Royaume du Maroc
Université Cadi Ayyad
Faculté des Sciences Semlalia
Marrakech

Programmation Orientée objet Java/Python

Pr. *Ilyass OUAZZANI TAYBI*

E-mail : i.ouazzani@uca.ac.ma

Département : Informatique FSSM

Université Cadi Ayyad

Licence ISI S5

Année universitaire : 2025/2026

Programmation orientée objet (POO)

Paradigme de programmation

- Un paradigme de programmation est une approche particulière de la programmation qui définit un ensemble de concepts, de principes et de styles de codage permettant de résoudre des problèmes spécifiques.
- Les paradigmes de programmation offrent des méthodologies pour la conception et la structuration de programmes, et ils peuvent influencer la façon dont les programmes sont écrits, organisés et exécutés.

Programmation orientée objet (POO)

Paradigme de programmation

Il existe plusieurs paradigmes de programmation, chacun avec ses propres caractéristiques, avantages et limitations. Voici quelques paradigmes de programmation:

- Paradigme de programmation structurée (Pascal, C, Algol, etc.)
- Paradigme de programmation procédurale (souvent considéré comme une forme de programmation structurée) (C, Basic, etc.)
- Paradigme de programmation orientée objet (Smalltalk, C++, Java, etc.)
- Paradigme de programmation logique (Prolog)
- Etc

Programmation orientée objet (POO)

Programmation procédurale vs Programmation OO

La programmation procédurale :

- Centrée sur les procédures (fonctions), qui représentent des opérations.
- Repose sur la décomposition d'un programme en procédures s'exécutant séquentiellement.
- Met l'accent sur l'écriture des étapes nécessaires pour résoudre un problème.
- Forte dépendance entre les procédures et les données :
  - Les données sont passées en arguments aux procédures.
  - Les données du programme peuvent être facilement accessibles et modifiées.

Programmation orientée objet (POO)

Programmation procédurale v/s Programmation OO

La programmation OO:

La programmation orientée objet (POO) offre de nombreux avantages :

- une meilleure organisation du code,
- la réutilisation des composants,
- la possibilité d'héritage,
- une facilité de maintenance et de correction,
- et des projets plus simples à gérer.

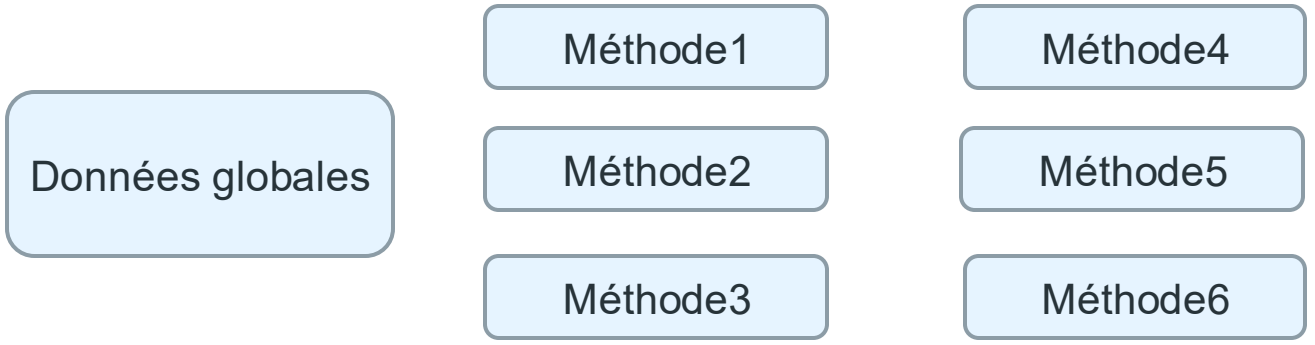
L'intérêt principal de la POO réside dans le fait qu'elle permet de structurer le code autour d'objets, qui sont des ensembles cohérents de données (attributs) et de fonctions (méthodes) interagissant entre eux.

Au lieu de décrire les actions à exécuter de manière linéaire, on organise le programme en objets représentant les concepts du problème à résoudre.

Programmation orientée objet (POO)

Programmation procédurale v/s Programmation OO

La programmation procédurale :



La programmation OO:

Programmation orientée objet (POO)

Concepts généraux de l'OO

La POO se base sur les notions clés suivantes :

- Classe
- Objet
- Encapsulation
- Héritage
- Polymorphisme
- Abstraction

Programmation orientée objet (POO)

Notion de classe

- Une classe regroupe des **propriétés** et des **comportements**.
- Par exemple, la classe Humain définit des propriétés (bras, jambes...) et des comportements (marcher, parler, voir...).

- En OO, les **comportements** sont appelés **méthodes** et les **propriétés** sont appelées **variables d'instance** ou **attributs.**
- Notons que les propriétés des classes peuvent elles-mêmes être des objets. Par exemple, la propriété voiture de la classe « MaisonDeVente » peut être un objet de type ‘Voiture'.

Programmation orientée objet (POO)

Notion d’objet

- Un **objet** est une **instance de classe**, c'est-à-dire un exemplaire utilisable crée à partir de cette classe.
- Par exemple, Ahmed ou Reda sont des **instances de la classe** Humain, c'est-à-dire des humains ayant des propriétés spécifiques (Ahmed est un humain aux yeux noirs et Reda un humain aux yeux marrons).
- Tout objet possède un type: Ahmed est de type Humain. L'instance d'une classe est du type de sa classe. C'est le type initial de l'objet.
- La **classe est le moule**, le **plan de tout objet**.
- La plupart du temps, les objets sont initialisés lors de leur instanciation.

Programmation orientée objet (POO)

Les classes : création

En Java, on déclare une classe comme suit:

```
[Modificateurs] class NomClass
{
   Liste des attributs
   Liste des méthodes
}
```

[Modificateurs]: peuvent être utilisés pour définir l'accès et le comportement de la classe. Exemples :

- **Public** : la classe est visible partout, y compris les autres paquetages
- **Abstract** : la classe ne peut pas être instanciée vue que la classe est abstraite
- **Final** : la classe ne peut pas être étendue.

