

Développement Orienté Objet : Programmation objet/Java

Pr. Mariame Nassit

Plan

1. Introduction à Java
2. Classes et Objets
3. Constructeurs
4. Héritage
5. Polymorphisme
6. Tableaux et collection
7. Classes abstraites, interfaces et packages

Chapitre 1

Introduction à Java

Introduction à Java

Introduction et historique

■ Programmation traditionnelle:

- Dans la **programmation classique**, les **données** sont séparées des **traitements**. Une application est généralement structurée en **deux étapes principales** :

1. Définition des structures de données :

Cette étape consiste à créer des structures capables de stocker les informations nécessaires pour gérer le système.

2. Conception des fonctions et procédures :

Cela implique de définir les opérations ou traitements qui vont agir sur les structures de données établies lors de la première étape

Introduction à Java

Introduction et historique

■ Programmation traditionnelle:

Exemple

- Dans une application de gestion de bibliothèque, on peut distinguer deux types de modèles conceptuels :
 - **Modèle Conceptuel des Données (MCD)** : Il représente les informations relatives aux données, telles qu'une personne avec ses différentes caractéristiques : **nom**, **prénom**, **adresse**, et **CNE**.
 - **Modèle Conceptuel de Traitement (MCT)** : Il décrit les opérations réalisées sur ces données. Par exemple, le traitement d'une **demande d'emprunt de livre**, avec des conditions spécifiques comme la disponibilité ou non du livre.
- Ce modèle illustre clairement la séparation entre **les données** et **les traitements**, un principe essentiel de la programmation traditionnelle.
-

Introduction à Java

Introduction et historique

■ Programmation orientée objet

- La programmation orientée objet a été introduite pour répondre aux limites des approches précédentes. Ses principes fondamentaux sont :
 - Encapsulation : rassembler au sein d'une même unité (appelée *classe*) les données et les traitements.
 - Contrôle d'accès : déclarer les données comme **privées** pour forcer l'utilisateur à employer les traitements encadrés par la classe.
- En général, les langages orientés objet reposent sur quatre caractéristiques essentielles : **Encapsulation, Abstraction, Héritage et Polymorphisme**.
- En conception et analyse orientées objet, **UML** est le langage de modélisation le plus utilisé.
- Concernant la programmation orientée objet, **Java** est l'un des langages les plus populaires.

Introduction à Java

Introduction et historique

■ Programmation orientée objet

- La programmation orientée objet a été introduite pour répondre aux limites des approches précédentes. Ses principes fondamentaux sont :
 - Encapsulation : rassembler au sein d'une même unité (appelée *classe*) les données et les traitements.
 - Contrôle d'accès : déclarer les données comme **privées** pour forcer l'utilisateur à employer les traitements encadrés par la classe.
- En général, les langages orientés objet reposent sur quatre caractéristiques essentielles : **Encapsulation, Abstraction, Héritage et Polymorphisme**.
- En conception et analyse orientées objet, **UML** est le langage de modélisation le plus utilisé.
- Concernant la programmation orientée objet, **Java** est l'un des langages les plus populaires.

Introduction à Java

Introduction et historique

■ Historique

- Le projet Java, lancé en **1991** par **Sun Microsystems** et racheté par **Oracle** en **2010**, avait pour objectif de développer des logiciels capables de fonctionner indépendamment de la plateforme matérielle utilisée.

■ Évolution des versions du JDK (Java Development Kit) :

- . **JDK 1.0** : publié en **1996**, première version officielle de Java.
- . **JDK 8** : lancé en **2014**
- . **JDK 21** : publié en **septembre 2024**

Introduction à Java

La plate forme Java

- La plateforme Java repose sur trois éléments principaux :
 - **Le langage Java.**
 - **La machine virtuelle Java (JVM).**
 - **Les bibliothèques Java.**
- La compilation en Java se déroule ainsi :
 - Le **code source** (avec l'extension **.java**) est traduit en **bytecode** (avec l'extension **.class**).
 - Ce **bytecode** est stocké dans un fichier exécutable par la **JVM**, permettant une portabilité totale du code.

Introduction à Java

Premier programme en Java

- Le code source en Java qui affiche le message "**Premier programme en Java**" :

```
public class PremierProgramme {  
    public static void main(String[] args) {  
        System.out.println("Premier programme en Java");  
    }  
}
```

Remarque:

- Le programme doit impérativement être enregistré dans un fichier dont le nom correspond à celui de la classe, soit **PremierExemple.java**.
- Pour compiler le programme précédent, il est nécessaire d'abord d'installer l'environnement de développement **JDK (Java Development Kit)**.

Introduction à Java

Installation de JDK

■ Installation sous Windows

- Télécharger la version de jdk ([jdk-17.0.12_windows-x64_bin.exe](https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html)) correspondant à votre architecture (64 bits) de:

<https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html>

- Exécuter le fichier téléchargé et ajouter

C:\Program Files\Java\jdk1.7.0_\xy\bin au chemin, en modifiant la variable path.

- Pour modifier la valeur de path:

cliquer sur **démarrer** puis **Panneau de configuration** puis **Système et sécurité** puis **Paramètres système avancés**

cliquer sur **Variables d'environnement** puis chercher dans **variables système Path** et modifier son contenu.

Introduction à Java

Installation de JDK

□ Compilaion

- Dans une console, naviguez vers le répertoire contenant votre fichier en utilisant la commande `cd`, puis tapez la commande suivante :

```
javac PremierExemple.java
```

- Après la compilation, si votre programme ne contient aucune erreur, le fichier `PremierExemple.class` sera généré.

□ Exécution

- Pour exécuter le fichier `.class`, tapez la commande suivante ([sans l'extension](#)) :

```
java PremierExemple
```

- Après l'exécution, le message suivant sera affiché : **Premier Programme Java.**

Introduction à Java

Installation de JDK

□ Description du programme

- Première classe
 - En Java, toutes les déclarations et instructions doivent être faites à l'intérieur d'une classe.
 - **public class PremierExemple** signifie que vous avez déclaré une classe nommée **PremierExemple**.
- Méthode main

Pour qu'un programme Java puisse être exécuté, il doit contenir la méthode spéciale **main()**, qui est l'équivalent de **main** dans le langage C.

 - **String[] args** dans la méthode **main** permet de récupérer les arguments transmis au programme lors de son exécution.
 - **String** est une classe, et les crochets **[]** indiquent que **args** est un tableau (nous aborderons l'utilisation des tableaux plus en détail plus tard).
 - Le mot-clé **void** désigne le type de retour de la méthode **main()**, indiquant qu'elle ne retourne aucune valeur lors de son appel.

Introduction à Java

Installation de JDK

□ Description du programme

Méthode main

- Le mot-clé **static** indique que la méthode est accessible et utilisable même sans qu'aucun objet de la classe ne soit créé.
- Le mot-clé **public** définit les droits d'accès. Il est obligatoire dans l'instruction **public static void main(String[] args)**, mais peut être omis dans la ligne **public class PremierExemple**.

Méthode println

- La méthode **println** sert à afficher un message. Par exemple, dans l'instruction **System.out.println("Premier Programme Java")** :
 - **System** est une classe.
 - **out** est un objet dans la classe **System**.
 - **println()** est une méthode (fonction) de l'objet **out**. Les méthodes sont toujours suivies de **()**
- Les points **.** séparent les classes, les objets et les méthodes.
- Chaque instruction doit se terminer par un point-virgule **(;)**.
- "Premier Programme Java" est une chaîne de caractères.

Introduction à Java

Commentaire en Java

- Les commentaires en Java peuvent être écrits sur **une seule ligne** ou **sur plusieurs lignes**. Voici un exemple :

```
/* Ceci est un commentaire  
sur plusieurs lignes */  
  
public class PremierExemple {  
  
    // Ceci est un commentaire sur une seule ligne  
  
    public static void main(String[] args) {  
  
        System.out.println("Premier Programme Java");  
  
    }  
  
}
```

Introduction à Java

Données primitives

- Voici un tableau regroupant les types primitifs en Java :

Type	Description	Taille	Valeurs possibles
byte	Entier	8 bits	-128 à 127
short	Entier	16 bits	-32,768 à 32,767
int	Entier	32 bits	- 2^{31} à $2^{31}-1$
long	Entier	64 bits	- 2^{63} à $2^{63}-1$
float	Nombre à virgule flottante	32 bits	$\pm 3.40282347E+38$ (précision simple)
double	Nombre à virgule flottante	64 bits	$\pm 1.7976931348623157E+308$ (haute précision)
char	Caractère Unicode	16 bits	0 à 65,535
boolean	Valeur booléenne	1 bit	true ou false

Introduction à Java

Données primitives

- Constante
- En Java, une constante est une variable dont la valeur est fixe et ne peut pas être modifiée après son initialisation. Pour déclarer une constante, on utilise le mot-clé **final**.
`final type nomConstante = valeur;`

Exemple

- . `final int AGE_MAX = 100;`
- . `final double PI = 3.14159;`

Introduction à Java

Opérateurs et expressions

- Comme en C, Java possède les opérateurs arithmétiques et de comparaison habituels.
- On retrouve également les expressions arithmétiques, les comparaisons et les boucles classiques du langage C.
- Des fonctionnalités spécifiques à Java seront abordées dans les chapitres suivants.

Introduction à Java

Opérateurs et expressions

Exemple

```
public class ExempleSimple {  
    public static void main(String[] args) { // Déclaration de variables  
        int a = 10, b = 5; int resultat;  
        // Opérateurs arithmétiques  
        resultat = a - b; // Soustraction  
        // Expression conditionnelle  
        if (resultat > 0) {  
            System.out.println("La différence est positive.");  
        } // Boucle for pour afficher les nombres de 1 à 3  
        for (int i = 1; i <= 3; i++) {  
            System.out.println("i = " + i);  
        }  
    }  
}
```

Introduction à Java

Exercices

- Exercice 1:

Déclarer deux variables avec des valeurs définies et afficher leur somme.

- Exercice 2:

Déclarer un nombre et afficher s'il est pair ou impair.

- Exercice 3:

Calculer et afficher la factorielle d'un nombre fixe, par exemple, 5.

- Exercice 4:

Afficher la table de multiplication d'un nombre fixe (par exemple, 3).

Chapitre 2

Classes et objets

Classes et objets

Introduction

- Java est un langage orienté objet. Tout élément doit se trouver à l'intérieur d'une classe.
- Le nom d'une classe (appelé **identifiant**) doit respecter les règles suivantes :
 - L'**identifiant** doit commencer par **une lettre, un tiret bas (_)** ou **un signe dollar (\$)**. Il ne peut pas commencer par **un chiffre**.
 - L'**identifiant** ne doit contenir que **des lettres, des chiffres, le caractère _**, et **le signe \$**.
 - Un identifiant ne peut pas correspondre à **un mot réservé** (voir tableau ci-dessous).
 - Un identifiant ne peut pas correspondre aux mots suivants : **true, false, ou null**. Bien que ces termes ne soient pas des mots réservés, ils représentent des types primitifs et ne peuvent donc pas être utilisés comme noms de classe.

Classes et objets

Introduction

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Table 1 : Table mots clés

Classes et objets

Introduction

- En Java, il est recommandé, par convention, de commencer les noms des classes par **une lettre majuscule**.
- De plus, il est conseillé d'utiliser des majuscules pour **le début des autres noms** afin d'améliorer la lisibilité du code.
- La table ci-dessous présente des mots qui peuvent être utilisés comme noms de classes.

Nom de la classe	Description
Etudiant	Commence par une majuscule
NotesEtudiant	Commence par une majuscule et le deuxième mot commence également par une majuscule
AnnéeScolaire2024	Commence par une majuscule et ne contient pas d'espace

Table 2 : Quelques noms de classes valides

Classes et objets

Introduction

- La table 3 liste des mots qui peuvent également être utilisés comme noms de classes, bien que leur utilisation soit **déconseillée**. Si vous choisissez d'utiliser l'un de ces mots, le programme compilera **sans erreur**

Nom de la classe	Description
etudiant	Ne commence pas par une majuscule
ETUDIANT	Le nom est entièrement en majuscules, ce qui n'est pas recommandé
Notes_Etudiant	L'usage du caractère _ ne reflète pas une convention pour marquer un nouveau mot
annéescolaire2024	Ne commence pas par une majuscule, et le deuxième mot n'est pas clairement identifiable, ce qui rend le nom difficile à lire

Table 3 : Quelques noms de classes non recommandés

Classes et objets

Introduction

- la table 4 regroupe les mots qui **ne peuvent pas être utilisés** comme noms de classes.

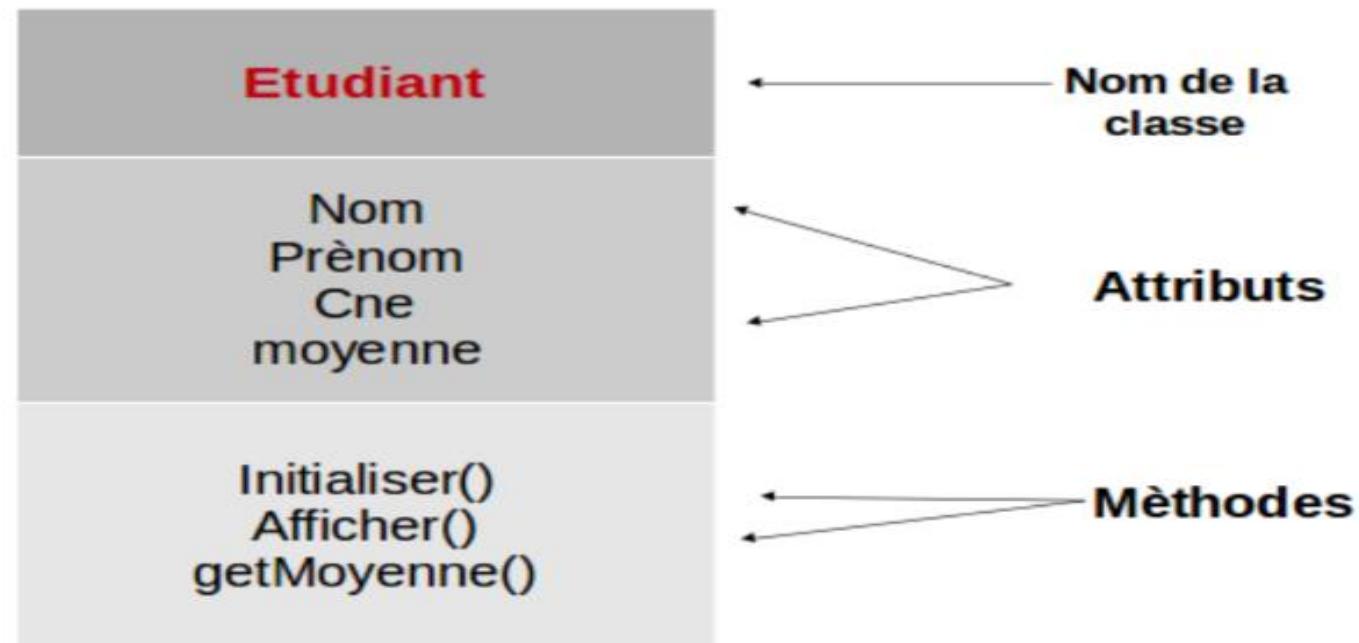
Nom de la classe	Description
Etudiant#	Contient #
double	Mot réservé
Notes Etudiant	Contient espace
2024annéescolaire	Commence par un chiffre

Table 4 : Quelques noms de classes non recommandés

Classes et objets

Introduction

- Représentation en UML la classe Etudiant



Classes et objets

Déclaration d'une classe

- Une classe peut inclure des méthodes (fonctions) et des attributs (variables).
- Pour créer une classe, on utilise le mot-clé class. Par exemple, voici le prototype de la classe Etudiant :

```
class Etudiant {  
    // Déclarations  
    // Attributs et méthodes  
}
```

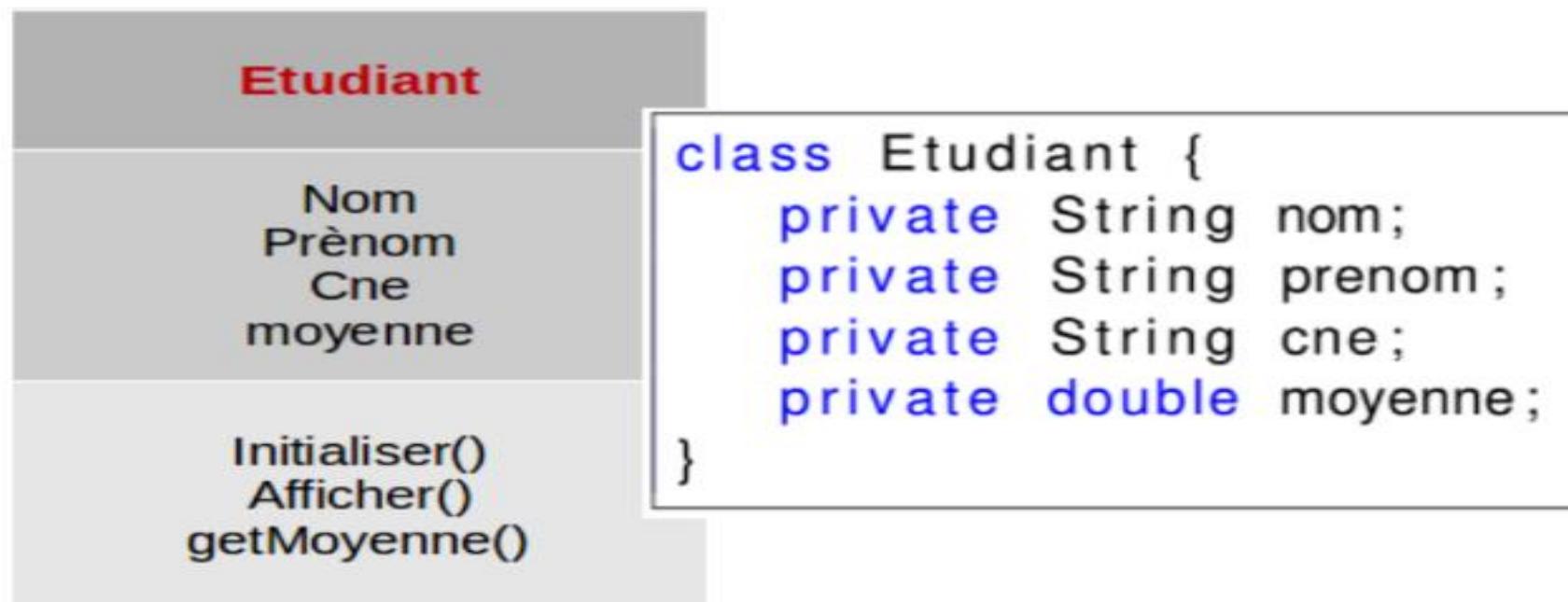
Remarques :

1. Il est possible de définir plusieurs classes dans un même fichier, mais une seule classe peut être précédée du mot-clé public. De plus, le fichier doit porter le même nom que cette classe publique.
2. Une classe peut être déclarée dans un fichier séparé, lequel doit porter le même nom que la classe, suivi de l'extension .java.
3. Pour que la machine virtuelle Java (JVM) puisse accéder à une classe contenant la méthode main, cette classe doit obligatoirement être publique.

