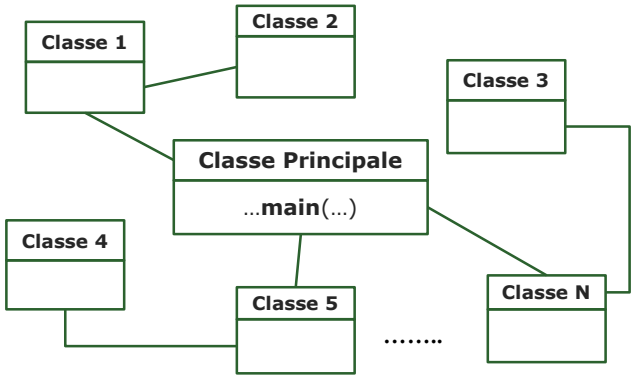
Chapitre 3

POO en Java

Application Orientée Objet

- Une application est créée à partir d'un ensemble de classes qui coopèrent pour effectuer une tâche.



Classe

- Une ***classe*** est une collection
  - D'***attributs*** contenant des valeurs
  - De ***méthodes*** définissant des traitements
- Les attributs et les méthodes sont nommés ***membres*** de la classe:
  - Les attributs sont les ***membres données***
  - Les opérations sont les ***membres fonctions***

Classe

- Exemple :

```
public class Etudiant {

   // les attributs

    private String CNE;
    private String nom;
    private String prenom;

   // constructeur paramétré

    public Etudiant(String CNE, String nom, String prenom) {
       this.CNE = CNE;
       this.nom = nom;
       this.prenom = prenom;
    }
```

Classe

```
    // getters & setters

      public String getCNE() {   return CNE;   }
      public void setCNE(String CNE) {  this.CNE = CNE; }

      public String getNom() { return nom; }
      public void setNom(String nom) { this.nom = nom; }

      public String getPrenom() { return prenom;  }
      public void setPrenom(String prenom) { this.prenom = prenom;   }

     // méthode afficher

    public void afficher(){
        System.out.println (" CNE = " + this.CNE+ " Nom = " + this.nom
                                + " Prénom = "   + this.prenom );
    }
}
```


Classe : Utilisation

- Pour pouvoir utiliser une classe il faut créer un objet.
- Relation entre Objet et Classe :
  - Une **classe** peut être considérée comme un **Plan** à partir duquel on crée une ou plusieurs **maisons** qui sont des **objets**.
- Un objet est appelé **instance** d'une classe
- Un objet est caractérisé par :
  - Un **état** (valeur pour des attributs) et
  - Un **ensemble d'opérations** qui décrivent son comportement

Classe : Utilisation

- Pour créer un objet, il faut :
  1. Déclarer une variable qui va le contenir.
     - Ex : Etudiant e;
  2. Lui allouer de la mémoire et fournir des valeurs pour ses attributs en appelant son constructeur.
     - Ex : e= new Etudiant ("1712150256", "Ali", "Taha");
- Remarque :
  - Il est possible de tout réunir en une seule déclaration :

    Etudiant e=new Etudiant ("1712150256", "Ali", "Taha");

Classe : Utilisation

- Appel des attributs et des méthodes selon visibilité :
  - Attributs : nomObjet.nomAttributs
  - Méthodes : nomObjet.nomMethodes()
- Exemple:
  - e.afficher()

Classe : Utilisation

- **Application**:
  - Écrire une classe Main (contenant la méthode main()) dans laquelle vous allez :
    - Créer un objet e1 de type Etudiant et l'afficher
    - Modifier son nom et le réafficher
    - Créer un objet e2 de type Etudiant et l'afficher
    - Affecter le prénom de e1 à e2 et le réafficher

Classe : Utilisation

```
public class Main {
//méthode
 public static void main (String args[]) {
   Etudiant e1= new Etudiant ("1712150256", "Ali", "Taha");
   e1. afficher();

   e1.setNom("Tahiri") ;
   e1. afficher();

   Etudiant e2=new Etudiant("1712130251", "Omar", "Omari");
   e2.afficher();

   e2.setPrenom(e1.getPrenom());
   e2.afficher();
 }
}
```

Classe : Constructeurs

- Un objet est toujours créé par le biais d'une méthode appelée constructeur.
- Un constructeur est une méthode spéciale qui sert à initialiser les attributs (après réservation de la mémoire) d'un objet lors de sa création.
- Il porte toujours le nom de la classe pour laquelle il est définie.
- Il est public et n'as pas de type de retour
- Une classe peut avoir un ou plusieurs constructeurs.

Classe : Constructeurs

- Il existe trois types de constructeurs :
  - Par défaut : pour la création puis l'initialisation d'un objet dans le cas ou le programmeur omet de donner des valeurs.
  - Paramétré: pour la création puis l'initialisation d'un objet avec des valeurs données par le programmeur .
  - Par recopie: pour la création puis l'initialisation d'un objet en copiant les valeurs d'un autre objet.

Classe : Constructeurs

- La classe Etudiant de l'exemple précédent comporte un constructeur paramétré.
- Dans le cas ou l'utilisateur ne fournit pas de valeurs explicites, des valeurs par défaut peuvent être proposées.
- Ce type de constructeurs est appelé : **constructeur par défaut**

Classe : Constructeurs

- **Exemple 1**: public Etudiant() { }
  - Dans ce cas Java fournit des valeurs par défaut aux attributs selon leurs types au moment de leurs création.
- Les valeurs par défaut lors de l'initialisation sont :
  - boolean : false
  - byte, short, int, long : 0
  - float, double : 0.0
  - Char : \u0000
  - Classe : null

Classe : Constructeurs

- Dans notre exemple le constructeur :
  - `public Etudiant() { }`
- Est équivalent à :
  -
```
public Etudiant() {
    this.CNE = null;
    this.nom = null;
    this.prenom = null;
}
```


Classe : Constructeurs

- **Exemple 2** :

```
public Etudiant() {
    this.CNE = "0000000000";
    this.nom = "Pas de nom";
    this.prenom = "Pas de prénom";
}
```

- Dans cet exemple nous avons choisi de donner d'autres valeurs par défaut en accord avec notre besoin

Classe : Constructeurs

- Un constructeur par recopie permet la recopie des attributs d'un objet dans un autre:
- Exemple :

```
public Etudiant(Etudiant etudiant) {
                    this.CNE = etudiant.CNE;
                    this.nom = etudiant.nom;
                    this.prenom = etudiant.prenom;
  }
```

- Remarque : ***this*** veut dire l'objet en cours de création

Classe : Constructeurs

```
public class Main {
//méthode
 public static void main (String args[]) {
     Etudiant e1= new Etudiant ();
     e1. afficher();
     e1.setCNE("1712130250");
     e1.setNom("Ali");
     e1.setPrenom("Jamali");
     e1. afficher();
     Etudiant e2=new Etudiant("1712130251", "Omar", "Omari");
     e2.afficher();
     Etudiant e3=new Etudiant(e2);
     e3.afficher();
     e3.setCNE("1712130252");
     e3.setNom("Jamali");
     e3.afficher();
 }
}
```

Classe : Constructeurs

- Soit les instructions :
  - Etudiant e1= new Etudiant("1712130251", "Omar", "Omari");
  - Etudiant e2;
  - e2=e1;
- L'instruction « e2=e1 » :
  - Ne définit pas un nouvel objet
  - Elle copie la référence de l'objet e2 dans e1

Classe : Destructeurs

- Un destructeur est une méthode qui libère la mémoire allouée par les constructeurs.
- En java, cette fonction est prise en charge automatiquement par le **garbage collector**
- Cependant pour des traitements avancés referez vous à la méthode finalize() de la classe Object.

