

# Programmation Orientée Objet

Dr.-Ing. Afef Mdhaffar

École Nationale d'Ingénieurs de Sfax

Université de Sfax

# Objectifs de ce cours

- Maîtriser les concepts principaux de la programmation orientée objet
- Être capable de résoudre un problème en suivant le paradigme orienté objet
- Maîtriser la syntaxe du langage Java

# Plan de ce cours

- Chapitre 1 : Introduction
- Chapitre 2 : Classes et Objets
- Chapitre 3 : Encapsulation
- Chapitre 4 : Héritage et Polymorphisme
- Chapitre 5 : Classes abstraites et interfaces
- Chapitre 6 : Les collections
- Chapitre 7 : Les exceptions

# Chapitre 1 : Introduction

# Objectifs

- Comprendre les avantages de la Programmation Orientée Objet (POO) par rapport à la programmation procédurale
- Terminologie
- Avoir une idée sur les langages POO existants
- Maîtriser l'histoire de Java et son fonctionnement
- Avoir une idée sur les environnements de développement existants

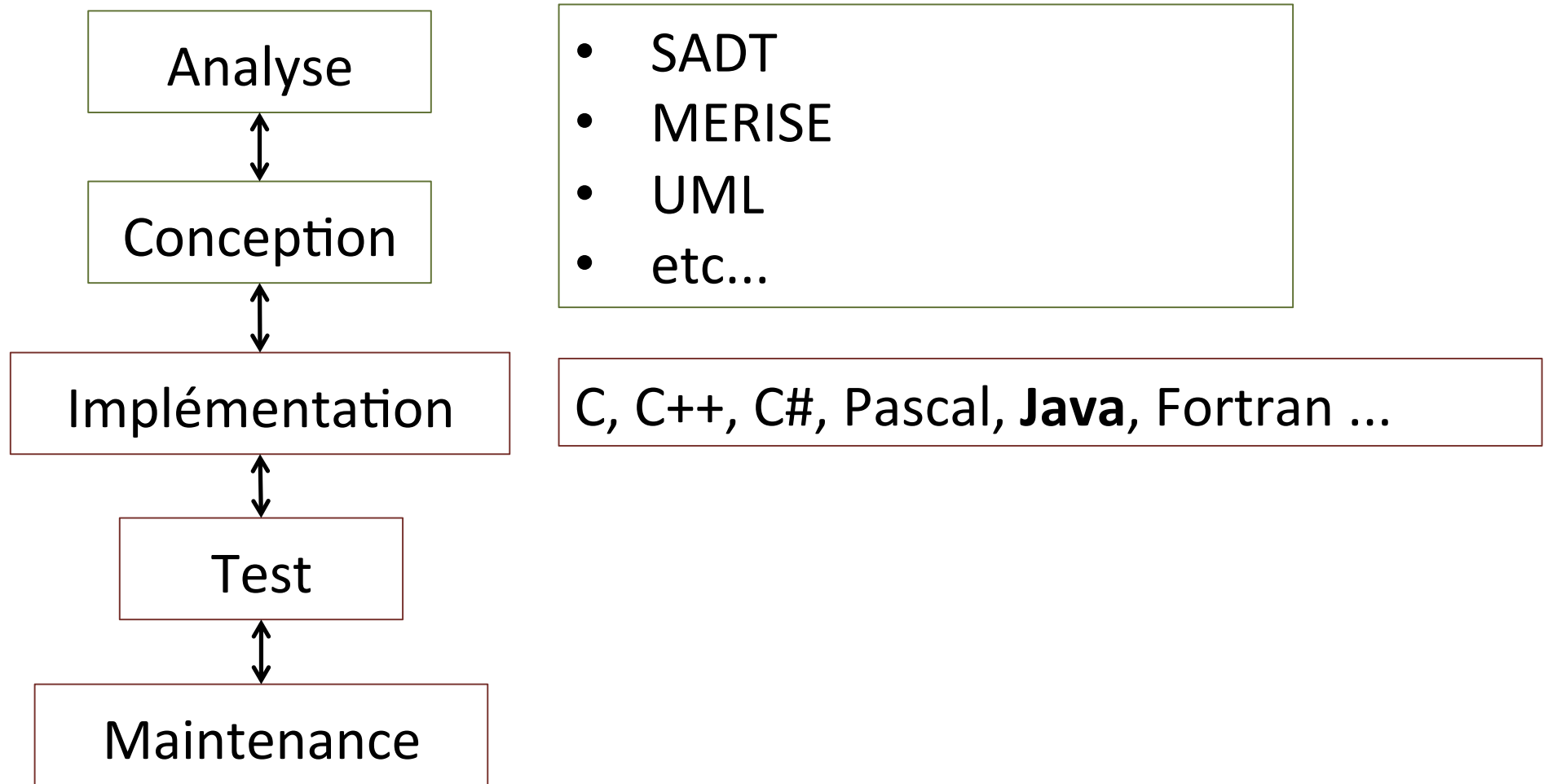
# Plan

- Du problème au programme
- De la programmation procédurale à la programmation modulaire
- Introduction à la POO : notions de base et terminologie
- Langages de POO existants
- Java
- Les environnements de développement existants
- Bilan

# Du problème au programme (1/2)

- Comment transformer un problème en un programme ?
  - Analyser
  - Traduire en algorithme
  - Développer une programme informatique
- Évident, mais pourtant !!
  - Pour construire un bâtiment, il faut faire un plan.
  - Même chose en informatique :
    - conception de l'algorithme : une réponse (rationnelle) au problème !
    - mise en œuvre technique - le *codage* - dans un langage de programmation.

# Du problème au programme (2/2)





# Analyse du problème

- Il faut être :
  - **efficace** : quelle méthode me permettra d'obtenir le **plus simplement**, le **plus clairement** et le **plus vite** les résultats attendus ?
  - **paresseux** : Comment pourrais je réutiliser ce que j'ai développé ?
  - **prévoyant** : Comment s'assurer que le programme pourra être réutilisable et extensible ?

# De la programmation procédurale à la programmation modulaire (1/3)

## Programmation procédurale

« le focus est mis sur **l'exécution du programme** »

- Organiser les traitements
  - Définition des fonctions qui admettent des arguments (inputs) et rendent des valeurs (outputs)
  - Chainage et réutilisation de fonctions
  - Découpage fonctionnel permettant de factoriser certains comportements
- Mais maintenance complexe en cas d'évolution
  - Fonctions interdépendantes
  - Beaucoup de lignes de code

# De la programmation procédurale à la programmation modulaire (2/3)

## Programmation modulaire

- Module : Centraliser (données et traitements) autour d'un type
- Besoin de la notion
  - d'objets
  - de classes
  - de polymorphisme
  - ...

# De la programmation procédurale à la programmation modulaire (3/3)

- **Récapitulons !**

- Approche procédurale : « Que doit faire mon programme ? »
- Approche modulaire : « De quoi est composé mon programme ? »

# Introduction à la programmation Orientée Objet (POO) : Notions de base et terminologie (1/2)

- La programmation orientée objet est axée :
  1. sur les données / encapsulées autour d'un concept appelé : « objet »
  2. sur les traitements qui y sont associés
  - La complexité des programmes est alors répartie sur chaque « objet »
  - Les langages objets (C++, java, ...) facilitent grandement cette approche de programmation

# Introduction à la programmation Orientée Objet (POO) : Notions de base et terminologie (2/2)

Le modèle objet est utilisé au niveau de l'analyse, la conception et le codage

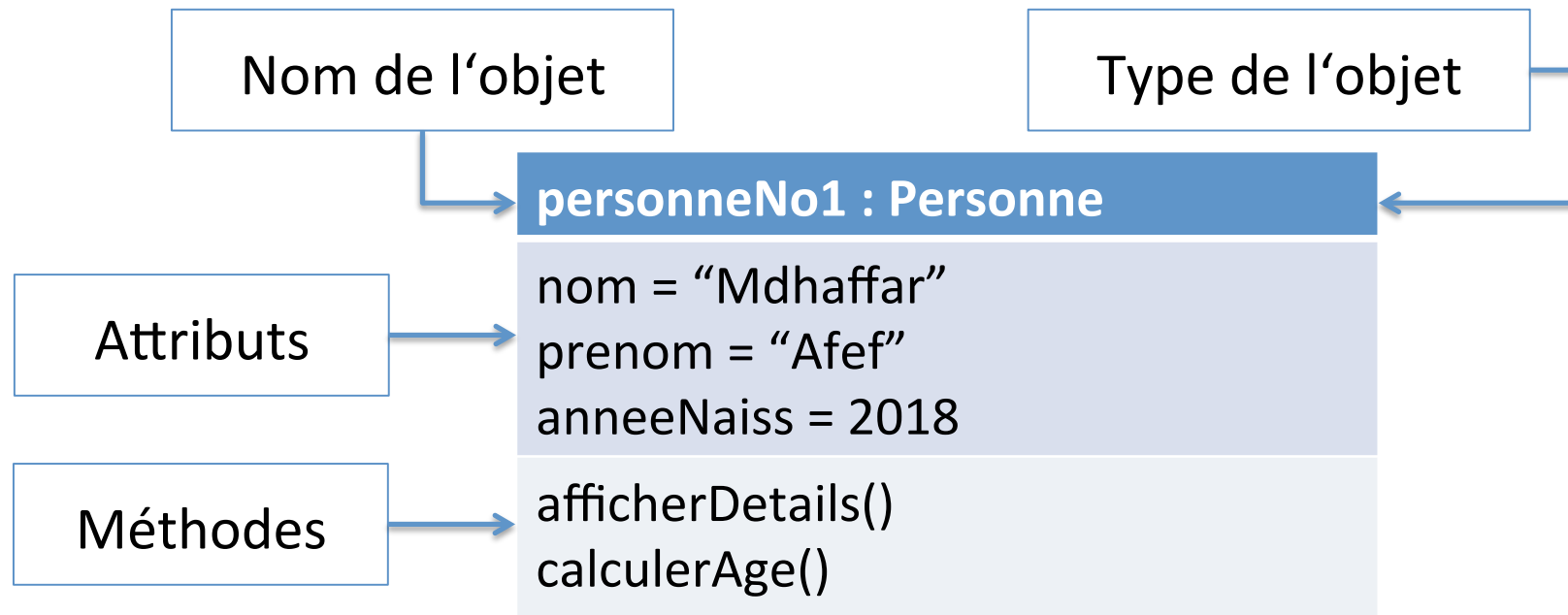
- cohérence pendant le développement
- Facilite la réalisation de programme : développement et exploitation de gros logiciels

## Principes de base

- Abstraction
- Encapsulation
- Héritage
- Polymorphisme
- ...