Classes et objets

Définition des attributs

- Un étudiant est caractérisé par les propriétés suivantes : son **nom**, son **prénom**, son **numéro CNE** et sa **moyenne**.
- En Java, **ces attributs** peuvent être codés dans notre classe de la manière



Classes et objets

Définition des attributs

- ```
class Etudiant {
 // Attributs de la classe Etudiant
 private String nom; // Le nom de l'étudiant
 private String prenom; // Le prénom de l'étudiant
 private String cne; // Le numéro CNE de l'étudiant
 private double moyenne; // La moyenne de l'étudiant
}
```
- Par convention, les noms des **attributs** et des **methodes** doivent être en **minuscule**.
- Le premier mot doit commencer en minuscule et les autres mots doivent commencer en majuscules (**ceciEstUneVariable**, **ceciEstUneMethode**).

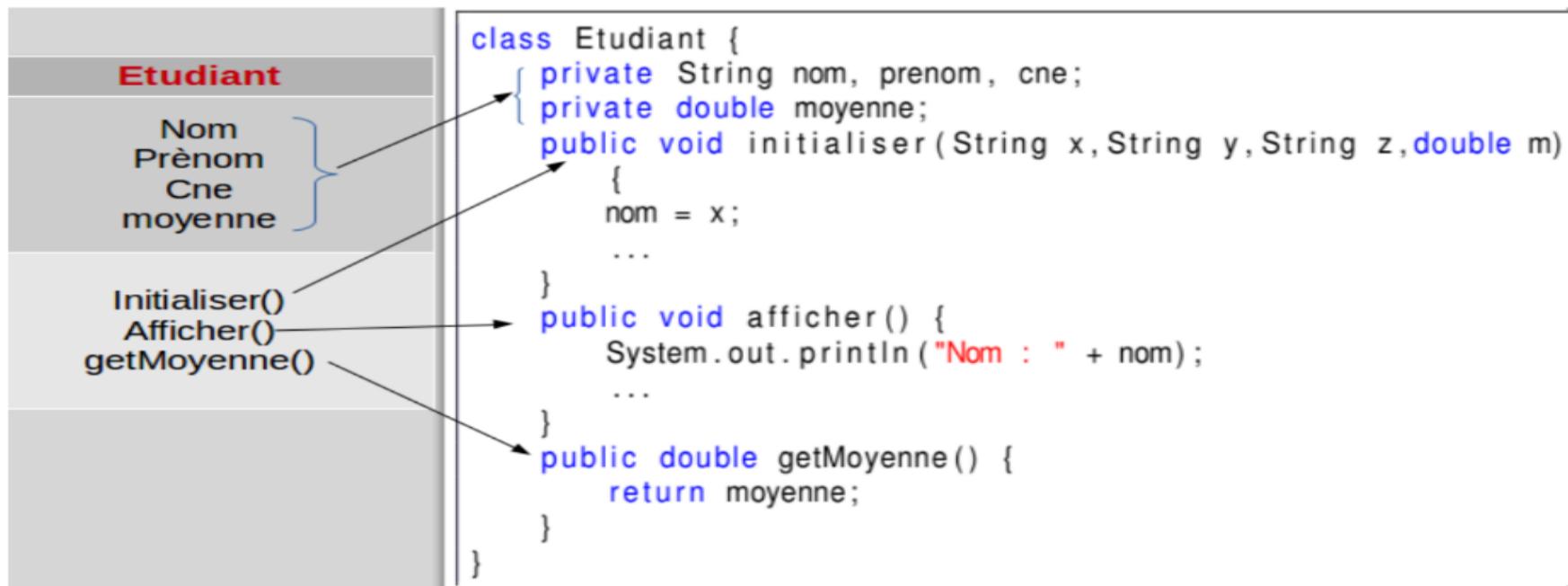
### Remarque :

- La présence du mot clé **private (prive)** indique que les variables ne seront pas accessibles de l'extérieur de la classe où elles sont définies.
- C'est possible de déclarer les variables **non privées**, mais c'est **déconseillé**.

# Classes et objets

## Définition des méthodes

- Dans la classe **Etudiant**, nous avons défini trois méthodes :
  - **initialiser()** : cette méthode permet d'initialiser les informations relatives à un étudiant.
  - **afficher()** : cette méthode affiche les informations d'un étudiant.
  - **getMoyenne()** : cette méthode retourne la moyenne d'un étudiant.



# Classes et objets

## Exemple: classe Etudiant

```
public class Etudiant {
 // Attributs de la classe
 private String nom;
 private String prenom;
 private String cne;
 private double moyenne;

 // Méthode pour initialiser les informations d'un étudiant
 public void initialiser(String nomParam, String prenomParam, String cneParam, double moyenneParam) {
 nom = nomParam;
 prenom = prenomParam;
 cne = cneParam;
 moyenne = moyenneParam;
 }

 // Méthode pour afficher les informations de l'étudiant
 public void afficher() {
 System.out.println("Nom : " + nom);
 System.out.println("Prénom : " + prenom);
 System.out.println("CNE : " + cne);
 System.out.println("Moyenne : " + moyenne);
 }

 // Méthode pour obtenir la moyenne de l'étudiant
 public double getMoyenne() {
 return moyenne;
 }
}
```

- Une classe peut contenir **plusieurs méthodes** définies en son sein.
- Lorsque le mot-clé **public** est utilisé, cela indique que les méthodes sont accessibles depuis **l'extérieur de la classe**. Cependant, il est également possible de déclarer des méthodes privées.

# Classes et objets

## Utilisation des classes

- Après avoir déclaré une classe, celle-ci peut être utilisée pour déclarer **un objet** (c'est-à-dire une variable de type classe) dans n'importe quelle méthode.
- Pour utiliser la classe Etudiant :

`Etudiant et1;`

- Contrairement aux types primitifs, cette déclaration **ne réserve pas d'espace mémoire** pour l'**objet** de type Etudiant, mais crée uniquement **une référence** à cet objet.
- Pour réservrer de la mémoire, il est nécessaire d'utiliser le mot-clé **new**, comme suit :

`et1 = new Etudiant();`

- Ces deux instructions peuvent être combinées en une seule :

`Etudiant et1 = new Etudiant();`

# Classes et objets

## Utilisation des classes: Référence



- Les objets sont manipulés à l'aide de références. Prenons l'exemple classique de l'animal et du chien :

**Chien** = **Objet**.

**Animal** = **Référence**.

- Pour interagir avec **l'objet** (chien), on utilise **la référence** (animal).
- Avoir une référence ne signifie pas forcément posséder l'objet (par exemple, on peut avoir une référence à un animal sans qu'il s'agisse d'un chien).
- Maintenant, nous pouvons appliquer n'importe quelle méthode à **l'objet et1**.
- Par exemple, pour initialiser **les attributs de et1**, on procède de la manière suivante :

**et1.initialiser(« SALIM », "Mohammed", "A8899", 12.5);**

# Classes et objets

## Utilisation des classes: Exemple

- Dans l'exemple ci-dessous, on utilisera la classe **ExempleEtudiant** pour tester la classe **Etudiant**, qui représente notre **classe principale** (celle contenant la méthode main()) :

```
class Etudiant {
 private String nom, prenom, cne;
 private double moyenne;
 public void initialiser(String x, String y, String z, double m)
 {
 nom = x;
 prenom = y;
 cne = z;
 moyenne = m;
 }
 public void afficher() {
 System.out.println("Nom: " + nom + ", prenom: " + prenom +
 ", CNE: " + cne + " et moyenne: " + moyenne);
 }
 public double getMoyenne() {
 return moyenne;
 }
}
```

```
public class ExempleEtudiant {
 public static void main(String[] args) {
 double moy;
 Etudiant et1 = new Etudiant();
 et1.initialiser("Sami", "Ali", "A8899", 12.5);
 et1.afficher();
 et1.initialiser("Rahim", "Lina", "A7788", 13);
 moy = et1.getMoyenne();
 System.out.println("Moyenne : " + moy);
 }
}
class Etudiant {
 ...
}
```

# Classes et objets

## Utilisation des classes: Exemple

Résultats:

Nom : Salim, prenom : Mohammed, CNE : A8899 et moyenne : 12.5

Moyenne : 13.0

Nom : Oujdi, prenom : Ahmed, CNE : A7788 et moyenne : 13.0

# Classes et objets

## Portée des attributs

- Les attributs d'une classe sont accessibles dans toutes les méthodes de cette même classe.
- Il n'est donc pas nécessaire de les transmettre en tant qu'arguments. Cependant, à l'extérieur de la classe, les attributs privés ne sont pas directement accessibles.
- Par exemple, dans la classe **ExempleEtudiant**, une instruction comme :

`Etudiant et = new Etudiant();`

`moy = et.moyenne;`

- provoquera une erreur de compilation avec le message : *The field Etudiant.moyenne is not visible.*
- Cela s'explique par le fait que l'attribut **moyenne** est privé et n'est accessible qu'à l'intérieur de la classe **Etudiant**, mais pas dans la classe **ExempleEtudiant**. C'est pourquoi, dans l'exemple précédent, nous avons utilisé la méthode **getMoyenne()** pour accéder à la valeur de **moyenne**.

# Classes et objets

## Surchage des méthodes

- La surcharge se produit lorsqu'il existe plusieurs méthodes portant le même nom au sein d'une classe.
- Ces méthodes doivent se différencier par le nombre ou le type des arguments qu'elles acceptent.

Exemple :

- Dans la classe **Etudiant**, on peut ajouter trois méthodes ayant le même nom, mais avec des signatures différentes :
  1. Une méthode prenant trois arguments de type **double**.
  2. Une méthode prenant deux arguments de type **double**.
  3. Une méthode prenant deux arguments de type **float**

# Classes et objets

## Surchage des méthodes

```
class Etudiant {
 ...
 //calcul de la moyenne de trois nombres
 public double calculMoyenne(double m1, double m2, double m3) {
 double moy = (m1 + m2 + m3)/3;
 return moy;
 }

 //calcul de la moyenne de deux nombres (doubles)
 public double calculMoyenne(double m1, double m2) {
 double moy = (m1 + m2)/2;
 return moy;
 }

 //calcul de la moyenne de deux nombres (float)
 public double calculMoyenne(float m1, float m2) {
 double moy = (m1 + m2)/2;
 return moy;
 }
}
```

# Classes et objets

## Surchage des méthodes

### □ Utilisation de la classe Etudiant

- À présent, nous utiliserons la classe principale ExempleEtudiant pour effectuer des tests sur la classe modifiée Etudiant

```
public class ExempleEtudiant {
 public static void main(String[] args) {
 double moy;
 Etudiant et1 = new Etudiant();
 et1.initialiser("Salim", "Mohammed", "A8899", 12.5);

 //Appel de calculMoyenne(double m1, double m2, double m3)
 moy = et1.calculMoyenne(10.5, 12, 13.5);

 //Appel de calculMoyenne(double m1, double m2)
 moy = et1.calculMoyenne(11.5, 13);

 //Appel de calculMoyenne(float m1, float m2)
 moy = et1.calculMoyenne(10.5f, 12f);

 //Appel de calculMoyenne(double m1, double m2)
 //13.5 est de type double
 moy = et1.calculMoyenne(11.5f, 13.5);
 }
}
```

# Classes et objets

## Surchage des méthodes

### □ Conflit

```
class Etudiant {
 ...
 public double calculMoyenne(double m1, float m2) {
 return = (m1 + m2)/2;
 }

 public double calculMoyenne(float m1, double m2) {
 return (m1 + m2)/2;
 }
}
```

# Classes et objets

## Surchage des méthodes

### Conflit

```
public class ExempleEtudiant {
 public static void main(String[] args) {
 double moy;
 Etudiant et1 = new Etudiant();
 et1.initialiser("Salim", "Mohammed", "A8899", 12.5);

 //Appel de calculMoyenne(double m1, float m2)
 moy = et1.calculMoyenne(11.5, 13f);

 //Appel de calculMoyenne(float m1, double m2)
 moy = et1.calculMoyenne(10.5f, 12.0);

 //11.5 et 13.5 sont de type double
 //Erreur de compilation
 moy = et1.calculMoyenne(11.5, 13.5);

 //11.5f et 13.5f sont de type float
 //Erreur de compilation
 moy = et1.calculMoyenne(11.5f, 13.5f);
 }
}
```

L'exemple précédent génère des erreurs de compilation :

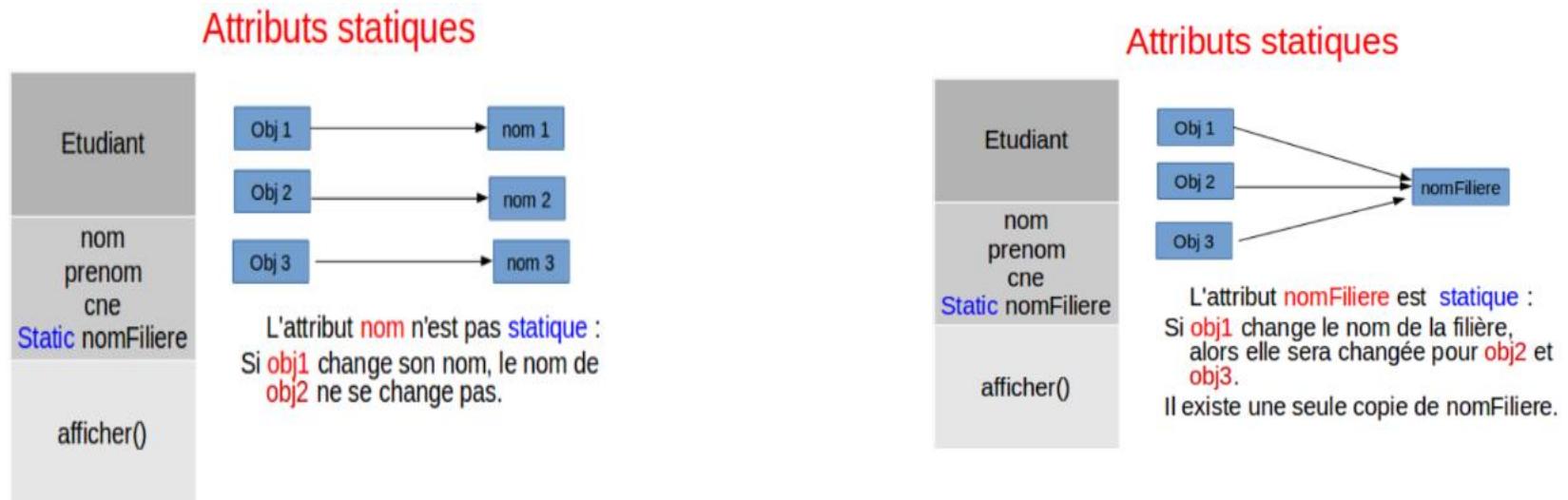
- moy = et.calculMoyenne(11.5,13.5) aboutit à l'erreur de compilation : The method calculMoyenne(double, float) in the type Etudiant is not applicable for the arguments (double, double);
- moy = et.calculMoyenne(11.5f,13.5f) aboutit à l'erreur de compilation : The method calculMoyenne(double, float) is ambiguous for the type Etudiant.

# Classes et objets

## Méthodes statiques et finales

### □ Attributs statiques

- Les **variables statiques**, également appelées **variables de classe**, sont partagées entre **toutes les instances** d'une même classe.
- Contrairement aux variables d'instance, **une seule copie** d'une variable statique est créée pour toute la classe, indépendamment du nombre d'objets créés.
- Lorsqu'un objet est instancié à l'aide de l'opérateur new, aucune nouvelle mémoire n'est allouée pour une variable statique.
- Pour déclarer une variable statique, le mot-clé **static** doit être utilisé



# Classes et objets

## Méthodes statiques et finales

### □ Constantes final et static

- Une constante partagée par toutes les instances d'une classe peut être déclarée à l'aide des modificateurs **final static**. Par exemple :

```
class A { final static double PI = 3.1415927;
 static final double Pi = 3.1415927; }
```

- La classe Math fournit des constantes statiques comme **Math.PI** (valant 3.14159265358979323846).
- Dans la classe Math, la constante PI est déclarée comme suit : **public final static double PI = 3.14159265358979323846;**
- Les caractéristiques de PI sont les suivantes :
  - **public** : Elle est accessible depuis n'importe quel endroit.
  - **final** : Sa valeur ne peut pas être modifiée.
  - **static** : Une seule copie de cette constante existe, et elle est accessible sans qu'il soit nécessaire de créer un objet de la classe Math.

# Classes et objets

## Méthodes statiques et finales

### □ Méthodes statiques

- Une méthode peut être déclarée comme **statique** en utilisant le mot-clé **static**.
- Une méthode statique peut être appelée directement à partir de **la classe**, sans avoir besoin de créer une instance de celle-ci.
- Ces méthodes sont également appelées **méthodes de classe**.
- **Restrictions des méthodes statiques :**
  - Elles ne peuvent appeler que d'autres **méthodes statiques**.
  - Elles ne peuvent utiliser que des **attributs statiques**.

# Classes et objets

## Méthodes statiques et finales

### □ Méthodes statiques: exemple

```
public class MethodesClasses {
 public static void main(String[] args) {
 Scanner clavier = new Scanner(System.in);
 int n;
 System.out.print("Saisir un entier : ");
 n = clavier.nextInt();
 System.out.print("Factorielle : " + n + " est : "
 + Calcul.factorielle(n));
 }
}

class Calcul {
 private int somme;
 static int factorielle(int n) {
 //ne peut pas utiliser somme
 if (n <= 0)
 return 1;
 else
 return n * factorielle(n - 1);
 }
}
```

# Classes et objets

## Lecture à partir du clavier

- Pour effectuer une lecture depuis le clavier, Java fournit la classe **Scanner**.
- Afin de rendre cette classe accessible au compilateur, il est nécessaire d'ajouter l'instruction suivante :

```
import java.util.Scanner;
```

- **Quelques méthodes principales de la classe Scanner :**

- **nextShort()** : Permet de lire une valeur de type short.
- **nextByte()** : Permet de lire une valeur de type byte.
- **nextInt()** : Permet de lire une valeur de type int.
- **nextLong()** : Permet de lire une valeur de type long (sans besoin d'ajouter L après le nombre saisi).
- **nextFloat()** : Permet de lire une valeur de type float (sans besoin d'ajouter F après le nombre saisi).
- **nextDouble()** : Permet de lire une valeur de type double.
- **nextLine()** : Permet de lire une ligne complète et la retourne sous forme de chaîne (String).
- **next()** : Permet de lire le mot suivant sous forme de chaîne (String).

