Université Sultan Moulay Slimane
Ecole Supérieure de Technologie Béni Mellal
Département ITG
DUT- IDIA 1 - S2
M122 - Ingénierie Logicielle avec Java et UML
Année universitaire: 2024-2025

Support de cours

Ingénierie logicielle avec Java et UML

Partie I: Programmation Orientée Objet en Java

Chapitre II: Programmation Orientée Objet : Java

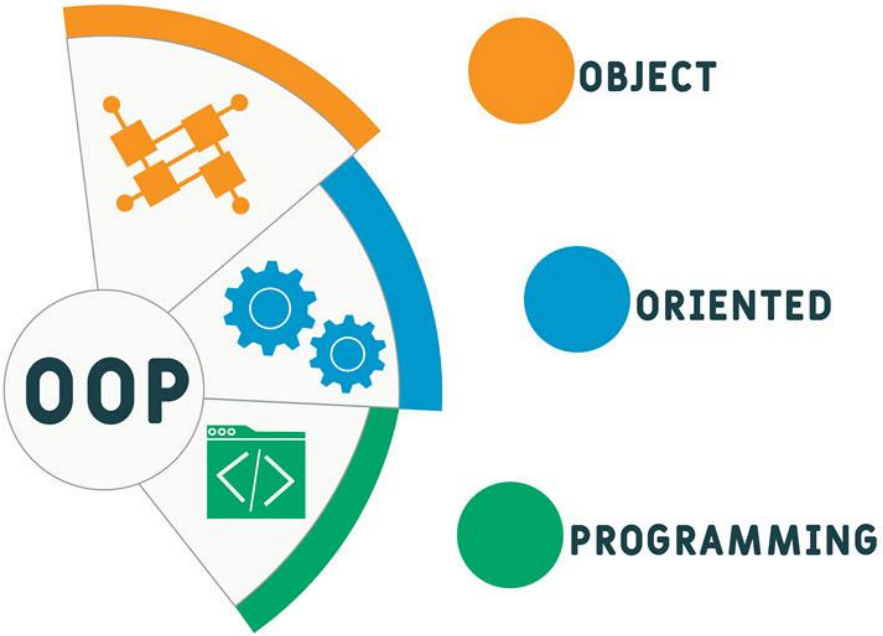
Pr. Siham BAKKOURI
Email: s.bakkouri@usms.ma

Plan

- **LES PRINCIPES DE LA POO**
- **LES CLASSES ET LES OBJETS**
- **L'ENCAPSULATION DES DONNÉES ET DES MÉTHODES**
- **L'HÉRITAGE**
- **LE POLYMORPHISME**
- **L'ABSTRACTION**
- **LES INTERFACES**
- **EXERCICES**

Les principes de la POO

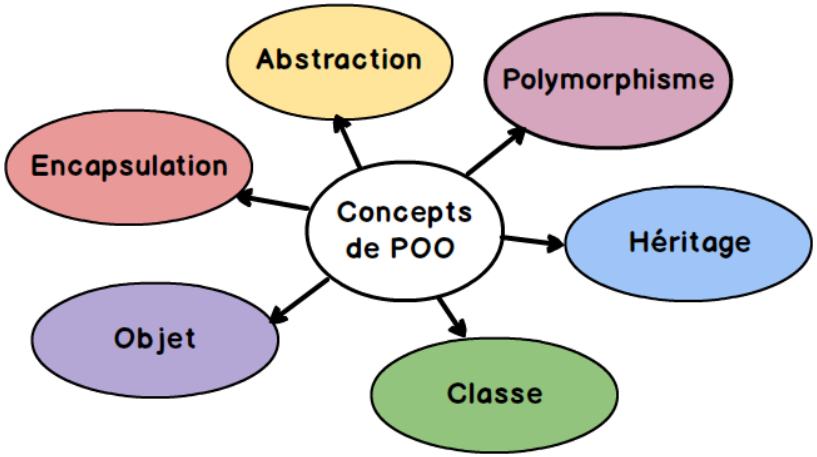
- La Programmation Orientée Objet (**POO**) est un paradigme de programmation basé sur la structuration du code autour d'objets, qui sont des instances de classes.
- Contrairement à la programmation procédurale qui repose sur des fonctions et des procédures, la POO permet une meilleure organisation du code, une réutilisation efficace et une plus grande évolutivité des applications.



Les principes de la POO

→ Les quatre piliers de la POO

- **Encapsulation** : Regroupe les données et les méthodes associées dans une classe tout en contrôlant l'accès via des modificateurs de visibilité (private, public, protected).
- **Héritage** : Permet à une classe d'hériter des attributs et méthodes d'une autre classe, favorisant la réutilisation du code.
- **Polymorphisme** : Autorise une même méthode à avoir plusieurs comportements selon le contexte (surcharge et redéfinition de méthodes).
- **Abstraction** : Cache les détails d'implémentation pour ne montrer que l'essentiel, via des classes abstraites et des interfaces.



→ Objets et Classes

- Un **objet** se définit par ses états (on peut aussi parler de ses caractéristiques) et son comportement.

Exemple d'un objet voiture

Etats	Comportements
marque	accélérer
modèle	passer rapport supérieur
cylindrée	passer rapport inférieur
quantité d'essence	tourner volant
niveau d'huile	ouvrir porte
pression des pneus	fermer porte
nombre de tours	freiner

- Une **classe** est un ***plan*** ou un ***moule*** pour fabriquer des objets.
  - → les ***états*** d'un objet vont être représentés par des ***variables***
  - → les ***comportements*** d'un objet seront représentés par des ***méthodes***.
- Un **objet** est une **instance** d'une classe.