Programmation orientée objet (POO)

Les classes : choix d'identificateurs

Les noms des classes en Java doivent respecter les conventions suivantes énoncées par Sun Microsystems (maintenant Oracle) :

- Le nom de la classe doit être écrit en PascalCase (la première lettre de chaque mot est en majuscule),
- Le nom doit commencer par une lettre majuscule et ne doit pas contenir d'underscore,
- Les noms des classes doivent être des noms simples et descriptifs,
- Eviter les acronymes.

De plus, d'une manière générale :

- N'utiliser que les lettres [a-z] et [A-Z] et [0-9] : Ne pas utiliser de tiret '-', d'underscore '_', ou d'autres caractères ($, *, accents, ...).

Programmation orientée objet (POO)

Les classes : les attributs de classes

- Les **attributs** sont des **variables déclarées à l'intérieur du corps de la classe** mais **en dehors des méthodes**.
- La déclaration des attributs se fait de préférence en début de classe.
- Les identificateurs des attributs suivent les mêmes règles que ceux des variables que nous avons déjà vues.
- Pour définir un attribut, il faut spécifier :
  - Son **type**, qui peut être un **type primitif** tel que int, double, boolean, etc, ou **un objet**, par exemple, les chaînes de caractères sont des instances de la classe String.
  - Ainsi que son **nom (**son **identificateur)**.

Programmation orientée objet (POO)

Les classes : la portée des attributs de classes

Les modificateurs d'accès indiquent le niveau d'accessibilité de l'attribut.

- [**public**]: Un attribut publique est visible de partout du moment que sa classe est visible.
- [**protected**]: un attribut protégé est visible à l'intérieur de son paquetage et hérité par une sous-classe.
- [**private**]: un attribut privé n'est visible que depuis sa propre classe.
- **Si on indique pas un modificateur**, l'attribut est visible à l'intérieur du paquetage de sa classe.
- [**static**]: indique qu'un attribut n'appartient pas à une instance particulière de la classe mais ils appartient à la classe elle-même.
- [**final**]: permet de déclarer une constante.

Programmation orientée objet (POO)

Les classes : déclaration d'attributs et de constantes

```
public class Film
{
	// déclarations des constantes de classe
	public static final String ENFANT= "En";

	// déclarations des attributs de classe
	private static int nbFilms = 0;

	// déclarations des attributs d'instance
	private String titre;
	private boolean comique;
}
```

Programmation orientée objet (POO)

Les méthodes

En Java, une méthode se définit comme suit :

```
[modificateurs] typeRetourné nomMéthode(les paramètres)
{
      corps de la méthode
}
```

Modificateurs : de façon générale, on met un accès public aux méthodes.

- Le but premier des méthodes **public** est de présenter les services de la classe, c-à-d l'interface public de la classe à ses clients. Les clients ne se soucient pas de la façon dont la classe accomplit ses tâches,
- Les méthodes **private**, que l'on qualifie de méthodes utilitaires ou d'assistance, supportent les opérations des autres méthodes de la classe et ne sont pas accessibles ailleurs.

Programmation orientée objet (POO)

Les méthodes

En Java, une méthode se définit comme suit :

```
[modificateurs] typeRetourné nomMéthode(les paramètres)
{
      corps de la méthode
}
```

TypeRetourné : type de données primitif de la valeur retournée ou classe de l'objet retourné par la méthode.

- L'instruction **return** permet de retourner la valeur du type précisé,
- Une méthode ne peut retourner qu'une seule valeur,
- Si la méthode ne retourne rien, on met **void** comme typeRetourné.

Programmation orientée objet (POO)

Les méthodes

En Java, une méthode se définit comme suit :

```
[modificateurs] typeRetourné nomMéthode(les paramètres)
{
      corps de la méthode
}
```

Les paramètres : variables locales à la méthode, ils reçoivent leur valeur seulement quand la méthode est appelée.

- Une méthode peut avoir aucun, un ou plusieurs paramètres.
- Lors de l'appel de la méthode, il faut respecter le type et l'ordre des paramètres d'entrée.

Programmation orientée objet (POO)

Encapsulation des attributs

- **L'encapsulation** consiste à **cacher l'état interne d'un objet** et **d'imposer de passer par des méthodes permettant un accès sécurisé à l'état de l'objet.**
- Pour mettre en œuvre l'encapsulation, la première étape consiste à **privatiser** les attributs. Pour ce faire, un mot clé spécial vous est proposé : **private**.

```
public class Film {
                    private String titre;
                    private boolean comique;
}
```

- Ensuite, il faut fournir les méthodes d'accès sécurisées : ce qu'on appelle généralement **getters/setters** en Java.
- L'encapsulation garantit un meilleur contrôle des données (niveaux d'accès) et aide à concevoir des objets immuables.

Programmation orientée objet (POO)

Les méthodes d'accès

Les **getters** ou les **méthodes de lecture** ou **accesseurs** désignées aussi par méthodes «**get**» permettent d'obtenir les valeurs des attributs et comportent les caractéristiques suivantes :

- Le nom de la méthode commence par **get** et est suivi du nom de l'attribut désiré (on met une majuscule après get),
- Si l'attribut demandé est une variable de type boolean, on écrit souvent « **is** » plutôt que get,
- Elle ne reçoit aucun paramètre,
- Elle retourne la valeur de l'attribut demandé,
- Si l'attribut demandé est **static**, la **méthode accesseur** l'est aussi.

Les méthodes d'accès

Les méthodes de **mutation** (**mutateurs** ou **modificateurs**) qualifiées aussi de méthodes «**set**» permettent de modifier les valeurs des attributs et comportent les caractéristiques suivantes :

- Le nom de la méthode commence par **set** et est suivi du nom de l'attribut désiré (on met une majuscule après set),
- Elle reçoit un ou plusieurs paramètres permettant d'établir la nouvelle valeur à donner à l'attribut demandé,
- Elle ne retourne habituellement aucune valeur,
- Si l'attribut à modifier est **static**, la méthode de mutation l'est aussi.

