fill() {
// TODO Auto-generated method stub
        return "La couleur est bleue";
    }
}
  
```

```

}
////////////////////////////////////
public class Red implements Color {
    @Override
    public String fill() {
// TODO Auto-generated method stub
        return "La couleur est rouge";
    }
}
////////////////////////////////////
public abstract class Shape {
    protected Color color;

    public Shape(Color color) {
        this.color = color;
    }

    abstract public String draw();
}
////////////////////////////////////
public class Square extends Shape {
    public Square(Color color) {

        super(color);
    }

    @Override
    public String draw() {
// TODO Auto-generated method stub
        return "Carré dessiné. " + color.fill();
    }
}
////////////////////////////////////
public class Triangle extends Shape {
    public Triangle(Color color) {
        super(color);
    }

    @Override
    public String draw() {
// TODO Auto-generated method stub
        return "Triangle dessiné. " + color.fill();
    }
}
////////////////////////////////////
public class Main {
    public static void main(String[] args) {
        Square c = new Square(new Red());
        System.out.println(c.draw());
        Square c1 = new Square(new Blue());

```

```
        System.out.println(c1.draw());
        Triangle t = new Triangle(new Blue());
        System.out.println(t.draw());
    }
}
```

Exercice

- Tester ce programme.
- Ajouter une forme Cercle et la couleur vert dans l'exemple précédent. Tester.
- Des véhicules -voitures, camions...- peuvent être équipés d'une boîte de vitesses manuelle ou d'une boîte automatique.

Proposer un diagramme de classes en utilisant le pattern bridge pour modéliser l'énoncé précédent.

19. Les « java bean »

Un JavaBean est une classe Java qui respecte les règles suivantes:

- les attributs (propriétés) sont privés,
 - les attributs sont accessibles grâce aux accesseurs getters -getX- et setters -setX-.
- Pour un booléen, le nom du getter isX est autorisé.
- la classe est munie d'un constructeur sans argument,
 - la classe est Serializable.