Concepts des classes et des objets

→ Objets et Classes

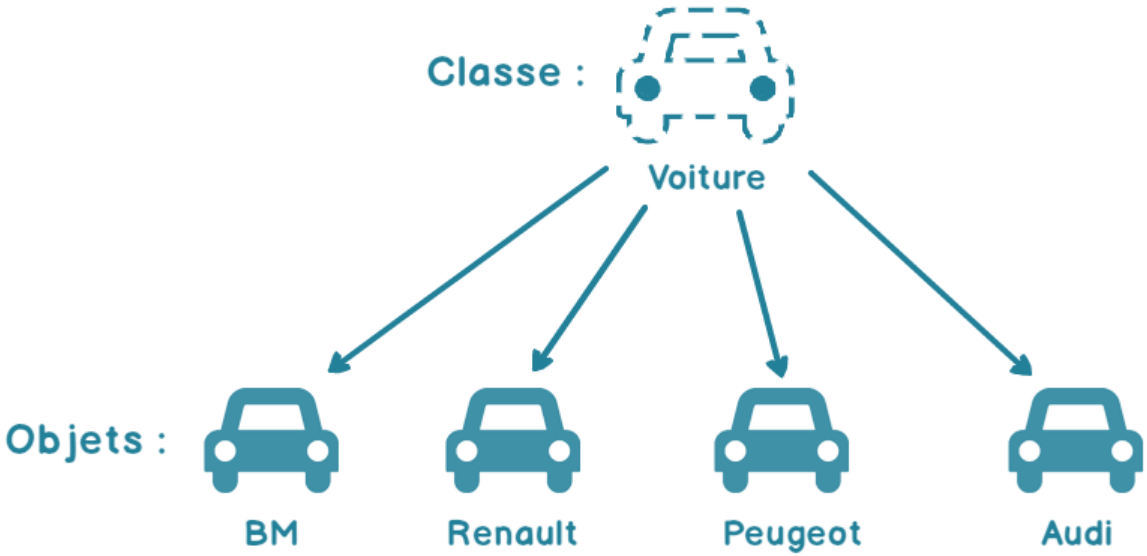
- Une **classe** est un *type abstrait* caractérisé par des propriétés (attributs et méthodes) communes à un ensemble d'objets et permettant de créer des objets ayant ces propriétés.
- Un **objet** ou une **instance de classe** possède un comportement et un état qui ne peut être modifié que par les actions du comportement.

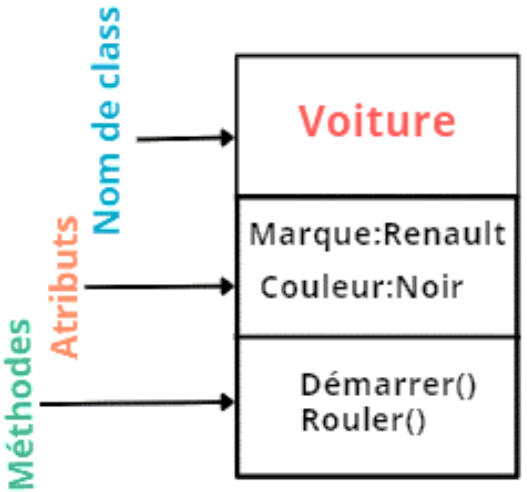
On peut créer une classe **Personnage** car tous les personnages partagent les mêmes caractéristiques. Lorsqu'on veut créer un personnage, on instancie la classe **Personnage**.

Par convention en Java, le **nom d'une classe** commence toujours par une **MAJUSCULE**, tandis que le nom des autres éléments (méthodes, variables, etc.) commence par une minuscule.

→ Objets et Classes

Exemple





→ La bibliothèque de classes Java

- **Java** possède une vaste bibliothèque de classes. Cette bibliothèque est structurée en différents **packages** et **sous-packages**.

  → Documentation Java

Toutes ces classes sont organisées en *packages* (ou bibliothèques) dédiés à un thème précis. Parmi les packages les plus utilisés, on peut citer les suivants :

Package	Description
java.awt	Classes graphiques et de gestion d'interfaces
java.io	Gestion des entrées/sorties
java.lang	Classes de base (***importé par défaut***)
java.util	Classes utilitaires
javax.swing	Autres classes graphiques

→ La bibliothèque de classes Java

- ***Par exemple***, le package **`java.lang`** regroupe des classes fondamentales du langage Java. On y trouve également une classe dédiée à la manipulation des chaînes de caractères, appelée **String**.

```java
public class GInfo {
    public static void main(String[] args) {
        String phrase = "Exemple d'utilisation de la classe String.";

        // Obtention de la longueur de la chaîne
        int longueur = phrase.length();
        System.out.println("Longueur de la phrase : " + longueur);

        // Conversion en majuscules
        String enMajuscules = phrase.toUpperCase();
        System.out.println("Phrase en majuscules : " + enMajuscules);

        // Recherche d'un mot dans la phrase
        boolean contientMot = phrase.contains("utilisation");
        System.out.println("La phrase contient le mot 'utilisation' : " + contientMot);
        // Affiche : La phrase contient le mot 'utilisation' : true
    }
}
```

→ Attributs et variables de classe

- **Variables d'instance** ou **attributs** : ces variables définissent les caractéristiques de l'objet.

  → initialisation ***optionnelle***.

  - **Accès** : **<nom objet>.<nom attribut>**
  - ***Exemple*** : monVelo.couleur

- **Variables de classe** : ces variables sont communes à *toutes* les instances de la classe,

  → déclaration avec le mot clé **static**

  → initialisation **obligatoire**

  - **accès** : **<nom de classe>.<nom variable de classe>**
  - ***Exemple*** : Float.MAX_VALUE

Remarque: Classe **Float** pour encapsuler un nombre flottant float.

Variables de classes : **MAX_VALUE**, **MAX_EXPONENT**, **NaN**, etc.