# Chapitre 3

## Constructeurs

# Constructeurs

## Introduction

- Dans le chapitre 2, nous avons appris à initialiser les attributs de la classe **Etudiant** en définissant une méthode appelée **initialiser()**.
- De plus, dans la classe principale **ExempleEtudiant**, pour créer une instance, il était nécessaire d'appeler une méthode spéciale **Etudiant()**, comme dans l'instruction : **Etudiant eti = new Etudiant();**

```
class Etudiant{
 private String nom, prenom, cne;
 private double moyenne;

 // Initialiser les information concernant l'etudiant
 public void initialiser(String x, String y, String z, double m) {
 nom = x;
 prenom = y;
 cne = z;
 moyenne = m;
 }
}
```

```
public class ExempleEtudiant{
 Run main | Debug main | Run | Debug
 public static void main(String[] args) {
 double moy;
 Etudiant et1 = new Etudiant();
 et1.initialiser(x:"Sami", y:"Ali", z:"A8899", m:12.5);
 moy=et1.getMoyenne();
 System.out.println("Moyenne:"+ moy);
 }
}
```

# Constructeurs

## Introduction

- Cette approche **n'est pas recommandée** pour plusieurs raisons :
  - La méthode spéciale `Etudiant()` n'est pas utilisée.
  - Chaque objet créé doit être initialisé manuellement en appelant une méthode spécifique.
  - Si l'initialisation est oubliée, l'objet sera initialisé avec des valeurs par défaut, ce qui peut entraîner des problèmes à l'exécution, même si le programme compile sans erreur.
- Pour résoudre ces inconvénients, on utilise **les constructeurs**.

# Constructeurs

## Définition

- Un **constructeur** est une méthode qui **n'a pas de type de retour** et qui porte **le même nom que la classe**.
- Il est automatiquement appelé lors de **la création d'un objet**.
- Une classe peut avoir plusieurs constructeurs grâce à la **surcharge**, tant que le nombre ou les types d'arguments des constructeurs sont différents.
- Pour créer et initialiser un objet, on peut remplacer ces deux instructions:

```
Etudiant et1 =new Etudiant();
et1.initialiser("Sami", "Ali", "A8899", 12.5);
```

Par une seule instruction:

```
Etudiant et1 =new Etudiant("Sami", "Ali", "A8899", 12.5);
```

```
class Etudiant{
 private String nom, prenom, cne;
 private double moyenne;

 // Constructeur
 public Etudiant(String x, String y, String z, double m) {
 nom = x;
 prenom = y;
 cne = z;
 moyenne = m;
 }
}
```

# Constructeurs

## Utilisation **this**

- Dans les arguments du **constructeur Etudiant**, les variables **x**, **y**, **z**, et **m** ont été utilisées.
- Il est également possible d'utiliser **les mêmes noms que les attributs privés de la classe** en utilisant le mot-clé **this**.

### Exemple

```
class Etudiant{
 private String nom, prenom, cne;
 private double moyenne;

 // Constructeur
 public Etudiant(String nom, String prenom, String cne, double moyenne) {
 this.nom = nom;
 this.prenom= prenom ;
 this.cne= cne ;
 this.moyenne= moyenne ;

 }
}
```

- Dans l'instruction **this.nom = nom**, **this.nom** fait référence à l'attribut **nom** de la classe, tandis que **nom** correspond à la variable locale (argument du constructeur).
- En général, le mot-clé **this** désigne l'instance actuelle de la classe.

# Constructeurs

## Surchage des constructeurs

- La classe **Etudiant** sera modifiée pour inclure deux constructeurs :
  - Un constructeur prenant **trois arguments**.
  - Un autre constructeur prenant **quatre arguments**

### Exemple 1:

```
class Etudiant{
 private String nom, prenom, cne;
 private double moyenne;

 // Constructeur 1
 public Etudiant(String nom, String prenom, double moyenne) {
 this.nom = nom;
 this.prenom= prenom ;
 this.moyenne= moyenne ;
 }

 // Constructeur 2
 public Etudiant(String nom, String prenom, String cne, double moyenne) {
 this.nom = nom;
 this.prenom= prenom ;
 this.cne= cne ;
 this.moyenne= moyenne ;
 }
}
```

```
public class ExempleEtudiant{
 Run main | Debug main | Run | Debug
 public static void main(String[] args) {
 double moy;

 Etudiant et =new Etudiant(nom:"KAMAL", prenom:"Lina", moyenne:11.5);
 Etudiant et1 =new Etudiant(nom:"Sami", prenom:"Ali", cne:"A8899", moyenne:14.5);

 moy=et1.getMoyenne();
 System.out.println("Moyenne:"+ moy);
 moy=et.getMoyenne();
 System.out.println("Moyenne2:"+ moy);
 }
}
Pr.NASSIT
```

# Constructeurs

## Surcharge des constructeurs

- Dans le constructeur 2 de l'exemple 1, trois instructions sont répétées.
- Pour éliminer cette redondance, on peut utiliser l'instruction **this(arguments)**.
- Ainsi, l'exemple 1 se réécrit comme suit :

### Exemple 2:

```
class Etudiant{
 private String nom, prenom, cne;
 private double moyenne;

 // Constructeur 1
 public Etudiant(String nom, String prenom, double moyenne) {
 this.nom = nom;
 this.prenom= prenom ;
 this.moyenne= moyenne ;
 }

 // Constructeur 2
 public Etudiant(String nom, String prenom , double moyenne, String cne) {
 // Appel du constructeur 1
 this(nom, prenom, moyenne);
 this.cne= cne ;
 }
}
```

- L'instruction **this(arguments)** doit être **la première** instruction du constructeur. Si elle est placée ailleurs, le compilateur génère **une erreur**.

# Constructeurs

## Constructeur par défaut

- Le constructeur par défaut est un constructeur qui n'a pas d'arguments.

```
public class ExempleEtudiant{
 Run main | Debug main | Run | Debug
 public static void main(String[] args) {

 // Utilisation du constructeur
 Etudiant et1 =new Etudiant();

 }
}
class Etudiant{
 private String nom, prenom, cne;
 private double moyenne;
 // Constructeur par defaut
 public Etudiant() {
 this.nom = "";
 this.prenom= "" ;
 this.cne="";
 this.moyenne= 0.0 ;
 }

 // Constructeur 1
 public Etudiant(String nom, String prenom, double moyenne) {
```

# Constructeurs

## Constructeur par défaut

Remarques :

1. Si aucun constructeur n'est défini ou utilisé, le compilateur initialise automatiquement les attributs de l'objet avec des valeurs par défaut.
2. Dans les exemples 1 et 2 de la section ([Surcharge des constructeurs](#)), l'instruction Etudiant **et1 = new Etudiant();** n'est pas valide, car les deux constructeurs définis nécessitent des arguments.
3. Contrairement aux autres méthodes, un constructeur ne peut pas être directement appelé.

Par exemple, l'instruction ci-dessous n'est invalide et ne sera pas acceptée

**et1.Etudiant("Sami", "Ali", 14.5, "A8899");**

# Constructeurs

## Constructeur de copie

- En Java, il est possible de créer la copie d'une instance à l'aide d'un **constructeur de copie**.
- Ce type de constructeur permet d'initialiser une nouvelle instance en recopiant les attributs d'une instance existante du même type.

```
public class ExempleEtudiant{
 Run main | Debug main | Run | Debug
 public static void main(String[] args) {

 Etudiant et1 =new Etudiant(nom:"Sami", prenom:"Ali", moyenne:14.5, cne:"A8899");
 Etudiant et2 =new Etudiant(et1);
 }
 class Etudiant{
 private String nom, prenom, cne;
 private double moyenne;

 // Constructeur de copie
 public Etudiant(Etudiant autreEt) {
 nom = autreEt.nom;
 prenom= autreEt.prenom;
 cne= autreEt.cne;
 moyenne= autreEt.moyenne;

 }
 // Constructeur par defaut
```

- Remarque: e1 et e2 sont deux références différentes pour deux objets ayant les mêmes valeurs d'attributs

# Chapitre 4

## Héritage

# Héritage

## Introduction

- Java prend en charge l'héritage comme d'autres langages orientés objet.
- L'héritage permet de créer de nouvelles classes à partir de classes existantes.
- Cela favorise la réutilisation en modifiant ou ajoutant des attributs et méthodes.
- Une classe qui hérite d'une autre s'appelle *classe dérivée, sous-classe* ou *classe-fille*.
- La classe dont elle hérite s'appelle *super-classe, classe-mère* ou *classe-parente*.

Syntaxe :

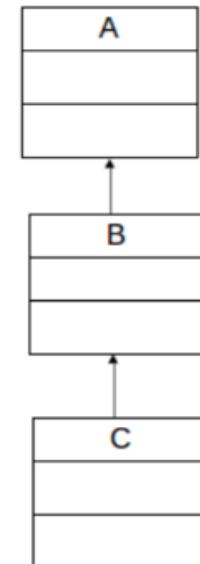
class SousClasse extends SuperClasse

# Héritage

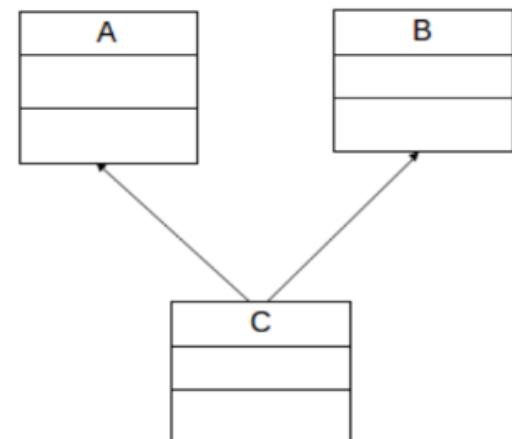
## Introduction

Remarques :

- Java ne permet **pas l'héritage multiple** : une classe ne peut hériter que **d'une seule** autre classe.
- Une **classe dérivée** peut hériter d'une autre **classe dérivée**.
- Par exemple :
  - La classe **A** est la **super-classe** de **B**.
  - B** est la **super-classe** de **C**.
  - A** devient ainsi la **super-super-classe** de **C**.



Héritage valide

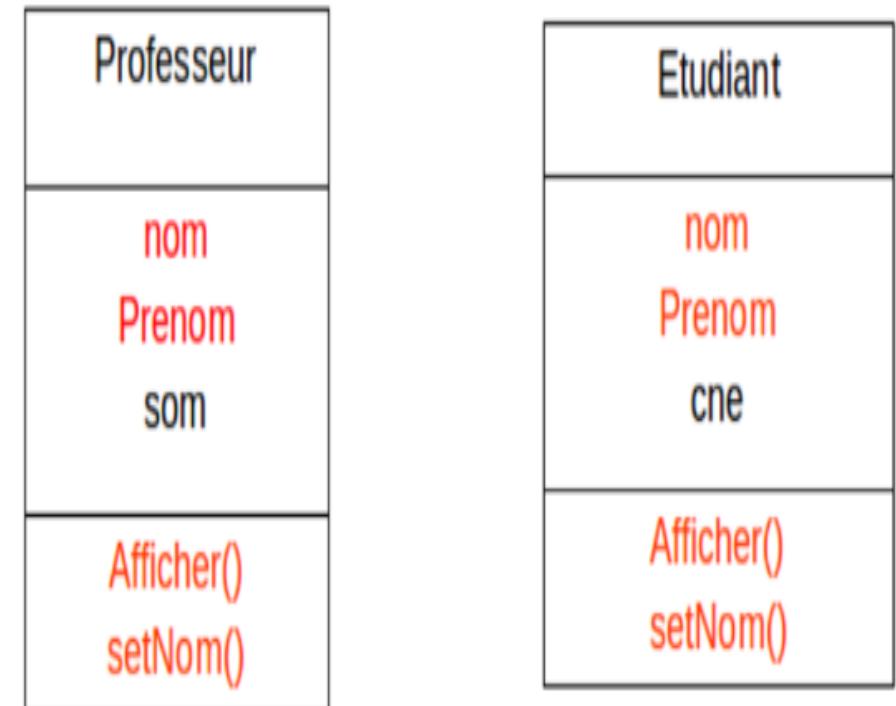


Héritage multiple non valide en Java

# Héritage

## Exemple introductif

- Prenons deux classes : **Etudiant** et **Professeur**.
  1. Les deux classes ont en **commun** les attributs **nom** et **prenom**.
  2. Elles partagent **aussi** les méthodes **afficher()** et **setNom()**.
  3. La classe **Etudiant** possède l'attribut spécifique **cne**.
  4. La classe **Professeur** possède l'attribut spécifique **som** .
- Présentation en UML des deux classes Etudiant et Professeur.



# Héritage

## Exemple introductif

- Voici une version reformulée et simplifiée des deux classes

```
class Etudiant {
 private String nom, prenom, cne;

 void afficher() {
 System.out.println("Nom : " + nom);
 System.out.println("Prénom : " + prenom);
 }

 void setNom(String nom) {
 this.nom = nom;
 }
}
class Professeur {
 private String nom, prenom, som;

 void afficher() {
 System.out.println("Nom : " + nom);
 System.out.println("Prénom : " + prenom);
 }

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

 void setSom(String som) {
 this.som = som;
 }
}
```

A

# Héritage

## Utilisation de l'héritage

Un étudiant et un professeur **sont des personnes**. Nous définissons donc une nouvelle **classe Personne**.

```
class Personne {
 private String nom, prenom;

 void afficher() {
 System.out.println("Nom : " + nom);
 System.out.println("Prénom : " + prenom);
 }

 void setNom(String nom) {
 this.nom = nom;
 }
}
```

# Héritage

## Utilisation de l'héritage

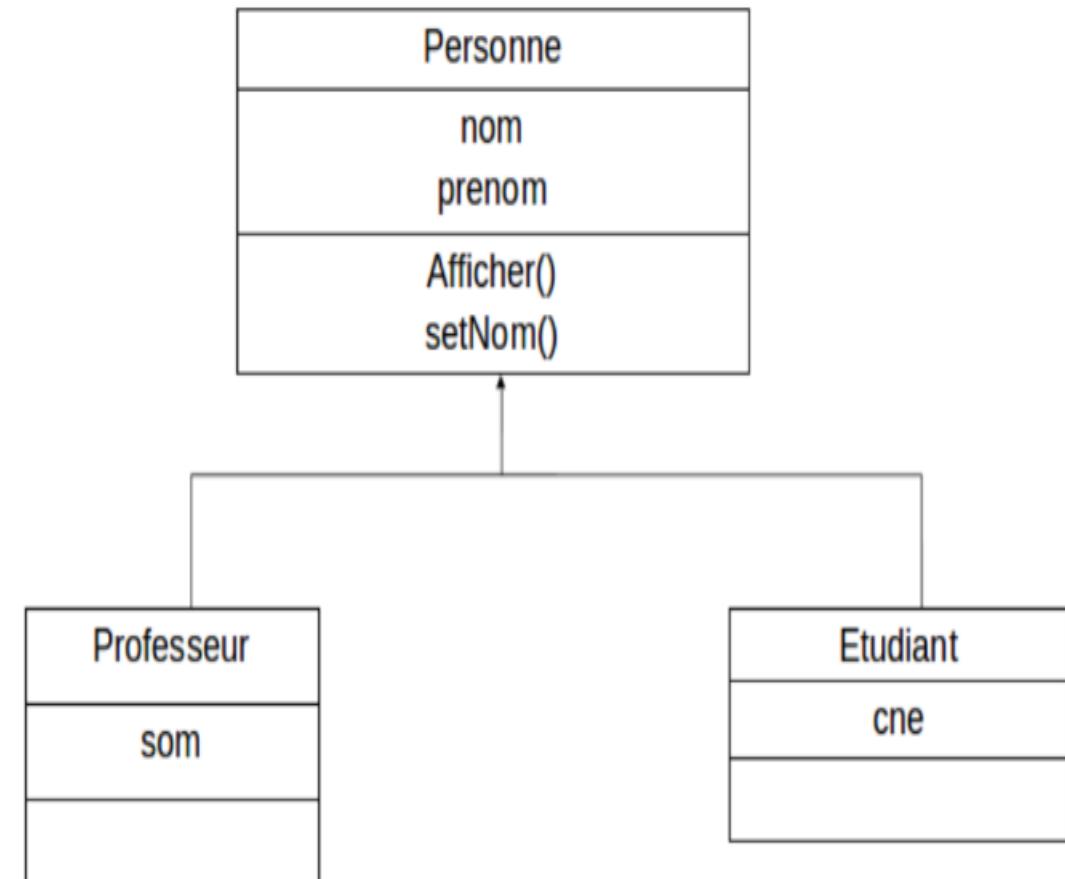
Les classes **Professeur** et **Étudiant** héritent de la classe **Personne**

```
class Etudiant extends Personne {
 private String cne;

 void setCne(String cne) {
 this.cne = cne;
 }
}

class Professeur extends Personne {
 private String som;

 void setSom(String som) {
 this.som = som;
 }
}
```



# Héritage

## Accès aux attributs

L'accès direct aux attributs privés (`private`) d'une super-classe n'est pas autorisé.

Par exemple, si l'on souhaite ajouter une méthode `getNom()` dans la classe **Etudiant** pour retourner le nom, l'instruction suivante est interdite.

