

Langage Java Chapitre 4

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.

1. Les modificateurs d'accès

Tester l'exemple suivant

```
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 `String prenom` sont précisés **public**.

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

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

La classe test00 suivante est à l'extérieur du package agenda, elle met en évidence le fonctionnement des modificateurs d'accès **private** et **public**.

```
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) ;
    }
}
```

Exercice 1

- 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 ?

Quelle est la solution pour afficher la donnée membre 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**

```
public void setNom(String Nouveaunom) {
    Nom = Nouveaunom ;
}
public void setPrenom(String Nouveauprenom) {
    Prenom = Nouveauprenom ;
}
```

Remarque : il est 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

Individu
- Age : int - Nom : String - Prenom : String
+ Individu (nom : String , prenom : String, age :int) + getAge() : int + getPrenom() : String + getNom():String + afficheIndividu() :void

Exercice 3:

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

3. Le mot clé this

On choisit un exemple pour expliquer l'opérateur **this**.

Travail

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 **réfère 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.

Tester le fonctionnement des classes avec cette variante du constructeur.

Exercice 4

Soit les classes `Individu` et `Test01` 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,.
// A compléter pour la saisie de l'âge.
// A compléter pour créer un objet pers1 de type Individu

    pers1.afficheIndividu();
}
}

```

Travail

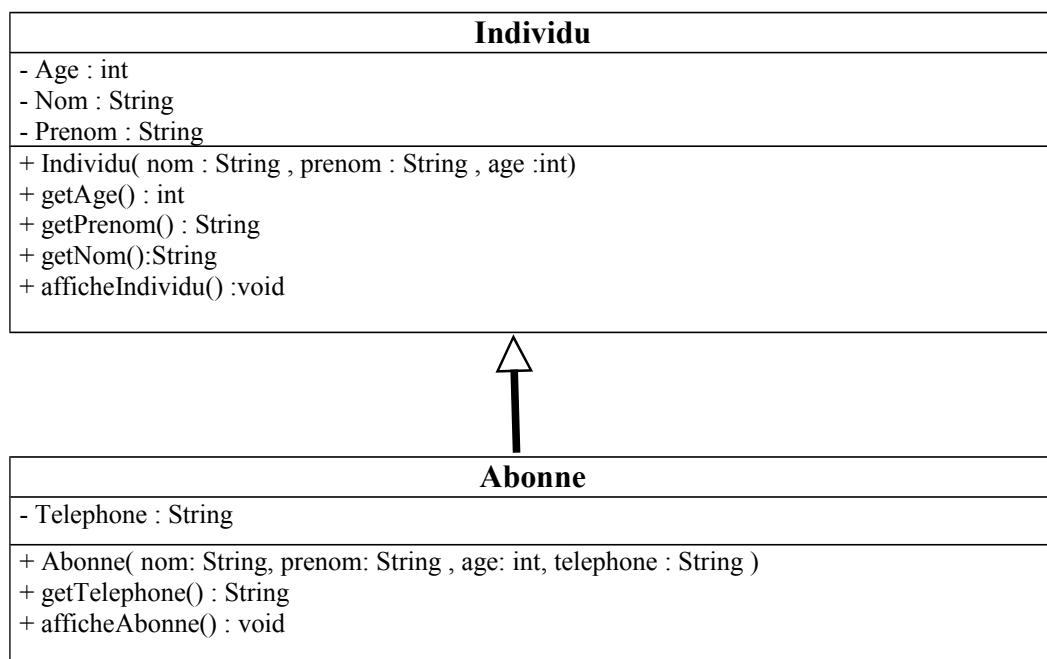
Compléter et compiler le fichier Individu.java.

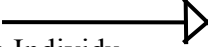
Compléter le fichier Test01.java

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. 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;
import java.io.*;

public class Individu {
    private String Nom;
    private String Prenom;
    private int Age ;
    public Individu(String nom,String prenom, int age ){
        Nom = nom;
        Prenom = prenom;
        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;
import java.io.*;
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
        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 nom et prenom. La classe Abonne ajoute une donnée telephone, en réutilisant les données de sa super classe.