Programmation orientée objet (POO)

Les méthodes d'accès

```
public class Film
{
	// déclarations des constantes de classe
	public static final String ENFANT= "En",
	// déclarations des attributs de classe
	private static int nbFilms = 0;
	// déclarations des attributs d'instance
	private String titre;
	private boolean comique;
	// déclaration des méthodes d'accès
	public static int getNbFilms(){ return nbFilms; }
	public String getTitre()   { return titre; }
	public boolean isComique() { return comique; }
	// déclaration des méthodes de mutation
	public void setTitre(String unTitre)     { titre = unTitre; }
	public void setComique(boolean c) { comique = c; }
}
```

Les méthodes surchargées

- Quand on travaille avec les librairies de classes de Java, on rencontre souvent des classes possédant de nombreuses méthodes portant le même nom. A titre d'exemple, la classe String comprend plusieurs méthodes **indexOf()** différentes.
- La surcharge de méthode (**overloading**) est le fait de donner le même nom à plus d'une méthode dans une même classe. Cependant, la liste des paramètres de ces méthodes doit être différente. Au moment de l'appel de la méthode, le compilateur détermine laquelle des méthodes sera appelée en fonction des paramètres fournis.

Programmation orientée objet (POO)

Les méthodes surchargées

```
int somme(int a, int b)
{ return a + b; }

int somme(int a, int b, int c)
{ return a + b + c; }

float somme(float a, float b)
{ return a + b; }
```

```
int x = 1,
y = 2,
z = 9;
float xx = 1.3,
      yy = 2.6;

int   iSomme = somme(x, y, z);
float fSomme = somme(xx, yy);
```

Les méthodes surchargées

- **Attention !** Il n'est pas possible d'avoir deux méthodes (de même nom) dont tous les paramètres sont identiques, et dont seul le type de retour est diffèrent. Le compilateur ne saurait pas quelle méthode employer. L'exemple ci-dessous n'est pas valide :

```
int somme(float a, float b)
{
        return (int)(a + b);
}

float somme(float a, float b)
{
        return a + b;
}
```

La méthode toString()

- Comme **toute classe en Java hérite de la classe Object**, toute classe possède la méthode spéciale suivante : **String toString()**
- qui peut être utilisée pour convertir un objet de cette classe en String. On peut alors l'utiliser pour afficher les informations contenues dans un objet comme ceci :

```
public String toString()
{       return "Vous etes "+nom+" "+prenom+", vous avez "+age+" ans
        et vous mesurez "+taille+" m.";
}
// appel explicite
System.out.println("Informations de l'objet :" +nomObjet.toString());
// appel implicite
System.out.println("Informations de l'objet :" + nomObjet);
```

Programmation orientée objet (POO)

Atelier I

Chaque étudiant est défini par : nom , prénom, age, CNE, et un tableau de note (on va spécifier uniquement 3 notes).

* Créer la classe étudiant en ajoutant les setters et les getters des différents attributs,

*Ajouter une méthode **calculMoyenne()** qui retourne la moyenne de l'étudiant.

Programmation orientée objet (POO)

Constructeurs de classes

- Une classe sert à construire des objets, ces objets sont créés (construits) par une méthode particulière de la classe appelée **constructeur**,
- Un **constructeur sert à initialiser les attributs d'instance d'un objet lors de sa création**. C'est le constructeur d'une classe qu'on appelle lorsqu'on crée un objet avec l'opérateur **new**,
- Un **constructeur porte le même nom que la classe dans laquelle il est défini**,
- Un **constructeur n'a pas de type de retour** (même pas void),
- Un **constructeur peut avoir des arguments**,
- **On peut définir plusieurs constructeurs pour une classes**.

Programmation orientée objet (POO)

Constructeurs de classes

- Une classe possède toujours un constructeur. Si aucun n'est défini explicitement, le compilateur Java crée automatiquement un constructeur par défaut sans arguments.
- Signature d'un constructeur:
  - Modificateur d'accès « public, protected et private » (en général public)
  - Pas de type de retour
  - Le même nom que la classe
  - Les paramètres du constructeur sont utilisés pour initialiser les variables de la classe

Programmation orientée objet (POO)

Constructeurs de classes : Exemple

```
class Personne{
        String nom, prenom;
        int age;

        Personne()
            {
                this.nom="";
                this.prenom="";
                this.age = 0;
            }

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

Constructeur par défaut et surcharge de constructeur

- Le constructeur par défaut, aussi appelé constructeur sans paramètre, initialise :
  - à 0 les attributs de type numérique,
  - à false les attributs de type boolean,
  - Et à null les attributs de type String et ceux dont le type est une classe.
- **Attention** : Lorsque le programmeur définit au moins un constructeur pour la classe, le compilateur Java ne crée pas automatiquement son constructeur par défaut. Si on a besoin d'un constructeur sans paramètre et qu'on a déjà un autre constructeur, il faut le définir de façon explicite.

Constructeur par défaut et surcharge de constructeur

- **Surcharge de constructeur** : On parle de surcharge de constructeur lorsqu'on fournit plus d'un constructeur dans une classe. Tout comme pour une méthode surchargée, ces constructeurs doivent avoir une liste de paramètres différente.

```
class Personne{
	String nom, prenom;
	int age;
	Personne(String nom, String prenom){
		this.nom = nom;
		this.prenom = prenom;
	}
	Personne(String nom, String prenom, int age){
		this(nom,prenom);
		this.age = age;
	}
}
```

Programmation orientée objet (POO)

Constructeur par défaut et surcharge de constructeur

```
class Personne{
        String nom, prenom;
        int age;
        Personne(String nom, String prenom){
                this.nom = nom;
                this.prenom = prenom;
        }
        Personne(String nom, String prenom, int age){
                this(nom,prenom);
                this.age = age;
        }
}
```

```
public class TestPersonne {
    public static void main(String[] args){
      Personne pers = new Personne("Karimi", "Karim");
    }
}
```

Programmation orientée objet (POO)

Utilisation de « this »

- Dans une classe, le mot-clé «**this**» représente l'objet courant de cette classe,
- L'utilisation de «**this**» permet d'utiliser les mêmes noms que ceux des attributs comme paramètres.
- S'il est suivi de parenthèses, **this()** signifie plutôt l'appel à un autre constructeur de la classe. Si on l'utilise, il doit être la première instruction du constructeur.

Programmation orientée objet (POO)

Utilisation de « this »