Classe : Unicode

- Java adopte Unicode comme jeu de caractères
- Unicode est un jeu de caractère standards englobant les alphabets du monde entier.
- Grace à Unicode on peut coder dans n'importe quelle langue (à condition de respecter les mots réservés)
- Un caractère Unicode est codé sur 16 bits
- Ceci permet de coder 65536 caractères
- Programmer en arabe est juste une décision :)

Classe : Unicode

Exemple :

```
public class طالب{
  private String  اسم;
  private String  لقب;
  public إنسان(){
      اسم="علي ";
      لقب="ياسين ";
  }
  public String  خذ_الاسم() {   return اسم;   }
  public void    بدل_الاسم (String اسم_جديد){
      اسم=اسم_جديد ;
  }
  public String  خذ_اللقب() {   return لقب;  }
  public void  بدل_اللقب( String لقب_جديد) {
     لقب=لقب_جديد ;
  }
  public void عرض(){
    System.out.println( اسم+","+لقب );
  }
}
```

Classe : Unicode

```
//الصنف تجربة

public class تجربة{

//الطريقة الرئيسية

 public static void main (String args[]) {
    طالبة ط =new طالبة();
    ط.إعرض();
    طالبة ط =new طالبة();
    ط.بدل_الاسم("عمر");
    ط.بدل_اللقب"الفاروق");
    ط.إعرض();
}
}
```


Classe : modificateurs

- Les modificateurs de visibilité d'une classe sont :
  - **public** : La classe est accessible partout
  - **Sans modificateur** : La classe n'est accessible que par les autres classes de son package
- Autres modificateurs
  - **abstract** : indique que la classe est abstraite
  - **final** : L'héritage de la classe est bloqué.
  - **private** : dans le cas d'une classe imbriquée
- Remarque : Les modificateurs **abstract** et **final** (resp. **public** et **private**) sont mutuellement exclusifs.

Attributs : modificateurs

- Les données d'une classe sont contenues dans des variables nommées attributs.
- Il existe 3 modificateurs de visibilité pour un attribut :
  - **public** : n'importe quelle classe peut accéder à cet attribut.
  - **protected** : seule la classe, ses sous-classes et les classes du même package peuvent accéder à cet attribut.
  - **private** : seule la classe elle-même peut accéder à cet attribut.

Attributs : modificateurs

- Autres modificateurs
  - **volatile:** prévient le compilateur de ne pas mettre les valeurs de l'attribut en cache de peur d'avoir des inconsistances dans le cas ou de multiple threads accèdent à cet attribut.
  - **transient** : empêche la sauvegarde de la valeur de l'attribut en cas de sérialisation de son objet car la valeur de l'objet peut changer en cours de temps.

Attributs : encapsulation

- Il est déconseillé de déclarer les attributs comme ***public***.
- Il faut les déclarer ***private*** (***protected*** à la rigueur) et les doter de méthodes d'accès qui ont la forme :
  - getNomAttribut () : pour lire un attribut
  - setNomAttribut (nouvelleValeur) : pour le modifier
- L'emploi de ces méthodes garantit la protection des attributs de fausses modifications.
- C'est le fameux principe de l'encapsulation.
- **Remarque** : un attribut disposant de getter/setter est appelé propriété

Attributs : types

- Un attribut peut être :
  1) Une constante
  2) Une variable d'instance,
  3) Ou une variable de classe
- Les attributs constantes :
  - Les constantes sont définies avec le mot clé final
    ⇨ leur valeur ne peut pas être modifiée une fois qu'elles sont initialisées..
  - Exemple :

```
public class MaClasse {
  final double pi=3.14 ;
}
```

Attributs : variables d'instances

- Les variables d'instances :
  - Une variable d'instance nécessite simplement une déclaration de la variable dans le corps de la classe.
  - Chaque instance de la classe a sa propre valeur de la variable.
  - Exemple :
    - Dans la classe ***Etudiant***, les attributs ***nom*** et ***prenom*** sont des variables d'instances.
    - Ainsi, dans un objet **e1** l'attribut nom, par exemple, vaut « *Ahmed* » et dans un autre objet **e2** elle vaut « *Omar* ».

Attributs : variables de classes

- Les variables de classes :
  - **Exercice** : Dans un parc de voitures, à chaque achat d'une voiture un objet de type de classe Voiture doit être créé.
    Écrire une classe voiture ayant :
    - Un attribut ***marque*** de type **String**
    - Un attribut ***total*** de type **int** (représentant le nombre total d'objets Voiture créés)
    - Un constructeur paramétré
    - Une fonction qui affiche la marque de la voiture
    - Une fonction qui affiche le nombre total de voitures

    Écrire un programme d'essai

Attributs : variables de classes

- La classe Voiture :

```
public class Voiture {

    private String marque;
    private int total=0;

    public Voiture(String m) {
        marque=m;
        total++;
    }
    public void afficherMarque(){
      System.out.println("la marque est "+marque);
    }
    public void afficherTotal(){
      System.out.println("le nombre de voitures est "+total);
    }
}
```


Attributs : variables de classes

- Le programme d'essai

```
public class Essai {

    public static void main(String[] args) {

      Voiture v1= new Voiture("Laraki Fulgura");

      v1.afficherMarque(); v1.afficherTotal();

      Voiture v2= new Voiture("Laraki Borac");

      v2. afficherMarque(); v2.afficherTotal();

      Voiture v3= new Voiture("Laraki Epitome");

       v3. afficherMarque(); v3.afficherTotal();

    }

}
```

Attributs : variables de classes

- Le résultat de l'exécution :
  - la marque de la voiture est Laraki Fulgura
  - le nombre de voitures est 1
  - la marque de la voiture est Laraki Borac
  - le nombre de voitures est 1
  - la marque de la voiture est Laraki Epitome
  - le nombre de voitures est 1
- **problème** : Le nombre de voitures n'augmente pas.
- **Cause** : chaque objet Voiture a sa propre version de l'attribut **total**.
- **Solution** : l'attribut **total** doit être commun à tous les objets Voitures

Attributs : variables de classes

- L'attribut **total** doit être une variable de classe et non d'instance.
- Les variables de classes sont définies avec le mot clé static
- Chaque objet de la classe partage la même variable.
- La classe voiture devient :

Attributs : variables de classes

- La classe Voiture modifiée:

```
public class Voiture {
    private String marque;
    private static int total=0;

    public Voiture(String m) {
        marque=m;
        total++;
    }
    public void afficherMarque(){
      System.out.println("la marque est "+marque);
    }
    public static void afficherTotal(){
      System.out.println("le nombre de voitures est "+total);
    }
    }
```


Attributs : variables de classes

- **Remarques**:
  - Une méthode doit être déclarée comme statique lorsqu'elle manipule un attribut statique ou qu'elle ne manipule aucun attribut.
  - Appel d'une méthode statique : Voiture.afficherTotal()
  - Une méthode statique ne peut manipuler que des attributs statiques.