La sous-classe (classe dérivée) dispose directement des attributs et des méthodes de la super classe (classe mère) 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)** ;

Attention : Il est obligatoire de déclarer un constructeur dans une classe dérivée dès lors qu'un constructeur avec arguments est déclaré dans la 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**.

Si aucun constructeur n'est défini dans la classe mère, 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	Constructeur sans argument	Constructeur avec arguments
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().

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 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() ;
        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 sous-classe 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 à 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() {---}**

6. Exercices

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

6.2 Variante : Ecrire les 2 classes dans le même fichier, puis 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.

Ecrire un second constructeur de la classe Abonne pour saisir au clavier le téléphone.

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 sans utiliser les accesseurs. 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!) 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 **String**, on aurait pu choisir un **float** (attribut protected, car #).

Vitesse est un **booléen**, précise l'urgence (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

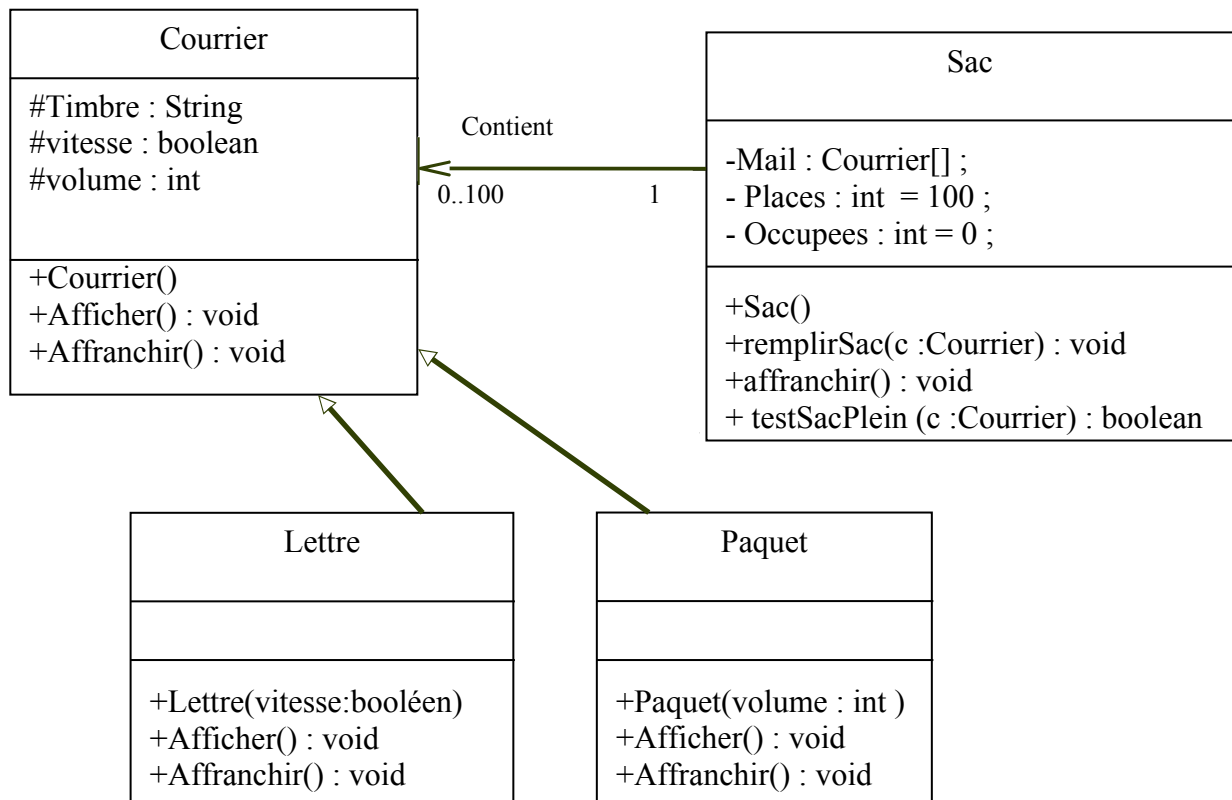
Elles dérivent de la classe `Courrier`.

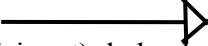
Les constructeurs sont spécialisés pour chacun de ces courriers, l'un reçoit la vitesse comme argument (pour une Lettre), l'autre le volume (pour un Paquet).


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 de test).