# Concept Objet : la notion d'Objet

- Les objets / Encapsulation
  - Un objet est constitué
    - d'une collection d'attributs (définissant l'état de l'objet)
    - et de méthodes associées (définissant son comportement)



# Concept Objet : la notion de **Classe**

Les classes / Abstraction de données :

*« Une classe est un prototype qui définit la nature des méthodes et des d'attributs communs à tous les objets d'un certain type. »*

## Personne

nom  
prenom  
anneeNaiss

afficherDetails()  
calculerAge()



# Classe et Objet

- La classe est le **patron** d'objet
- Un objet est **l'instance** d'une classe
- Une classe permet par instanciation de construire des objets

**personneNo1 : Personne**

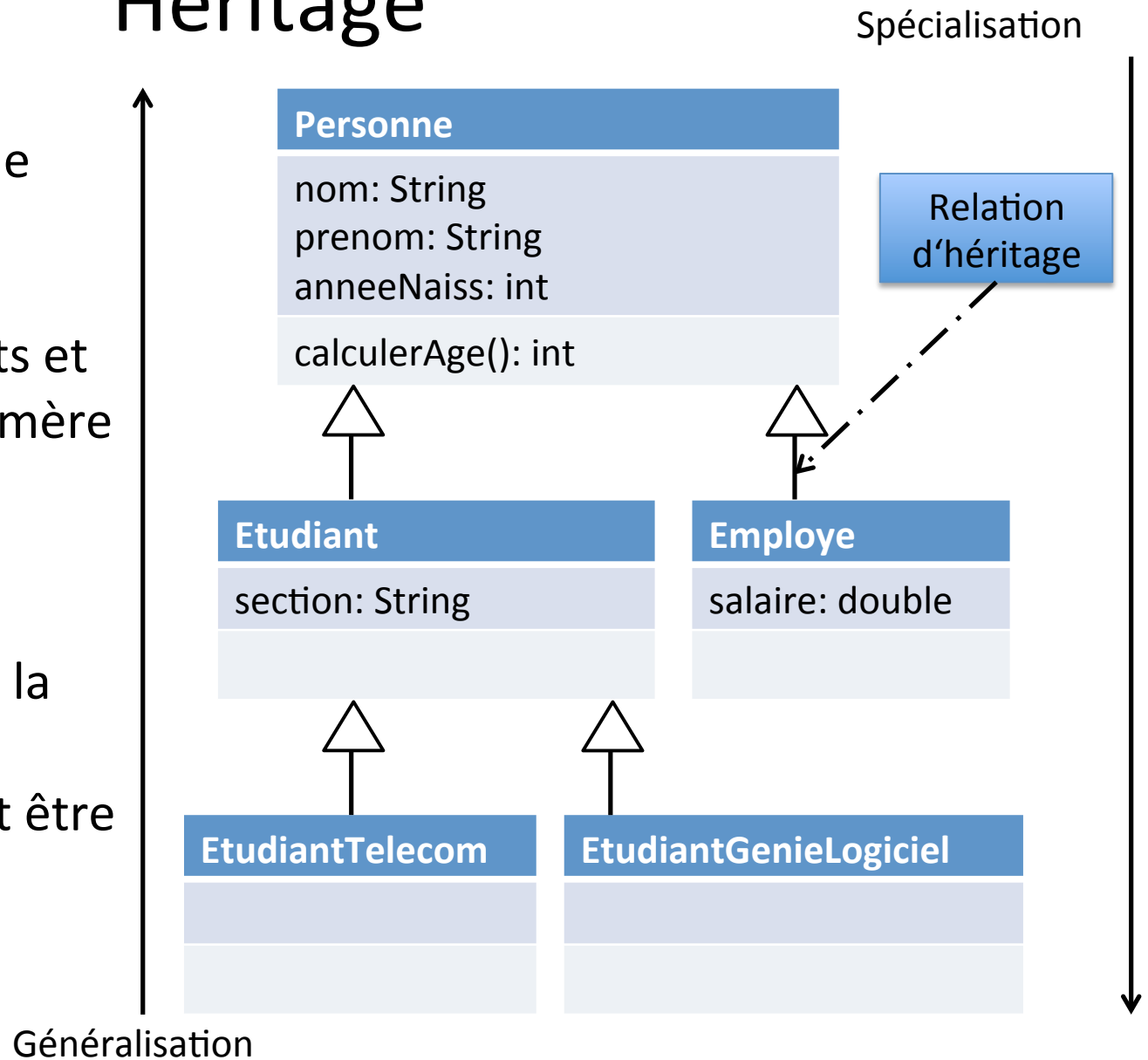
nom = "Mdhaftar"  
prenom = "Afef"  
anneeNaiss = 2018

afficherDetails()  
calculerAge()

« L'objet **personneNo1** est une instance de la classe **Personne** »

# Héritage

- Héritage = Une classe fille (dérivée ou sous-classe) hérite toutes les caractéristiques (attributs et méthodes) d'une classe mère (super-classe)
- Une classe dérivée peut ajouter ou modifier les attributs et méthodes de la classe mère
- Une classe (dérivée) peut être spécialisée à partir d'une autre classe (mère)



# Héritage et polymorphisme

Polymorphisme : mécanisme qui autorise l'appel d'une « **même** » méthode **redéfinie** sur **différents objets** et provoque des **actions différentes** selon la **nature de l'objet**.

```
for (int i=0; i<nbPersonne; i++){  
    personnes[i].afficherDetails();  
}
```

En Java

## Personne

nom: String  
prenom: String  
anneeNaiss: int

calculerAge(): int  
**afficherDetails(): void**

## Etudiant

section: String

**afficherDetails(): void**

```
afficherDetails(){  
    //affiche nom, prenom,  
    //anneeNaiss et section  
}
```

## Employe

salaire: double

**afficherDetails(): void**

```
afficherDetails(){  
    //affiche nom, prenom,  
    //anneeNaiss et salaire  
}
```

# Récapitulons !

- Les concepts fondateurs de la POO sont :
  - La notion d'objet et de classe
  - L'encapsulation
  - L'héritage
  - ... etc
- Les avantages de la POO :
  - Élaboration et évolution d'applications complexes
  - Réutilisation du code

# Langages de POO existants

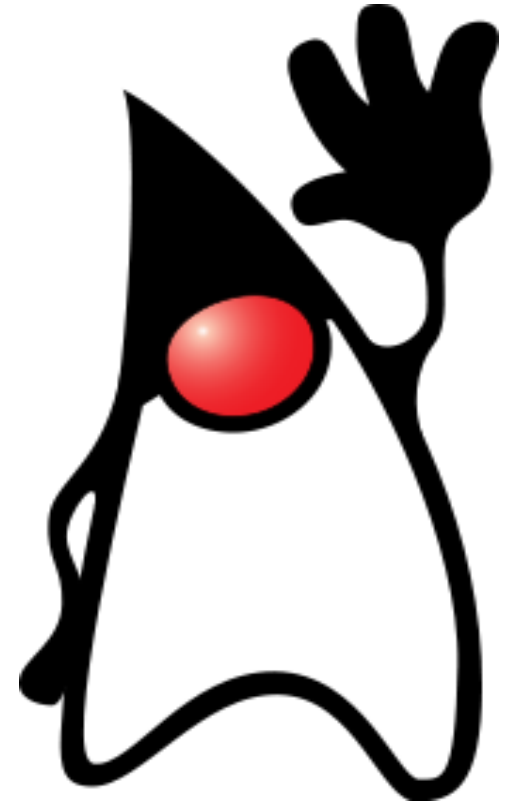
- Java
- C++
- C#
- Objective C
- Eiffel
- SmallTalk
- Scala
- ...

# C++

- Extension du langage C
- Développé chez Bell Laboratory (ATT) par Bjarne Stroustrup en 1983
- A conquis les marchés des langages orientés objet
- (++) Langage très performant
- (- -) Mais syntaxe assez difficile

# Java

- Un langage de programmation orienté objet créé par Sun Microsystems
- Concurrent plus récent de C++
- Présenté officiellement le 23 Mai 1995 au SunWorld
- Java est multiplateforme
- Java est sous Licence libre depuis 13 Novembre 2006
- En 2009, Oracle a acheté Sun Microsystems : le logo Oracle apparaît dans la documentation Java
- Avec JAVA, on peut développer :
  - des applications sous forme de fenêtres ou de console
  - des applications pour appareils mobiles
  - des applets
  - et bien d'autres !