→ Méthodes d'instance et méthodes de classe

- **Méthode d'instance** : ces méthodes permettent de *modifier* ou d'*accéder* à l'*état* de l'objet.
- **Méthode de classe** : ces méthodes *ne* modifient *pas* l'état interne d'un objet.

Exemple

- la classe **Float**

→ **Méthode d'instance**: **String toString()**: retourne une représentation en chaîne de caractères de l'*objet courant*

→ **Méthode de classe: static String toString(Float f)**: retourne une représentation en chaîne de caractères du float passé en paramètre

```
Float f;
System.out.println(f.toString());
System.out.println(Float.toString(3.1419));
```

→ Les modificateurs d'accès

Les comportements et états d'un objet peuvent être définis par la visibilité de leurs membres :

- **Public** (**public**) : Les membres déclarés comme publics sont accessibles depuis n'importe quelle classe. Ils forment l'interface publique de la classe et peuvent être utilisés par d'autres classes.
- **Privé** (**private**) : Les membres déclarés comme privés ne sont accessibles qu'à l'intérieur de la classe où ils sont définis. Ils ne peuvent pas être utilisés ou modifiés depuis l'extérieur de la classe.
- **Protégé** (**protected**) : Les membres protégés sont accessibles dans le même paquet (package) ainsi que par les sous-classes, même si elles sont situées dans un autre paquet.
- **Sans modificateur / par défaut (**parfois appelé **"package-private") :** L'absence de modificateur signifie que le membre est accessible uniquement dans le même paquet (**package**). Cela permet de limiter la visibilité aux classes du même package.

→ Les modificateurs d'accès

- **Autorisations d'accès**

	public	protected	défaut	private
Dans la même classe	Oui	Oui	Oui	Oui
Dans une classe du même package	Oui	Oui	Oui	Non
Dans une sous-classe d'un autre package	Oui	Oui	Non	Non
Dans une classe quelconque d'un autre package	Oui	Non	Non	Non

→ Les constructeurs

En programmation orientée objet, un constructeur est une méthode spéciale appelée lorsqu'un objet est **instancié**. Son rôle principal est **d'initialiser** les membres de l'objet et de préparer l'objet à être utilisé.. Ils sont caractérisés par :

- **Nom et Signature :** Un constructeur porte le même nom que la classe à laquelle il appartient. Il n'a pas de type de retour défini.
- **Appel par Défaut :** Un constructeur sans arguments est appelé "**constructeur par défaut**". Il est automatiquement appelé lorsqu'un objet est créé sans spécifier d'arguments.

→ On appelle constructeur **par défaut** le constructeur **sans** arguments :

```
public class <nom classe> {
// déclaration des variable d'instances et
// variables de classe
...
// constructeur par défaut
public <nom classe>(){
// corps de la méthode
 }
 }
```

→ Les constructeurs

- **Surcharge de Constructeurs**

La surcharge de constructeurs permet de définir **plusieurs constructeurs** pour une classe, chacun ayant une liste **différente de paramètres**. Cela offre une flexibilité lors de la création d'objets de la classe.

Exemple

```
public class MaClasse {
    private int valeur;

    // Constructeur par défaut
    public MaClasse() {
        this.valeur = 0;
    }

    // Constructeur avec un paramètre
    public MaClasse(int valeur) {
        this.valeur = valeur;
    }

    // Autres membres de la classe...
}
```

Signature d'un constructeur

- Modificateur d'accès (en général public)
- Pas de type de retour
- Le même nom que la classe
- Les arguments sont utilisés pour initialiser les variables de la classe

→ Création d'une instance d'objet

- **Déclaration** : comme pour les types primitifs :

<Nom de la classe> <nom objet>;

- **Création/initialisation** à l'aide du mot clé **new** et appel du constructeur :

new <Nom de classe>(<liste d'arguments>);

- Comme pour les types primitifs, on peut déclarer et initialiser plusieurs objets du même type en même temps.

```
public class MonProgramme {
    public static void main(String[] args) {
        /* Création d'une instance de MaClasse
        en utilisant le constructeur par défaut */
        MaClasse monObjet = new MaClasse();
    }
}
```

→ Création d'une instance d'objet

Exemple

```
public class Personne
{
 public String nom;
  // constructeur par défaut
   public Personnage(){
   nom = ”Inconnu";
}
  //constructeur avec un paramètre
   public Personnage(String name){
    nom = name;
}
}
```

```
public static void main(String[] args) {
    Personne p1 = new Personne("Laila");
    Personne p2 = new Personne("Amina");
    Personne p3 = new Personne();
}
```

→ Destruction d'un objet

- La destruction des objets est prise en charge par Java à l'aide d'un "**Garbage Collector**" (**GC**).
- Le **GC** détruit les objets (i.e. efface la mémoire) qui ne sont référencés par aucun autre objet.
- Les destructions sont ***asynchrones*** et il n'y a *pas de garanties* que les objets soient détruits.
- Une méthode optionnelle nommée ***finalize()*** est appelée lorsque l'objet est détruit. Elle peut par exemple s'assurer que des fichiers ou des connexions sont bien fermées avant la destruction de l'objet.