Méthodes : déclaration

- Les méthodes sont des fonctions qui implémentent les traitements de la classe.
- L'ordre des méthodes n'as pas d'importance
- Une méthode est identifiée par sa signature
- La signature comprend, le nom de la méthode , les types des paramètres et le type de retour.
- Exemple : `public int somme(int a, int b)`

Méthodes : arguments

- En Java les arguments d'une méthode sont passés :
  - Par valeur : lorsqu'il s'agit d'un type primitif
  - Par Reference : lorsqu'il s'agit d'un objet
- La possession de la référence d'un objet permet de le modifier par le biais de ses méthodes.

Méthodes : arguments

- Exemple :

```
public class Point {
   private int x;
   private int y;

   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
 //on suppose que nous avons les getters et setters
 public void afficher() {
       System.out.println("Point : x=" + x + ", y=" + y );
 }
}
```


Méthodes : arguments

```
public class PassageDeParametres {

   public static void modifier(int x) {
      x = x + 5;
      System.out.println(" A l'intérieur de la méthode : " + x);
   }

    public static void modifier(Point p) {
      p.setX(30);
      p.setY(40);
      System.out.println("A l'intérieur de la méthode :");
      p.afficher();
   }
```

Méthodes : arguments

```
public static void main(String[] args) {
      System.out.println("-------- Cas d'un argument primitif----------");
      int y = 10;
      System.out.println(" Avant la méthode = " + y);
      modifier(y);
      System.out.println(" Après la méthode  = " + y);
      System.out.println("-------- Cas d'un argument objet ----------");
      Point pt=new Point(10,20);
      System.out.print(" Avant la méthode = ");
      pt.afficher();
      modifier(pt);
      System.out.print(" Après la méthode = ");
      pt.afficher();
  }
```


Méthodes : arguments

- Résultat de l'exécution :

```
-------- Cas d'un argument primitif----------
 Avant la méthode = 10
 A l'intérieur  : 15
 Après la méthode  = 10
-------- Cas d'un argument objet ----------
 Avant la méthode = Point : x=10, y=20
 A l'intérieur : Point : x=30, y=40
 Après la méthode = Point : x=30, y=40
```

Méthodes : modificateurs

- Les modificateurs de méthodes sont :
  - **public :** La méthode est accessible aux méthodes des autres classes
  - **private :** L'usage de la méthode est réservé aux méthodes de la même classe
  - **protected :** La méthode ne peut être invoquée que par des méthodes de la classe ou de ses sous classes ou package.
  - **final :** La méthode ne peut être modifiée (redéfinition lors de l'héritage interdite)
  - **static** : la méthode appartient simultanément à tous les objets de la classe.

Méthodes : modificateurs

- **abstract** : la méthode n'as pas de corps et doit être redéfinie par les sous-classes. Exemple : abstract void afficher();
- Sans modificateur : Utilisation réservée exclusivement aux classes du même package.

- Autres modificateurs
  - **synchronized :** la méthode fait partie d'un thread. Lorsqu'elle est appelée, elle barre l'accès à son instance jusqu'à la fin de son exécution.
  - **native :** le code source de la méthode est écrit dans un autre langage et appelée en Java.

Méthodes : modificateurs

- Une méthode ne peut pas être à la fois
  - public et private (resp. public et protected ou private et protected)
  - abstract et final
  - abstract et static

Méthodes : surcharge

- Lorsque dans une classe, plusieurs méthodes portent :

  Le même nom, Le même type de retour, des arguments différents : On dit que la méthode est surchargée
- Exemple:

```
class Affichage{
   public void afficherValeur(int i) {
    System.out.println(" nombre entier =" + i);
   }
   public void afficherValeur(float f) {
    System.out.println(" nombre flottant = " + f);
   }
}
```

- Le compilateur choisi la méthode qui doit être appelée en fonction du nombre et du type des arguments.

Méthodes : surcharge

- La méthode println() est surchargée !!!

void	println()	Terminates the current line by writing the line separator string.
void	println(boolean x)	Prints a boolean and then terminate the line.
void	println(char x)	Prints a character and then terminate the line.
void	println(char[] x)	Prints an array of characters and then terminate the line.
void	println(double x)	Prints a double and then terminate the line.
void	println(float x)	Prints a float and then terminate the line.
void	println(int x)	Prints an integer and then terminate the line.
void	println(long x)	Prints a long and then terminate the line.
void	println(Object x)	Prints an Object and then terminate the line.
void	println(String x)	Prints a String and then terminate the line.

Méthodes : surcharge

- Remarque :
  - Il n'est pas possible d'avoir deux méthodes qui ont deux signatures identiques.
- Exemple FAUX de surcharge:

```
class Affichage{
   public float convertir (int i){
      return((float) i);
   }
   public double convertir (int i){
     return((double) i);
   }
}
```

Méthodes : surcharge

- Résultat à la compilation :
  - C:\>javac Affiche.java
  - Affiche.java:5: Methods can't be redefined with a different return type: double
  - convert(int) was float convert(int)
  - public double convert(int i){
  - ^ 1 error

Méthodes : Enchaînement de références

- Une classe peut disposer d'un attribut qui est de type une classe aussi.
- On peut à partir d'un objet de la première classe, accéder à l'attribut objet de la deuxième classe et appeler ses méthodes.
- Ceci est appelé enchainement des références.
- On peut aller dans la chaine aussi longtemps que nous avons des relations entre les objets.

Méthodes : Enchaînement de références

- Exemple:
  - Soit l'instruction System.out.println("bonjour");
  - Deux classes sont impliquées dans l'instruction :
    - *System* et *PrintStream*.
  - La classe System possède un attribut nommé out qui est un objet de type PrintStream. Println() est une méthode de la classe PrintStream.
  - L'instruction signifie : utilise la méthode *Println()* de la variable out de la classe System

Packages

- Un **package** permet regrouper des classes qui couvrent un même domaine dans un même dossier après compilation.
- L'utilisation de packages permet de simplifier la maintenabilité et l'évolutivité d'une application.
- Pour mettre une classe dans un package il faut commencer son codage par :
  - **package nompackage**;
- Après compilation un dossier avec le même nom du package est créé où le fichier .class est entreposé.