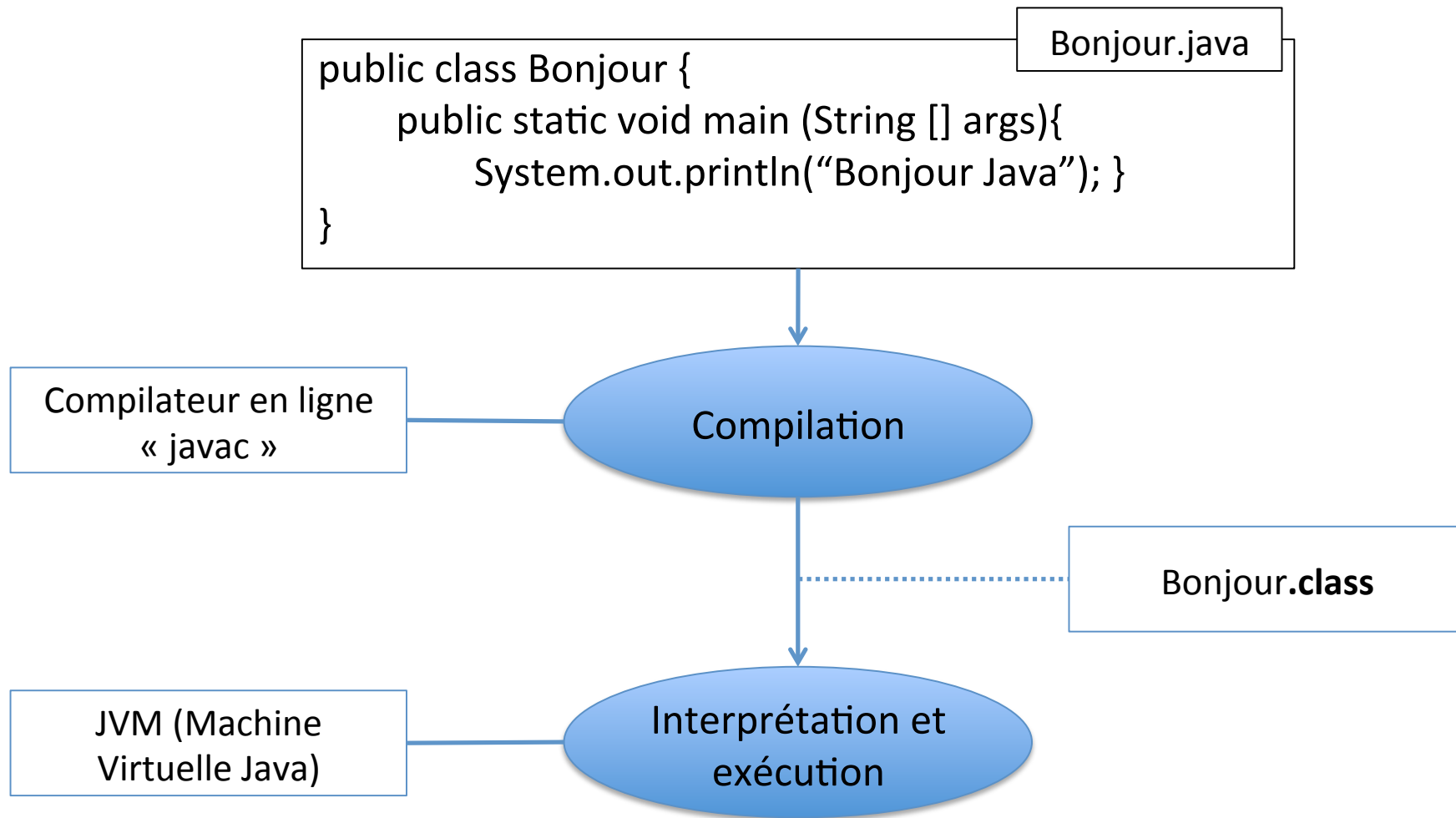
Nous allons utiliser Java comme langage de programmation dans le cadre de ce module !



# JAVA (1/3)

- Java est un langage hybride : compilé et interprété à la fois
- Compilé : le code source est soumis à un compilateur pour générer un fichier binaire compréhensible par un micro-processeur (fichier exe)
- Interprété : le code source est directement interprété sans compilation. C'est l'interprète qui exécute le code source. Il l'interprète à la volée (JVM)

# JAVA (2/3)



# JAVA (3/3)

- L'interpréteur Java s'appelle machine virtuelle Java (JVM : Java Virtual Machine)
- Une machine virtuelle est un ordinateur fictif (virtuel) qui tourne sur un ordinateur réel :
  - Possède un langage natif propre et traduit un programme écrit dans ce langage vers le langage natif de l'ordinateur
  - Définit les spécifications hardware de la plateforme
  - Lit le bytecode compilé (indépendant de la plateforme)

# Version (Release) de Java

- JDK1.0 (January 23, 1996)
- JDK1.1 (February 19, 1997)
- J2SE1.2 (December 8, 1998)
- J2SE 1.3 (May 8, 2000)
- J2SE1.4 (February 6, 2002)
- J2EE 5.0 (September 30, 2004)
- Java SE 6 (December 11, 2006)
- Java SE 7 (July 28, 2011)
- Java SE 8 (March 18, 2014)
- Java SE 9 (September 21, 2017)
- Java SE 10, AKA SE 18.3 (2018)

# Identificateurs en JAVA

- Nommer les classes, les attributs, les méthodes, ...
- Un identificateur Java
  - est de longueur quelconque
  - commence par une lettre Unicode
  - peut ensuite contenir des lettres ou des chiffres ou le tiret bas « \_ »
  - ne doit pas être un mot réservé du langage (mot clé : if, for, true, false, ...)

# Les règles de nommage

- Classe:
  - 1<sup>ère</sup> lettre en majuscule
  - Mélange de minuscule, majuscule avec la première lettre de chaque mot en majuscule
  - Donner des noms simples et descriptifs
- Packages
  - Tout en minuscule
  - Utiliser seulement [a-z], [0-9] et le point « . »
  - **Ne pas utiliser** le tiret « - », le tiret bas (underscore) « \_ », l'espace, ou d'autres caractères (\$, \*, accents, ...).
- Constante
  - Les constantes sont en majuscules
  - Les mots sont séparés par le tiret bas « \_ »
  - Exemple : UNE\_CONSTANTE

# IDE, SDK, JDK, JRE (1/3)

- On développe un programme JAVA en utilisant un IDE (Integrated Development environment)
- Pour créer une application JAVA consistante, on a besoin du JDK (Java Development Kit)
- Pour tester et lancer un code JAVA, on a besoin d'un JRE (Java Runtime Environment)

## IDE, SDK, JDK, JRE (2/3)

- IDE : Un software qui regroupe un ensemble d'outils (éditeur de texte, compilateur, débogueur, ...) pour le développement de logiciels. Exemple : Eclipse, Netbeans
- SDK (Software Development Kit) : C'est un Kit qui comporte un JRE, un compilateur, de nombreux programmes utiles, des exemples de programmes Java et les sources de toutes les classes de l'API



# IDE, SDK, JDK, JRE (3/3)

- JRE : C'est l'environnement qui permet l'exécution des applications Java. Il comporte une JVM en particulier
- JDK : Plusieurs outils permettant de :
  - Développer, compiler (**javac**), déboguer (**jdb**) et exécuter un programme Java
  - Créer des archives **jar** (outils d'archivage)
  - Générer de la documentation (**javadoc**)

# Environnement de développement intégré (EDI / IDE)

- Exemple: Eclipse, Netbeans, Jcreator,...
- La compilation se fait implicitement lorsque il y a des changements dans les fichiers sources
- L'interprétation se lance graphiquement avec le bouton « Run ».
- Le résultat apparaît dans une console intégrée à l'EDI

# Bilan

- Avantages de la POO par rapport à la programmation procédurale
- Notions de base de la POO
- JAVA
- Plus de détails dans les chapitres suivants !