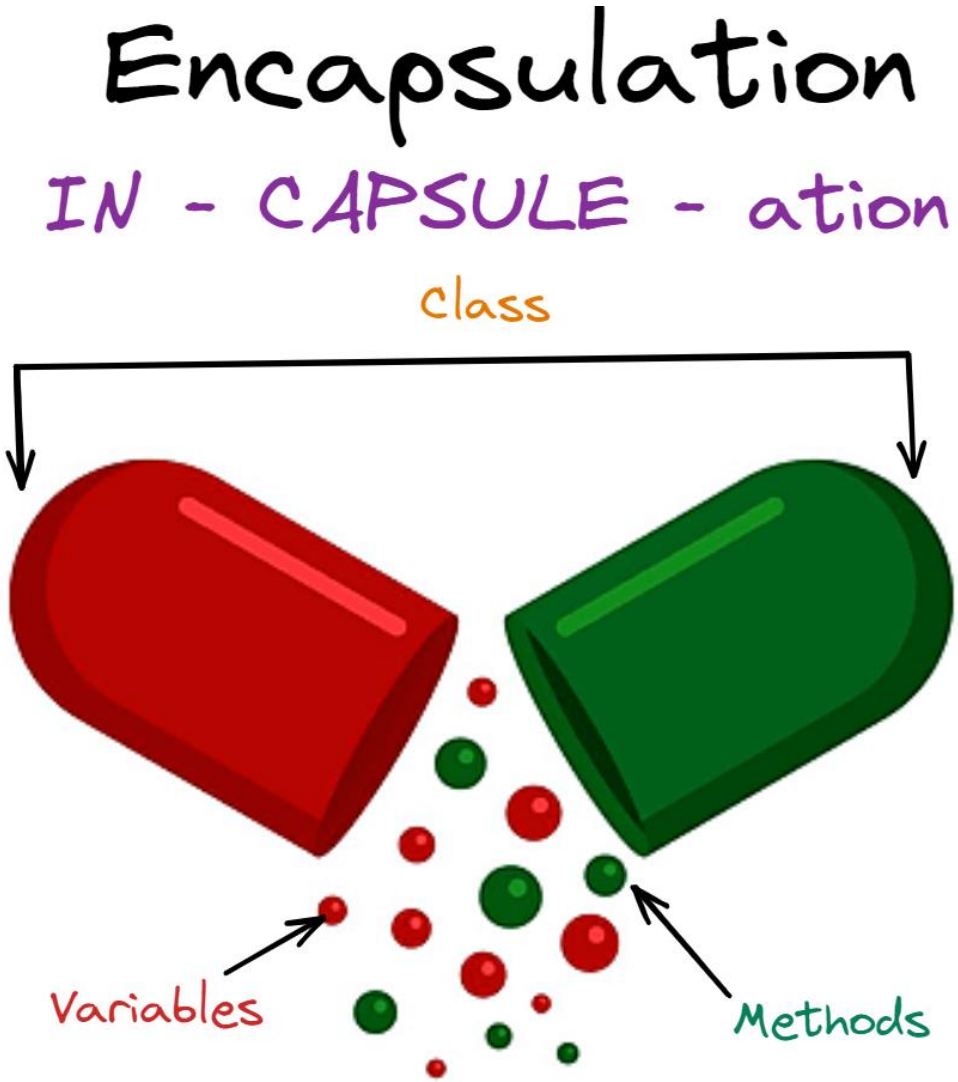
L’encapsulation des données et des méthodes

L'encapsulation est un principe fondamental de la programmation orientée objet (POO) qui consiste à regrouper les données (attributs) et les méthodes qui agissent sur ces données au sein d'une même entité, généralement une classe.

Les points clés de l'encapsulation :

- **Rassemblement des données et des méthodes :** Les données et les méthodes sont regroupées à l'intérieur d'une classe, formant ainsi une unité logique et fonctionnelle.
- **Protection des données :** Les attributs de la classe sont généralement déclarés comme **privés**, ce qui signifie qu'ils ne peuvent être directement accessibles depuis l'extérieur de la classe.
- **Accès contrôlé aux données :** L'accès aux attributs se fait généralement via des méthodes spéciales appelées "**getters**" (pour accéder aux données) et "**setters**" (pour les modifier). Ces méthodes permettent de contrôler et de valider l'accès aux données de la classe.

L’encapsulation des données et des méthodes



L'encapsulation des données et des méthodes

→ Accesseurs et mutateurs

- Les **accesseurs** sont des méthodes et sont déclarés public afin d'être accessibles depuis d'autres classes, comme depuis la méthode ***main***.
- Les **mutateurs** sont déclarés avec le type ***void***. en effet, ces méthodes ne retournent aucune valeur, elles se contentent de mettre à jour les variables de la classe.
- Les **accesseurs** sont nommés en commençant par "**get**" et les mutateurs par "**set**", conformément à la convention de nommage standard. On se réfère souvent à eux en utilisant les termes "**Getters**" et "**Setters**".

→ L'utilisation de ***this*** en Java à l'intérieur des méthodes peut être employée pour référencer les attributs de l'objet courant.

Par exemple, **this.nom** ferait référence à l'attribut **nom** de l'objet actuel à l'intérieur d'une méthode de la classe ➔ Cela est particulièrement utile pour éviter les ambiguïtés lorsqu'un nom de variable local est similaire au nom d'un attribut de la classe.

L'encapsulation des données et des méthodes

→ Accesseurs et mutateurs

- Un **accesseur** est une méthode qui permet d'accéder aux variables de nos objets en lecture.
- Un **mutateur**, quant à lui, permet de modifier ces variables.
- Les **accesseurs** vous permettent d'afficher les valeurs des variables de vos objets, tandis que les **mutateurs** vous autorisent à les modifier.

Exemple:

```
public class Personne {
    private String nom;

    // Constructeur par défaut
    public Personne() {
        nom = "Inconnu";
    }
    // Constructeur avec un paramètre
    public Personne(String name) {
        nom = name;
    }
    // Accesseur pour l'attribut 'nom'
    public String getNom() {
        return nom;
    }
    // Mutateur pour l'attribut 'nom'
    public void setNom(String nouveauNom) {
        nom = nouveauNom;
    }
```

L'héritage

- L'héritage permet à un objet d'acquérir les propriétés d'un autre objet, favorisant la factorisation des connaissances. Dans ce concept :

→ **La classe mère** (ou **classe de base**) est plus générale et contient les propriétés communes à toutes les **classes filles** (ou **classes dérivées** ou **héritées**).