Packages

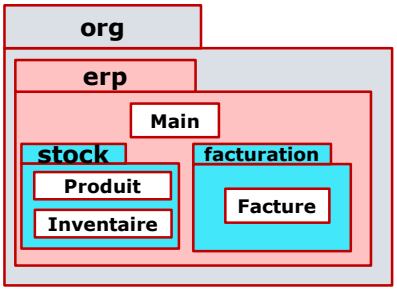
- Exemples :

```
package org.erp.stock;
public class Produit {}
```

```
package org.erp.stock;
public class Inventaire {}
```

```
package org.erp.facturation;
public class  Facture {}
```

```
package org.erp;
public class Main {}
```



Packages : Déclaration

- Remarques:
  - Les packages peuvent être créés graphiquement dans les IDE tels que Netbeans ou Eclipse.
  - Une classe doit être déclarée public pour être visible en dehors de son package de base.
- Convention de nommage des packages:
  - Tout en minuscule, seulement [a-z], [0-9] et le point « . »
  - Tout package doit avoir comme racine par : **com**, **edu**, **gov**, **mil**, **net**, **org** ou code pays comme **ma, fr, dz, tn**… (Standard ISO 3166, 1981).

Packages

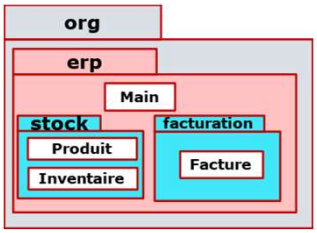
- Il existe plusieurs types de packages :
  - **Les packages standards** : Les packages de la bibliothèque java. Comme le package *java.lang* qui est importé implicitement.
  - **Les packages personnels** : les packages créés par les utilisateurs.
  - **Le package par défaut** : le dossier courant lorsque vous ne spécifiez aucun package particulier.

Packages : utilisation

- Pour utiliser les classes d'un package, il y a 2 méthodes :
  - On importe toutes les classes du package grâce à l'instruction, par exemple : import monpackage.*;
  - On importe juste les classes qui nous intéressent, par exemple :
    - import monpackage.Classe1;
    - import monpackage.Classe2;
- **Remarque** 1: Les classes du même package n'ont pas besoin de s'importer les unes les autres.

Packages : utilisation

- **Remarque 2** : « * » n'importe pas les sous-packages.



- **Exemple :**

  L'instruction « import org.erp.* » n'importe que la classe « Main » et **non pas** les packages « stock » et « facturation » et leurs contenus.

Packages : Collision de classes

- Deux classes portant le même nom dans un programme, on dit qu'il y 'a collision de classes.
- Solution :
  - Qualifier explicitement le nom de la classe avec le nom complet du package.
  - L'import est inutile dans ce cas
- Exemple :
  - Nous souhaitons utiliser deux classes qui portent le même nom (Humain) mais qui se trouvent dans deux différents packages (collision.pkg1 et collision.pkg2).

Packages : Collision de classes

```
package collision.pkg1;

public class Humain{
      public void parler(){
       System.out.println("Je parle");
      }
}
```

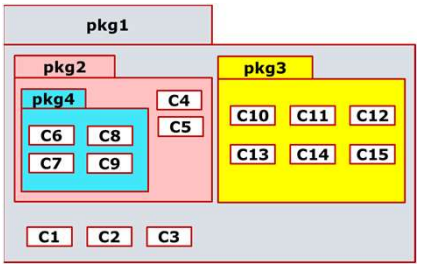
```
package collision.pkg2;

public class Humain{
      public void discuter(){
       System.out.println("Je discute");
      }
}
```

```
package collision;
public class Main {
   public static void main(String[] args) {
      collision.pkg1.Humain h1= new  collision.pkg1.Humain();
       h1.parler();
      collision.pkg2.Humain h2= new  collision.pkg2.Humain();
       h2.discuter();
   }
}
```


Packages : Exercice d'application

- Donner les déclarations des classes C1, C6 et C9 sachant que :
  - La classe C1 utilise les classes C9 et C10
  - La classe C6 hérite de la classe C13 et utilise la classe C3
  - La classe C9 hérite de la classe C5 et utilise les classes C10 à C14.



Packages : Solution

✓ La classe C1 se trouve dans pkg1 et utilise les classes C9 et C10 :

```
package pkg1;
import pkg1.pkg2.pkg4.C9;
import pkg1.pkg3.C10;
public class C1{}
```

✓ La classe C6 se trouve dans pkg4, hérite de la classe C13 et utilise la classe C3 :

```
package pkg1.pkg2.pkg4;
import pkg1.pkg3.C13;
import pkg1.C3;
public class C6 extends C13{}
```

✓ La classe C9 se trouve dans pkg4, hérite de la classe C5 et utilise les classe C10 à C14.

```
package pkg1.pkg2.pkg4;
import pkg1.pkg2.C5;
import pkg1.pkg3.*;
public class C9 extends C5{}
```


Packages : import statique

- Utilisation d'un un membre statique d'une classe :
- Exemple :

```
import java.lang.Math;
public class TestStaticImportOld {
    public static void main(String[] args) {
      System.out.println(Math.PI);
      System.out.println(Math.sin(0));
    }
}
```

Packages : import statique

- L'import statique permet d'accéder directement aux membres statiques d'un classe.
- Exemple :

```
import static java.lang.Math.*;
public class TestStaticImport {
    public static void main(String[] args) {
      System.out.println(PI);
      System.out.println(sin(0));
    }
}
```


Packages : java archive

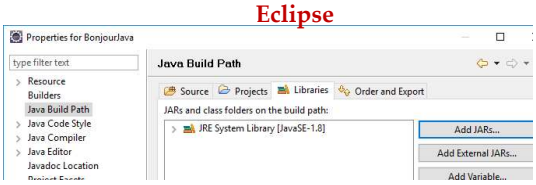
- Une application java (ou une bibliothèque) est composée d'un ensemble de classes rangées dans zéro ou plusieurs packages.
- Livraison application à un client : un fichier .jar (java archive).
- Le fichier jar est généré :
  - Avec Eclipse : menu Fichier puis exporter le projet en fichier Jar et suivre les instructions.
  - Avec Netbeans : La commande « clean & build project » du menu « Run » crée un dossier « dist » contenant le fichier jar.

Packages : java archive

- Pour pouvoir utiliser une bibliothèque externe (fichier .jar) il faut y ajouter une référence:
  - Eclipse : Java Build Path/Add JARs
  - Netbeans : Libraries/Add Library ou Add JAR/Folder
  - DOS : variable d'environnement Classpath

Netbeans

Eclipse



Classes imbriquées

- Une classe imbriquée est une classe définie au sein d'une autre classe.
- Une ***classe imbriquée*** peut être déclarée *private*, *public*, *protected* ou *package*
- Les ***classes externes*** ne peuvent être déclarées que *public* ou *package*
- Une classe imbriquée peut être :
  - Classe imbriquée interne (non statique)
  - Classe imbriquée statique

Classes internes

- Dispose d'un accès direct aux méthodes et aux champs de cet objet.
- Ne peut pas elle-même définir de membres statiques.
- Comme c'est un membre de la classe englobante, elle n'est accessible qu'à partir d'un objet de cette classe si sa visibilité le permet.