```
public class Film{
    // déclarations des constantes de classe
    public static final char GENERAL = 'G';
    // déclarations des attributs de classe
    private static int nbFilms = 0;
    // déclarations des attributs d'instance
    private String titre;
    private char cote;
    private boolean comique;
    // déclaration du constructeur avec 3 paramètres
    public Film(String titre, char cote, boolean comique){
       this.titre = titre;
       this.cote = cote;
       this.comique = comique;
       nbFilms++;
    }
    public void setComique(boolean comique){
       this.comique = comique;}
}
```

Programmation orientée objet (POO)

Création et utilisation d'objets : Création d'un objet

- Créer un objet signifie **créer une instance d'une classe** à l'aide de l'opérateur **new**. Celui-ci appelle le constructeur approprié de la classe. L'opérateur **new alloue de la place en mémoire pour l'objet tandis que le constructeur initialise les attributs de l'objet**.
- Après la création d'un objet, pour utiliser une méthode d'instance (c'est-à-dire une méthode qui agit sur les attributs de l'objet), on doit faire précéder cette méthode par la référence de l'objet en utilisant le symbole point ".". Par exemple : chaine.toUpperCase(), chaine.length().

Programmation orientée objet (POO)

Création et utilisation d'objets : Exemple

```
public class ManipulerPersonne{
	public static void main(String args[]){
		Personne enfant;
		// crée des objets en appelant le constructeur
		enfant = new Personne("Saad", 2);
		Personne homme = new Personne();
		Personne femme = new Personne("Fadwa", 23);

		homme.setPrenom("Ahmed");
		homme.setAge(42);

		System.out.println("le prenom de votre enfant est " +enfant.getPrenom());
		System.out.println(enfant.getPrenom()+" avez "+enfant.getAge
				+" ans et maintenant il a "+enfant.setAge(3)+" ans");
	}
}
```

Programmation orientée objet (POO)

Création et utilisation d'objets : Exemple

- Créer une classe Personne qui a :
  - Nom, prenom, age, taille,
  - Constructeur sans paramètre et constructeur avec paramètres,
  - Méthodes d'accès et de mutation,
  - Et la méthode toString qui permet d'afficher un objet personne de la manière suivante : "Bonjour " + nom + " " + prenom + " vous avez "+ age +" ans et vous mesurez "+ taille +" m.".
- En utilisant la classe Personne :
  - Créer deux objets de type personne un avec un constructeur sans paramètre et l'autre avec un constructeur avec paramètres,
  - Afficher les deux objets à l'aide de la méthode toString déjà redéfinit.

Création et utilisation d'objets : Exemple

```
class Personne{
        private String nom, prenom;
        private int age;
        private double taille;
        Personne(){
                this.nom="";
                this.prenom="";
                this.age=0;
                this.taille=0.0; }
        Personne(String nom, String prenom, int age, double taille){
                 this.nom=nom;
                 this.prenom=prenom;
                 this.age=age;
                 this.taille=taille;}
        // méthodes d'accès et de mutation
        public String getNom() {return nom;}
        public String getPrenom(){return prenom;}
        public int getAge(){return age;}
        public double getTaille(){return taille;}
```

Création et utilisation d'objets : Exemple