→ **Les classes filles** ont des propriétés plus spécifiques, élargissant ainsi les fonctionnalités de **la classe de base**.

➔ Cela conduit à la formation d'une hiérarchie de classes.

- Pour exprimer qu'une classe est une **classe fille**, on utilise le mot-clé **extends** dans la déclaration de la classe :

class *<nom classe fille>* **extends** *<nom classe mère>*

- **En Java, on hérite d'une seule et unique classe.**
- Une classe qui n'hérite d'aucune classe héritent en fait implicitement de la classe **Object**

❑ *Exemple*

```
public class Personne {
    protected String nom;

    // Constructeur par défaut
    public Personne() {
        nom = "Inconnu";
    }

    public String presentation() {
        return "Mon nom est " + nom;
    }
}
```

Classe mère : Personne

Classe fille : Etudiant

```
public class Etudiant extends Personne { ... }
```

❑ Les méthodes de la classe mère

Pour les **méthodes public** ou **protected** héritées d'une **classe mère**, deux options se présentent :

→ Si le comportement doit rester le même dans la classe fille, il est possible (et souvent recommandé) d'omettre la **ré-écriture de la méthode**.

→ Si le comportement doit être différent dans la classe fille, il est nécessaire de **ré-écrire la méthode**.

- L'**annotation** ***@Override*** peut être utilisée pour indiquer explicitement que l'on redéfinit une méthode de la classe mère en Java.

➔ Cette annotation permet à Java de vérifier si la méthode de la classe fille est réellement une redéfinition de celle de la classe mère.

❑ Les méthodes de la classe mère

- Deux **références** sont utilisées pour naviguer dans la hiérarchie des classes :

  - **this** : fait référence à l'instance de la classe **courante**.
  - **super** : fait référence à l'instance de la classe **parente**.

- Il est tout à fait possible d'ajouter des méthodes spécifiques à la classe fille en plus des méthodes héritées de la classe mère.

➔ Cela permet d'ajouter des fonctionnalités spécifiques à la classe fille sans altérer le comportement des méthodes **héritées.**

❑ Constructeur de la classe fille

- Le constructeur de la classe fille suit généralement la même signature que celui de la classe mère. Son implémentation implique deux étapes :

  - **Appeler le constructeur de la classe mère** : Cela se fait à l'aide de la méthode **super()** avec la liste des arguments nécessaires, si applicable.
  - **Effectuer les traitements spécifiques à la classe fille** à l'intérieur du constructeur de la classe fille lui-même.

- Si l'appel explicite au constructeur de la classe mère n'est pas effectué dans le constructeur de la classe fille, Java tentera automatiquement d'appeler le constructeur **par défaut** (sans argument) **de la classe mère.**

❑ *Exemple*

```
public class Etudiant extends Personne {
    private String CIN;

    public Etudiant() {
        super();
        CIN = "Non défini";
    }
    public String getCIN() {
        return CIN;
    }
    public void setCIN(String nouveauCIN) {
        this.CIN = nouveauCIN;
    }

    @Override
    public String presentation() {
        return "Je suis un étudiant avec un CIN : " + CIN
                + " " + super.presentation();
    }
}
```

Appel au constructeur de la classe mère

Surcharge de la méthode ***presentation()*** de la classe **Personne** pour inclure des informations spécifiques à l'étudiant

Utilisation de ***super.presentation()*** pour appeler la méthode de la classe mère

❑ Opérateur – *instanceof*

L'opérateur **instanceof** en Java est utilisé pour vérifier si un **objet est une instance d'une classe, d'une de ses sous-classes ou d'une classe implémentant une interface**.

➔ Il retourne ***true*** si l'objet est une instance de la classe spécifiée ou de l'une de ses superclasses, sinon il retourne ***false***.

- ***Exemple (1/2)***

```
public class Personne {
    // Code de la classe Personne
}

public class Etudiant extends Personne {
    // Code de la classe Etudiant
}

public class Enseignant extends Personne {
    // Code de la classe Enseignant
}

public class EnseignantChef extends Enseignant {
    // Code de la classe EnseignantChef
}
```

❑ Opérateur – *instanceof*

Exemple (2/2)

```
public class Main {
    public static void main(String[] args) {
        EnseignantChef professeur = new EnseignantChef();

        // Vérification des instances avec l'opérateur instanceof
        System.out.println(professeur instanceof Personne);
        System.out.println(professeur instanceof Etudiant);
        System.out.println(professeur instanceof Enseignant);
        System.out.println(professeur instanceof EnseignantChef);
    }
}
```

```
true
false
true
true
```

→ **professeur** est une instance de ***Personne***
→ **professeur** n'est pas une instance de ***Etudiant***
→ **professeur** est une instance de ***Enseignant***
→ **professeur** est une instance de ***EnseignantChef***

Le polymorphisme

- Le **polymorphisme** en POO fait référence à la capacité d'un objet de prendre plusieurs formes.

➔ Cela signifie que même si un objet est instancié à partir d'une classe spécifique, il peut être manipulé comme s'il était d'un type différent, à condition que ce type soit lié à la même hiérarchie de classes.