# Chapitre 2 : Classes et Objets

# Objectifs

- Maîtriser les notions Objet / Classe de la POO, à travers un exemple
- Maîtriser les notions suivantes :
  - Constructeur / constructeur surchargé
  - Méthode surchargée
  - Mot clé « this »
  - Classe principale
  - Règles de nommage (Java)

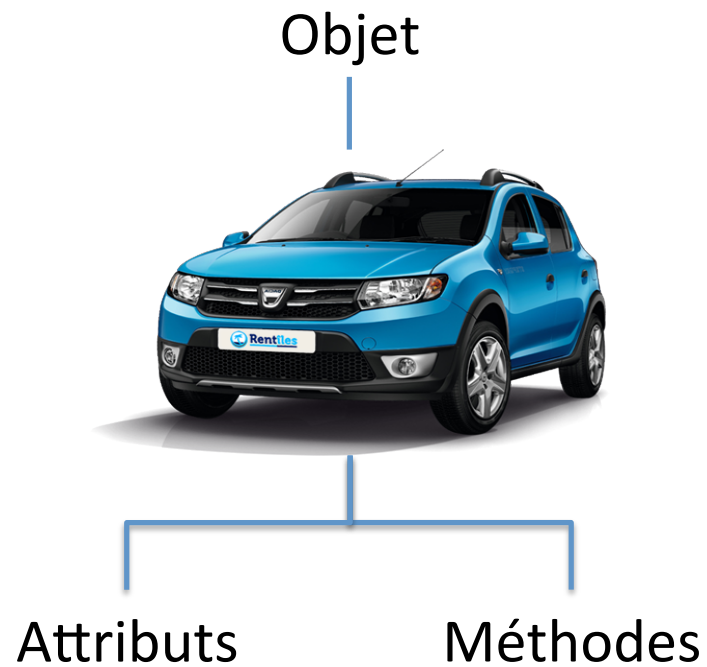
# Plan


- Analyse d'un problème avec l'approche Orientée Objet
  - Identification de l'objet
  - Classe et Objet Java
  - Déclaration d'une Classe
  - Déclaration des attributs
- Types de données en Java
- Conversion des types primitifs
  - La méthode toString()
- Tableaux
- Déclaration des méthodes
- Classe Voiture
  - Création des objets
  - Référence d'un objet
- Notion de référence
- Les constructeurs
- Le mot clé « this »
- Les attributs static
- Les méthodes static
- Classe principale
- Passage de paramètres
- Surcharge
- Exemple
- Associations

# Analyse d'un problème avec l'approche OO

- Développer une application pour une agence de location de voiture
- Chaque voiture est caractérisée d' :
  - Un identifiant : matricule
  - Une couleur: noir, vert, bleu, rouge
  - Une marque : Seat, VolksWagen, Renault
  - Un prix de location / jour
  - Une description : type du carburant, nombre de portes, etc ...
  - nombre de jours de location
- Il est possible de :
  - Augmenter ou de diminuer le nombre de jours de location
  - Calculer le prix total de location

# Identification de l'objet



Voiture	
id prix couleur description nbJourLocation	
augmenterNbJours() diminuerNbJours() calculerPrixLocation()	



# Classe et Objet Java

- Une classe n'est pas un objet
- Une classe est un **patron** d'objet.



Classe « Voiture »

Objet 1

Id: 134Tunisie1999

Prix : 100 DT

Couleur : Move

NbreJL : 14



Objet 2

Id: 157Tunisie1897

Prix : 200 DT

Couleur : Gris

NbreJL : 10



# Classe et Objet Java

- Classe :
  - structure d'un objet
  - la déclaration de l'ensemble des entités qui définissent un objet
- Un objet est une instantiation d'une classe objet = instance
- Une classe est composée de deux parties :
  - Les attributs (appelés aussi données membres) : il s'agit des données représentant l'état de l'objet
  - Les méthodes (appelées aussi fonctions membres) : il s'agit des opérations applicables aux objets

# Déclaration d'une classe

```
public class Voiture {  
    /*Déclaration des attributs*/  
    /*Déclaration des méthodes*/  
  
    //commentaire sur une seule ligne  
  
    /*commentaires sur  
    plusieurs lignes*/  
}
```

Le nom de la classe doit commencer par une **MAJISCULE**  
Exemple : **C**ompteBancaire, **A**genceVoyage

# Déclaration des attributs

## Syntaxe :

```
type nom_attribut [=value];
```

```
int id = 0;
```

Le nom de l'attribut doit commencer par une lettre  
**miniscule**

Exemple: **age**, **quantiteStock**

# Types de données en Java

- Deux grands groupes de types de données :
  - types **primitifs**
  - types **objets** (instances de classe)
- Les types de données utilisés sont :
  - les nombres entiers
  - les nombres réels
  - les caractères et chaînes de caractères
  - les booléens
  - les objets

# Types de données en Java

Déclaration du type primitif → réservation de l'espace mémoire

```
int nbJourLocation;  
boolean disponible;  
char couleur;
```

nbJourLocation



disponible



couleur



La valeur affectée sera stockée dans l'espace mémoire réservé

```
nbJourLocation = 12;  
disponible = false;  
couleur = 'R';
```

nbJourLocation



disponible



couleur



# Types de données en Java

- Types primitifs
  - Valeur logique
    - boolean (true/false)
  - Nombres entiers
    - byte (1 octet)
    - short (2 octets)
    - int (4 octets)
    - long (8 octets)
  - types réels
    - float (4 octets)
    - double (8 octets).
  - Caractère (un seul)
    - char (2 octets)

```
boolean disponible = true
```

```
float prix=2.5f  
double poids=45.9d
```

```
char sexe='F'
```

# Types de données en Java

Type primitif	Espace mémoire	Signification
byte	1 octet	$-128 < \text{Entier très court} < +127$
short	2 octets	$-32768 < \text{Entier} < +32767$
int	4 octets	$-2147483648 < \text{Entier} < +2147483647$
long	8 octets	$-2^{63} < \text{Entier long} < +2^{63} - 1$
float	4 octets	$-1.4 * 10^{-45} < \text{Nombre réel} < +3.4 * 10^{38}$
double	8 octets	$4.9 * 10^{-324} < \text{Nombre réel double précision} < +1.7 * 10^{308}$
char	2 octets	Caractère unicode (65536 caractères possibles)
boolean	1 octet	Variable booléenne (valeurs : true ou false)



# Types de données en Java

## Types primitifs : initialisation par défaut

Type	Valeur par défaut
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false

# Types de données en Java

## Déclaration des constantes

- On déclare une constante avec le mot **final**
- Le nom de la constante doit être en **MAJUSCULE**
- Si le nom est composé de plusieurs mots, on utilise le tiret bas « \_ » pour la séparation des mots
- Exemples :

```
final int TAILLE;  
final int MAX_STOCK;  
final int CAPACITE_DANS_STOCK;
```

# Types de données en Java

## Types objets

- Ce type inclut :
  - les instances de classe
  - les tableaux (« arrays »)
- Les objets sont créés dynamiquement en utilisant l'opérateur « `new` ».
- Exemples : String, Voiture, Etudiant, ...

# Types de données en Java

## Les références en Java : String

- Création de la « référence » à un String
- Déclaration de type objet : Réservation de l'espace mémoire pour la référence

String msg;



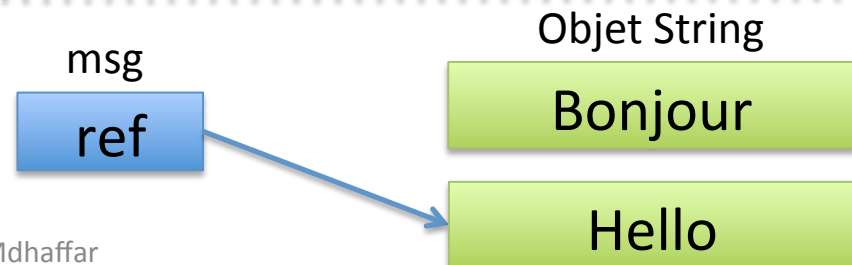
msg = "Bonjour";

//équivalent à

Msg = new String ("Bonjour");



msg = "Hello";



# Types de données en Java

## Types objets Enveloppes : Wrappers

- Un type enveloppe (Wrapper) : Un type objet qui encapsule un type primitif
- A chaque type primitif est associé une classe enveloppe

Type Primitif	Classe Enveloppe (Wrapper)
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character
boolean	java.lang.Boolean

# Types de données en Java

## Types objets Enveloppes : Wrappers

- Chaque classe enveloppe (Wrapper) possède une méthode permettant de convertir une instance de la classe enveloppe en une valeur du type primitif associé : **primitifValue()**

**Integer entier= new Integer (25) ;**

**int age = entier. intValue ( ) ;**

- Généralement, les objets enveloppes permettent notamment de convertir une chaîne de caractères en un entier, un réel, un booléen, etc ...

**int quantite=Integer.parseInt( "6" ) ;**

**double prix=Double.parseDouble( "6 .89" ) ;**

**boolean test= Boolean.parseBoolean( "false" ) ;**

# Conversion des types primitifs

- Conversion **implicite** (automatique)
  - byte -> short -> int -> long -> float -> double

– Exemple :

*// 64 bit long integer*

*Long myLongInteger;*

*// 32 bit standard integer*

*int myInteger;*

*myLongInteger = myInteger;*

- Conversion **explicite** : Casting

*myInteger = (int) myLongInteger;*

- Conversion d'un objet en String : méthode **toString()**

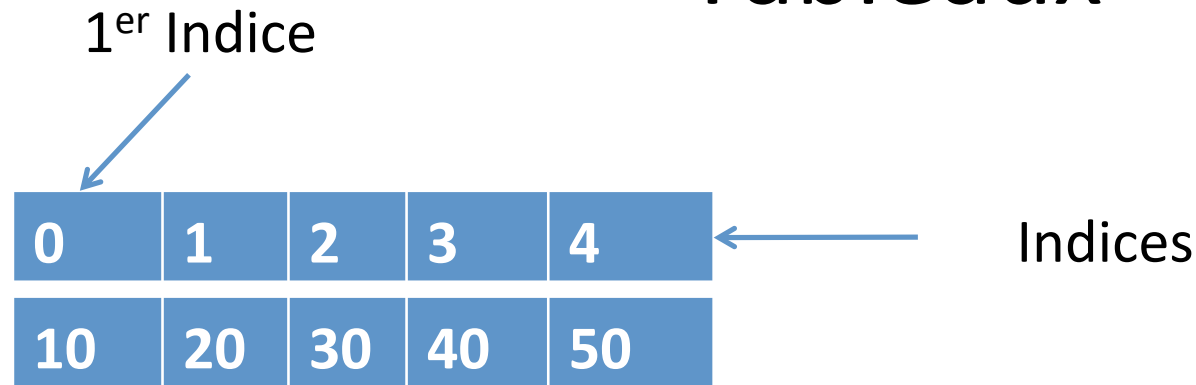
# La méthode `toString()`

- La méthode `toString()` de la classe `Object` renvoie le nom de la classe de l'objet concerné suivi de l'adresse de cet objet
- Quand la méthode `toString()` est redéfinie, elle renvoie une chaîne de caractères servant à décrire l'objet concerné

```
package ch1;
/**
 * @author mdhaffar
 */
public class Personne {
    String nom;
    String prenom;
    int age;
    public String toString(){
        return prenom+
            " "+nom+" a "+age+" ans.";
    }
}
```