```
        public void setNom(String nom)         {this.nom = nom;}
        public void setPrenom(String prenom)   {this.prenom = prenom;}
        public void setAge(int age)            {this.age = age;}
        public void setTaille(double taille)   {this.taille = taille;}
// retourne une chaîne contenant les attributs de la personne
public String toString(){
   return "Bonjour " + nom + " " + prenom + " vous avez " + age
          + " ans et vous mesurez " + taille + " m.";
}
// méthode main pour tester la classe
public static void main(String args[]){
    Personne personne1= new Personne();
    Personne personne2= new Personne("KARMI","Karim",21,1.80);
    System.out.println("Vous avez créé deux personnes");
    System.out.println("La première personne est :\n\t"+personne1);
    System.out.println("Et la deuxième personne est :\n\t"+personne2);
   }
}
```

Programmation orientée objet (POO)

Atelier II

- Prenons la classe Etudiant et ajoutons un constructeur par défaut ainsi qu'un autre prenant des paramètres.
- Effectuez une surcharge de la méthode calculMoyenne, afin qu'elle prenne en paramètre un tableau de coefficients.

Programmation orientée objet (POO)

L’héritage

- L’héritage est un mécanisme qui permet à une classe de disposer des champs et des méthodes d’une autre classe,

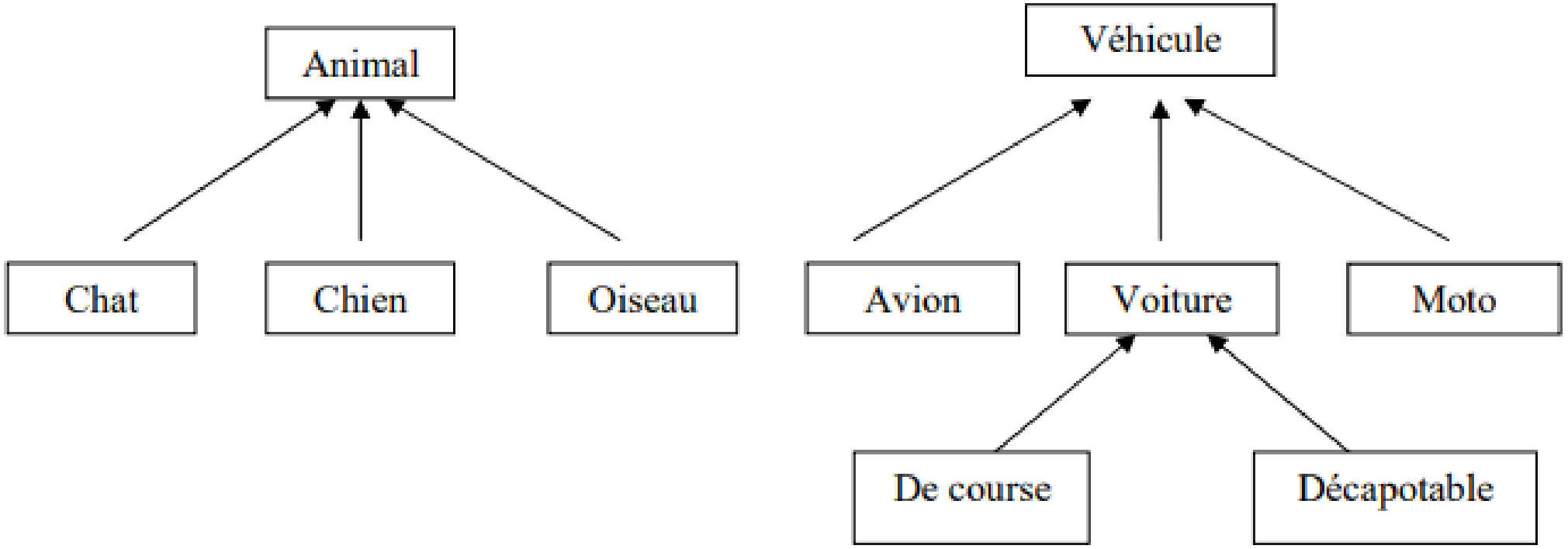
Exemples :

- Un chat est un animal
- Une moto est un véhicule
- Un cercle est une forme

- L'emploi de l'héritage conduit à un style de programmation par raffinements successifs et permet donc une programmation incrémentielle.
- L'héritage est mis en œuvre par la construction de classes dérivées.

Programmation orientée objet (POO)

L'héritage



- La classe dont on dérive est dite CLASSE MERE
- Les classes obtenues par dérivation sont dites CLASSES FILLES

Programmation orientée objet (POO)

L’héritage

L'héritage en POO permet de :

- Réutiliser le code existant (même si ce code est inaccessible) et éviter la duplication.
- Créer une hiérarchie de classes pour organiser les fonctionnalités de manière logique et cohérente.
- Simplifier la maintenance du code.
- Faciliter la compréhension du code en utilisant des noms cohérents pour les méthodes et les propriétés.

Programmation orientée objet (POO)

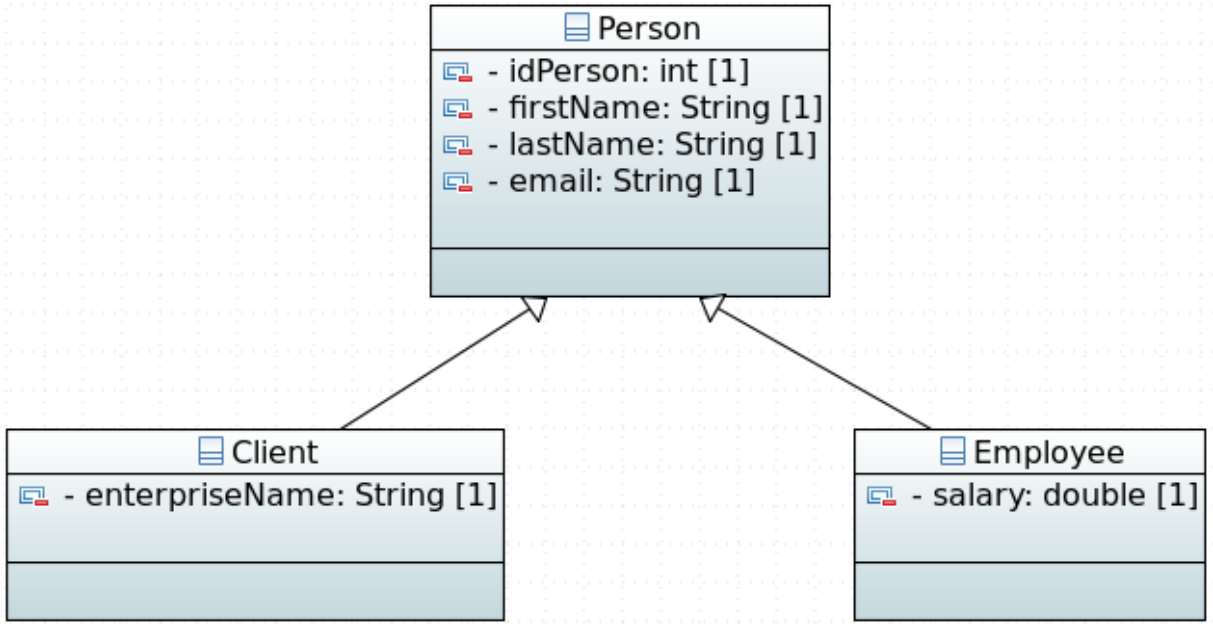
L'héritage

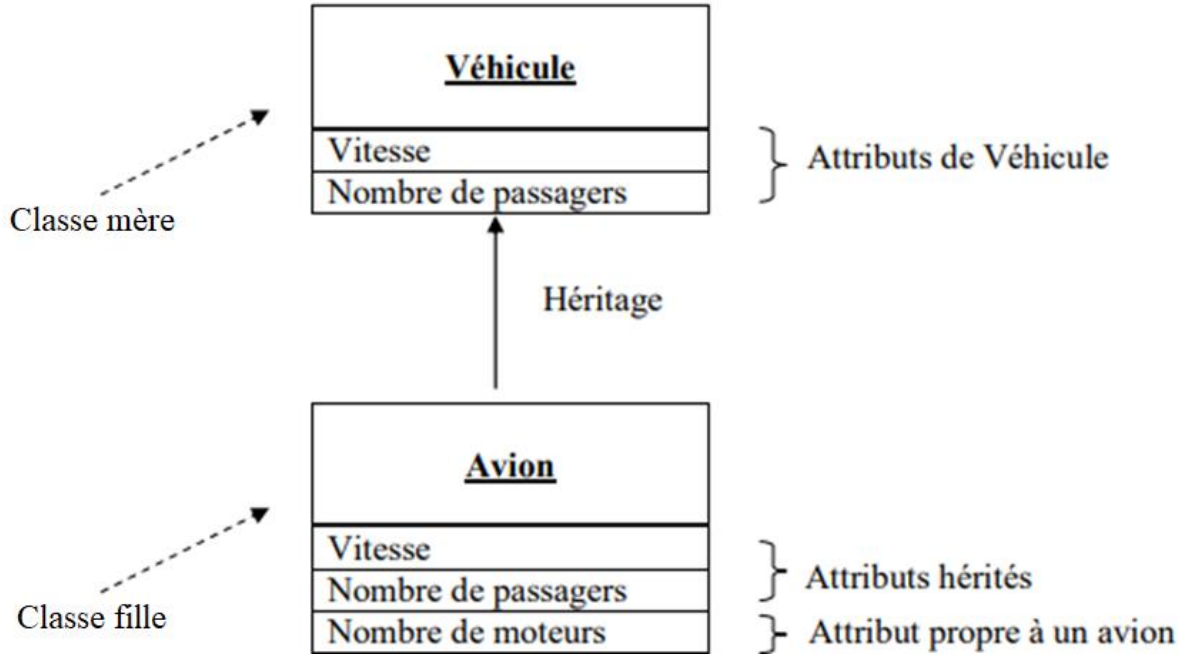
Une classe fille modélise un cas particulier de la classe de base. Elle possède les propriétés suivantes :

- contient les données membres de la classe de base,
- peut en posséder de nouvelles,
- possède les méthodes de la classe de base,
- peut redéfinir certaines méthodes,
- peut posséder de nouvelles méthodes.

Programmation orientée objet (POO)

L'héritage





Programmation orientée objet (POO)

L'héritage en Java

En Java, pour mettre en œuvre l'héritage on utilise le mot clé **extends**.