- ***Exemple (1/4)*** **:** Reprenons l'exemple des classes **Personne** (classe mère) **et Etudiant** (classe fille)

```
public class Personne {
    protected String nom;

    // Constructeur par défaut
    public Personne() {
        nom = "Inconnu";
    }

    public String presentation() {
        return "Mon nom est " + nom;
    }
}
```

```
public class Etudiant extends Personne {
    private String CIN;

    public Etudiant() {
        super();
        CIN = "Non défini";
    }
    public String getCIN() {
        return CIN;
    }
    public void setCIN(String nouveauCIN) {
        this.CIN = nouveauCIN;
    }

    @Override
    public String presentation() {
        return "Je suis un étudiant avec un CIN : " + CIN
                + ". " + super.presentation();
    }
}
```

Exemple (2/4)

```
public class Main {
    public static void main(String[] args) {
        Personne personne1 = new Etudiant("Asmae", "12345");
        Personne personne2 = new Etudiant("Ilham", "67890");
        Personne personne3 = new Personne("Imane");

        Personne[] personnes = new Personne[3];
        personnes[0] = personne1;
        personnes[1] = personne2;
        personnes[2] = personne3;

        for (Personne p : personnes) {
            System.out.println(p.presentation());
        }
        Personne personne4 = new Etudiant("Meryem", "12345");

        if (personne4 instanceof Etudiant) {
            String cne = ((Etudiant) personne4).getCNE();
            System.out.println("Le CNE de l'étudiant est : " + cne);
        }
    }
}
```

Exemple (3/4)

- Les objets ***personne1*** et ***personne2*** sont déclarés comme des **Personne**, mais ils sont en fait des instances de **Etudiant**.
- Chaque objet a sa propre présentation spécifique grâce à la méthode ***presentation()*** redéfinie dans la classe **Etudiant**.

```
Personne personne1 = new Etudiant("Asmae", "12345");
Personne personne2 = new Etudiant("Ilham", "67890");
Personne personne3 = new Personne("Imane");

Personne[] personnes = new Personne[3];
personnes[0] = personne1;
personnes[1] = personne2;
personnes[2] = personne3;

for (Personne p : personnes) {
    System.out.println(p.presentation());
}
```

- Même si ***personne1*** et ***personne2*** sont déclarés comme des Personne, lorsqu'on appelle ***p.presentation()*** dans la boucle, le **polymorphisme** permet d'appeler la méthode appropriée selon le type réel de chaque objet.

Exemple (4/4)

- Appeler la méthode **getCNE()** directement sur l'objet ***personne4*** provoque une erreur de compilation car **getCNE()** appartient spécifiquement à la classe **Etudiant**.
- Pour éviter cette erreur, on vérifie d'abord si l'objet est une instance de **Etudiant** (avec **instanceof**), puis on effectue un "**cast**" vers le type **(Etudiant)** avant d'appeler la méthode **getCNE().**

```
Personne personne4 = new Etudiant("Meryem", "12345");
if (personne4 instanceof Etudiant) {
    String cne = ((Etudiant) personne4).getCNE();
    System.out.println("Le CNE de l'étudiant est : " + cne);
}
```

On ne peut pas appeler **String cne = personne4.getCNE() - cela générera une erreur de compilation.**

→ Mot clés - *final*

→ **Pour une classe** : une classe déclarée comme ***final*** ne peut pas avoir de classe fille. Cela est mis en place pour des raisons de sécurité, dans le but d'éviter toute forme de modification ou de "détournement".

➔ Par exemple, la classe **String** est déclarée comme **finale**, elle ne peut donc pas être héritée.

```
public final class String { ....}
```

→ **Pour une méthode** : si une méthode est déclarée comme final, elle **ne peut pas être redéfinie** dans une classe dérivée. Cela signifie que son comportement ne peut pas être modifié par une classe héritée.

→ **Pour une variable** : une variable déclarée comme final **ne peut pas être modifiée** après son initialisation. Une fois qu'elle a été assignée une valeur, elle ne peut être changée.

Le polymorphisme

→ La classe 'Object'

- Tout **objet** en Java hérite de la classe **Object**.
- La classe **Object** en Java possède **11 méthodes publiques**

1. protected Object **clone()**
2. boolean **equals(**Object obj**)**
3. protected **void finalize()**
4. **Class<?>** getClass**()**
5. int **hashCode()**
6. void **notify()**
7. void **notifyAll()**
8. String **toString()**
9. void **wait()**
10. void **wait(**long timeout**)**
11. void **wait(**long timeout, int nanos**)**

- Ces méthodes sont héritées par toutes les autres classes Java, car toutes les classes Java sont des sous-classes de Object.
- Certaines de ces méthodes sont utilisées pour des opérations avancées de gestion de mémoire et de synchronisation, tandis que d'autres, comme **equals**(), **toString**(), et **hashCode**(), sont couramment utilisées dans le développement pour des comparaisons d'objets, des représentations en chaîne de caractères et la gestion des codes de hachage.

Le polymorphisme

→ La classe 'Object'

Méthodes	Description
protected Object **clone()**	Crée et renvoie une copie de cet objet.
boolean **equals(Object obj)**	Indique si un autre objet est "égal à" celui-ci.
protected void **finalize()**	Appelée par le garbage collector sur un objet lorsque la collecte des déchets détermine qu'il n'y a plus de références à l'objet.
String toString()	Renvoie une représentation sous forme de chaîne de caractères de l'objet.
Class<?> getClass()	Renvoie la classe d'exécution de cet objet.
int **hashCode()**	Renvoie une valeur de code de hachage pour l'objet

Si vous ne redéfinissez pas ces méthodes de la classe **Object**, **l'implémentation par défaut** sera celle de la classe **Object**.

L’abstraction

- En Java, une **classe abstraite** est une classe déclarée avec le mot-clé **abstract**. Elle ne peut pas être instanciée directement, ce qui signifie qu'on ne peut pas créer d'objets à partir d'une classe abstraite.

➔ Elle est conçue dans le but d’être utilisée comme classe de base pour d'autres classes.

- Une classe abstraite peut contenir à la fois **des méthodes concrètes** (avec une implémentation) et **des méthodes abstraites** (sans implémentation).
- Les **méthodes abstraites** sont des méthodes déclarées sans implémentation dans la classe parente, et leur implémentation doit être fournie par les sous-classes qui étendent cette classe abstraite.