# Tableaux

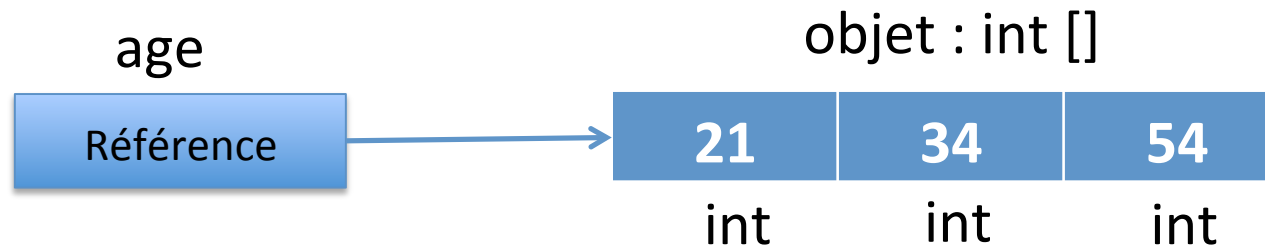


tab.length = 5

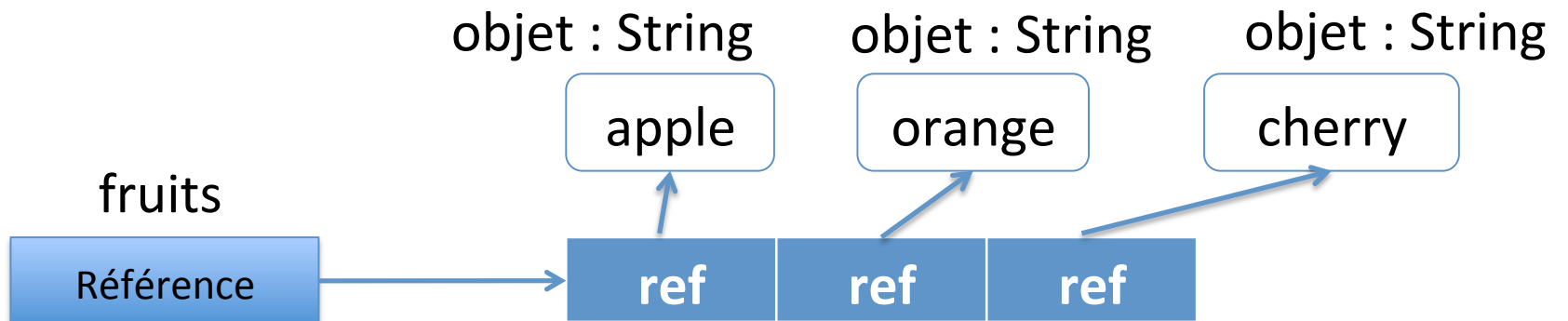
```
int[] tab;  
tab = new int[5];  
tab[0] = 10; // initialiser le premier élément  
tab[1] = 20; // initialiser le second élément  
int[] tab = {10, 20, 30, 40, 50}; //déclaration + initialisation
```

# Tableaux

- Un tableau qui contient des éléments de type primitif  
**int[] age={21,34,54};**



- Un tableau qui contient des éléments de type objet  
**String[] fruits={"apple","orange","cherry"};**



# Déclaration des méthodes

Syntaxe :

```
Type_retour nom_methode([arguments]) {  
}  
void afficher(){  
}
```

Le nom de la méthode doit commencer par un verbe

# Exemple : Classe Voiture

```
public class Voiture {
```

```
    int id;  
    char couleur;  
    float prix;  
    String description;  
    int nbJourLocation;
```

**Attributs**

```
    void augmenterNbJours(int nbr){
```

```
        nbJourLocation +=nbr;}
```

```
    void diminuerNbJours(int nbr){
```

```
        nbJourLocation -=nbr;}
```

```
    void afficher(){
```

```
        System.out.println("voiture "+id+" de couleur "+couleur+  
        ", dont le prix de location par jour est "+prix);
```

```
    }}
```

**Méthodes**

# Création des objets

Voiture

**Voiture**

id  
prix  
couleur  
description  
nbJourLocation



augmenterNbJours()  
diminuerNbJours()  
calculerPrixLocation()

Référence 1

Objet 1 : voitureNo1

Id: 134Tunisie1999

Prix : 100 DT

Couleur : Move

NbreJL : 14



Référence 2

Objet 2 : voitureNo2

Id: 157Tunisie1897

Prix : 200 DT

Couleur : Gris

NbreJL : 10

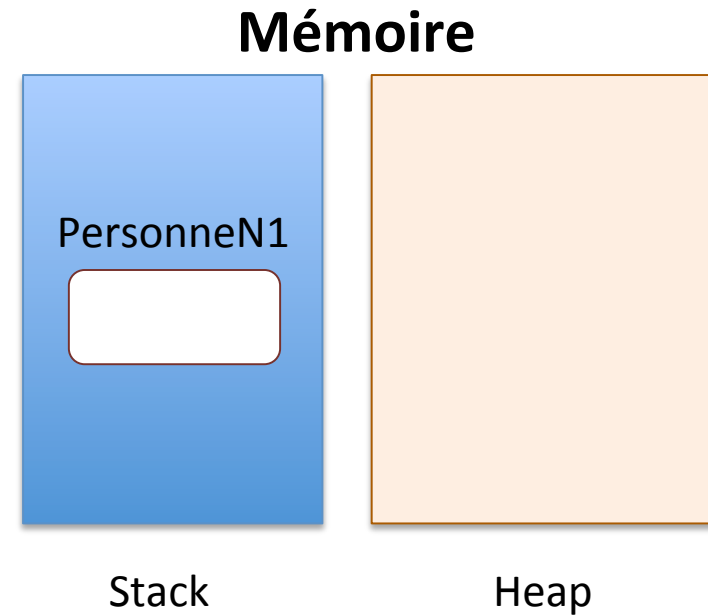


# Référence d'un objet Vs. Adresse d'une maison

- Une référence peut être vue comme une « **adresse** »
- Une adresse nous permet de trouver la maison  
→ **Une référence nous permet de trouver l'objet**
- En utilisant une adresse, on peut envoyer une lettre à la maison  
→ **Une référence nous permet d'accéder aux attributs et méthodes d'un objet**

# Référence

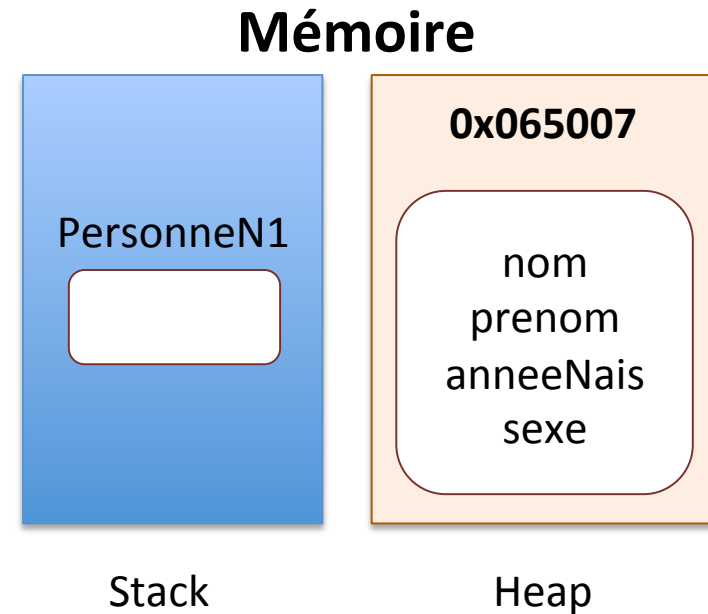
```
public class TestRef{  
    public static void main(String[] args) {  
        Personne personneN1;  
        personneN1=new Personne();  
        personneN1.sexe='F';  
    }  
}
```



Création d'une variable `personneN1` de type `Personne`

# Notion de référence

```
public class TestRef{  
    public static void main(String[] args) {  
        Personne personneN1;  
        personneN1 = new Personne();  
        personneN1.sexe='F';  
    }  
}
```