Classes internes : Exemple

```
public class Etudiant {
   private String cne;
   private String nom;
   private Filière filière;
   public class Filière{ //peut être private, protected ou public
      private String idF;
      private String nomF;
      public Filière(String idF, String nomF) { this.idF = idF;  this.nomF = nomF; }
      @Override
      public String toString() { return "idF = " + idF + ", nomF = " + nomF ; }
   }
   public Etudiant(String cne, String nom, String idF, String nomF) {
      this.cne = cne; this.nom = nom; this.filière = new Filière(idF, nomF);
   }
@Override
   public String toString() {
      return "Etudiant{ cne = " + cne + ", nom = " + nom + ", filière : " + filière + '}';
   }
}
```

Classes internes : Exemple

```
public class Main {
   public static void main(String[] args) {
      Etudiant et=new Etudiant("8562", "Alaoui Ali", "A1", "SMI");
      System.out.println( et.toString());
     // objet filière indépendant
      Etudiant.Filière f= et.new Filière("A9","SMP");
      System.out.println(f.toString());
   }
}
```


Classes internes : Exemple

- Remarque :
  - Si la classe englobante et la classe imbriquée ont des membres qui ont les mêmes noms alors le membre imbriqué masque celui de la classe englobante
  - Pour appeler explicitement le membre de la classe englobante, il faut le précéder par le nom de sa classe

Classes internes : Exemple

```
public class Englobante {
   public int x = 0;
   class Imbriquee {
      public int x = 1;
      void methodeImbriquee(int x) {
         System.out.println("x = " + x);
         System.out.println("this.x = " + this.x);
         System.out.println(" Englobante.this.x = " + Englobante.this.x);
      }
   }

   public static void main(String[] args) {
     Englobante eng = new Englobante();
     Englobante. Imbriquee imb= eng.new Imbriquee();
      imb. methodeImbriquee(23);
   }
}
```

```
x = 23
this.x = 1
Englobante.this.x = 0
```


Classes imbriquées statiques

- Sont des membres statiques d'une classe englobante
- Peuvent donc accéder mutuellement aux membres privés avec leurs classes englobantes
- Accèdent directement aux attributs et méthodes statiques de la classe englobante mais doivent passer par un objet pour les attributs d'instance.

Classes imbriquées statiques : Exemple

```
public class Etudiant {
   private String cne;
   private String nom;
   private Filière filière;
   public static class Filière{ //peut être private, protected ou public
      private String idF;
      private String nomF;
       public Filière(String idF, String nomF) { this.idF = idF;   this.nomF = nomF; }
      @Override
       public String toString() { return "idF = " + idF + ", nomF = " + nomF ; }
   }
   public Etudiant(String cne, String nom, String idF, String nomF) {
      this.cne = cne; this.nom = nom; this.filière = new Filière(idF, nomF);
   }
  @Override
   public String toString() {
      return "Etudiant{ cne = " + cne + ", nom = " + nom + ", filière : " + filière + '}';
   }
}
```


Classes imbriquées statiques : Exemple

```
public class Main {
   public static void main(String[] args) {
      Etudiant et=new Etudiant("8562", "Alaoui Ali", "A1", "SMI");
      System.out.println( et.toString());
       //objet filière indépendant
      Etudiant.Filière f= new Etudiant.Filière("A9","SMP");
      System.out.println(f.toString());
   }
}
```

Héritage

- L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution.
- Il définit une relation entre deux classes :
  1. une **classe mère** ou **super-classe** ou **classe de base**
  2. une **classe fille** ou **sous-classe** ou **classe dérivée** qui hérite de sa classe mère
- Grâce à l'héritage, les objets d'une classe fille ont accès aux données et aux méthodes de la classe parent et peuvent les étendre.

Héritage : mise en œuvre

- Mise en œuvre de l'héritage : mot clé ***extends***
  - Exemple : class Fille extends Mere { .. }
- Une classe peut avoir plusieurs sous-classes mais ne peut hériter que d'une seule classe.
- Appel des méthodes héritées : mot clé ***super***
  - Exemple : **super**.afficher()
- Appel des constructeurs : ***super(paramètres)***
  - Exemples : **super(**nom, prenom**)**, super**()**

Héritage : Exemple

```
public class Personne {
        private String nom;
        private String prenom;
        public Personne(String nom,String prenom){
          this.nom=nom; this.prenom=prenom;
        }
        public void afficher(){
        System.out.println(" nom= " +nom+"prenom= "+prenom);
       }
}
```


Héritage : Exemple

```
public class Etudiant extends Personne{
     private String CNE;
     public Etudiant (String CNE, String nom, String prenom){
          super(nom,prenom); // toujours le premier
          this.CNE=CNE;
     }
     @Override
     public void afficher(){
       super.afficher();
       System.out.println("CNE = "+CNE);
     }
}
public class Main{
  public static void main(String[] args) {
     Etudiant e=new Etudiant("2008/201","Alaoui","Omar" );
     e.afficher();
  }
}
```

Héritage : affectation mère/fille

- Il est possible d'affecter un objet de la classe Fille à un objet de la classe Mère :
  - **Exemple** : ClasseMere m=new ClasseFille()
  - Le **contraire** est **FAUX**
- Etudiant hérite de Personne, nous pouvons écrire :
  - **Personne p= new Etudiant**("Ali", "Omari", "2008/201");
  - Aussi, toute méthode ayant un paramètre de type **Personne** accepte lors de l'appel un paramètre de type **Etudiant**

Redéfinition

- Lors de l'héritage, si une méthode héritée nous ne convient pas alors on peut la redéfinir.
- La redéfinition consiste à réécrire la méthode:
  - On la précédent de l'annotation @Override
  - Avec **exactement la même signature**
  - Un contenu différent
- Remarque 1 :
  - Il est possible de redéfinir un attribut en changeant son type dans la classe fille.

Redéfinition

- Remarque 2 :
  - Il est possible de bloquer l'héritage à partir d'une classe en lui ajoutant le mot clé « final »
  - Exemple : public **final** class Personne {…..}
- Remarque 3:
  - Il est possible de bloquer la redéfinition d'une méthode en ajoutant le mot clé « **final** »
  - Exemple : public **final** void afficher(){…..}
- Remarque 4:
  - Une méthode statique ne peut pas être redéfinie

Classe Object

- **Object** est la classe mère de toutes les classes en Java (soit directement soit indirectement)
- Ainsi nous pouvons écrire par exemple :
  - **Object o= new Etudiant**("Ali", "Omari", "2008/201");
- Toute méthode ayant un paramètre de type **Object** accepte lors de l'appel un paramètre de n'importe quelle classe.