Exemple (1/2)

Classe abstraite : **Personne**

```
public abstract class Personne {
    protected String nom;

    // Constructeur par défaut
    public Personne() {
        nom = "Inconnu";
    }

    public String presentation() {
        return "Mon nom est " + nom;
    }

    public abstract String identite();
}
```

Méthode abstraite à implémenter dans les sous-classes

Exemple (2/2)

```
public class Etudiant extends Personne {
   private String CIN;
   public Etudiant() {
        super();
        CIN = "Non défini";
   }

   @Override
   public String identite() {
        return "Je suis un étudiant.";
   }

}
```

Implémentation de la méthode abstraite ***identite()*** de la classe Personne

Les interfaces

- En Java, l'héritage est **simple**, ce qui signifie qu'une classe peut hériter directement d'une seule autre classe. Cependant, les **interfaces** offrent un moyen de réaliser un type **d'héritage multiple**.
- Une interface représente un **standard** que les classes peuvent choisir de suivre. Pour respecter ce standard, une classe doit mettre en œuvre (ou "**implémenter**") les méthodes et les constantes déclarées dans l'interface correspondante.
- Lorsqu'une classe met en œuvre une interface, on dit qu'elle "**implémente**" cette interface.
- Une classe Java peut **implémenter plusieurs interfaces**.

Les interfaces

- Pour définir une interface en Java, on utilise un fichier **.java** portant le même nom que l'interface.
- Le nom d'une interface commence toujours par une majuscule. La structure d'une interface suit généralement ce schéma :

```
[public] interface NomInterface [extends NomInterface1,
NomInterface2, ...] {
// Déclaration de méthodes ou d'attributs statiques  }
```

  - Toute méthode déclarée dans une interface est **abstraite**
  - Les méthodes sont implicitement déclarées comme telles (i.e. il n'est pas nécessaire d'ajouter le mot-clé ***abstract***)
  - Tout attribut est implicitement déclaré **static** et **final**.

- La syntaxe pour déclarer une classe en Java qui implémente une interface est la suivante :

```
[<modificateurs>] class NomClasse implements
NomInterface {     // Corps de la classe }
```

Exemple

Déclaration de l'interface **Vehicule**

Déclaration de la classe **Voiture** qui **implémente** l'interface **Vehicule**

```
public interface Vehicule {
    void deplacer();
}

public class Voiture implements Vehicule {
    @Override
    public void deplacer() {
        System.out.println("La voiture se déplace sur la route.");
    }
}
```

Dans cet exemple, la classe **Voiture** implémente l'interface **Vehicule** en utilisant le mot-clé **implements**, et fournit une implémentation pour la méthode ***deplacer()*** définie dans l'interface **Vehicule**.

Les interfaces

On peut maintenant utiliser l'interface comme un type normal !

On sait qu'on ne peut pas instancier l'interface, mais on peut créer des instances d'une classe qui implémente l'interface.

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

        Vehicule monVehicule = new Voiture();
        monVehicule.deplacer();
    }
    }
```

- Dans cet exemple, dans la méthode main(), nous déclarons ***monVehicule*** comme une référence de type **Vehicule**, une interface. Ensuite, nous créons une instance de la classe **Voiture** et l'assignons à cette référence **Vehicule**.
- Même si nous déclarons ***monVehicule*** comme une interface **Vehicule**, nous sommes capables d'instancier et d'utiliser une classe concrète (comme **Voiture**) qui implémente cette interface.

Exercice 1

Crée une classe ***CompteBancaire*** avec les caractéristiques suivantes :

- Un attribut ***solde*** de type double.
- Un constructeur qui prend un montant initial comme paramètre.
- Un accesseur pour obtenir le solde.
- Un mutateur pour modifier le solde (en s'assurant que le solde ne devient pas négatif).
- Une méthode ***deposer(double montant)*** qui ajoute de l'argent au compte.
- Une méthode ***retirer(double montant)*** qui enlève de l'argent si le solde est suffisant.

Dans la classe Main, crée une instance de **CompteBancaire**, teste les méthodes et affiche le solde avant et après des opérations.

Correction

Exercice 2

Une entreprise compte plusieurs employés. Chaque employé est caractérisé par son nom, un identifiant unique (matricule) et un indice salarial. Le salaire est déterminé en multipliant l'indice salarial par une valeur fixe.

1. Créer une classe nommée **Employe** possédant les attributs nom, matricule et indiceSalarial.
2. Ajouter à la classe **Employe** une méthode ***afficherCaracteristiques***() pour présenter les détails de chaque employé et une méthode ***calculerSalaire***() pour déterminer son salaire.
3. Développer une sous-classe appelée **Responsable** héritant de **Employe** et comprenant un tableau pour enregistrer les employés placés directement sous sa supervision (inferieurs). Ajouter la méthode ***afficherInferieursDirects***() à cette sous-classe pour afficher les subordonnés directs.
4. Redéfinir la méthode ***afficherCaracteristiques***() dans la classe **Responsable** pour inclure des informations spécifiques aux responsables en plus de celles des employés.

Utiliser cette méthode redéfinie dans la méthode main() pour afficher les caractéristiques à la fois des employés ordinaires et des responsables, mettant ainsi en évidence l'utilisation du polymorphisme.

→ Correction

Exercice 3

Une application de dessin doit gérer diverses formes géométriques telles que des cercles et des rectangles. Chaque forme possède ses caractéristiques distinctes tout en partageant des fonctionnalités communes.

1. Créer une interface nommée **Forme** définissant une méthode ***calculerSurface***() pour uniformiser le calcul de la surface de toutes les formes géométriques.
2. Élaborer une classe abstraite appelée **FormeGenerique** implémentant l'interface **Forme**. Cette classe contiendra des méthodes pour gérer les dimensions communes à toutes les formes.
3. Créer deux classes abstraites, **Cercle** et **Rectangle**, qui héritent des attributs et méthodes de la classe **FormeGenerique**. Ces classes devront également implémenter la méthode ***calculerSurface***() adaptée à chaque forme.
4. Créer des classes concrètes telles que **CercleRouge** et **RectangleBleu**, héritant respectivement des classes abstraites **Cercle** et **Rectangle**. Ces classes devront définir les attributs spécifiques à chaque forme et redéfinir la méthode ***calculerSurface***().

Dans la méthode **main**(), créer des instances de différentes formes géométriques et appeler la méthode ***calculerSurface***() pour illustrer le calcul de la surface pour chaque forme.

→ Correction

ANNEXE

Exercice 1 (1/2)