En effet, le champ `Personne.nom` n'est pas accessible directement ([The field Personne.nom is not visible](#)).

```
class Etudiant extends Personne {
 private String cne;

 String getNom() {
 return nom; // Non autorisé
 }
}
```

# Héritage

## Accès aux attributs

- Pour accéder à un attribut d'une super-classe, il existe trois solutions possibles :
  1. Rendre l'attribut public : Cela permet l'accès à l'attribut depuis toutes les classes, mais cette pratique est déconseillée, car elle viole l'encapsulation.
  2. Ajouter des méthodes dans la classe super-classe : Utiliser des **getters** et **setters** pour accéder aux attributs privés.
  3. Déclarer l'attribut comme protégé : Utiliser le mot-clé **protected** pour permettre l'accès dans les sous-classes tout en limitant son accès aux autres classes

# Héritage

## Accès aux attributs

### Exemple:

- Dans cet exemple, l'attribut nom est déclaré comme protégé.
- La classe Etudiant, qui hérite de la classe Personne, peut y accéder directement :

```
class Personne {
 protected String nom;
 // ...
}

class Etudiant extends Personne {
 String getNom() {
 return nom; // Accès autorisé grâce à "protected"
 }
}
```

- Remarques

1. Un attribut déclaré comme protégé (protected) est accessible par toutes les sous-classes et par toutes les classes du même package (la notion de package sera expliquée plus tard). Cela peut casser l'encapsulation.
2. Le mode protégé est peu utilisé en Java.

# Héritage

## Héritage hiérarchique

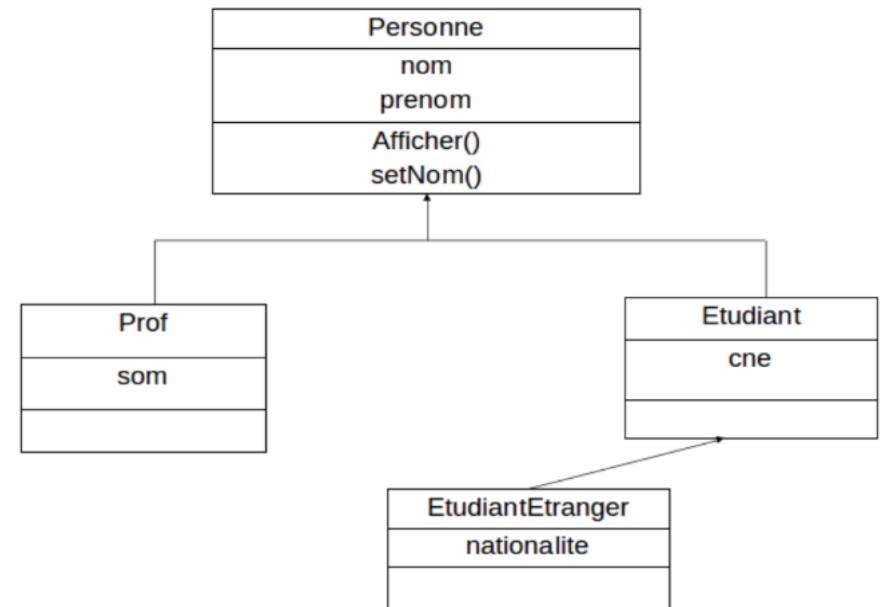
- Comme mentionné dans l'introduction, une classe peut être la super-classe d'une autre classe, elle-même héritée par une troisième.
- Dans l'exemple précédent, ajoutons une nouvelle classe : **EtudiantEtranger**.  
Un étudiant étranger est un étudiant avec un attribut supplémentaire pour enregistrer sa nationalité.

Exemple: La classe **EtudiantEtranger** hérite de **Etudiant**, et **Etudiant** hérite de **Personne**.

```
class Personne {
}

class Etudiant extends Personne {
}

class EtudiantEtranger extends Etudiant {
 private String nationalite;
}
```



# Héritage

## Redéfinition

- **Redéfinition (overriding)** : Comme pour la surcharge à l'intérieur d'une classe, une méthode déjà définie dans une super-classe peut être modifiée avec une nouvelle définition dans une sous-classe.

Remarques : Ne pas confondre **surcharge** et **redéfinition** !

- La redéfinition sera expliquée en détail dans le chapitre sur le polymorphisme.

---

```
class A
{
 public void f(int a, int b)
 {
 // instructions
 }
 // Autres méthodes et attributs
}

class B extends A
{
 public void f(int a, int b)
 {
 // la méthode redéfinie f() de la super-classe
 }
 // Autres méthodes et attributs
}
```

# Héritage

## Redéfinition: exemple

- Reprenons l'exemple précédent et ajoutons à la [classe Personne](#) une méthode `afficher()`.
- Cette méthode sert à afficher les attributs nom et prénom.
- Dans les classes [Etudiant](#), [EtudiantEtranger](#) et [Professeur](#), la méthode `afficher()` peut être [redéfinie](#) pour inclure les informations spécifiques à chaque classe.
- Pour éviter de répéter le code déjà présent dans la méthode de base, on utilise le mot-clé [super](#).
- Cela permet d'appeler la méthode de la super-classe tout en ajoutant les nouvelles instructions nécessaires dans la sous-classe

# Héritage

## Redéfinition: exemple

### Remarques :

- La méthode `super.afficher()` doit être placée en **première position** dans la méthode `afficher()`.
- Dans la classe **EtudiantEtranger**, `super.afficher()` appelle la méthode `afficher()` de la classe **Etudiant**.
- Si la classe **Etudiant** ne possède pas de méthode `afficher()`, alors, **par transitivité**, `super.afficher()` dans **EtudiantEtranger** appellera la méthode `afficher()` de la classe **Personne**.
- Il n'existe pas de syntaxe du type `super.super`.

```
class Personne {
 private String nom, prenom;
 void afficher() {
 System.out.println("Nom : " + nom);
 System.out.println("Prenom : " + prenom);}
 // Autres méthodes...
}
class Etudiant extends Personne {
 private String cne;
 void afficher() {
 super.afficher();
 System.out.println("CNE : " + cne);}
 // Autres méthodes...
}
class EtudiantEtranger extends Etudiant {
 private String nationalite;
 void afficher() {
 super.afficher();
 System.out.println("Nationalite : " + nationalite);}
 // Autres méthodes...
}
class Professeur extends Personne {
 private String som;
 void afficher() {
 super.afficher();
 System.out.println("SOM : " + som);}
 // Autres méthodes...
}
```

# Héritage

## Héritage et constructeurs

Les constructeurs ne sont pas hérités par les sous-classes, mais une sous-classe peut appeler le constructeur de sa super-classe en utilisant le mot-clé `super`.

```
class Personne {
 private String nom, prenom;

 //Constructeur
 public Personne(String nom, String prenom) {
 this.nom = nom;
 this.prenom = prenom;
 }
 ...
}

class Etudiant extends Personne {
 ...
 private String cne;
 //Pas de constructeur
 ...
}

class Professeur extends Personne{
 ...
 private String cne;
 //Pas de constructeur
 ...
}
```

# Héritage

## Héritage et constructeurs: exemple

Dans cet exemple, le **constructeur** de la classe **EtudiantEtranger** **invoque** celui de la classe **Etudiant**, qui, à son tour, **appelle le constructeur** de la classe **Personne**.

```
class Personne {
 private String nom;
 private String prenom;
 // Constructeur de la classe Personne
 public Personne(String nom, String prenom) {
 this.nom = nom;
 this.prenom = prenom; }
}
class Etudiant extends Personne {
 private String cne;
 // Constructeur de la classe Etudiant
 public Etudiant(String nom, String prenom, String cne) {
 super(nom, prenom); // Appel au constructeur de la classe Personne
 this.cne = cne; }
}
class EtudiantEtranger extends Etudiant {
 private String nationalite;
 // Constructeur de la classe EtudiantEtranger
 public EtudiantEtranger(String nom, String prenom, String cne, String nationalite) {
 super(nom, prenom, cne); // Appel au constructeur de la classe Etudiant
 this.nationalite = nationalite;
 }
}
```

# Héritage

## Héritage et constructeurs: exemple

Remarques :

1. L'appel à **super** doit être **la première instruction** dans le constructeur et ne peut pas être effectué plusieurs fois.
2. Il n'est pas **nécessaire d'appeler super** lorsque la super-classe possède un **constructeur par défaut**. Cette tâche sera réalisée par le compilateur
3. **Les arguments** passés à **super** doivent correspondre à ceux d'un des constructeurs de la super-classe.
4. Il n'existe pas de syntaxe **super.super**.

# Héritage

## Héritage et constructeurs: Opérateur « instanceof »

- L'opérateur **instanceof** en Java est utilisé pour tester si un objet est une instance d'une classe spécifique ou d'une de ses sous-classes.
- Il permet de vérifier le type d'un objet à l'exécution, ce qui peut être particulièrement utile dans le cadre de l'héritage et du polymorphisme.

### Syntaxe :

**object instanceof ClassName**

- **object** : l'objet que l'on souhaite tester.
- **ClassName** : le nom de la classe (ou d'une de ses sous-classes) contre laquelle on teste l'objet.
- L'opérateur retourne une valeur **true** si l'objet est une instance de la classe spécifiée ou d'une de ses sous-classes, sinon il retourne **false**.

# Héritage

## Opérateur « instanceof »

- L'opérateur **instanceof** en Java est utilisé pour tester si un objet est une instance d'une classe spécifique ou d'une de ses sous-classes.
- Il permet de vérifier le type d'un objet à l'exécution, ce qui peut être particulièrement utile dans le cadre de l'héritage et du polymorphisme.

### Syntaxe :

**object instanceof ClassName**

- **object** : l'objet que l'on souhaite tester.
- **ClassName** : le nom de la classe (ou d'une de ses sous-classes) contre laquelle on teste l'objet.
- L'opérateur retourne une valeur **true** si l'objet est une instance de la classe spécifiée ou d'une de ses sous-classes, sinon il retourne **false**.

# Héritage

## Opérateur « instanceof »: exemple

```
class Animal {}
class Chien extends Animal {}
class Chat extends Animal {}

public class TestInstanceOf {
 Run main | Debug main | Run | Debug
 public static void main(String[] args) {
 Animal a = new Chien();

 // Test si 'a' est une instance de la classe Chien
 if (a instanceof Chien) {
 System.out.println("a est une instance de Chien");
 }

 // Test si 'a' est une instance de la classe Animal
 if (a instanceof Animal) {
 System.out.println("a est une instance de Animal");
 }

 // Test si 'a' est une instance de la classe Chat
 if (a instanceof Chat) {
 System.out.println("a est une instance de Chat");
 } else {
 System.out.println("a n'est pas une instance de Chat");
 }
 }
}
```

Résultats :

a est une instance de Chien ; a est une instance de Animal ; a n'est pas une instance de Chat

# Héritage

## Héritage et méthodes finales

- Pour empêcher la redéfinition d'une méthode lors de l'héritage, on peut utiliser le mot-clé **final**.
- Une méthode déclarée comme **final** ne peut pas être redéfinie dans une sous-classe.

```
class A {
 final void meth() {
 System.out.println("Méthode finale.");
 }
}

class B extends A {
 void meth() { // Erreur : meth() ne peut pas être redéfinie.
 System.out.println("Illegal !");
 }
}
```

- Dans cet exemple, la tentative de redéfinir la méthode `meth()` dans la classe `B` entraînera **une erreur** de compilation, car la méthode est marquée comme **final** dans la classe `A`.

# Héritage

## Héritage et classes finales

- Pour empêcher qu'une classe soit héritée, il faut la déclarer comme **final** en ajoutant le mot-clé **final** avant sa définition.
- Une classe déclarée comme **final** rend également toutes ses méthodes implicitement finales, ce qui signifie qu'elles ne peuvent pas être redéfinies.

```
final class A {
 // Corps de la classe
}

class B extends A {
 // Erreur : B ne peut pas hériter de A
}
```

# Héritage

## Exercices:

- **Exercice 1 :**
  - 1. Créer une classe Personne avec des attributs nom et prenom, ainsi qu'une méthode afficher().
  - 2. Créer une sous-classe Etudiant qui ajoute un attribut cne et redéfinit la méthode afficher().
- **Exercice 2**
  - 1. Créez une classe Cheval avec trois attributs privés : nom, couleur et anneeNaissance.
    - Ajoutez des méthodes getters et setters pour chacun des attributs.
  - 2. Créez une sous-classe appelée ChevalCourse qui hérite de la classe Cheval et ajoute un nouvel attribut privé : nombreCourses (représentant le nombre de courses terminées par le cheval).
    - Ajoutez des méthodes getters et setters pour cet attribut supplémentaire.
  - 3. Testez les deux classes (Cheval et ChevalCourse) dans une classe de test nommée TestChevaux.

# Héritage

## Exercices:

```
• class Personne {
 protected String nom;
 protected String prenom;

 public Personne(String nom, String prenom) {
 this.nom = nom;
 this.prenom = prenom;
 }

 public void afficher() {
 System.out.println("Nom : " + nom + ", Prénom : " + prenom);
 }
}

class Etudiant extends Personne {
 private String cne;

 public Etudiant(String nom, String prenom, String cne) {
 super(nom, prenom); // Appel au constructeur de la super-classe
 this.cne = cne;
 }

 // Redéfinition de la méthode afficher() sans utiliser @Override
 public void afficher() {
 super.afficher(); // Appel à la méthode afficher() de Personne
 System.out.println("CNE : " + cne);
 }
}
```

```
public class Main {
 Run main | Debug main | Run | Debug
 public static void main(String[] args) {
 Personne p = new Personne(nom:"Dupont", prenom:"Jean");
 p.afficher();

 Etudiant e = new Etudiant(nom:"Durand", prenom:"Alice", cne:"CNE12345");
 e.afficher();
 }
}
```

# Chapitre 5

## *Polymorphisme*

# Polymorphisme

## Introduction

- Le terme *polymorphisme* provient du grec, où *poly* signifie "plusieurs" et *morph* signifie "forme".
- Cela indique qu'une même chose peut prendre différentes formes.
- Nous avons déjà abordé cette notion dans le chapitre 4 avec la redéfinition des méthodes.
- Une même méthode peut avoir des définitions différentes en fonction de la classe dans laquelle elle est utilisée.

# Polymorphisme

## Liaison dynamique

- Considérons la classe B qui hérite de la classe A :

```
class A {
 public void message() {
 System.out.println("Je suis dans la classe A");
 }

 public void g() {
 System.out.println("Méthode g() de la classe A");
 }
}

class B extends A {
 public void message() {
 System.out.println("Je suis dans la classe B");
 }

 public void f() {
 System.out.println("Méthode f() de la classe B");
 }
}
```

- Dans cet exemple : La méthode message() a été **redéfinie** dans la classe B et la méthode f() a été **ajoutée** dans la classe B.

# Polymorphisme

## Liaison dynamique

- Prenons maintenant des instructions pour observer le comportement de ces méthodes.

\*Dans l'exécution on aura les résultats suivants:

```
public class PolyMor {
 public static void main(String[] args) {
 A a = new A();
 B b = new B();

 a.message(); // Appelle la méthode message() de la classe A
 b.message(); // Appelle la méthode message() de la classe B
 b.f(); // Appelle la méthode f() de la classe B
 b.g(); // Appelle la méthode g() de la classe A
 a = new B(); // L'objet de type B est affecté à une référence de type A
 a.message(); // Appelle la méthode message() de la classe B
 }
}
```

Je suis dans la classe A

Je suis dans la classe B

Méthode f() de la classe B

Méthode g() de la classe A

Je suis dans la classe B

# Polymorphisme

## Liaison dynamique

- Lorsqu'une méthode est **redéfinie** dans une sous-classe, c'est toujours **la version la plus spécifique** (celle de la sous-classe) qui **est exécutée**.
- Cela est dû à un processus appelé **liaison dynamique** ou **liaison tardive** (en anglais : *dynamic binding* ou *late binding*).
- En effet, la recherche de la méthode à appeler se fait **à l'exécution** en fonction de la **classe réelle** de l'objet, et non de la classe de la référence utilisée lors de la compilation.
- **Synonymes de Liaison Dynamique** : Liaison tardive, *Dynamic binding*, *Late binding*, *Run-time binding*.

# Polymorphisme

## Polymorphisme et méthodes de classes

- Le polymorphisme ne s'applique pas aux **méthodes de classe (méthodes statiques)**.

```
public class MethodesInstances {

 public static void main(String[] args) {
 Etudiant_ e = new Etudiant_();
 e.message(); // Affiche "Je suis un étudiant"

 e = new EtudiantEtranger();
 e.message(); // Affiche "Je suis un étudiant étranger"
 }

 class Etudiant_ {
 public void message() {
 System.out.println("Je suis un étudiant");
 }
 }

 class EtudiantEtranger extends Etudiant_ {
 @Override
 public void message() {
 System.out.println("Je suis un étudiant étranger");
 }
 }
}
```

L'exécution du programme donnera:

Je suis un étudiant  
Je suis un étudiant étranger

# Polymorphisme

## Polymorphisme et méthodes de classes

- Si on modifie la méthode message() de la classe Etudiant\_, en la rendant **statique**

```
public class MethodesInstances {

 public static void main(String[] args) {
 Etudiant_ e = new Etudiant_();
 e.message(); // Affiche "Je suis un étudiant"

 e = new EtudiantEtranger();
 e.message(); // Affiche "Je suis un étudiant étranger"
 }

 class Etudiant_ {
 public static void message() {
 System.out.println("Je suis un étudiant");
 }
 }

 class EtudiantEtranger extends Etudiant_ {

 public static void message() {
 System.out.println("Je suis un étudiant étranger");
 }
 }
}
```

Alors l'exécution du programme donnera:

Je suis un étudiant  
Je suis un étudiant

C'est la classe du **type de référence** qui est **utilisée** (ici Etudiant), et non la **classe du type réel** de l'objet (ici EtudiantEtranger).

# Polymorphisme

## Utilité du polymorphisme

- Le **polymorphisme** permet d'**éviter** les codes contenant de **nombreux embranchements et tests conditionnels**.

### Exemple :

Considérons deux classes, Rectangle et Cercle, qui héritent d'une super-classe appelée FormeGeometrique. À l'intérieur d'un tableau hétérogène (concept qui sera abordé dans le chapitre suivant), nous pouvons stocker des objets de type Rectangle et Cercle.