```
public class classe_mere {
      Liste des attributs
      …
      Liste des méthodes
}
```

```
public class classe_fille extends classe_mere {
      Liste des attributs
      …
      Liste des méthodes
}
```

Programmation orientée objet (POO)

L'héritage en Java : exemple

```
public class Person {
private int identifier;
private String firstName;
private String lastName;
private String email;

public Person() { }
public Person(int id, String fName, String lName, String
mail){
      identifier = id;
      firstName = fName;
      lastName = lName;
      email = mail;}
// getters and setters
public String toString() {
return "Person [identifier=" + identifier + ", firstName="+
firstName+", lastName="+lastName+", email="+email+ "]";}
}
```

```
public class Client extends
Person {
String nomEntreprise;
}
```

```
public class Employee extends
Person {
double salaire;
}
```

Programmation orientée objet (POO)

L'héritage en Java

- Le simple fait d'avoir spécifié l'héritage lors de la déclaration de la classe **Client** et **Employee** garanti que nous avons déjà quatre propriétés et une méthode toString sur cette classe. Ces éléments ont été « **hérités** » de la **classe mère**. On peut donc lancer le test suivant:

```
public class MainClass {
    public static void main( String [] args ) {
          Client client = new Client();
          client.setIdentifier( 0 );
          client.setFirstName( "test" );
          client.setLastName( "test" );
          client.setEmail( "test@unknown.com" );
          System.out.println( client );}
}
```

- Voici le résultat produit par cet exemple de code

```
Person [identifier=0, firstName=test, lastName=test, email=test@unknown.com]
```

L'héritage en Java

- Lors de l'instanciation d'une sous-classe, il faut initialiser :
  - Les attributs hérités des super-classes
  - Les attributs propres à la sous-classe

MAIS...

- L'accès aux attributs de la classe mère peut notamment être interdit ! (**private**)
- L'initialisation des attributs hérités doit donc se faire au niveau des classes où ils sont explicitement définis.

Solution : l'initialisation des attributs hérités doit donc se faire en **invoquant** les **constructeurs des super-classes**.

L'héritage en Java : Constructeurs

- **L'invocation du constructeur de la classe mère se fait au début du corps** du **constructeur de la classe fille** au moyen du mot clé **super**.

- **Syntaxe** :

```
class A{
        private int i;
        // Constructeur de la classe A
        public A(int x){
                i = x;}
}
// Héritage
class B extends A{
      double z;
      // Constructeur de la classe B
      public B(int f, double w) {
      // Appel explicite du constructeur de A à la 1 ère ligne.
                super(f);
                z = w;}
}
```

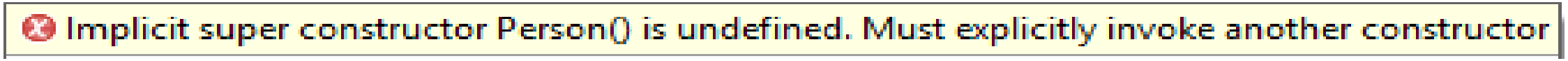
L'héritage en Java : Constructeurs

- Un **appel implicite à super()** sera effectué par le compilateur **si vous n'utilisez pas explicitement l'instruction super** dans votre constructeur.

```
public Client()
{super();
entrepriseName = null;}

public Client(int identifier, String firstName, String lastName, String email, String entreprise)
{super(identifier, firstName, lastName, email); //appel du constructeur de la classe mère
entrepriseName = entreprise;}
```

- **Attention :** Etant donné qu'un appel à super() sera produit implicitement par le compilateur, la classe mère doit obligatoirement définir le constructeur par défaut si elle contient d'autres constructeurs avec des paramètres. Sinon, vous obtiendrez une erreur de compilation.



L’héritage en Java : Constructeurs

- **Règles** :
  - Le constructeur d’une classe fille appelle le **constructeur de la classe** mère avec **super(...)**,
  - Les arguments fournis à **super()** doivent correspondre à ceux d'au moins un des constructeurs de la super-classe,
  - L'appel à **super()** doit être la première instruction dans le corps d'un constructeur de sous-classe. Si l'appel est effectué plus tard dans le corps du constructeur, une erreur se produira,
  - Aucune autre méthode ne peut appeler **super(...)**.

L’héritage en Java : Constructeurs

- Rappel : le constructeur sans paramètre est particulier
  - Il existe par défaut pour chaque classe qui n’a aucun autre constructeur
  - Il disparaît dès qu’il y a un autre constructeur
- Si on oublie l’appel à super(...) ?
  - L'appel automatique à super() peut être pratique dans certains cas, mais cela peut causer une erreur si le constructeur sans paramètre n'existe pas dans la super-classe
- Afin d'éviter des problèmes avec les hiérarchies de classes, il est recommandé de :
  - Toujours déclarer au moins un constructeur dans chaque classe
  - Toujours effectuer l'appel à super() dans les constructeurs de sous-classes.

L'héritage en Java : Constructeurs

```
class Rectangle {
        private double largeur;
        private double hauteur;
        // il y a un constructeur par defaut !
        public Rectangle(){
                largeur = 1;
                hauteur = 1;
        }// le reste de la classe...
}
```