```java
class CompteBancaire {
    double solde;

    public CompteBancaire(double montantInitial) {
        if (montantInitial >= 0) {
            solde = montantInitial;
        } else {
            solde = 0;
            System.out.println("Solde initial invalide, mis à 0.");
        }
    }

    public double getSolde() {
        return solde;
    }

    public void setSolde(double montant) {
        if (montant >= 0) {
            solde = montant;
        } else {
            System.out.println("Le solde ne peut pas être négatif !");
        }
    }

    public void deposer(double montant) {
        if (montant > 0) {
            solde += montant;
            System.out.println("Dépôt de " + montant + " effectué.");
        } else {
            System.out.println("Montant invalide !");
        }
    }

    public void retirer(double montant) {
        if (montant > 0 && montant <= solde) {
            solde -= montant;
            System.out.println("Retrait de " + montant + " effectué.");
        } else {
            System.out.println("Retrait impossible, solde insuffisant !");
        }
    }
}
```

Exercice 1 (2/2)

```
public class Main {
    public static void main(String[] args) {
        CompteBancaire compte = new CompteBancaire(1000);
        System.out.println("Solde initial : " + compte.getSolde());
        compte.deposer(500);
        System.out.println("Solde après dépôt : " + compte.getSolde());
        compte.retirer(300);
        System.out.println("Solde après retrait : " + compte.getSolde());
        compte.setSolde(2000);
        System.out.println("Solde après modification avec setter : " + compte.getSolde());
        compte.retirer(2500);
    }
}
```

Exercice 2 (1/3)

```
class Employe {
    protected String nom;
    protected String matricule;
    protected double indiceSalarial;

    public Employe(String nom, String matricule, double indiceSalarial) {
        this.nom = nom;
        this.matricule = matricule;
        this.indiceSalarial = indiceSalarial;
    }

    public void afficherCaracteristiques() {
        System.out.println("Nom : " + nom);
        System.out.println("Matricule : " + matricule);
        System.out.println("Indice Salarial : " + indiceSalarial);
    }

    public double calculerSalaire(double valeurFixeSalaire) {
        return indiceSalarial * valeurFixeSalaire;
    }
}
```

Exercice 2 (2/3)

```
class Responsable extends Employe {
    private Employe[] inferieurs;

    public Responsable(String nom, String matricule, double indiceSalarial, Employe[] inferieurs) {
        super(nom, matricule, indiceSalarial);
        this.inferieurs = inferieurs;
    }

    public void afficherInferieursDirects() {
        System.out.println("Subordonnés directs du responsable " + nom + ":");
        for (Employe employe : inferieurs) {
            employe.afficherCaracteristiques();
        }
    }

    @Override
    public void afficherCaracteristiques() {
        super.afficherCaracteristiques();
        System.out.println("Information spécifique au responsable...");
    }
}
```

Exercice 2 (3/3)

```
public class Main {
    public static void main(String[] args) {
        Employe emp1 = new Employe("Imane", "123", 1.5);
        Employe emp2 = new Employe("Fatima", "456", 1.7);

        Employe[] employesSousResponsable = { emp1, emp2 };
        Responsable responsable = new Responsable("Paul", "789", 2.0, employesSousResponsable);

        Employe emp3 = new Employe("Laila", "789", 1.8);

        Employe[] employes = { responsable, emp3 };

        for (Employe emp : employes) {
            emp.afficherCaracteristiques();
            if (emp instanceof Responsable) {
                ((Responsable) emp).afficherInferieursDirects();
            }
            System.out.println("---------------");
        }
    }
}
```

Exercice 3 (1/3)

```java
public interface Forme {
    double calculerSurface();
}
```

```java
public abstract class FormeGenerique implements Forme {
    @Override
    public abstract double calculerSurface();
}
```

```java
public abstract class Cercle extends FormeGenerique {
    double rayon;
    public Cercle(double rayon) {
        this.rayon = rayon;
    }

    @Override
    public double calculerSurface() {
        return Math.PI * rayon * rayon;
    }
}
```

Exercice 3 (2/3)

```java
public abstract class Cercle extends FormeGenerique {
    double rayon;
    public Cercle(double rayon) {
        this.rayon = rayon;
    }

    @Override
    public double calculerSurface() {
        return Math.PI * rayon * rayon;
    }
}
```

```java
public abstract class Rectangle extends FormeGenerique {
    double longueur;
    double largeur;

    public Rectangle(double longueur, double largeur) {
        this.longueur = longueur;
        this.largeur = largeur;
    }

    @Override
    public double calculerSurface() {
        return longueur * largeur;
    }
}
```

Exercice 3 (3/3)

```java
public class CercleRouge extends Cercle {

    public CercleRouge(double rayon) {
        super(rayon);
    }
```

```java
public class RectangleBleu extends Rectangle {

    public RectangleBleu(double longueur, double largeur) {
        super(longueur, largeur);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        CercleRouge cercle = new CercleRouge(5.0);
        RectangleBleu rectangle = new RectangleBleu(4.0, 6.0);

        double surfaceCercle = cercle.calculerSurface();
        double surfaceRectangle = rectangle.calculerSurface();

        System.out.println("Surface du cercle : " + surfaceCercle);
        System.out.println("Surface du rectangle : " + surfaceRectangle);
    }
}
```