- Voici une implémentation de cet exemple :

# Polymorphisme

## Utilité du polymorphisme

- Voici une implémentation de cet exemple :

```
class FormeGeometrique {
 public void dessineRectangle() {} // Méthode vide par défaut
 public void dessineCercle() {} // Méthode vide par défaut
}

class Rectangle extends FormeGeometrique {
 public void dessineRectangle() {
 System.out.println("Un rectangle");}
}
class Cercle_ extends FormeGeometrique {
 public void dessineCercle() {
 System.out.println("Un cercle");}
}

public class TestFormesGeometriques {
 public static void main(String[] args) {
 FormeGeometrique[] figures = new FormeGeometrique[3];
 figures[0] = new Rectangle();
 figures[1] = new Cercle_();
 figures[2] = new Cercle_();
 for (int i=0;i< figures.length; i++) {
 if (figures[i] instanceof Rectangle) {
 figures[i].dessineRectangle();
 } else if (figures[i] instanceof Cercle_) {
 figures[i].dessineCercle();}
 }}}
```

# Polymorphisme

## Utilité du polymorphisme

### Inconvénients du code précédent:

- Le code précédent présente deux principaux inconvénients :
  - Si l'on souhaite ajouter une nouvelle forme géométrique, comme un triangle, il est nécessaire de modifier le code source de la super-classe. En particulier, on doit ajouter une méthode vide, telle que public void dessineTriangle().
  - Pour chaque nouvelle forme géométrique, un nouveau test conditionnel doit être ajouté dans la classe Test, par exemple if (figures[i] instanceof Triangle). Ainsi, plus le programme gère de formes, plus le code devient complexe et encombré de tests.

# Polymorphisme

## Utilité du polymorphisme

- En exploitant le polymorphisme, on peut éviter ces inconvénients et améliorer le code comme suit:

```
class FormeGeometrique {
 public void dessine() {
 // Méthode vide par défaut
 }
}
class Rectangle extends FormeGeometrique {
 @Override
 public void dessine() {
 System.out.println("Un rectangle");}
}
class Cercle_ extends FormeGeometrique {
 @Override
 public void dessine() {
 System.out.println("Un cercle");}

public class TestFormesGeometriques {
 public static void main(String[] args) {
 FormeGeometrique[] figures = new FormeGeometrique[3];
 figures[0] = new Rectangle();
 figures[1] = new Cercle_();
 figures[2] = new Cercle_();
 for (int i=0;i< figures.length; i++) {
 figures[i].dessine(); // Appel polymorphique
 }
 }
}
```

# Polymorphisme

## Utilité du polymorphisme

- Avec cette approche, plusieurs avantages se dégagent :
  1. Réduction des tests conditionnels :
    - Les nombreux tests **instanceof** sont éliminés, rendant le code plus simple et lisible.
  2. Extensibilité du code :
    - Il devient possible d'ajouter de nouvelles sous-classes (par exemple, une classe Triangle) sans modifier le code existant, notamment dans la super-classe ou dans les autres parties du programme. Cela facilite la maintenance et l'évolution du code.

# Polymorphisme

## Exercices:

### Exercice 1 :

- Créer une super-classe appelée Animal avec une méthode parle().
- Créer trois sous-classes : Chien, Chat, et Oiseau, chacune redéfinissant la méthode parle() pour afficher leur cri respectif.
- Dans une classe TestPolymorphisme, utiliser un tableau d'objets Animal pour appeler la méthode parle() sur différents types d'animaux.

### Exercice 2 :

- Créer une super-classe Employe avec une méthode afficheDetails() pour afficher des informations générales sur un employé.
- Créer deux sous-classes, Manager et Developpeur, qui redéfinissent la méthode pour afficher leurs rôles spécifiques.
- Écrire un programme pour afficher les détails de différents types d'employés à l'aide de références de type Employe.

# Polymorphisme

## Solution 1 :

```
public class TestPolymorphismeAnimal {
 public static void main(String[] args) {
 Animal[] animaux = { new Chien(), new Chat(), new Oiseau() };
 for (Animal animal : animaux) {
 animal.parle(); // Appel polymorphique
 }
 }
}

class Animal {
 public void parle() {
 System.out.println("un animal.");
 }
}

class Chien extends Animal {
 @Override
 public void parle() {
 System.out.println("Woof Woof !");
 }
}

class Chat extends Animal {
 @Override
 public void parle() {
 System.out.println("Miaou !");
 }
}

class Oiseau extends Animal {
 @Override
 public void parle() {
 System.out.println("Cui Cui !");
 }
}
```

# Polymorphisme

## Solution 2 :

```
public class TestEmploye {
 public static void main(String[] args) {
 Employe[] employes = { new Manager(), new Developpeur(), new Employe() };
 for (int i=0;i< employes.length; i++) {
 employes[i].afficheDetails(); // Appel polymorphique
 }
 }
}

class Employe {
 public void afficheDetails() {
 System.out.println("Détails de l'employé.");
 }
}

class Manager extends Employe {
 @Override
 public void afficheDetails() {
 System.out.println("Je suis un Manager, je gère des équipes.");
 }
}

class Developpeur extends Employe {
 @Override
 public void afficheDetails() {
 System.out.println("Je suis un Développeur, j'écris du code.");
 }
}
```

# Chapitre 6

## *Tableaux et collections*

# Tableaux et collections

## Tableaux: Déclaration et initialisation

- En Java, un tableau à une dimension peut être déclaré de deux manières équivalentes :  
`type tab[];` ou `type[] tab;`
- Ici, tab est une référence vers un tableau.

### Exemple de déclaration :

- `int tab[];`
- `int[] tab;`
- Contrairement au langage C, l'instruction `int tab[10];` n'est pas autorisée en Java, car la taille du tableau ne peut pas être spécifiée lors de sa déclaration.

# Tableaux et collections

## Tableaux: Déclaration et initialisation

### Remarques :

- En Java, un tableau :
  - est un objet ;
  - est alloué dynamiquement avec l'opérateur new ;
  - contient un nombre fixe d'éléments de même type.
- Un tableau peut être créé soit au moment de sa déclaration, soit en utilisant l'opérateur **new**.

### Exemples de création de tableaux :

#### 1. Initialisation directe lors de la déclaration :

```
int[] tab = {10, 20, 5 * 6};
```

#### 2. Création avec l'opérateur new :

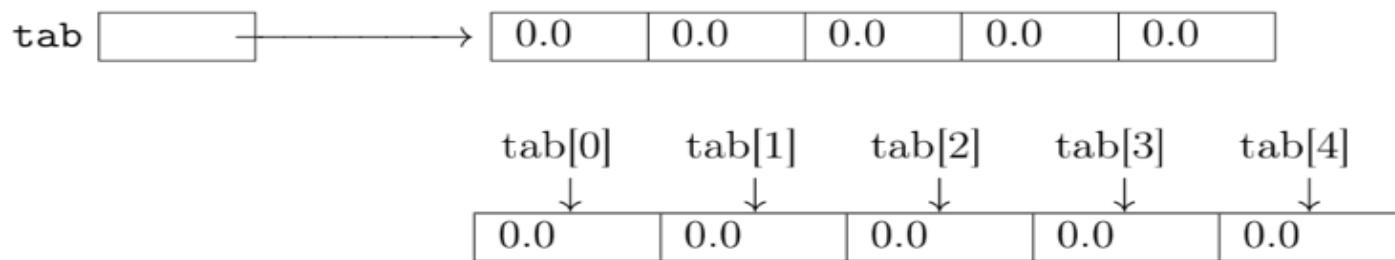
```
int[] tab = new int[5];
```

# Tableaux et collections

## Tableaux: Déclaration et initialisation

*La déclaration :*

- `double[] tab = new double[5];`
- Crée un emplacement pour un tableau de 5 réels (double) et fait la référence à tab comme illustré par le schéma suivant :



- Les valeurs du tableau sont initialisées aux valeurs par défaut (0 pour int, 0.0 pour double, null pour String ...).

# Tableaux et collections

## Tableaux: Déclaration et initialisation

### *Déclaration mixte :*

- Il est possible de déclarer un tableau en même temps que d'autres variables du même type.
- Par exemple, la déclaration suivante :

double scores[], moyenne;

- signifie que :
  - **scores** est un tableau de type double non initialisé,
  - **moyenne** est une simple variable de type double.
- En revanche, la déclaration suivante :

double[] scores, moyenne;

  - crée **deux tableaux** non initialisés de type double.

# Tableaux et collections

## Tableaux: Déclaration et initialisation

*Taille d'un tableau :*

- La taille d'un tableau en Java est accessible grâce à l'attribut **length**, qui est un champ public final de l'objet tableau. Il permet de connaître le nombre d'éléments d'un tableau.

Exemple :

- `int[] scores = new int[10];`
- `System.out.println(scores.length); // Affichera 10`

# Tableaux et collections

## Tableaux: Déclaration et initialisation

*Taille d'un tableau*

Remarque :

- La taille d'un tableau en Java est **fixe** et ne peut pas être modifiée après sa création. Cependant, la **référence** d'un tableau peut être réaffectée à un autre tableau de taille différente.

Exemple :

```
double[] tab1 = new double[10]; // Création d'un tableau de 10 éléments

double[] tab2 = new double[5]; // Création d'un tableau de 5 éléments

tab1 = new double[7]; // tab1 pointe maintenant vers un nouveau tableau de 7 éléments

tab2 = tab1; // tab2 et tab1 référencent désormais le même tableau
```

- Dans cet exemple, **tab1** était initialement associé à un tableau de 10 cases, mais en lui affectant un **nouveau tableau**, il pointe désormais vers un tableau de 7 cases.
- De même, en affectant **tab1** à **tab2**, les deux références désignent le même tableau en mémoire.

# Tableaux et collections

## Tableaux: Parcourir un tableau

- Tout comme en langage C, il est possible d'accéder aux éléments d'un tableau en utilisant **des indices** placés entre **crochets [ ]**.

Exemple :

```
double[] tab = new double[10];

int i;

// Remplissage et affichage du tableau

for (i = 0; i < tab.length; i++) {

 tab[i] = i * i;

 System.out.println("tab[" + i + "] = " + tab[i]);

}
```

# Tableaux et collections

## Tableaux: Parcourir un tableau

Remarque :

- Il est possible de parcourir un tableau avec une **boucle for améliorée (foreach)** :

```
for (double t : tab) {
 System.out.println(t);
}
```

- Cette méthode est uniquement **valable pour la lecture** des éléments du tableau.
- Elle ne permet **pas de modifier** les valeurs.
- Son avantage principal est d'éviter l'utilisation explicite des indices, rendant le code plus clair et lisible.

# Tableaux et collections

## Tableaux: Parcourir un tableau

Exemple :

```
public class BoucleForExemple {
 public static void main(String[] args) {
 // Déclaration et initialisation du tableau
 double[] tab = new double[10];

 // Utilisation d'une boucle for classique pour remplir et afficher le tableau
 System.out.println("Affichage avec la boucle for classique :");
 for (int i = 0; i < tab.length; i++) {
 tab[i] = i * i; // Remplissage du tableau avec i^2
 System.out.println("tab[" + i + "] = " + tab[i]);
 }

 // Utilisation de la boucle foreach pour afficher les valeurs du tableau
 System.out.println("\nAffichage avec la boucle foreach :");
 for (double t : tab) {
 System.out.println(t);
 }
 }
}
```

# Tableaux et collections

## Tableaux: Parcourir un tableau

Exemple : Calcul de la somme des éléments d'un tableau avec une boucle foreach

- Dans cet exemple, nous utilisons une **boucle for-each** pour parcourir un tableau sans avoir besoin d'accéder aux indices et calculer la somme de ses éléments.

```
public class SommeTableau {

 public static void main(String[] args) {
 // Déclaration et initialisation du tableau
 double[] tab = {1.5, 2.0, 3.5, 4.0, 5.5};

 // Initialisation de la somme
 double somme = 0.0;

 // Utilisation de la boucle foreach pour calculer la somme
 for (double t : tab) {
 somme += t; // Ajout de chaque élément à la somme
 }

 // Affichage du résultat
 System.out.println("Somme des éléments du tableau = " + somme);
 }
}
```

# Tableaux et collections

## Tableaux: Copie de tableau

- Comme on l'a vu précédemment, pour deux tableaux référencés par « tab1 » et « tab2 », l'instruction **tab1 = tab2;** ne copie que la référence et non le contenu.

### Exemple :

Dans le programme ci-dessous, après l'affectation **tab1 = tab2;**, **tab1 et tab2 référencent le même tableau**. Toute modification apportée à tab2 se reflétera également dans tab1.

```
public class CopieTableau {
 public static void main(String[] args) {
 // Déclaration et initialisation de trois tableaux
 int[] tab1 = {1, 2, 3, 4, 5};
 int[] tab2 = {0, 2, 4, 6, 8};
 int[] tab3 = {0, 2, 4, 6, 8};

 // Affectation : tab1 pointe maintenant vers le même tableau que tab2
 tab1 = tab2;

 // Comparaison des références des tableaux
 System.out.println("tab1 et tab2 référencent le même objet ? " + (tab1.equals(tab2)));
 System.out.println("tab3 et tab2 référencent le même objet ? " + (tab3.equals(tab2)));

 // Modification du premier élément de tab2
 tab2[0] = 3;

 // Affichage des premiers éléments des tableaux
 System.out.println("tab1[0] = " + tab1[0]); // Affecté par la modification
 System.out.println("tab2[0] = " + tab2[0]); // Modifié
 System.out.println("tab3[0] = " + tab3[0]); // Resté inchangé
 }
}
```

### Résultats:

tab1 et tab2 référencent le même objet ? true  
tab3 et tab2 référencent le même objet ? false  
tab1[0] = 3  
tab2[0] = 3  
tab3[0] = 0

# Tableaux et collections

## Tableaux: Copie de tableau

### Copie du contenu d'un tableau

- Pour copier un tableau dans un autre, deux approches sont possibles :

1. **Copie manuelle élément par élément** (méthode classique) :

```
for (int i = 0; i < tab1.length; i++) {
 tab2[i] = tab1[i]; }
```

- Cette boucle parcourt chaque élément du tableau tab1 et le copie dans tab2.

2. **Utilisation de méthodes prédéfinies de la classe Arrays :**

- **System.arraycopy()**
- **Arrays.copyOf()**
- **Arrays.copyOfRange()**
- Ces méthodes ne seront pas abordées dans ce cours.

# Tableaux et collections

## Tableaux: Passage et retour d'un tableau d'une méthode

- Considérant le programme suivant:

```
public class TableMethodes {
 public static void main(String[] args) {
 int[] tab = {1, 2, 3, 4};
 int i;
 System.out.print("Début de main: ");
 afficher(tab);
 for (i = 0; i < tab.length; i++) {
 test(tab[i]);}
 System.out.print("Fin de main: ");
 afficher(tab);
 }
 public static void test(int x) {
 System.out.print("Début :\t" + x);
 x = 10;
 System.out.println("\tfin :\t" + x);
 }
 public static void afficher(int[] tableau) {
 int i;
 for (i = 0; i < tableau.length; i++) {
 System.out.print(tableau[i] + "-");
 }
 System.out.println();
 }
}
```

### Résultats:

```
Début de main: 1-2-3-4-
Début : 1 fin : 10
Début : 2 fin : 10
Début : 3 fin : 10
Début : 4 fin : 10
Fin de main: 1-2-3-4-
```

Selon le résultat de l'exécution, le contenu du tableau n'a pas été modifié car **x** est local à la méthode **test()**, et toute modification de cette valeur n'est pas visible à l'extérieur. Par conséquent, **le tableau reste inchangé**.

# Tableaux et collections

## Tableaux: Passage et retour d'un tableau d'une méthode

Maintenant, nous passons le tableau en argument à la méthode test

```
public class TableMethodesArg {
 public static void main(String[] args) {
 int[] tab = {1, 2, 3, 4};
 System.out.print("Début de main : ");
 afficher(tab); // Affiche le tableau avant modification
 test(tab); // Modifie les éléments du tableau
 System.out.print("Fin de main : ");
 afficher(tab); // Affiche le tableau après modification
 System.out.println();
 }

 // Méthode qui modifie tous les éléments du tableau en les remplaçant par 10
 public static void test(int[] tableau) {
 for (int i = 0; i < tableau.length; i++) {
 tableau[i] = 10;
 }
 }

 // Méthode qui affiche les éléments du tableau séparés par un tiret
 public static void afficher(int[] tableau) {
 for (int i = 0; i < tableau.length; i++) {
 System.out.print(tableau[i] + " - ");
 }
 System.out.println();
 }
}
```

### Résultats:

Début de main : 1 - 2 - 3 - 4 -  
Fin de main : 10 - 10 - 10 - 10 -

- À la différence de l'exemple précédent, le tableau sera bien modifié, car ici, il est **passé par référence** plutôt que par valeur.
- La méthode test() remplace chaque élément du tableau par 10, et cette modification est visible en dehors de la méthode

# Tableaux et collections

## Tableaux: Passage et retour d'un tableau d'une méthode

### *Retour d'un tableau dans une méthode*

- Une méthode peut retourner un tableau

```
public class RetourneTable {
 public static void main(String[] args) {
 // Appel de la méthode qui retourne un tableau
 int[] tab = scoreInitial();
 // Affichage du contenu du tableau
 System.out.print("Contenu du tableau : ");
 afficherTableau(tab);
 }
 // Méthode qui retourne un tableau d'entiers
 public static int[] scoreInitial() {
 int score[] = {2, 3, 6, 7, 8};
 return score;
 }
 // Méthode pour afficher le contenu d'un tableau
 public static void afficherTableau(int[] tableau) {
 for (int i = 0; i < tableau.length; i++) {
 System.out.print(tableau[i] + " ");
 }
 System.out.println(); // Nouvelle ligne après l'affichage
 }
}
```

Résultats:

Contenu du tableau : 2 3 6 7 8

# Tableaux et collections

## Tableaux: Tableaux d'objets

- L'utilisation des tableaux ne se limite pas aux types primitifs ; il est également possible de créer des **tableaux d'objets**.
- Par exemple, dans la méthode **main(String[] args)**, les tableaux d'objets sont déjà utilisés avec la classe String, qui est une classe en Java.

### Exemple avec la classe Etudiant

- Considérons la classe Etudiant suivante :

```
class Etudiant {
 private String nom, prenom, cne;

 public Etudiant() {
 nom = "";
 prenom = "";
 cne = "";
 }
 // Autres méthodes...
}
```

# Tableaux et collections

## Tableaux: Tableaux d'objets

- La déclaration suivante :

```
Etudiant[] etudiants = new Etudiant[30];
```

- réserve un espace mémoire pour stocker 30 objets de type Etudiant.
  - Toutefois, cette instruction ne crée que des **références aux objets sans les initialiser**.
  - Ainsi, tenter d'exécuter l'instruction suivante :
- ```
etudiants[0].afficher();
```
- entraînera une erreur à l'exécution, car **etudiants[0]** n'a pas encore été **instancié**.
 - Avant d'utiliser un élément du tableau, il est nécessaire de créer un objet et de l'affecter à la case correspondante, par exemple :

```
etudiants[0] = new Etudiant();
```

Tableaux et collections

Tableaux: Tableaux d'objets

- Pour initialiser **tous** les objets du tableau, on peut utiliser une boucle :

```
for (int i = 0; i < etudiants.length; i++) {  
    etudiants[i] = new Etudiant();  
}
```

- Cette approche garantit que chaque élément du tableau contient une instance valide de Etudiant.