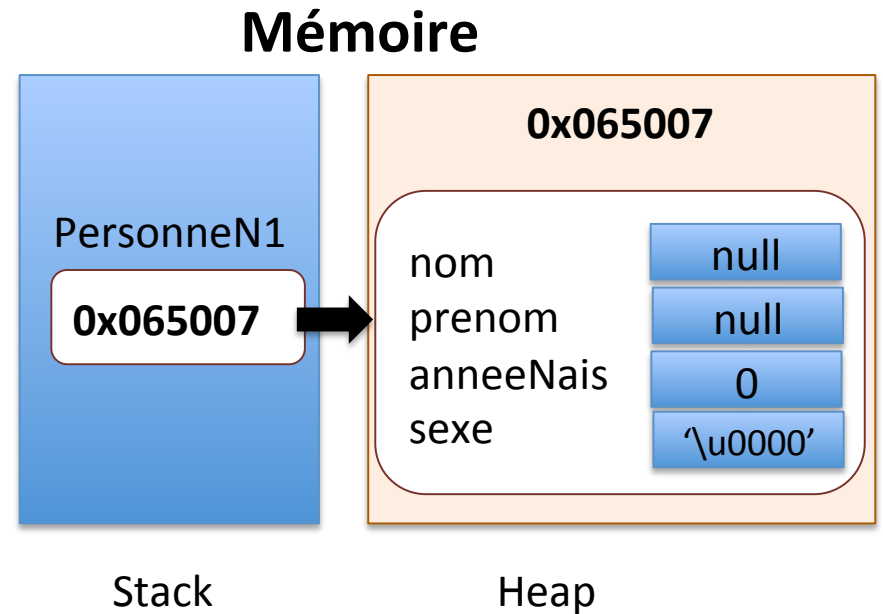


- Instanciation de la classe `Personne` → Création d'un objet
- Cet objet possède une adresse de son emplacement dans la mémoire (**0x065007**)



# Notion de référence

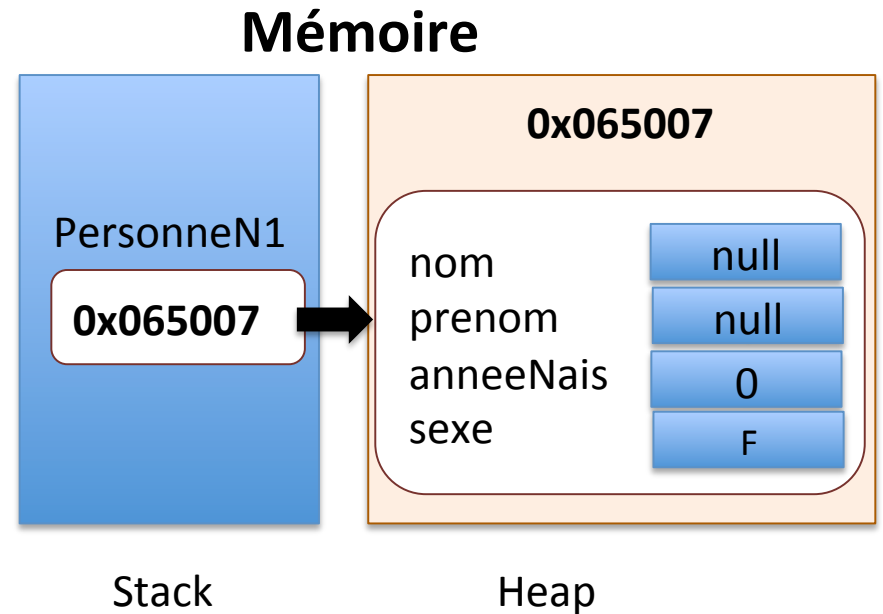
```
public class TestRef{  
    public static void main(String[] args) {  
        Personne personneN1;  
        personneN1=new Personne();  
        personneN1.sexe='F';  
    }  
}
```



Lier l'objet créé et la variable **personneN1**  
→ **personneN1** est la référence de l'objet créé

# Notion de référence

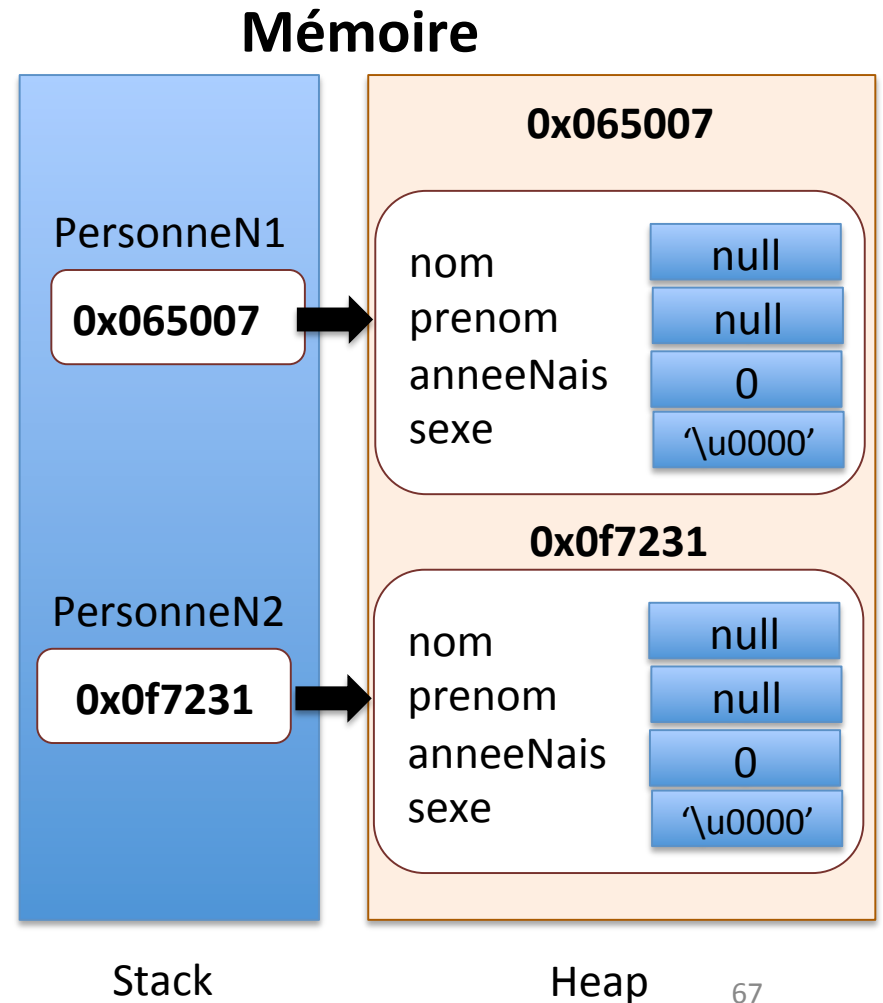
```
public class TestRef{  
    public static void main(String[] args) {  
        Personne personneN1;  
        personneN1=new Personne();  
        personneN1.sexe='F';  
    }  
}
```



En utilisant la référence **personneN1**, on peut accéder aux attributs de l'objet

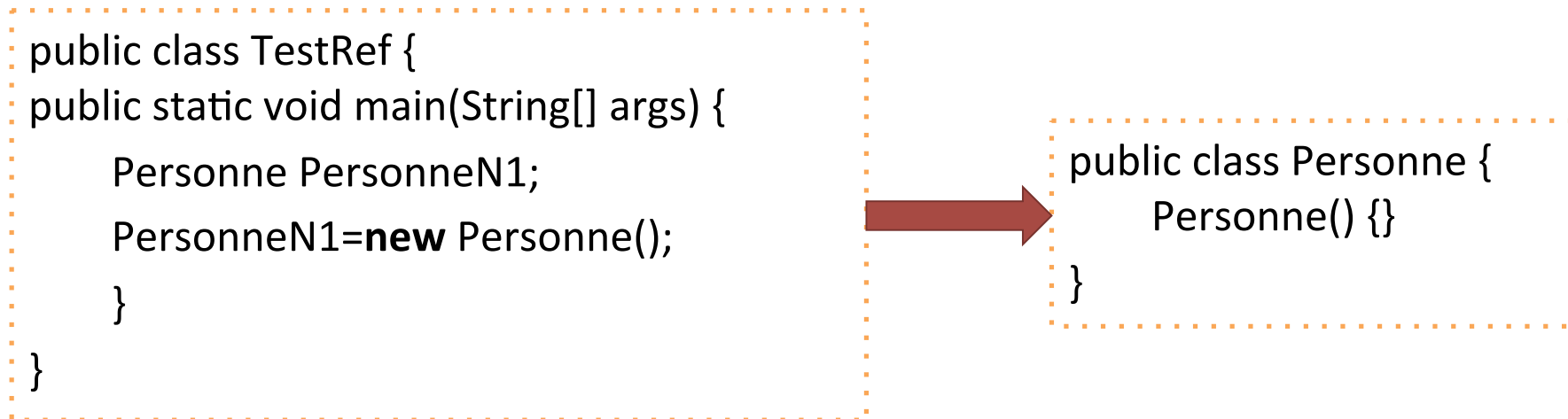
# Référence

```
public class TestRef{  
    public static void main(String[] args) {  
        Personne PersonneN1=new Personne();  
        Personne PersonneN2=new Personne ();  
    }  
}
```



# Les constructeurs

Pour créer un objet à partir d'une classe, on utilise l'opérateur **new**.