```
class Rectangle3D extends Rectangle {
                private double profondeur;
                public Rectangle3D(double p){
                     profondeur = p;
                }// le reste de la classe...
}
```

Ici il n'est pas nécessaire d'invoquer explicitement le constructeur de la classe mère puisque celle-ci admet un constructeur par défaut.

L'héritage en Java : Constructeurs

```
class Rectangle {
        private double largeur;
        private double hauteur;
        //Constructeur avec parametres!
        public Rectangle(double lar, double hau){
                this.largeur = lar;
                this.hauteur = hau;}
}
```

```
class Rectangle3D extends Rectangle {
        private double profondeur;
        public Rectangle3D(double haut, double lar,double p){
                super(haut,larg);
                profondeur=p;}}
```

Mais dans ce cas, il est nécessaire d'invoquer explicitement le constructeur de la classe mère puisque celle-ci admet un constructeur avec paramètres.

L'héritage en Java : Droits d'accès

- Jusqu'à maintenant, l'accès aux membres (attributs et méthodes) d'une classe est :
  - soit public : visibilité totale à l'intérieur et à l'extérieur de la classe (mot-clé **public**)
  - soit privé : visibilité uniquement à l'intérieur de la classe (mot-clé **private**) la classe fille doit alors utiliser les getters/setters déjà définit dans la classe mère pour manipuler les membres privés
  - soit par défaut (aucun modificateur) : visibilité depuis toutes les classes du même paquetage.
- Le quatrième type d'accès qui régit l'accès aux attributs/méthodes au sein d'une hiérarchie de classes est :
  - l'accès protégé : assure la visibilité des membres d'une classe mère dans ses classes filles et dans les autres classes du même paquetage (mot clé **protected**).

L'héritage en Java : Redéfinition des méthodes

- Une méthode définie dans une classe peut être redéfinie dans une classe dérivée **sans changer sa signature**.
- Exemple de code où on n'a pas redéfinit la méthode m de la classe mère :

```
class B {
        void m(){
         System.out.println("méthode m de B");}
}
class D extends B{
}
```

- Exemple d'appel :

```
B b = new B();
D d = new D();
b.m();
d.m();
```

L'héritage en Java : Redéfinition des méthodes

- Exemple de code avec redéfinition de la méthode m de la classe mère :

```
class B {
        void m(){
        System.out.println("méthode m de B");}
}
class D extends B{
        void m(){
        System.out.println("redéfinition de la méthode m dans D");}
   }
```

- Exemple d'appel :

```
B b = new B();
D d = new D();
b.m();
d.m();
```

L'héritage en Java : Redéfinition des méthodes

Remarques :

- Il ne faut pas confondre la redéfinition avec la surcharge :
  - **Redéfinition** : mêmes paramètres et même type de retour.
  - **Surcharge** : paramètres différents [et type de retour], les paramètres au moins doivent être différents.
- La référence super permet, dans une méthode redéfinie, d'appeler la version de cette méthode définie dans la classe mère. On parle alors de spécialisation de la méthode héritée.

```
public class A {
        public void f() { ... }
}
public class B extends A {
        public void f() {
                super.f();
                ... }
}
```

Programmation orientée objet (POO)

L'héritage en Java : Redéfinition des méthodes

```
public class Pixel {
	private int x,y;
	...
	public String toString() {
		return "(" + x + "," + y + ")";
	}
}

public class PixelCouleur extends Pixel {
	private String couleur;
	...
	public String toString() {
		return super.toString() + "\nCouleur : " + this.couleur;
	}
}
```

L'héritage en Java : Redéfinition des méthodes

On peut aussi surcharger une méthode redéfinie.

```
public class Logement {
	public void vendre(String nomProprietaire) {
		this.proprietaire = nomProprietaire;
	}
}
public class Appartement extends Logement {
	public void vendre(String nomProprietaire, boolean cave){
		super.vendre(nomProprietaire);
		this.cave = cave;
	}
	public void vendre(String nomProprietaire) {
		vendre(nomProprietaire,false);
	}
}
```

Programmation orientée objet (POO)

Le polymorphisme : définition

- Le polymorphisme peut être vu comme la capacité de choisir dynamiquement la méthode qui correspond au type réel de l'objet.
- En clair, ça signifie que lorsqu'on appelle une méthode sur un objet, la méthode choisie dépend du type réel de l'objet (la classe qui a crée l'objet) et ne dépends pas du type de la variable qui stocke l'objet.
- Cela permet d'appeler la même méthode sur des objets sans se soucier de leur type!!

Programmation orientée objet (POO)

Le polymorphisme : exemple

```
class Animal {
    public void speak() { System.out.println("Qui suis-je?");}
}
class Chien extends Animal{
    public void speak() { System.out.println("Je suis un chien");}
}
class Chat extends Animal{
    public void speak() { System.out.println("Je suis un chat");}
}
class Souris extends Animal{
    public void speak() { System.out.println("Je suis une souris");}
}
class Poule extends Animal{
    public void speak() { System.out.println("Je suis une poule");}
}
class Perroquet extends Animal{
    public void speak() { System.out.println("Je suis un perroquet");}
}
```

Programmation orientée objet (POO)

Le polymorphisme : exemple

```
class Main {
  public static void main(String[] args) {
        Animal[] animals = new Animal[5];
        animals[0] = new Chien();
        animals[1] = new Chat();
        animals[2] = new Souris();
        animals[3] = new Poule();
        animals[4] = new Piroquet();

        for(int i=0;i<animals.length;i++) {
                animals[i].speak();
    }
  }
}
```

Classes finales

- Lors de la conception d'une classe, le concepteur peut empêcher que d'autres classes héritent d'elle (classe finale).

```
final public class A { }
```

- Par exemple : la classe String est une classe finale.
- On peut empêcher la redéfinition d'une méthode d'instance d'une classe dans une de ses sous-classes en la déclarant final.

```
public class A {
                final public void f() {}
}
public class B extends A {
                // on ne peut pas redéfinir f() !
}
```

Programmation orientée objet (POO)

Classes abstraites

- De la même façon qu'il est possible d'empêcher quelqu'un d'étendre une classe, ou de redéfinir une méthode, il est possible de forcer l'extension ou la redéfinition, en utilisant le mot-clé **abstract**.
- Formellement une classe abstraite n'est pas différente d'une classe normale. Simplement, elle est déclarée en ajoutant le mot-clé abstract, comme dans l'exemple suivant.