Tableaux et collections

Tableaux à plusieurs dimensions

- Il est possible de créer des tableaux à plusieurs dimensions en ajoutant **des crochets ([]).**
- Par exemple, l'instruction suivante :

double[][] matrice; déclare un tableau à deux dimensions de type double.

- Comme pour les tableaux à une seule dimension, la création d'un tableau multidimensionnel se fait à l'aide de l'opérateur **new**.

Exemple :

- L'instruction suivante :
matrice = new double[4][3]; crée un tableau de **4 lignes et 3 colonnes**.
- Il est également possible de **combiner la déclaration et l'initialisation** en une seule étape :

double[][] matrice = new double[4][3];

Tableaux et collections

Tableaux à plusieurs dimensions

Remarques:

- En langage C, un tableau multidimensionnel est en réalité stocké comme un tableau **unidimensionnel** en mémoire.
Par exemple, la déclaration suivante :

double matrice[4][3];

- alloue un bloc **contigu** de 12 éléments double en mémoire.
- En Java, un tableau multidimensionnel **n'est pas stocké de manière contiguë** en mémoire.
- En effet, un tableau à plusieurs dimensions est en réalité un **tableau de tableaux**, où chaque ligne est une référence vers un autre tableau.
- **Flexibilité en Java** : Contrairement à C, en Java, il est possible de définir un tableau à deux dimensions **dont les colonnes n'ont pas la même taille**.

Tableaux et collections

Tableaux à plusieurs dimensions

Exemple 1:

```
public class TableDeuxDemExemple {  
    public static void main(String[] args) {  
        // Déclaration d'un tableau à 2 dimensions de type double  
        double[][] tab = new double[2][]; // tab a 2 lignes, mais les colonnes ne sont pas encore définies  
  
        // Initialisation des colonnes pour chaque ligne  
        tab[0] = new double[3]; // La première ligne a 3 éléments  
        tab[1] = new double[4]; // La deuxième ligne a 4 éléments  
  
        // Affichage de la taille des lignes et des colonnes  
        System.out.println(tab.length); // Affiche 2 (le nombre de lignes du tableau)  
        System.out.println(tab[0].length); // Affiche 3 (le nombre d'éléments dans la première ligne)  
        System.out.println(tab[1].length); // Affiche 4 (le nombre d'éléments dans la deuxième ligne)  
        // Remplissage du tableau avec des valeurs et affichage  
        for (int i = 0; i < tab.length; i++) {  
            for (int j = 0; j < tab[i].length; j++) {  
                tab[i][j] = i + j; // Remplissage des cases avec la somme de l'indice i et j  
                System.out.print("tab[" + i + "][" + j + "] = " + tab[i][j] + " ");  
            }  
            System.out.println(); // Saut de ligne après chaque ligne du tableau  
        }  
    }  
}
```

✓ 39 ⌂ ⌃

2
3
4

tab[0][0] = 0.0 tab[0][1] = 1.0 tab[0][2] = 2.0
tab[1][0] = 1.0 tab[1][1] = 2.0 tab[1][2] = 3.0 tab[1][3] = 4.0

- Cet exemple montre comment créer un tableau à 2 dimensions avec des lignes de tailles différentes (3 éléments pour la première ligne et 4 éléments pour la deuxième ligne),
- Puis comment le remplir avec des valeurs en fonction des indices des lignes et des colonnes, et enfin afficher ces valeurs.

Résultats

Tableaux et collections

Tableaux à plusieurs dimensions

Exemple 2:

```
public class TableauTriangulaire {  
    public static void main(String[] args) {  
        final int N = 4; // Nombre de lignes  
        int[][] tabTriangulaire = new int[N][]; // Déclaration du tableau triangulaire  
  
        // Initialisation des lignes du tableau, où chaque ligne i a i+1 éléments  
        for (int n = 0; n < N; n++) {  
            tabTriangulaire[n] = new int[n + 1];  
        }  
  
        // Affichage du tableau triangulaire  
        for (int i = 0; i < tabTriangulaire.length; i++) {  
            for (int j = 0; j < tabTriangulaire[i].length; j++) {  
                System.out.print(tabTriangulaire[i][j] + "\t"); // Affiche chaque élément suivi d'un tab  
            }  
            System.out.println(); // Saut de ligne après chaque ligne du tableau  
        }  
    }  
}
```

⚠ 1 ✓ 18 ^

Résultats

```
0  
0 0  
0 0 0  
0 0 0 0
```

- Ce programme crée un tableau triangulaire où chaque ligne a un nombre d'éléments croissant.
- La première ligne a un élément, la deuxième ligne en a deux, etc.
- Ensuite, il remplit ce tableau avec des valeurs **par défaut (0)** et les affiche dans un format triangulaire

Tableaux et collections

Parcourir un tableau multidimensionnel

Tout comme pour un tableau à une seule dimension, on peut utiliser une **boucle "for-each"** pour accéder aux éléments d'un **tableau multidimensionnel**.

```
public class TableauMultidimensionnelExp {
    public static void main(String[] args){
        double[][] matrice = new double[4][3];
        // Remplissage du tableau
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 3; j++) {
                matrice[i][j] = i + j;
            }
        }
        // Calcul de la somme des éléments du tableau
        double somme = 0;
        for (double[] ligne : matrice) {
            for (double val : ligne) {
                somme += val;
            }
        }
        // Affichage du résultat
        System.out.println("Somme = " + somme);
    }
}
```

Tableaux et collections

Initialisation d'un tableau multidimensionnel

- L'initialisation d'un tableau multidimensionnel peut se faire de plusieurs manières :

1. Déclaration avec des tailles différentes pour chaque ligne :

```
double[][] matrice = {new double[4], new double[5]};
```

- Ici, matrice est un tableau à deux dimensions où la première ligne contient **4 éléments** et la seconde **5 éléments**.

2. Initialisation directe avec des valeurs :

```
double[][] tabMultiBis = { {1, 2}, {3, 5}, {3, 7, 8, 9, 10} };
```

Tableaux et collections

Initialisation d'un tableau multidimensionnel

Exemple

```
public class TableauMultidimensionnelExp2 {  
    public static void main(String[] args) {  
        // Déclaration et initialisation d'un tableau irrégulier  
        double[][] tabMulti = {new double[4], new double[5]};  
        // Affichage des dimensions du tableau  
        System.out.println("Nombre de lignes de tabMulti : " + tabMulti.length); // Affiche 2  
        System.out.println("Nombre de colonnes dans la première ligne : " + tabMulti[0].length); // Affiche 4  
        System.out.println("Nombre de colonnes dans la deuxième ligne : " + tabMulti[1].length); // Affiche 5  
        // Déclaration et initialisation d'un tableau avec des valeurs  
        double[][] tabMultiBis = {  
            {1, 2},  
            {3, 5},  
            {3, 7, 8, 9, 40}  
        };  
        // Affichage des dimensions de tabMultiBis  
        System.out.println("Nombre de lignes de tabMultiBis : " + tabMultiBis.length); // Affiche 3  
        System.out.println("Nombre de colonnes dans la première ligne : " + tabMultiBis[0].length); // Affiche 2  
        System.out.println("Nombre de colonnes dans la deuxième ligne : " + tabMultiBis[1].length); // Affiche 2  
        System.out.println("Nombre de colonnes dans la troisième ligne : " + tabMultiBis[2].length); // Affiche 5  
    }  
}
```

Tableaux et collections

La collection ArrayList

- En Java, les tableaux ont une taille fixe, ce qui peut être un problème si on ne connaît pas à l'avance le nombre d'éléments à stocker.
- Pour résoudre cela, Java propose l'**API Collections**, qui permet d'utiliser des structures de données flexibles et adaptées à différents besoins.
- Une **collection** est un objet capable de contenir d'autres objets.
- Parmi ces collections, la classe **ArrayList**, située dans le package **java.util**, permet de manipuler **des listes dynamiques**.
- Contrairement aux tableaux classiques, un **ArrayList ajuste sa taille automatiquement** en fonction des ajouts et suppressions d'éléments.

Tableaux et collections

La collection ArrayList

Principales méthodes de ArrayList

- Voici quelques méthodes couramment utilisées avec **ArrayList** :
 - **add(Object element)** → Ajoute un élément à la liste.
 - **get(int index)** → Retourne l'élément situé à l'indice donné.
 - **remove(int index)** → Supprime l'élément à l'indice spécifié.
 - **isEmpty()** → Retourne true si la liste est vide, sinon false.
 - **removeAll()** → Supprime tous les éléments de la liste.
 - **contains(Object element)** → Retourne true si l'élément spécifié est présent dans la liste.
 - **size()** → Retourne le nombre d'éléments dans la liste.

Tableaux et collections

La collection ArrayList

Exemple 1

```
import java.util.ArrayList;
public class ArrayListExample {
    public static void main(String[] args) {
        // Création d'une ArrayList sans type spécifique
        ArrayList Ar = new ArrayList();
        // Ajout d'éléments de différents types
        Ar.add(12);
        Ar.add("Bonjour tout le monde !");
        Ar.add(12.2);
        Ar.add('d');
        // Affichage des éléments avant suppression
        System.out.println("Éléments de la liste avant suppression :");
        for (int i = 0; i < Ar.size(); i++) {
            System.out.println("Indice " + i + " : " + Ar.get(i));
        }
        // Suppression de l'élément à l'index 2 (12.2)
        Ar.remove( index: 2);
        // Affichage après suppression
        System.out.println("\nÉléments de la liste après suppression :");
        for (int i = 0; i < Ar.size(); i++) {
            System.out.println("Indice " + i + " : " + Ar.get(i));
        }
        // Vérification si l'élément 'd' est toujours présent
        System.out.println("\nL'élément 'd' est-il présent ? " + Ar.contains('d'));
    }
}
```

Résultats:

Éléments de la liste avant suppression :
Indice 0 : 12
Indice 1 : Bonjour tout le monde !
Indice 2 : 12.2
Indice 3 : d

Éléments de la liste après suppression :
Indice 0 : 12
Indice 1 : Bonjour tout le monde !
Indice 2 : d

L'élément 'd' est-il présent ? true

Tableaux et collections

La collection ArrayList

Exemple 2

```
import java.util.ArrayList;
public class TabDynamique {
    public static void main(String[] args) {
        ArrayList<Etudiant2> listeEtudiants = new ArrayList<Etudiant2>();
        Etudiant2 et1 = new Etudiant2( nom: "Ali", prenom: "Ahmed", cne: "A7788");
        Etudiant2 et2 = new Etudiant2( nom: "Salim", prenom: "Mohammed", cne: "A8899");
        listeEtudiants.add(et1);listeEtudiants.add(et2);
        System.out.println("Liste des étudiants avant suppression :");
        for (Etudiant2 etudiant : listeEtudiants) {
            etudiant.afficher();}
        listeEtudiants.remove( index: 1);
        System.out.println("\nListe des étudiants après suppression :");
        for (Etudiant2 etudiant : listeEtudiants) {
            etudiant.afficher();}
        System.out.println("\nL'étudiant 'Ali Ahmed' est-il dans la liste ? " + listeEtudiants.contains(et1));}
    class Etudiant2 {
        private String nom, prenom, cne;
        public Etudiant2(String nom, String prenom, String cne) {
            this.nom = nom; this.prenom = prenom; this.cne = cne;}
        public void afficher() {
            System.out.println("Nom : " + nom + ", Prénom : " + prenom + ", CNE : " + cne);}}
```

Résultats:

Liste des étudiants avant suppression :
Nom : Ali, Prénom : Ahmed, CNE : A7788
Nom : Salim, Prénom : Mohammed, CNE : A8899

Liste des étudiants après suppression :
Nom : Ali, Prénom : Ahmed, CNE : A7788
L'étudiant 'Ali Ahmed' est-il dans la liste ? true

Tableaux et collections

Tableaux

- **Exercice 1 :**
- Écrivez un programme en Java qui manipule un tableau d'entiers de taille 10. Ce programme doit accomplir les tâches suivantes :
 1. Afficher tous les entiers du tableau.
 2. Afficher tous les entiers du tableau dans l'ordre inverse.
 3. Calculer et afficher la somme des 10 entiers.
 4. Trouver et afficher l'entier minimum du tableau.
 5. Trouver et afficher l'entier maximum du tableau.
 6. Calculer et afficher la moyenne des entiers du tableau.
 7. Afficher toutes les valeurs du tableau qui sont supérieures à la moyenne calculée.

Tableaux et collections

Tableaux

- **Exercice 1 :**

```
public class TableEntiers {  
    public static void main(String[] args) {  
        // Initialisation du tableau de 10 entiers  
        int[] entiers = {12, 45, 67, 23, 89, 34, 56, 78, 90, 14};  
  
        // 1. Afficher tous les entiers  
        System.out.println("Tous les entiers :");  
        for (int i = 0; i < entiers.length; i++) {  
            System.out.print(entiers[i] + " ");  
        }  
        System.out.println("\n");  
  
        // 2. Afficher tous les entiers dans le sens inverse  
        System.out.println("Entiers dans le sens inverse :");  
        for (int i = entiers.length - 1; i >= 0; i--) {  
            System.out.print(entiers[i] + " ");  
        }  
        System.out.println("\n");  
    }  
}
```

```
// 3. Calculer et afficher la somme des entiers  
int somme = 0;  
for (int i = 0; i < entiers.length; i++) {  
    somme += entiers[i];  
}  
System.out.println("La somme des entiers : " + somme);  
  
// 4. Trouver et afficher l'entier minimum  
int min = entiers[0];  
for (int i = 1; i < entiers.length; i++) {  
    if (entiers[i] < min) {  
        min = entiers[i];  
    }  
}  
System.out.println("L'entier minimum : " + min);  
  
// 5. Trouver et afficher l'entier maximum  
int max = entiers[0];  
for (int i = 1; i < entiers.length; i++) {  
    if (entiers[i] > max) {  
        max = entiers[i];  
    }  
}  
System.out.println("L'entier maximum : " + max);  
  
// 6. Calculer et afficher la moyenne des entiers  
double moyenne = (double) somme / entiers.length;  
System.out.println("La moyenne des entiers : " + moyenne);  
  
// 7. Afficher toutes les valeurs supérieures à la moyenne  
System.out.println("Les valeurs supérieures à la moyenne :");  
for (int i = 0; i < entiers.length; i++) {  
    if (entiers[i] > moyenne) {  
        System.out.print(entiers[i] + " ");  
    }  
}}}
```

Tableaux et collections

Tableaux

Exercice 2

1. **Créer une classe Personne** qui contient :

- Trois attributs privés :nom (type String) , prenom (type String) et tel (type int)
- Un **constructeur** pour initialiser ces attributs.
- Une méthode **afficher()** qui affiche les informations de la personne.

1. **Créer une classe CarnetTel** avec une méthode main() où :

- a) Un **tableau de 10 personnes** est déclaré.
- b) L'utilisateur **saisit les informations** de ces 10 personnes.
- c) Le programme **affiche toutes les personnes** enregistrées.
- d) L'utilisateur saisit un **nom**, et le programme affiche ses informations si elles existent.

1. **Reprendre l'exercice précédent avec une ArrayList :**

- a) Déclarer une ArrayList et y **ajouter cinq personnes**.
- b) **Supprimer** la troisième personne.
- c) **Afficher** les personnes avant et après suppression.
- d) L'utilisateur saisit un **nom**, et le programme affiche ses informations si elles existent.

•

Tableaux et collections

Tableaux

Exercice 2

1. **Créer une classe Personne** qui contient :

- Trois attributs privés :nom (type String), prenom (type String) et tel (type int)
- Un **constructeur** pour initialiser ces attributs.
- Une méthode afficher() qui affiche les informations de la personne.

2. **Créer une classe CarnetTel** avec une méthode main() où :

- a) Un **tableau de 10 personnes** est déclaré.
- b) L'utilisateur **saisit les informations** de ces 10 personnes.
- c) Le programme **affiche toutes les personnes** enregistrées.
- d) L'utilisateur saisit un **nom**, et le programme affiche ses informations si elles existent.

3. **Reprendre l'exercice précédent avec une ArrayList** :

- a) Déclarer une ArrayList et y **ajouter cinq personnes**.
- b) **Supprimer** la troisième personne.
- c) **Afficher** les personnes avant et après suppression.
- d) L'utilisateur saisit un **nom**, et le programme affiche ses informations si elles existent.

Tableaux et collections

Tableaux

```
// Suppression de la troisième personne
if (carnet.size() >= 3) {
    carnet.remove( index: 2);
}

// Affichage après suppression
System.out.println("\nListe des personnes après suppression:");
for (Personne_2 p : carnet) {
    p.afficher();
}

// Recherche par nom
System.out.print("\nSaisir un nom à rechercher: ");
String nomRecherche = scanner.next();
boolean trouve = false;

for (Personne_2 p : carnet) {
    if (p.getNom().equalsIgnoreCase(nomRecherche)) {
        p.afficher();
        trouve = true;
        break;
    }
}
```

Chapitre 7

Classes abstraites, interfaces et packages

Classes abstraites, interfaces et packages

Classes abstraites: Méthodes abstraites

- Une méthode est qualifiée d'**abstraite** lorsqu'elle est déclarée sans être implémentée.
- Autrement dit, seule sa signature est définie, comprenant son type de retour et ses paramètres, sans corps de méthode.
- Les méthodes abstraites sont spécifiées à l'aide du mot-clé **abstract** et doivent obligatoirement être **publiques**.

Exemple :

```
public abstract void f(int i, float x);  
public abstract double g();
```

- Une méthode abstraite définie dans une classe **A** doit être implémentée par ses classes dérivées.
- De plus, une méthode **statique** ne peut pas être abstraite, car il est impossible de la redéfinir dans une sous-classe.

Classes abstraites, interfaces et packages

Classes abstraites: Classes abstraites

- Une **classe abstraite** est une classe qui **ne peut pas être instanciée directement**, car certaines de **ses méthodes sont abstraites**.
- Elle peut toutefois contenir des **variables**, des **méthodes implémentées**, ainsi que des **méthodes abstraites**.
- Pour déclarer une classe abstraite, on utilise le mot-clé **abstract class**.
- Étant **non-instanciable**, une classe abstraite **ne peut pas être utilisée pour créer directement un objet**. Toutefois, il est possible de déclarer une **référence** vers un objet de cette classe.