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

La flèche ouverte  signifie une association (lien) entre les 2 classes : 1 objet « sac » peut contenir de 0 à 100 objets « courrier ».

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

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

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 le stockage : Mail et un tableau de 100 Courrier.

La méthode **remplirSac(c :Courrier)** de la classe Sac remplit le tableau de Courrier (Mail[]) avec des objets Lettre ou des objets Paquet. Attention, une Lettre occupe une case du tableau Mail mais un Paquet peut occuper plusieurs cases.

Cet exercice met en application le polymorphisme: on déclare un tableau de Courrier dans la classe Sac, et on remplit ce tableau par des objets de type Lettre ou Paquet, qui dérivent de la classe Courrier. 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 Mail) 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.

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;
import java.io.*;
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= 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.

```
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 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`.

Exercices :

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 e) 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 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+"");
    }
    abstract double surface() ;
    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;
    }
    double surface() {
        return Math.PI*Math.pow(rayon, 2.);
    }
    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.

9.3 Le sac postal, version 2

Modifier l'exercice du sac postal en déclarant abstraite la classe `Courrier` ainsi que les méthodes

<i>Courrier</i>
<code>#Timbre : String</code> <code>#vitesse : boolean</code> <code>#volume : int</code>
<code>+Courrier()</code> <code>+Afficher() : void</code> <code>+Affranchir() : void</code>

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*.

10. 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 contenant du code, alors qu'une interface ne contient que des déclarations de méthodes (prototypes de méthodes, méthodes non codées).

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

La classe qui implémente une interface doit coder 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 une seule méthode.

```
package agenda;
public interface affichable{
    void affichetout() ;
}
```

L'intérêt de l'interface, dans cet exemple est d'avoir une fonction affichetout() que l'on pourra coder entièrement à notre guise.

- Ecrivons une classe Abonne03 dérivée de Abonne et implémentant l'interface affichable.

```
import agenda.Abonne;
import agenda.affichable;
import java.io.*;

public class Abonne03 extends Abonne implements affichable {

////////// La fonction affichetout() de l'interface affichable est codée ici.
public void affichetout()
{
    System.out.println("Nom: "+getNom()+ " Prénom: "+getPrenom()+"\n");
    System.out.println("Téléphone : "+getTelephone()+"\n");
}
////////////////////////////////////
public Abonne03(String n,String p, int a, String t)
{
```

```

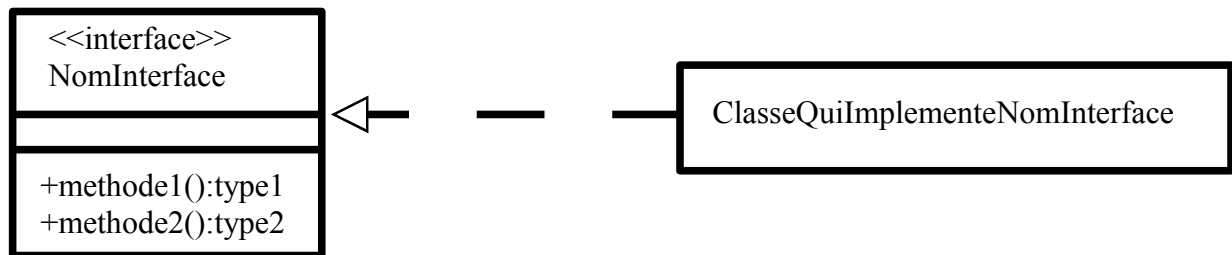
super(n,p,a);
....
}
....
} ///////////////Fin de la classe

```

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.

Règles

En JAVA, une classe peut implémenter plusieurs interfaces.

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

En JAVA, un objet peut avoir plusieurs types : le type de sa propre classe mais également 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

```

Exercices

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

10.2 Reprendre/Relire le paragraphe 13 page 12 du chapitre 2 (interface Comparable).

11. 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.

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 ; //l'objet véhicule référence un objet voiture , on référence
                    // ici un objet instance d'une classe dérivée par un objet
                    // instance classe mère : il y a transtypage, V1 est maintenant
        V1.f() ;     // un objet voiture
    }
}
```

Affichage :

Je suis la voiture
Je suis la voiture

V1 référence un objet Voiture

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 );
    }
}
```