```
// ajout de la clause abstract à la classe A
public  abstract  class A {
      // pas de corps de méthode
      public  abstract  void ouSuisJe();

}
```

Programmation orientée objet (POO)

Classes abstraites

- Une classe contenant au moins une méthode abstraite est nécessairement abstraite.
- Une classe ne contenant pas de méthode abstraite peut néanmoins être abstraite
- Alors qu'est ce qu'une méthode abstraite :
  - Une méthode abstraite, comme dans notre exemple, ne comporte qu'une déclaration de signature ; elle n'a pas de corps.
  - On peut alors se poser la question : comment fait-on pour exécuter une telle méthode ? La réponse est simple : on ne peut pas l'exécuter.

Classes abstraites

- Ainsi toute classe concrète qui étend une classe abstraite doit fournir un corps de méthode pour toutes les méthodes abstraites de cette classe abstraite.

- Dans notre exemple, on peut alors écrire le code de l'exemple suivant.

```
public  abstract  class A {
    public  abstract  void ouSuisJe();
}

public  class B extends A {
    public  void ouSuisJe() {
        System.out.println("Il est passé par ici") ;
    }
}
```

Programmation orientée objet (POO)

Classes abstraites

- On peut énoncer deux principes pour justifier la création d'une classe abstraite :
  - Une classe abstraite sert de modèle pour créer des classes concrètes (non abstraites) qui en hériteront
  - Ainsi, une classe abstraite définit une implémentation partielle (les méthodes concrètes ont des instructions, les méthodes abstraites n'en ont pas) et les classes qui en hériteront devront compléter

Héritage multiple

- Problèmes de l'héritage multiple :
  - Si on hérite de deux méthodes ayant même signature dans deux super classes, quelle code choisir ?
- **Solution :**
  - Il n'y a pas d'héritage multiple en Java
  - Java définit des interfaces et permet à une classe d'implémenter plusieurs interfaces

Programmation orientée objet (POO)

Interfaces

- Une interface définit un type sans code
- On utilise le mot-clé **interface**

```
public interface List {
      public Object get(int index);
      public void set(int index,Object element);
}
```

- Une interface déclare des méthodes sans indiquer le code (implantation) de celles-ci
  - On dit alors que les méthodes sont abstraites

Interfaces

- Il n'est pas possible d'instancier une interface car celle-ci ne définie pas le code de ses méthodes.

```
public interface List {
     public Object get(int index);
     public void set(int index,Object element);
}

public class Main {
     public static void main(String[] args) {
        List list = new List(); // illégal
     }
}
```

Interfaces : Implémentation

- Implémenter une interface consiste à déclarer une classe qui fournira le code pour l'ensemble des méthodes abstraites.

```
public interface List {
      public Object get(int index);
      public void set(int index,Object element);
}
public class FixedSizeList implements List {
      private final Object[] array;
      public FixedSizeList(int capacity) {
            Array = new Object[capacity];}
      public Object get(int index) {
            return array[index];}
      public void set(int index,Object element) {
            array[index]=element;}
}
```

Interfaces : Évolution

Avant Java 8 :

- Les interfaces ne contenaient que des méthodes abstraites et des constantes.
- Chaque classe implémentant l'interface devait redéfinir toutes les méthodes.

Depuis Java 8 :

- Les interfaces peuvent aussi contenir :
  - Des **méthodes default** (avec implémentation par défaut).
  - Des **méthodes static** (liées à l'interface elle-même).

Interfaces : Méthodes *default*

Une méthode *default* est une méthode d'instance avec une implémentation par défaut dans une interface.

```
interface Animal {
    default void dormir() {System.out.println("L'animal dort...");}
}
```

Caractéristiques :

- Peut être redéfinie dans les classes implémentantes.
- Sert à ajouter de nouvelles fonctionnalités à une interface sans obliger toutes les classes à modifier leur code.
- Peut être appelée via une instance de la classe implémentante.

```
class Chien implements Animal{}
new Chien().dormir(); // L'animal dort...
```

Interfaces : Méthodes static

Une méthode static appartient à l'interface elle-même (et non à ses implémenteurs).

```
interface UtilMath{
    static int carre(int n)
      {return n * n;}
}
```

Caractéristiques :

- Appelée via le nom de l'interface :

```
int x = UtilMath.carre(4);
```

- Non héritée par les classes implémentantes.
- Utile pour regrouper des méthodes utilitaires ou des valeurs communes.

Interfaces

Remarques :

- Le compilateur vérifie que toutes les méthodes de l'interface sont implémentées par la classe.
- Le fait qu'une classe implémente une interface implique que la classe est un sous-type de l'interface.
- Une interface peut hériter d'une ou plusieurs interfaces. Les méthodes de cette interface correspondent à l'union des méthodes des interfaces héritées.
- Les méthodes déclarées dans une interface sont implicitement **public** et **abstract.**
- Les attributs d'une interface sont **publics**, **statiques** et **finaux** par défaut.
- Une interface ne peut pas contenir de constructeur (car elle ne peut pas être utilisée pour créer des objets).

Programmation orientée objet (POO)

Atelier

- Créer une classe de base appelée "Forme" qui contient une méthode appelée "calculer aire".
- Créer Ensuite deux classes dérivées : "Rectangle" et "Triangle". Ces classes doivent hériter de la classe "Forme" et ajouter une méthode qui calcule l'aire spécifique à chaque forme.
- Utiliser le polymorphisme pour créer une liste de formes qui contient à la fois des rectangles et des triangles. Parcourir ensuite cette liste et appeler la méthode "calculer aire" pour chaque forme.

Royaume du Maroc
Université Cadi Ayyad
Faculté des Sciences Semlalia
Marrakech

Programmation Orientée objet Java/Python

Pr. *Ilyass OUAZZANI TAYBI*

E-mail : i.ouazzani@uca.ac.ma

Département : Informatique FSSM

Université Cadi Ayyad

Licence ISI S5

Année universitaire : 2025/2026