Exemple :

```
abstract class A {  
    abstract void methodeAbstraite();  
}  
  
A a;      // Autorisé (déclaration d'une référence)  
A a = new A(); // Erreur (instanciation impossible)
```

Classes abstraites, interfaces et packages

Classes abstraites: Classes abstraites

- Si une **classe B hérite** d'une classe abstraite **A** et que **B** est une **classe concrète** (non abstraite), alors il est possible d'écrire :

A a = new B();

- Cela signifie qu'une **référence de type A** peut pointer vers **une instance de B**.
- Une sous-classe d'une classe abstraite doit **obligatoirement** implémenter toutes les méthodes abstraites de la classe mère. Si elle ne le fait pas, elle doit elle-même être déclarée **abstraite**.

Classes abstraites, interfaces et packages

Classes abstraites: Utilité

L'intérêt principal des classes abstraites est de définir une **structure commune** pour un ensemble de classes dérivées, tout en laissant à ces dernières la responsabilité d'implémenter certains comportements spécifiques.

Exemple:

- Prenons l'exemple des classes Cercle et Rectangle, qui héritent d'une classe abstraite FigureGeometrique.
- La méthode de calcul de la surface est différente pour chaque forme.
- Une solution efficace consiste à déclarer une **méthode abstraite surface()** dans FigureGeometrique pour obliger ses sous-classes à fournir leur propre implémentation.

Classes abstraites, interfaces et packages

Classes abstraites: Utilité

Exemple:

```
abstract class FigureGeometrique {
    abstract double surface(); } // Méthode abstraite
class Cercle1 extends FigureGeometrique {
    private double rayon;
    Cercle1(double rayon) {
        this.rayon = rayon;}
    @Override
   💡 double surface() {return Math.PI * rayon * rayon;}}
class Rectangle_ extends FigureGeometrique {
    private double largeur, hauteur;
    Rectangle_(double largeur, double hauteur) {
        this.largeur = largeur;
        this.hauteur = hauteur;}
    @Override
    double surface() {
        return largeur * hauteur;}}
public class TestFigureGeometrique {
    public static void main(String[] args) {
        FigureGeometrique f1 = new Cercle1( rayon: 5);
        FigureGeometrique f2 = new Rectangle_( largeur: 4, hauteur: 6);
        System.out.println("Surface du cercle : " + f1.surface());
        System.out.println("Surface du rectangle : " + f2.surface());}}
```

Surface du cercle : 78.53981633974483

Surface du rectangle : 24.0

Classes abstraites, interfaces et packages

Interfaces: Définition et déclaration

- En C++, l'héritage multiple est pris en charge, tandis qu'en Java, il n'est pas autorisé. Pour pallier cette limitation, Java propose une alternative : les interfaces.

Définition

- Une **interface** est un ensemble de **méthodes abstraites** qui définissent un comportement sans fournir d'implémentation.

Déclaration d'une interface

- La déclaration d'une interface ressemble à celle d'une classe, à la différence qu'on utilise le mot-clé **interface** au lieu de **class**.

Exemple :

```
public interface Forme {  
    double perimetre();  
    double surface();  
}
```

Classes abstraites, interfaces et packages

Interfaces: Définition et déclaration

Propriétés

- Une interface est **implicitement abstraite**, il n'est donc pas nécessaire d'ajouter le mot-clé **abstract** dans sa déclaration.
- Toutes les méthodes d'une interface sont par défaut **publiques et abstraites**, ce qui signifie que les mots-clés **public** et **abstract** peuvent **être omis**.

Règles

- Une interface partage certaines similitudes avec une classe :
 - Elle peut contenir plusieurs méthodes abstraites.
 - Elle peut être définie dans un fichier Java distinct.
 - Elle peut appartenir à un package (voir section package).

Classes abstraites, interfaces et packages

Interfaces: Définition et déclaration

Règles des Interfaces en Java

Cependant, une interface présente aussi des particularités :

- Elle **ne peut pas être instanciée**, donc elle ne peut pas contenir de constructeurs.
- Elle **ne peut pas avoir d'attributs d'instances** ; tous ses attributs doivent être **static** et **final**.
- Elle peut hériter de plusieurs interfaces, ce qui permet un **héritage multiple** au niveau des interfaces.

Remarque :

- Depuis Java 8, une interface peut inclure des **méthodes statiques**.

Classes abstraites, interfaces et packages

Interfaces: Implémentation d'une interface

- Une classe implémente une interface à l'aide du mot-clé **implements**.
- Il s'agit **d'une implémentation et non d'un héritage**.

```
interface Forme {  
    double perimetre();  
    double surface();}  
  
// Implémentation de l'interface Forme par la classe Rectangle  
class Rectangle2 implements Forme1 {  
    private double largeur;  
    private double longueur;  
    // Constructeur  
    public Rectangle2(double largeur, double longueur) {  
        this.largeur = largeur;  
        this.longueur = longueur;}  
    // Méthode pour calculer le périmètre  
    @Override  
    public double perimetre() {  
        return 2 * (largeur + longueur);}  
    // Méthode pour calculer la surface  
    @Override  
    public double surface() {  
        return largeur * longueur;}}
```

```
public class InterfaceForme {  
    public static void main(String[] args) {  
        // Création d'un objet Rectangle  
        Rectangle2 rect = new Rectangle2( largeur: 5.0, longueur: 3.0);  
        // Affichage des résultats  
        System.out.println("Périmètre du rectangle : " + rect.perimetre());  
        System.out.println("Surface du rectangle : " + rect.surface());  
    }  
}
```

Classes abstraites, interfaces et packages

Interfaces: Implémentation d'une interface

Implémentation partielle

- Lorsqu'une classe implémente une interface, elle **doit obligatoirement** fournir une implémentation pour **toutes ses méthodes**. Si ce n'est pas le cas, la classe doit être déclarée comme **abstraite**.

```
// Définition de l'interface Forme
interface Forme {
    double perimetre();
    double surface();
}

// Classe abstraite Rectangle avec une implémentation partielle
abstract class Rectangle2 implements Forme {
    protected double largeur;
    protected double longueur;

    // Implémentation de la méthode surface
    public double surface() {
        return largeur * longueur;
    }
}
```

Classes abstraites, interfaces et packages

Interfaces: Implémentation d'une interface

Implémentation multiple

- Une **classe** a la possibilité d'implémenter **plusieurs interfaces** simultanément.

Exemple

Considérons les 2 interfaces **I1** et **I2** :

```
interface I1 {  
    final static int MAX = 20;  
    void meth1();  
    void meth2();  
}
```

```
interface I2 {  
    void meth3();  
    void meth4();  
}
```

Classes abstraites, interfaces et packages

Interfaces: Implémentation d'une interface

Implémentation multiple

La classe A peut implémenter les interfaces I1 et I2

```
class A implements I1 , I2{
    // attributs
    public void meth1(){
        int i=10;
        //on peut utiliser la constante MAX
        if ( i < MAX)
            i++;
        //implementation de la methode
    }
    public void meth2(){
        // ...
    }
    public void meth3(){
        // ...
    }
    public void meth4(){
        // ...
    }
}
```

Classes abstraites, interfaces et packages

Interfaces: Implémentation d'une interface

Implémentation et héritage

- Une classe B peut hériter de la classe A et implémenter les interfaces I1 et I2, comme illustré dans l'exemple suivant :

```
class B extends A implements I1, I2
```

Classes abstraites, interfaces et packages

Interfaces: Polymorphisme et interface

➤ Déclaration

- Il est possible de déclarer des variables de type interface. Par exemple :

`Forme forme;`

- Toutefois, pour instancier cette variable, une classe concrète implémentant l'interface Forme doit être utilisée. Par exemple :

`forme = new Rectangle(2.5, 4.6);`

- Utilisation avec des Tableaux
- Si plusieurs classes, comme Cercle, implémentent également l'interface Forme, et qu'une classe Carré hérite de Rectangle, il devient possible de manipuler ces objets de manière polymorphique à l'aide de tableaux.

Classes abstraites, interfaces et packages

Interfaces: Polymorphisme et interface

Exemple:

```
interface Forme {  
    double perimetre();  
    double surface();}  
  
// Implémentation de l'interface Forme par la classe Rectangle  
class Rectangle2 implements Forme {  
    private double largeur;  
    private double longueur;  
    // Constructeur  
    public Rectangle2(double largeur, double longueur) {  
        this.largeur = largeur;  
        this.longueur = longueur;}  
    // Méthode pour calculer le périmètre  
    @Override  
    public double perimetre() {  
        return 2 * (largeur + longueur);}  
    // Méthode pour calculer la surface  
    @Override  
    public double surface() {  
        return largeur * longueur;}}
```

```
// Classe Cercle implementant l'interface Forme  
class Cercle2 implements Forme {  
    private double rayon;  
    // Constructeur  
    public Cercle2(double rayon) {  
        this.rayon = rayon;}  
    // Méthode pour calculer le périmètre  
    @Override  
    public double perimetre() {  
        return 2 * Math.PI * rayon;}  
    // Méthode pour calculer la surface  
    @Override  
    public double surface() {  
        return Math.PI * rayon * rayon;}  
    // Méthode spécifique pour calculer le diamètre  
    public double diametre() {  
        return 2 * rayon;}}
```

```
// Classe Carré héritant de Rectangle  
class Carre extends Rectangle2 {  
    // Constructeur  
    public Carre(double largeur) {  
        super(largeur, largeur);}  
    // Programme principal  
public class InterfaceForme {  
    public static void main(String[] args) {  
        // Création d'un tableau de formes  
        Forme[] tabForme = new Forme[3];  
        tabForme[0] = new Rectangle2( largeur: 10, longueur: 20);  
        tabForme[1] = new Cercle2( rayon: 3);  
        tabForme[2] = new Carre( largeur: 10);  
        // Affichage des surfaces et périmètres  
        for (int i = 0; i < 3; i++) {  
            System.out.println("Surface : " + tabForme[i].surface()  
                + ", Périmètre : " + tabForme[i].perimetre());  
    }}}
```

Classes abstraites, interfaces et packages

Interfaces: Polymorphisme et interface

Exemple:

Conversion de type (Casting)

- Ajoutons une méthode **diametre()** à la classe **Cercle** :

```
class Cercle implements Forme {  
    private double rayon;  
    ...  
    public double diametre() {  
        return 2 * rayon;  
    }  
}
```

Classes abstraites, interfaces et packages

Interfaces: Polymorphisme et interface

Exemple:

Conversion de type (Casting)

- Prenons l'instruction suivante :

Forme forme = new Cercle(4);

- Si nous tentons d'appeler la méthode diametre() comme ceci :

double d = forme.diametre();

- Cela entraînera **une erreur de compilation**, car forme est déclaré comme une instance de l'interface Forme, qui ne définit pas la méthode diametre().

- Pour corriger cette erreur, **un cast explicite** est nécessaire afin de convertir forme en un objet de type Cercle :

double d = ((Cercle) forme).diametre();

- Cela permet d'accéder correctement à la méthode diametre() définie dans la classe Cercle.

Classes abstraites, interfaces et packages

Interfaces: Héritage et interface

- Une interface a la possibilité d'hériter **d'une** ou de plusieurs autres interfaces.
- L'interface **I3** hérite des interfaces **I1** et **I2**, ce qui signifie qu'elle hérite de toutes leurs méthodes. Ainsi, l'instruction :

```
interface I3 extends I1, I2 {  
    void meth5(); }
```

est équivalente à une interface regroupant toutes les méthodes des interfaces **I1** et **I2**, tout en ajoutant sa propre méthode **meth5()**.

```
interface I3 {  
    final static int MAX = 20;  
    void meth1();  
    void meth2();  
    void meth3();  
    void meth4();  
    void meth5();  
}
```

Classes abstraites, interfaces et packages

Interfaces: Exercice

1. Créer une interface **Animal** avec une méthode **faireSon()**.
2. Implémenter cette interface dans deux classes :
 - **Chien**, qui affiche "Le chien aboie".
 - **Chat**, qui affiche "Le chat miaule".
3. Dans la classe principale **TestAnimal**, créer un tableau d'objets **Animal** et afficher le son de chaque animal.

Classes abstraites, interfaces et packages

Packages:

- Un **package** est un regroupement de classes permettant d'organiser plus efficacement un programme.
- Si aucun package n'est spécifié, les classes sont placées automatiquement dans **le package par défaut (default package)**.
- Par exemple, pour la saisie à partir du clavier, nous avons utilisé **la classe Scanner**, pour cela nous avons importé le **package java.util**.

Classes abstraites, interfaces et packages

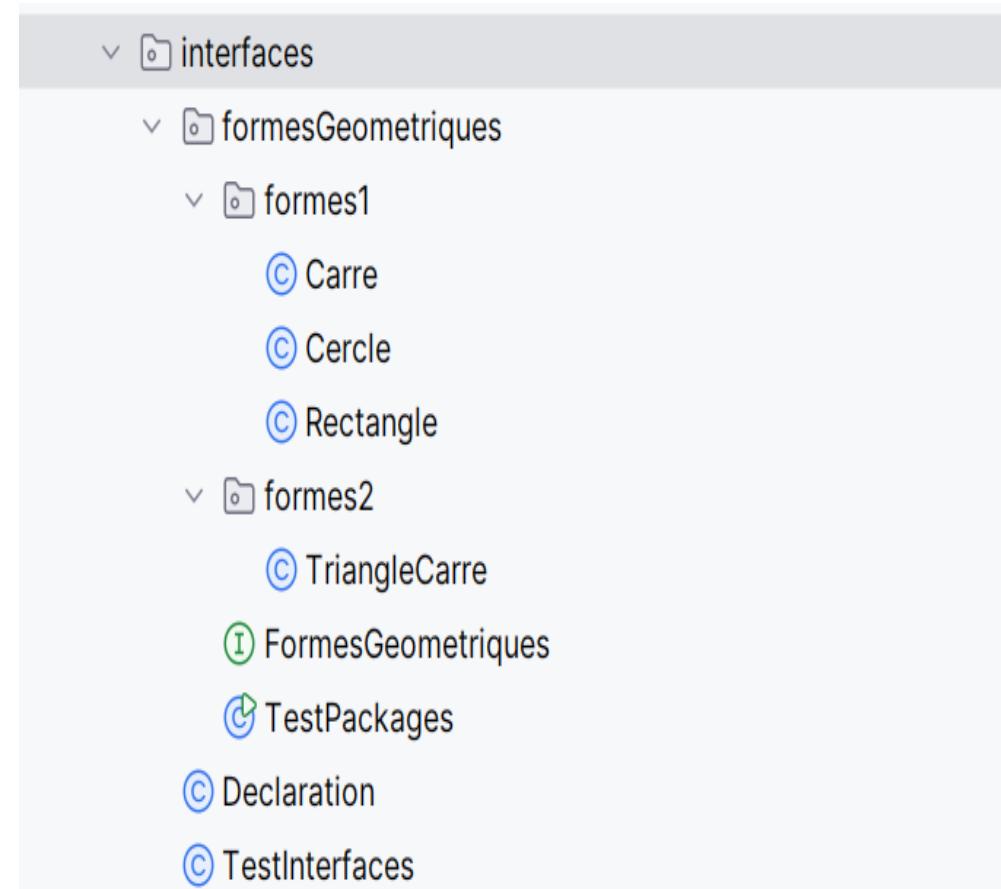
Packages: Création d'un package

- Pour définir un package, on utilise le mot-clé **package**.
- Il est obligatoire d'ajouter l'instruction suivante au début de chaque fichier :
package nomPackage;
- De plus, toute classe publique doit être déclarée dans un fichier distinct afin d'être accessible au sein du package.
- L'accès à une classe publique se fait en utilisant le nom du package correspondant.

Classes abstraites, interfaces et packages

Packages: Exemple

- On considère la hiérarchie suivante dont La structure du projet est organisée comme suit :
- Le répertoire **interfaces** contient :
 - Le répertoire **formesGeometriques**
 - Les fichiers Declaration.java et TestInterfaces.java
- Le répertoire **formesGeometriques** contient :
 - Les sous-répertoires **formes1** et **formes2**
 - Les fichiers FormeGeometriques.java et TestPackages.java
- Le répertoire **formes2** contient :
 - Le fichier TriangleCarre.java
- Le répertoire **formes1** contient :
 - Les fichiers Carre.java, Cercle.java et Rectangle.java



Classes abstraites, interfaces et packages

Packages: Exemple

- Étant donné que la classe `TriangleCarre` est située dans le répertoire `formes2`, il est nécessaire d'ajouter l'instruction suivante au début du fichier :

```
package interfaces.formesGeometriques.formes2;
```

- De plus, comme `TriangleCarre` hérite de la classe `Rectangle`, qui se trouve dans le répertoire `formes1`, il faut importer cette classe avec l'instruction :

```
import interfaces.formesGeometriques.formes1.Rectangle;
```

Classes abstraites, interfaces et packages

Packages: Exemple

- La classe **TestPackages** est utilisée pour tester les différentes classes. Par conséquent, il est nécessaire d'inclure les instructions suivantes au début du fichier :

```
package interfaces.formesGeometriques;  
  
import interfaces.formesGeometriques.formes2.TriangleCarre;  
  
import interfaces.formesGeometriques.formes1.Carre;  
  
import interfaces.formesGeometriques.formes1.Cercle;  
  
import interfaces.formesGeometriques.formes1.Rectangle;
```

- Pour simplifier l'importation des classes situées dans le package **formes1**, on peut regrouper les trois dernières instructions en une seule :

```
import interfaces.formesGeometriques.formes1.*;
```

Classes abstraites, interfaces et packages

Packages: Exemple

```
package interfaces.formesGeometriques;

import interfaces.formesGeometriques.formes2.TriangleCarre;
import interfaces.formesGeometriques.formes1.*;
public class TestPackages {
    public static void main(String[] args) {
        // Création d'un tableau de formes
        FormesGeometriques[] tabForme = new FormesGeometriques[2];
        // Initialisation des différentes formes
        tabForme[0] = new Rectangle( largeur: 10, hauteur: 20);
        tabForme[1] = new TriangleCarre( larg: 3, longu: 4, cote: 5);
        // Affichage des surfaces de chaque forme
        for (int i = 0; i < tabForme.length; i++) {
            System.out.println("Surface de la forme " + (i + 1) + " : " + tabForme[i].surface());
        }
    }
}

package interfaces.formesGeometriques;

public interface FormesGeometriques {
    public double surface();
}
```

```
✓ 9 . package interfaces.formesGeometriques.formes1;
import interfaces.formesGeometriques.FormesGeometriques;

public class Rectangle implements FormesGeometriques {

    private double largeur, longueur;
    public Rectangle (double largeur, double hauteur) {
        this.largeur = largeur;
        this.longueur = hauteur;
    }
    @Override
    public double surface() {
        return largeur * longueur;
    }
}

package interfaces.formesGeometriques.formes2;
import interfaces.formesGeometriques.formes1.Rectangle;

public class TriangleCarre extends Rectangle {
    private double cote;
    // Constructeur
    public TriangleCarre(double larg, double longu, double cote) {
        super(larg, longu);
        this.cote = cote;
    }
    // Calcul de la surface (moitié de celle du rectangle parent)
    public double surface() {
        return super.surface() / 2;
    }
}
```

Classes abstraites, interfaces et packages

Packages: Classes du même package

- Lorsqu'une classe se trouve dans le même package que les autres classes avec lesquelles elle interagit, il n'est pas nécessaire d'effectuer des importations.
- De plus, il est possible de déclarer plusieurs classes dans un même fichier, à condition qu'elles ne soient pas publiques, ce qui empêche leur accessibilité depuis d'autres packages.
- Prenons l'exemple du fichier Carre.java, qui contient les classes Carre et Cube

Classes abstraites, interfaces et packages

Packages: Classes du même package

```
package interfaces.formesGeometriques.formes1;

public class Carre extends Rectangle {
    private double largeur;
    public Carre(double largeur) {
        super(largeur, largeur);
        this.largeur = largeur;}
    double getLargeur() {
        return largeur;
    }
}
// Classe Cube non publique, accessible uniquement dans le même package
class Cube extends Carre {
    private double profondeur;
    public Cube(double largeur, double profondeur) {
        super(largeur);
        this.profondeur = profondeur;
    }
    double getProfondeur() {
        return profondeur;
    }
}
```

Classes abstraites, interfaces et packages

Packages: Classes du même package

Remarques:

- Les classes Rectangle et Carre sont dans le même package, donc on n'a pas besoin de faire des importations.
- La classe Cube n'est pas accessible depuis les autres packages, car elle n'a pas été déclarée comme public. Ainsi, dans la classe TestPackages, une instruction comme :

`Cube cube = new Cube();`

- entraînera une erreur de compilation. Cette erreur se produit car TestPackages est situé dans le package interfaces.formesGeometriques, tandis que Cube appartient au package interfaces.formesGeometriques.formes1 et n'est pas déclarée comme une classe publique.