Classe Object

- Quelques Méthodes (redéfinissable au besoin):
  - String **toString()** : retourne une chaine de caractères qui représente l'objet.
  - boolean **equals(**Object obj**) :** teste l'égalité entre le contenu de deux objets (**==** pour les primitifs)
  - public int hashCode() : génère un code de hachage permettant de réduire le temps de recherche par equals()
  - Class <? extends Object> **getClass() :** retourne la classe de l'objet sur lequel elle est appelée.
  - *protected* Object **clone() :** crée une copie de l'objet sur lequel elle est définie.

Classe Object

- Remarque :
  - Il existe une classe appelée Objects composée de la plupart des méthodes que nous retrouvons dans la classe Object mais statiques comme equals(),hashcode() et toString
  - Exemple d'appel :
    - boolean res=Objects.equals(e1,e2);
    - String ts=Objects.toString(e1);

Classe Object : méthode toString()

- Signature : public String **toString()**
- Permet de transformer un objet en chaine de caractère
- Exemple pour la classe Personne

```
@Override
public String toString() {
 return " Personne { nom=" + nom + " prénom=" + prénom + '}';
}
```


Classe Object : méthode equals()

- Signature : public boolean equals(Object obj)
- Teste l'égalité entre deux objets en comparant la valeur de leurs attributs respectifs
- L'objet défini en paramètre doit toujours être casté à l'objet d'origine :
- Exemple : Etudiant et= (Etudiant) obj;

Classe Object : méthode equals()

- Exemple :

```
@Override
   public boolean equals(Object obj) {
      if (this == obj) { return true; }
      if (obj == null) { return false; }
      if (this.getClass() != obj.getClass()) { return false;}
      final Etudiant other = (Etudiant) obj;
      if (!Objects.equals(this.cne, other.cne)) { return false;}
      if (!Objects.equals(this.nom, other.nom)) { return false;}
      if (!Objects.equals(this.prenom, other.prenom)) {
            return false;}
      return true;
   }
```


Classe Object : méthode hashcode()

- Signature : public int hashCode()
- Accompagne souvent la méthode equals()
- Permet de réduire le temps de recherche car elle génère un code de hachage qui permet de diviser les instances dans des sous-ensembles ayant chacune un code de hachage commun.
- Ainsi lors de la recherche, si le code de hachage de l'objet recherché ne correspond pas au code du hachage d'un sous-ensemble d'objets il l'ignore.

Classe Object : méthode hashcode()

- Contrat respecté par la méthode hashCode():
  - Plusieurs appels renvoient la même valeur entière
  - La valeur de hachage peut changer dans différentes exécutions de la même application.
  - Si deux objets sont égaux selon la méthode equals(), alors leur code de hachage doit être le même.
  - Si deux objets ne sont pas égaux selon la méthode equals(), leur code de hachage ne doit pas forcement être différent.

Classe Object : méthode hashcode()

- Exemple de redéfinition de hashCode():

```
@Override
  public int hashCode() {
     int hash = 5;
     hash = 17 * hash + Objects.hashCode(this.cne);
     hash = 17 * hash + Objects.hashCode(this.nom);
     hash = 17 * hash + Objects.hashCode(this.prenom);
     return hash;
  }
```

Classe Object : méthode getClass()

- Signature : public final Class<?> getClass()
- Retourne un objet de type *Class* contenat les informations sur la classe de l'objet sur laquelle elle est appelée.
- Exemple:
  - Si e est un objet de type Etudiant alors
  - e.getClass().getName() retourne la chaine « Etudiant »

Classe Object : méthode getClass()

- Soit Etudiant une classe implémentant une interface IEtudiant et héritant d'une classe Personne.
  - Etudiant e1= new Etudiant ("A12357i","Ali","Alaoui");
    - e1.getClass().getName() ➔ Etudiant
  - IEtudiant ie = new Etudiant ("A12357i","Omar","Omari");
    - ie.getClass().getName() ➔ Etudiant
  - Personne p= new Etudiant ("A12357i","Omar","Omari");
    - p.getclass().getName() ➔ Personne **// Attention !!!!**
- Solution : operateur **instanceof**

Operateur instanceof

- Syntaxe : ***objet instanceof classe***
- Détermine la classe de l'objet d'une classe fille affecté à un objet de sa classe mère
  - Exemple 1 : if (p **instanceof** Etudiant) { Etudiant e=(Etudiant) p;…}
  - Exemple 2 :

```
Object[] tab= {new Voiture(…),…, new Personne(…),.., new Maison(…)}
for ( Object o : tab){
 if (o instanceof Voiture)  {Voiture v= (Voiture) o;  v. afficherMarque(); }
 else if (o instanceof Personne)  { Personne p= (Personne) o; p.afficherNom(); }
 else if (o instanceof Maison) {Maison m= (Maison) o;     m.afficherAdresse(); }
 …
}
```


Héritage & constructeurs

- Dans un constructeur, il est possible d'initialiser un attribut par appel d'une méthode.
- Exemple :

```
public Etudiant (String CNE, String nom){
    this.CNE=CNE;
    this.Nom=nom;
    this.moyenne=calculerMoyenne();
}
```

- Dans ce cas il faut bloquer la redéfinition de la méthode appelée pour ne pas avoir un comportement anormal.

Héritage & constructeurs

- Dans ce cas la redéfinition peut être bloquée des façons suivantes :
  - Rendre la classe finale
  - Rendre la méthode finale
  - Rendre la méthode privée
  - Rendre la méthode statique

Polymorphisme

- Soit l'exemple de code suivant :

```
class Humain{
      public void parler(){
       System.out.println("Je
           parle");
      }
}

class Arabe extends Humain{
      @Override
      public void parler(){
       System.out.println(" أنا أتحدث
           بالعربية");
      }
}
```

```
class Anglais extends Humain{
      @Override
      public void parler(){
       System.out.println("I speak
           english");
      }
}
class Francais extends Humain{
      @Override
      public void parler(){
       System.out.println("Je parle
           français");
      }
}
```

Polymorphisme

```
class Main {
  public static void main (String[] args) {
      Humain h;
     h=new Arabe();
      h.parler();  // ➔ «  أنا أتحدث بالعربية  »
     h=new Anglais();
    h.parler();   // ➔ « I speak English »
    h=new Français();
    h.parler();  // ➔ « je parle Français »
  }
}
```

- La méthode *parler()* prend différents aspects selon l'objet affecté à l'objet Humain. D'où le nom polymorphisme !!