- L'opérateur « new » fait appel au constructeur de la classe
- Un constructeur porte le même nom que la classe dans laquelle il est défini
- Un constructeur n'a pas de type de retour (même pas void)

# Les constructeurs

- **Constructeur par défaut : `Personne(){}`** 
  - Il initialise les attributs de la classe aux valeurs par défaut
  - Il initialise les attributs de la classe à des valeurs données :

```
Personne(){  
    nom="Mdhaïffar";  
    prenom = "Afef";  
    sexe='F';  
    anneeNais=1999; }
```

- **Constructeur surchargé :**

```
Personne(String nom, String prenom, char sexe, int anneeNais){  
    this.nom=nom;  
    this.prenom=prenom;  
    this.sexe=sexe;  
    this.anneeNais=anneeNais}
```

# Les constructeurs

- Absence de constructeur dans la classe → le compilateur crée automatiquement un constructeur par défaut implicite
- Si le constructeur surchargé est créé, le constructeur par défaut implicite ne sera plus créé par le compilateur
- La plateforme java différencie entre les différents constructeurs déclarés au sein d'une même classe en se basant sur le nombre d'arguments (input) et leurs types.
- On ne peut pas créer deux constructeurs ayant le même nombre et types des arguments d'entrée.

```
Personne(String nom){  
    this.nom = nom+nom;}  
Personne(String nom{  
    this.nom = nom;}
```

 **ERREUR DE COMPILATION !**

# Les constructeurs

Quel constructeur va choisir Java lors de création de l'objet ?

```
public class Personne {
```

```
    String nom;
```

```
    String prenom;
```

```
    int anneeNais;
```

```
    char sexe;
```

```
    Personne(){} 
```

```
    Personne(String nom){
```

```
        this.nom = nom;
```

```
    Personne(String nom, String prenom, int
```

```
    anneeNais, char sexe){
```

```
        this.nom = nom;
```

```
        this.prenom = prenom;
```

```
        this.anneeNais = anneeNais;
```

```
        this.sexe = sexe;}}
```

Personne p1 = new Personne();

Personne p1 = new Personne("Mdhaffar");

Personne p1 = new Personne("Mdhaffar",  
"Afef", 2001, 'F');

# Le mot clé « this »

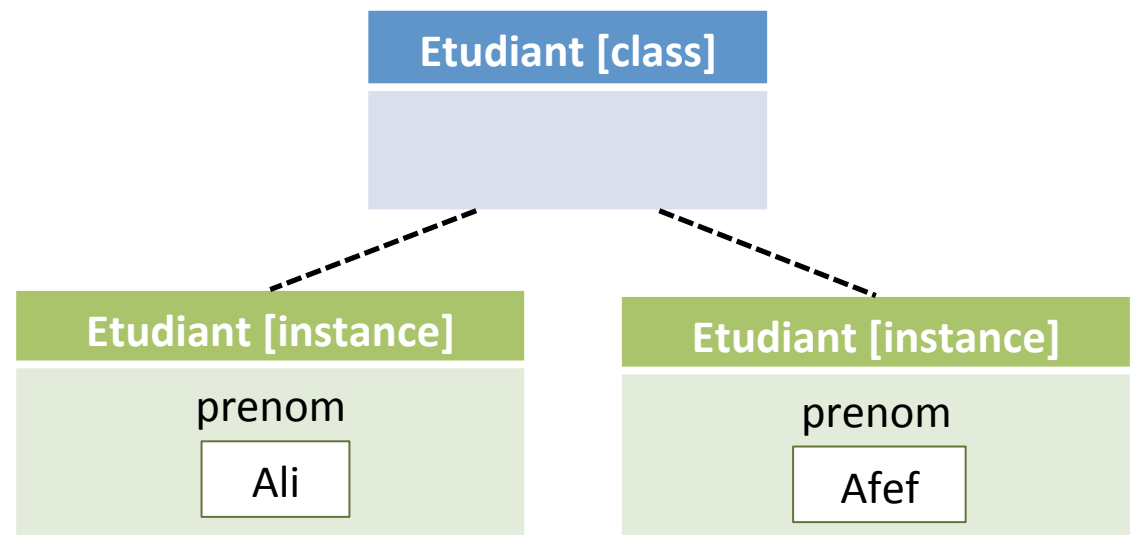
- Le mot clé « this » permet de désigner l'objet courant
- « **this** » permet d'accéder aux attributs et méthodes de l'objet courant
  - Pour manipuler un attribut de l'objet courant :  
**this.nom**
  - Pour manipuler une méthode de l'objet courant :  
**this.toString ()**
  - Pour faire appel au constructeur de l'objet courant :  
**this()**



# Les attributs

**Attribut d'instance** : Chaque instance de la classe possède ses propres valeurs d'attributs

```
Class Etudiant{  
String nom;  
Etudiant(String prenom){  
this.prenom=prenom; }  
}
```



```
Etudiant e1 = new Etudiant ("Ali");  
Etudiant e2 = new Etudiant ("Afef");
```

# Les attributs

## Attribut d'instance :

Utilisation : Les attributs **d'instance** sont appelés avec le **nom de l'instance**

```
Class Etudiant{  
String nom;  
Etudiant(String prenom){  
this.prenom=prenom; }  
}
```

```
class TestEtud{  
public static void main(String[] args){  
Etudiant e1=new Etudiant("Ali");  
System.out.println(e1.prenom);  
} }
```

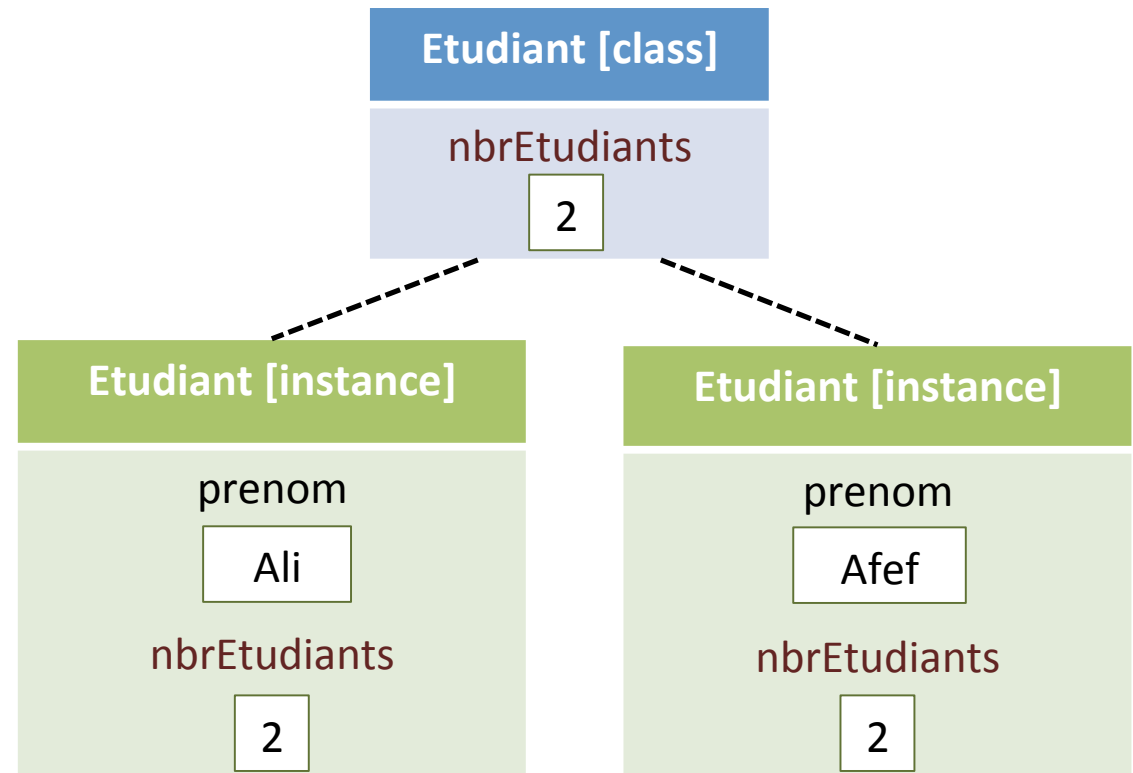
# Les attributs

## Attribut de classe :

- n'appartient pas à une instance particulière, il appartient à la classe
- est partagé par toutes les instances de la classe

```
Class Etudiant{ String nom;  
static int nbrEtudiants;  
Etudiant(String prenom){  
    this.prenom=prenom;  
    nbrEtudiants++;  
}}
```

```
Etudiant e1 = new Etudiant ("Ali");  
Etudiant e2 = new Etudiant ("Afef");
```



« `nbrEtudiants` » est le même pour toutes les instances de la classe

# Les attributs

## Attribut de classe :

→ Utilisation : Les attributs static sont appelés avec le **nom de la classe**

```
Class Etudiant{ String nom;  
static int nbrEtudiants;  
Etudiant(String prenom){  
this.prenom=prenom;  
nbrEtudiants++;  
}}
```

```
class TestEtud{  
public static void main(String[] args){  
    System.out.println(Etudiant.nbrEtudiants);  
}}
```

# Les méthodes statiques (static)

Le comportement d'une méthode statique ne dépend pas de la valeur des attributs d'instance

```
public class MaCalcullette {  
    static int max(int a, int b){  
        if(a < b)  
            return b;  
        else  
            return a;}  
}
```

L'appel de la méthode  
statique se fait avec le  
**nom de la classe**

```
public class TestCalcullette {  
    public static void main(String[] args) {  
        int x = MaCalcullette.max(3, 5);}  
}
```

# Les méthodes statiques (static) VS. Les méthodes d'instances

Méthodes statiques	Méthodes d'instances
<b>N'</b> accèdent <b>pas</b> aux attributs d'instance et méthodes d'instance	Accèdent aux attributs d'instance et méthodes d'instance
Accèdent aux attributs de classe (static) et méthodes de classe (static)	Accèdent aux attributs de classe (static) et méthodes de classe (static)