Classes abstraites, interfaces et packages

Packages: Classes du même package

Remarques:

Un fichier JAR (*Java ARchive*) est une archive compressée qui regroupe les classes compilées d'un projet Java.

Création d'un fichier JAR sous Eclipse

Pour générer un fichier JAR à partir d'un projet sous Eclipse, suivez ces étapes :

1. Faites un **clic droit** sur le nom du projet.
2. Sélectionnez **Export**.
3. Dans la section **Java**, choisissez **JAR** et cliquez sur **Next**.
4. Définissez un nom pour le fichier JAR dans la section **JAR file** (ex. : test.jar).
5. Cliquez deux fois sur **Next**, puis sélectionnez la classe principale.
6. Cliquez sur **Finish** pour finaliser l'exportation.

•

Classes abstraites, interfaces et packages

Packages: Classes du même package

Remarques: Exécution du fichier JAR

Une fois l'archive test.jar créée, placez-vous dans le répertoire où elle est enregistrée, puis exéutez-la avec la commande suivante :

```
java -jar test.jar
```

-

Chapitre 8

Gestion des exceptions

Gestion des exceptions

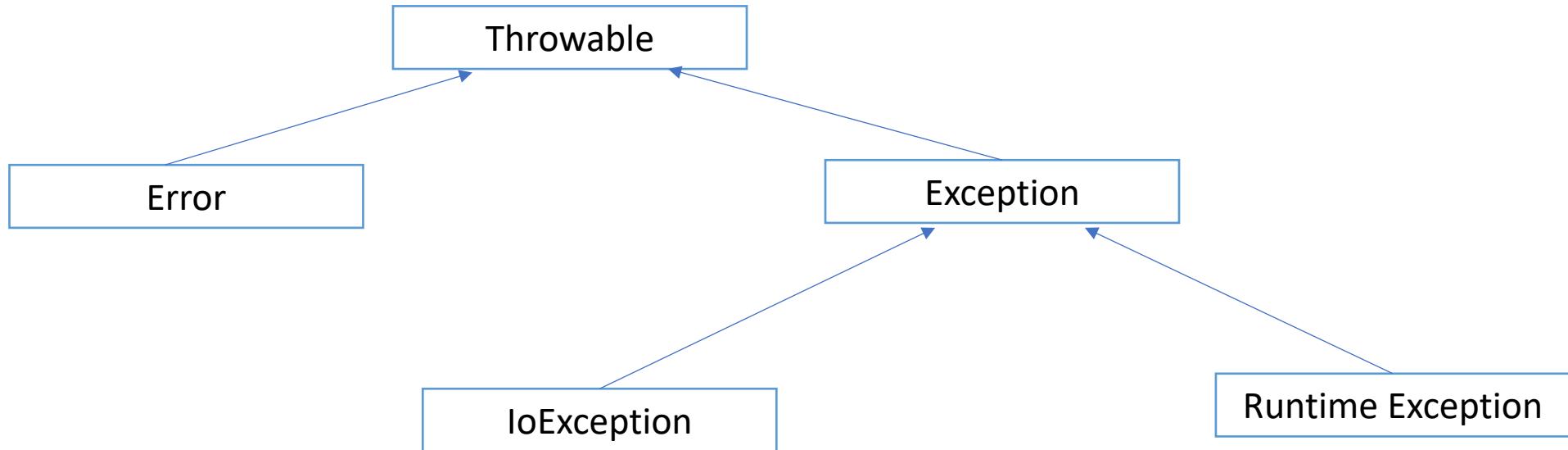
Introduction

- Une exception est un signal indiquant qu'un événement inhabituel, tel qu'une erreur, s'est produit.
- Elle interrompt le déroulement normal du programme afin de gérer séparément les situations anormales, évitant ainsi de complexifier le code principal.
- Cela permet de rendre le traitement standard plus clair et plus lisible.
- Voici quelques exemples d'exceptions courantes :
 - Division par zéro ;
 - Saisie incorrecte de l'utilisateur (ex : une chaîne de caractères au lieu d'un entier) ;
 - Dépassement des limites d'un tableau ;
 - Tentative de lecture ou d'écriture dans un fichier inexistant ou non accessible.

Gestion des exceptions

Exceptions

- En Java, la gestion des erreurs repose sur deux classes principales : **Error** et **Exception**.
- Ces deux classes sont des sous-classes de **Throwable**, comme illustré dans la figure suivante.



Gestion des exceptions

Exceptions

- La classe **Error** désigne des erreurs critiques qui ne peuvent pas être gérées par le programme.
 - Par exemple, un manque de mémoire empêchant l'exécution d'un programme.
- La classe **Exception** regroupe les erreurs moins graves qui peuvent être traitées dans le programme.
 - Elle se divise en deux sous-classes principales :
 - **IOException**, qui concerne les erreurs d'entrée/sortie;
 - **RuntimeException**, qui englobe les erreurs liées à des fautes de programmation.

Gestion des exceptions

Types d'exceptions

Exception

ArithmException

Description

Se produit lors d'une erreur arithmétique, comme une division par zéro.

Se déclenche lorsque l'on tente d'accéder à un index en dehors des limites d'un tableau.

```
double[] tab = new double[10];  
tab[10] = 1.0; // Erreur  
tab[-1] = 1.0; // Erreur
```

ArrayIndexOutOfBoundsException

Survient lorsqu'un tableau est déclaré avec une taille négative.

```
double[] tab = new double[-8]; // Erreur
```

NegativeArraySizeException

Se produit lorsqu'une tentative de conversion de type (cast) est invalide.

Se déclenche lorsqu'une chaîne de caractères ne peut pas être convertie en un type numérique.

```
String s = "tt";  
double x = Double.parseDouble(s); // Erreur
```

NumberFormatException

Se produit lorsqu'un index dépasse la taille d'une chaîne de caractères.

```
String s = "bb";  
char c = s.charAt(2); // Erreur  
char c = s.charAt(-1); // Erreur
```

StringIndexOutOfBoundsException

Se déclenche lorsqu'on tente d'accéder à un objet non initialisé (null).

```
A[] a = new A[2]; // A est une classe  
a[0].x = 2; // Erreur : a[0] n'a pas été initialisé
```

NullPointerException

Gestion des exceptions

Types d'exceptions

Exemple

```
class Animal {  
}  
  
class Chien extends Animal {  
    int taille = 80;  
}  
  
public class Conversion {  
    public static void main(String args[]) {  
        Animal animal = new Animal();  
        Chien chien = new Chien();  
        chien = (Chien)animal;  
    }  
}
```

On aura l'exception suivante: [Exception in thread "main" java.lang.ClassCastException: class Animal3 cannot be cast to class Chien3 \(Animal3 and Chien3 are in unnamed module of loader 'app'\)](#) at
Conversion.main(Conversion.java:8)

Gestion des exceptions

Méthodes des exceptions

- La **classe Throwable** fournit plusieurs méthodes utiles pour la gestion des exceptions. Voici quelques-unes des plus courantes

Méthode

`public String getMessage()`

`public String toString()`

`public void printStackTrace()`

Description

Renvoie un message décrivant l'exception qui s'est produite.

Retourne le nom de la classe de l'exception suivi du message renvoyé par `getMessage()`.

Affiche le résultat de `toString()` ainsi que la trace complète de l'erreur, facilitant le débogage.

Gestion des exceptions

Méthodes des exceptions

Exemple

```
public class TestException {  
  
    public static void main(String[] args) {  
        int i = 3, j = 0;  
        try {  
            System.out.println("resultat = " + (i / j)); // Division par zéro -> Exception  
        } catch (ArithmeticsException e) {  
            System.out.println("1. " + e.getMessage()); // Affiche le message de l'exception  
            System.out.println("2. " + e.toString()); // Affiche le nom de l'exception et son message  
            System.out.print("3.");  
            e.printStackTrace(); // Affiche la trace complète de l'erreur  
        }  
    }  
}
```

Exécution:

```
1. / by zero  
2. java.lang.ArithmeticsException: / by zero  
3.java.lang.ArithmeticsException: / by zero  
     at TestException.main(TestException.java:6)
```

Gestion des exceptions

Captures des exceptions

- Lors des différents tests, le programme s'est interrompu de manière abrupte.
- Cependant, il est possible de capturer ces exceptions (to catch) afin de permettre la poursuite de l'exécution du programme.
- Pour cela, on utilise cinq mots-clés : **try**, **catch**, **throw**, **throws** et **finally**.
- La structure générale d'un bloc try est la suivante :
- **TypeException** représente le type d'exception qui a été déclenché, et **excepObj** est l'objet associé à cette exception.

Remarque :

- **TypeException** peut être soit une classe d'exception intégrée à Java, soit une classe d'exception personnalisée définie par l'utilisateur.

```
try {  
    // Traitement normal  
    // Code susceptible de generer une erreur  
} catch (TypeException1 excepObj) {  
    // Traitement en cas d'exception de type TypeException1  
  
} catch (TypeException2 excepObj) {  
    // Traitement en cas d'exception de type TypeException2  
  
// ...  
} finally {  
    // Code a executer avant la fin du bloc try/catch  
}  
// Code apres le try
```

Gestion des exceptions

Utilisation du bloc try-catch

□ Un seul catch:

- Pendant l'exécution du programme suivant :

```
public class DivideZero {  
    public static void main(String[] args) {  
        int n = 0;  
        System.out.println("1/" + n + " = " + 1/n);  
    }  
}
```

- Une **ArithmException** a été générée en raison d'une division par zéro.
- Pour éviter cette erreur et gérer l'exception, on peut modifier le programme en intégrant un bloc try, comme illustré ci-dessous :

Gestion des exceptions

Utilisation du bloc try-catch

❑ Plusieurs catch: Voici un exemple qui génère au moins deux exceptions

```
import java.util.*; // Importation de la bibliothèque utilitaire
public class ExceptionBloc {
    public static void main(String[] args) {
        int n;
        Scanner clavier = new Scanner(System.in); // Création d'un objet Scanner pour la saisie utilisateur
        try {
            System.out.print("Saisir un entier : ");
            n = clavier.nextInt(); // Lecture de l'entier saisi
            System.out.println("1/" + n + " = " + (1 / n)); // Division par l'entier saisi
        }
        catch (ArithmetricException e) {
            System.out.println("Impossible de diviser par 0");
            System.out.println(e.getMessage()); // Affichage du message d'erreur
        }
        catch (InputMismatchException e) {
            System.out.println("Vous n'avez pas saisi un entier valide");
            System.out.println(e); // Affichage de l'exception sous forme de texte
            clavier.nextLine(); // Nettoyage du buffer d'entrée
        }
        System.out.println("Fin du programme"); // Message final
        clavier.close(); // Fermeture du scanner
    }
}
```

⚠ 1 ✅ 26 ^

Gestion des exceptions

Utilisation du bloc tr-catch

❑ *Plusieurs catch:*

- Puisque les classes `ArithmException` et `InputMismatchException` sont des sous-classes de `Exception`, elles peuvent être capturées simultanément dans un bloc `catch générique` en utilisant la classe `Exception`.

```
import java.util.*;
public class ExceptionGenerique {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.print("Saisissez un nombre entier : ");
            int n = scanner.nextInt();
            System.out.println("Résultat de 1 / " + n + " = " + (1 / n));
        } catch (Exception e) { // Capture toutes les exceptions qui héritent de Exception
            System.out.println("Une erreur s'est produite : " + e.toString());
        }
        System.out.println("Fin du programme.");
        scanner.close();
    }
}
```

Gestion des exceptions

Utilisation du bloc try-catch

❑ Bloc Finally:

- Le **bloc finally** (optionnel) est toujours exécuté, même en l'absence d'exception.

```
public class BlocFinally {
    public static void main(String[] args) {
        int tab[] = new int[2];

        try {
            tab[2] = 1; // Provoque une exception de dépassement de tableau
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception détectée : ");
            e.printStackTrace(); // Affiche les détails de l'exception
            // ou bien e.printStackTrace(System.out);
        } finally {
            tab[0] = 6; // Modification du tableau dans le bloc finally
            System.out.println("tab[0] = " + tab[0]);
            System.out.println("Le bloc finally est exécuté");
        }
    }
}
```

Gestion des exceptions

Exceptions personnalisées

□ Déclenchement d'une exception:

- Prenons une classe Point, qui possède un constructeur acceptant deux paramètres représentant la position du point dans un plan, ainsi qu'une méthode deplace() permettant de modifier cette position.
- Le programme s'interrompra si l'une des coordonnées du point est négative.
- Pour gérer cette situation, nous pouvons créer deux classes d'exception personnalisées, ErrCons et ErrDepl, qui étendent la classe Exception :

```
class ErrDepl extends Exception { }
```

```
class ErrCons extends Exception { }
```

- Une méthode peut indiquer qu'elle est susceptible de générer une exception en utilisant le mot-clé throws.

Gestion des exceptions

Exceptions personnalisées

□ Déclenchement d'une exception:

- Une méthode déclare qu'elle peut lancer (déclencher) une exception p en utilisant le mot-clé throws.
- Exemple

```
public void deplace(int dx, int dy) throws ErrDepl {  
    // Indique que la méthode deplace() peut déclencher une exception  
    ...  
}
```

- Ensuite, la méthode peut effectivement déclencher une exception en créant et en lançant un nouvel objet d'exception à l'aide du mot-clé throw

```
public void deplace(int dx, int dy) throws ErrDepl {  
    if ((x + dx < 0) || (y + dy < 0)) {  
        throw new ErrDepl(); // Détection et génération d'une exception  
    }  
    x = x + dx;  
    y = y + dy; // Traitement normal  
}
```

Gestion des exceptions

Exceptions personnalisées

Exemple complet:

```
// Définition des exceptions personnalisées
class ErrConst extends Exception {
    public ErrConst() {
        super("Erreur : Les coordonnées du point ne peuvent pas être négatives.");}
}
class ErrDepl extends Exception {
    public ErrDepl() {
        super("Erreur : Déplacement invalide entraînant des coordonnées négatives.");}
}

// Classe Point avec gestion des exceptions
class Point {
    private int x, y;
    // Constructeur avec vérification des coordonnées initiales
    public Point(int x, int y) throws ErrConst {
        if (x < 0 || y < 0) {
            throw new ErrConst();}
        this.x = x;
        this.y = y;}
    // Méthode pour déplacer le point avec gestion des erreurs
    public void deplace(int dx, int dy) throws ErrDepl {
        if ((x + dx < 0) || (y + dy < 0)) {
            throw new ErrDepl();}
        x += dx;
        y += dy;
    }
}
```

```
// Classe de test
public class TestPoint {
    public static void main(String[] args) {
        try {
            // Création et déplacement d'un point valide
            Point a = new Point( x: 1, y: 4);
            a.deplace( dx: 0, dy: 1);

            // Création d'un autre point valide
            a = new Point( x: 7, y: 4);
            a.deplace( dx: 3, dy: -5); // Déplacement valide

            // Déplacement entraînant une exception
            a.deplace( dx: -1, dy: -5);
        } catch (ErrConst e) {
            System.out.println("Erreur de Construction : " + e.getMessage());
            System.exit( status: -1);
        } catch (ErrDepl e) {
            System.out.println("Erreur de Déplacement : " + e.getMessage());
            System.exit( status: -1);
        }
    }
}
```

Gestion des exceptions

Exceptions personnalisées

Exercice 1:

Écrire un programme qui demande à l'utilisateur d'entrer deux nombres entiers, puis effectue une division. Votre programme doit gérer plusieurs types d'exceptions possibles.

- 1. Demander à l'utilisateur d'entrer deux nombres entiers.**
- 2. Effectuer la division du premier nombre par le second.**
- 3. Gérer les exceptions suivantes :**
 - InputMismatchException si l'utilisateur entre autres chose qu'un entier.
 - ArithmeticException si l'utilisateur tente de diviser par zéro.
 - Une exception générale pour capturer tout autre problème inattendu.
- 4. Affichage des messages appropriés en cas d'erreur**

Exercice 2:

Écrire un programme qui demande à l'utilisateur de saisir un nombre réel, puis calcule et affiche sa racine carrée en utilisant la méthode Math.sqrt(). Si l'utilisateur entre un nombre négatif, le programme doit afficher un message d'erreur approprié et gérer l'exception.