Polymorphisme

- Même exemple avec une boucle:

```
class Main {
  public static void main (String[] args) {
   Humain[]  tab={new Arabe(), new Anglais(), new Arabe(), new Français(),  new Anglais() };
     for(Humain h : tab)
        h.parler();
  }
}
```

Classes abstraites

✓ Soit la classe Enseignant suivante :

Enseignant.java

```
public class Enseignant {
 private int PPR;
 private String nom;
 private String prenom;
// constructeur
 public Enseignant(int PPR, String nom, String prenom) {
  this.PPR = PPR;
  this.nom = nom;
  this.prenom = prenom;
 }
//getters setters
 public String getNom() {  return nom;  }
 public void setNom(String nom) {  this.nom = nom; }
```


Classes abstraites

```
  public int getPPR() { return PPR;  }
  public void setPPR( int PPR ) {  this.PPR = PPR; }
  public String getPrenom() { return prenom; }
  public void setPrenom(String prenom) { this.prenom = prenom; }
    @Override
    public String toString() {
        return " PPR = " + this.PPR + ", Nom = " + this.nom
                + ", Prénom = " + this.prenom ;
    }
   //méthode
  public void quiSuisje(){  System.out.println("je suis un enseignant");  }
}
```

Classes abstraites

✓ Soit la classe Etudiant suivante :

Etudiant.java

```
public class Etudiant {
 private String CNE;
 private String nom;
 private String prenom;
// constructeur
 public Etudiant (String CNE, String nom, String prenom) {
  this.CNE = CNE;
  this.nom = nom;
  this.prenom = prenom;
 }
//getters setters
 public String getNom() {  return nom;  }
 public void setNom(String nom) {  this.nom = nom; }
```


Classes abstraites

```
  public String getCNE() { return CNE;  }
  public void setCNE( String CNE) {  this.CNE = CNE; }
  public String getPrenom() { return prenom; }
  public void setPrenom(String prenom) { this.prenom = prenom; }
 @Override
    public String toString() {
        return " CNE = " + this.CNE + ", Nom = " + this.nom
                    + ", Prénom = " + this.prenom ;
    }
  //méthode
  public void quiSuisje(){  System.out.println("je suis un étudiant");  }
}
```

Classes abstraites

- Les deux classes, ont des points en commun :
  - Attributs : « nom », « prenom »
  - Constructeurs : même initialisation « nom » et « prenom »
  - Getters /setters : pour « nom » et « prenom »
  - Méthode toString() sauf pour le PPR et le CNE
  - Méthode : quiSuisje() même signature, contenu différent
- Pour « factoriser » nous allons mettre ces points en commun dans une classe de base que nous appellerons Personne

Classes abstraites

- La classe Personne n'est là que pour la généralisation et n'est pas une classe métier donc Il faut bloquer son instanciation en la déclarant avec le mot clé **abstract** !!
- La classe Personne sera composée de :
  - Attributs : « nom », « prenom »
  - Constructeurs : initialisation « nom » et « prenom »
  - Getters /setters : pour « nom » et « prenom »
  - Méthode toString() pour pour « nom » et « prenom »
  - Méthode : quiSuisje() sans contenu qui sera redéfinie par les classes Enseignant et Etudiant→ c'est une **méthode abstraite**

Classes abstraites

- Le code de la classe Personne est comme suit :

Personne.java

```
public abstract class Personne{
  protected String nom;
  protected String prenom;
 // constructeur
 public Personne(String nom, String prenom) {
   this.nom = nom;
  this.prenom = prenom;
 }
//getters setters
 public String getNom() {  return nom;  }
 public void setNom(String nom) {  this.nom = nom; }
```


Classes abstraites

```
 public String getPrenom() { return prenom; }
  public void setPrenom(String prenom) { this.prenom = prenom; }
@Override
    public String toString() {
        return " Nom = " + this.nom + ", Prénom = " + this.prenom ;
    }
  //méthode
  public abstract void quiSuisje();
}
```

Classes abstraites

Enseignant .java

```
public class Enseignant extends Personne{
  private int PPR;
public Enseignant(int PPR, String nom, String prenom) {
    super(nom,prenom);  this.PPR = PPR;
 }
//getters setters
public int getPPR() { return PPR;  }
public void setPPR( int PPR ) {  this.PPR = PPR; }
@Override
public String toString() { return " PPR = " + this.PPR + super.toString();}
@Override
public void quiSuisje(){  System.out.println("je suis un enseignant");  }
}
```


Classes abstraites

Etudiant.java

```
public class Etudiant extends Personne{
 private String CNE;
 public Etudiant (String CNE, String nom, String prenom) {
   super(nom,prenom);    this.CNE = CNE;
 }
//getters setters
 public String getCNE() { return CNE;  }
 public void setCNE( String CNE) {  this.CNE = CNE; }
 @Override
 public String toString() { return " CNE = " + this.CNE + super.toString();
 @Override
 public void quiSuisje(){  System.out.println("je suis un etudiant");  }
}
```

Classes abstraites

```
public class Main {
   public static void main(String[] args) {
      Enseignant ens=new Enseignant("12354","Alaoui","Ali");
      ens.quiSuisJe();
      System.out.println(ens.toString());
      Etudiant et=new Etudiant("125468","Omari","Omar");
      ens.quiSuisJe();
      System.out.println(et.toString());
 }
}
```


Classes abstraites

- Remarques
  - Une classe abstraite ne peut pas être instanciée mais on peut lui affecter un objet de sa classe fille concrète :
    - **Exemple** : Personne p=new Enseignant() ou new Etudiant()
    - Le contraire est FAUX
  - Une méthode abstraite doit forcement être redéfinie au niveau de la classe fille (sauf si elle abstraite aussi)
  - Une classe abstraite peut ne pas contenir de méthodes abstraites.
  - Une classe doit être déclarée abstraite dès qu'une de ses méthodes est déclarée abstraite.

Classes abstraites

- Polymorphisme avec une classe abstraite

```
public abstract class Humain{
 public abstract void parler();
}

class Arabe extends Humain{
  @Override
  public void parler(){
      System.out.println(" أنا أتحدث
          بالعربية");
      }
}
```

```
class Français extends Humain{
      @Override
      public void parler(){
       System.out.println("Je parle
          Français");
      }
}
class Anglais extends Humain{
      @Override
      public void parler(){
       System.out.println("I speak
          english");
      }
}
```


Interfaces

- Une interface définit un ensemble de services (méthodes abstraites) attendus par un client externe.
- L'implémentation de ces services (redéfinition de ces méthodes) est assurée par une classe cachée du client externe
- On dit que la classe implémente l'interface
- L'implémentation est une forme d'héritage
- Ainsi : InterfaceMere f=new ClasseFille()

Interfaces

- Exemple :

```
interface AffichageType {
    void afficher();
}
```

```
class Personne implements AffichageType{
 @Override
  public void afficher() {
      System.out.println(" Je suis une personne ");
     }
}
```

```
class Voiture implements AffichageType{
   @Override
    public void afficher() {
      System.out.println(" Je suis une voiture ");
     }
}
```


Interfaces

```
class Main {
    public static void main (String[] args) {
     AffichageType f;
    f=new Personne();
    f.afficher();
     f=new Voiture();
    f.afficher();
     }
}
```

Interfaces

- Outre les méthodes abstraites (sans le mot clé abstract), une interface peut aussi contenir :
  - Constantes (sans mot clé final)
  - Méthodes par défaut (mot clé default)
  - Méthodes statiques (mot clé static)
- Une classe implémente une ou plusieurs interfaces , lorsqu'elle fournit une redéfinition pour leurs méthodes abstraites
- Une classe est obligée de redéfinir toutes les méthodes abstraites de l'interface implémentée

Interfaces: Exemple

```
import java.time.LocalDate;
public interface ICalcul {
   int effectuerCalcul(int a, int b);
   default void direBonjour() {System.out.println("Bonjour");};
   default void direAurevoir() {System.out.println("Au revoir");};
   static void afficherDate() {System.out.println(LocalDate.now());}
}
```

```
public class Somme implements ICalcul {
   @Override
   public int effectuerCalcul(int a, int b) {  return a+b; }
}
```

```
public class Multiplication implements ICalcul {
   @Override
   public int effectuerCalcul(int a, int b) {   return a*b;  }
   @Override
   public void direAurevoir() {  System.out.println("Bye Bye");  }
}
```

Interfaces: Exemple

```
public class Main {
   public static void main(String[] args) {
      ICalcul.afficherDate();
      ICalcul ic=new Somme();
      ic.direBonjour();
      System.out.println("Somme: "+ ic.effectuerCalcul(3,4));
      ic.direAurevoir();
      ICalcul ic=new Multiplication();
      ic.direBonjour();
      System.out.println("Multiplication: "+ic.effectuerCalcul(3,4));
      ic.direAurevoir();
   }
}
```


Interfaces

- Remarque:
  - Une classe peut à la fois hériter d'une classe (une seule) et implémenter plusieurs interfaces (séparées par des virgules).
  - Dans le cas où plusieurs interfaces (Interface1 et Interface2 par exemple) ont des mêmes nom de méthodes par défaut (méthode1() par exemple) l'appel suivant permet de faire la différence :
    - Interface1.super.methode1()
    - Interface2.super.methode1()

Interfaces

- Une interface peut être d'accès :
  - public : toutes ses méthodes sont implicitement publiques même si elles ne sont pas déclarées avec le modificateur public.
  - Package (sans modificateur) : accessible seulement aux classes et interfaces du même package.

Interfaces

- Remarque :
  - À partir d'une interface il n'est possible d'accéder qu'aux méthodes redéfinies de la classe d'implémentation.
  - Les méthodes propres de la classe ne sont pas accessibles à partir de l'interface.
  - Exemple : Etudiant e= (Etudiant) f;

Interfaces

- Polymorphisme avec une interface

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

class Arabe impelments Humain{
  @Override
  public void parler(){
      System.out.println(" أنا أتحدث
            بالعربية");
      }
}
```

```
class Français impelments Humain{
      @Override
      public void parler(){
       System.out.println("Je parle
            Français");
      }
}
class Anglais impelments Humain{
      @Override
      public void parler(){
       System.out.println("I speak
            english");
      }
}
```


Interfaces : Exercice d'application

- Soit l'interface suivante :

```
public interface Salutation {
    void saluer(String nom);
}
```

- Ecrire trois classes (SalutationArabe, SalutationAnglais, SalutationFrancais) qui implémentent cette interface et redéfinissent la méthode saluer.
- Ecrire une classe Main pour le test, qui demande à l'utilisateur son nom et le code de la langue qu'il souhaite (ar, en ou fr) et qui le salue avec la langue qu'il souhaite.

Interfaces : Solution

```
public class SalutationArabe implements Salutation{
    @Override
    public void saluer(String nom) {
        System.out.println(" Salam alikom "+nom);    }
}
```

```
public class SaluationAnglais implements Salutation{
    @Override
    public void saluer(String nom) {
        System.out.println(" Hi "+nom);    }
}
```

```
public class SalutationFrancais implements Salutation{
    @Override
    public void saluer(String nom) {
        System.out.println(" Bonjour "+nom);    }
}
```


Interfaces : Solution (suite)

```
public class Main {
    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);
        System.out.println("Saisir Nom :");
        String nom=sc.next();
        System.out.println("Saisir code langue Ar/Fr/En");
        String rep=sc.next().toLowerCase();
        Salutation sal;
        switch(rep){
            case "ar": sal=new SalutationArabe();break;
            case "en": sal=new SalutationAnglais();break;
            case "fr": sal=new SalutationFrancais();break;
            default: sal=new SalutationArabe();
        }
        sal.saluer(nom);
    }
}
```

Interface Cloneable

- Pour cloner un objet il faut :
  - Redéfinir la méthode clone() de la classe Object
  - Implémenter l'interface **Cloneable** pour ne pas avoir l'exception **CloneNotSupportedException.**
- Cloneable doit être implémentée pour indiquer à la méthode Object.clone () qu'il est légal d'effectuer une copie champ-à-champ des instances de cette classe.

Interface Cloneable

- Exemple :

```
public class Personne implements Cloneable{
    private String nom;
    private String prenom;
    public Personne(String nom, String prenom) {
          this.nom = nom; this.prenom = prenom;
    }
    public String getNom() { return nom; }
    public void setNom(String nom) {  this.nom = nom;  }
    public String getPrenom() {return prenom; }
    public void setPrenom(String prenom) {this.prenom = prenom;}
```

Interface Cloneable

```
    public void afficher(){
    System.out.println("Nom = "+nom+" , Prenom = "+prenom);    }
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

- Remarque :
  - Aucun code explicite n'est requis pour copier la valeur d'un objet dans un autre

Interface Cloneable

```
public class Main {
 public static void main(String[] args) {
      Personne p1=new Personne("Omar","Omari");
      try {     Personne p2=(Personne)p1.clone();
                p2.setNom("Alaoui");p2.setPrenom("Ali");
                p1.afficher(); // affichera Omar omari
                p2.afficher(); // affichera Ali Alaoui
 }
       catch(CloneNotSupportedException e){
          System.out.println(e.getMessage());
       }
 }
}
```

Classe générique

- Une classe générique est une classe qui admet un ou plusieurs paramètres.
- Exemple :

```
public class Boite<T> {
   private T t;
   public Boite(T t) {  this.t = t;  }
   public T get() { return t;  }
   public void set(T t) { this.t = t; }
}
```

```
public class Main {
   public static void main(String[] args) {
     Boite<Integer> boiteEntier = new Boite<>(10);
     Boite<String> BoiteChaine = new Boite<>("Hello World");
     System.out.println("Valeur entière : "+ boiteEntier.get());
     System.out.println("Valeur chaine : "+ BoiteChaine.get());
  }
}
```


Classe générique

- Remarque: une classe générique peut prendre plusieurs paramètres
- Dans ce cas ils doivent être séparés par des virgules
- Exemple :

```
class Boite<T, S> {
   private T t;
   private S s;

   public void set(T t, S s) {
      this.t = t;
      this.s = s;
   }

   public T getPrem() { return t; }
   public S getSecond() { return s;  }
}
```