Dans une méthode statique, on ne peut pas utiliser « **this** »

# Classe principale

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

# Passage de paramètres

Types primitifs	Types objets
passage de paramètres par valeur	passage de paramètres par adresse
La transmission des arguments de type simple est uniquement par valeur: les arguments ne peuvent pas être modifiés par la méthode appelée	Le passage des objets en paramètre d'une méthode est par adresse: si la méthode modifie l'objet, il sera donc toujours modifié après l'appel à cette méthode



# Passage de paramètres

```
public class Rectangle {  
    int longueur;  
    int largeur;  
    void changInt(int ent){ent = 22;}  
    void changeObj(Rectangle r){r.longueur=40;}}
```

```
public class TestRectangle {  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle();  
        //essayer de modifier un entier  
        int ent = 11;  
        r.changInt(ent); //passage de paramètre par valeur  
        System.out.println("valeur entière : " +ent); // affiche ??  
        //essayer de modifier le contenu d'un objet  
        r.longueur=25;  
        r.changeObj(r); //passage de paramètre par adresse  
        System.out.println("longueur de l'objet : "+r.longueur); // affiche ??  
    }  
}
```

# Surcharge

- Une classe peut contenir des méthodes de **même nom**, avec des **arguments différents**  
→ On dit que la méthode est **surchargée**

```
public class Affiche {  
    void afficher(int i){System.out.println("l'entier est : "+i);}  
    void afficher(String s){System.out.println("la chaine de caractère est : "+s);}  
    void afficher(float f){System.out.println("le réel est : "+f);}  
}
```

```
public class TestAffiche {  
    public static void main(String[] args) {  
        Affiche a = new Affiche();  
        a.afficher(2); // affiche ??  
        a.afficher("Bonjour G11"); // affiche ??  
        a.afficher(2.4f); // affiche ??  
    }  
}
```

# Exemple 1 (1/2)

```
public class Rectangle {  
    int longueur;  
    int largeur;  
    Rectangle(){  
        longueur = 0;  
        largeur = 0;}  
    void allongerRect(int lg){  
        longueur+=lg;}  
    void augmenterRect(){  
        allongerRect(2);}  
    void affiche(){  
        System.out.println("longueur = "+longueur+", largeur = "+largeur);  
    }  
    int getLongueur(){  
        return longueur;}  
    int getLargeur(){  
        return largeur;}}
```

# Exemple 1 (2/2)

```
public class TestRectangle {  
    public static void main(String[] args) {  
        Rectangle r1 = new Rectangle(); //Constructeur par défaut, longueur = ?, largeur = ?  
  
        //invocation de la méthode affiche()  
        r1.affiche(); //longueur = ?, largeur = ?  
  
        //accès aux attributs de la classe Rectangle  
        r1.longueur = 20;  
        r1.largeur = 10;  
  
        //invocation des méthodes de la classe Rectangle  
        int larg = r1.getLargeur();  
        System.out.println("largeur = "+larg); // largeur = ?  
        System.out.println("longueur = "+r1.getLongueur()); // longueur = ?  
        r1.augmenterRect();  
        r1.affiche(); // longueur = ?, largeur = ?  
    }  
}
```

## Exemple 2

```
public class Rectangle {  
    static int compte = 0;  
    int id, longueur, largeur;  
    Rectangle(){  
        compte++; id = compte; longueur = 20; largeur = 10;}}
```

```
public class TestRectangle {  
    public static void main(String[] args) {  
        Rectangle r1, r2;  
        r1 = new Rectangle();  
        r2 = new Rectangle();  
        System.out.println("Compte avec le nom de la classe "+Rectangle.compte); // affiche ??  
        System.out.println("Compte avec l'instance r1 "+r1.compte); // affiche ??  
        System.out.println("Compte avec l'instance r2 "+r2.compte); // affiche ??  
        System.out.println("id = "+r1.id); // affiche ??  
        System.out.println("id = "+r2.id); // affiche ??  
    }  
}
```

# Example 3

```
public class MathTool {  
    final static double PI = 3.14;  
    static double getPI(){return PI;}  
    static double diametre(double rayon){return (2*PI*rayon);}  
    static double power(double x){return x*x;}}
```

```
public class TestMathTool {  
    public static void main(String[] args) {  
        // méthode 1  
        double i = MathTool.power(6);  
        System.out.println("i = "+i); //affiche ??  
  
        //méthode 2  
        MathTool MT = new MathTool();  
        double j = MT.power(6);  
        System.out.println("j = "+j); //affiche ??  
    }  
}
```

# Exemple 4

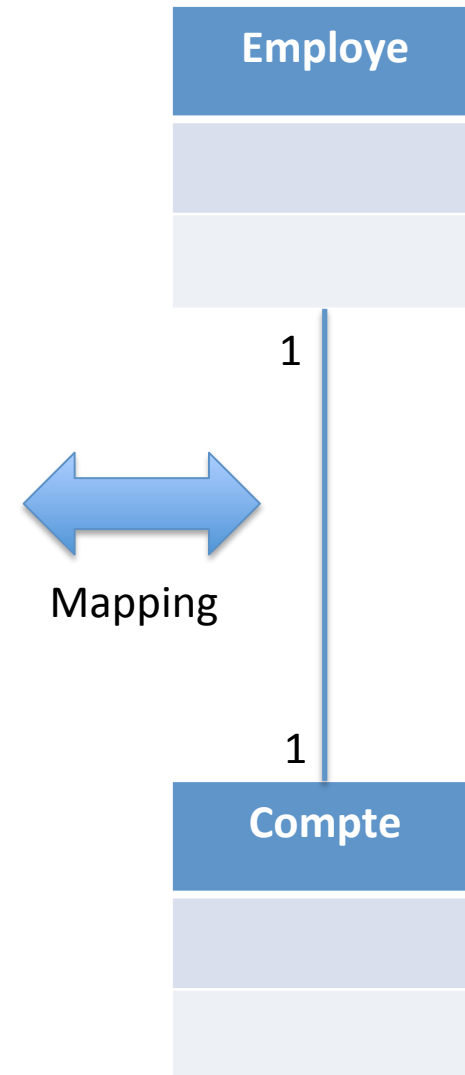
```
public class Rectangle {  
    int longueur, largeur; String nom;  
    Rectangle() { // premier constructeur  
        longueur = 20;  
        largeur = 10;}  
    Rectangle(String nom){ //deuxième constructeur  
        this.nom = nom+nom;}  
    Rectangle(int longueur, int largeur, String nom){ //troisième constructeur  
        this(nom); //appel au deuxième constructeur  
        this.longueur = longueur;  
        this.largeur = largeur;}  
    void affiche(){  
        System.out.println("nom = "+nom+"longueur = "+longueur+", largeur = "+largeur);}}}
```

```
public class TestRectangle {  
    public static void main(String[] args) {  
        Rectangle r1 = new Rectangle(5,3,"REC");  
        r1.affiche(); //affiche ??  
    }  
}
```

# Les associations (1/3)

```
public class Employe {  
    public Compte compte;  
    public void setCompte (Compte compte){  
        this.compte = compte;  
    }  
    public Compte getCompte (){  
        return compte;  
    }  
}
```

```
public class Compte {  
    public Employe employe;  
    public void setEmploye (Employe employe){  
        this.employe = employe;  
    }  
    public Employe getEmploye (){  
        return employe;  
    }  
}
```

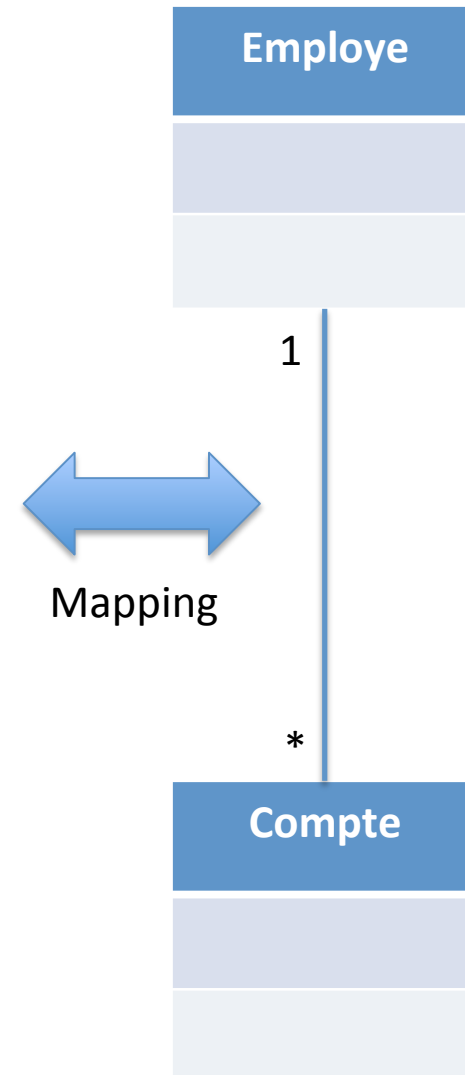




# Les associations (2/3)

```
public class Employe {  
    public Compte[] comptes;  
    public void setCompte (Compte[] comptes){  
        this.comptes = comptes;  
    }  
    public Compte[] getCompte (){  
        return comptes;  
    }  
}
```

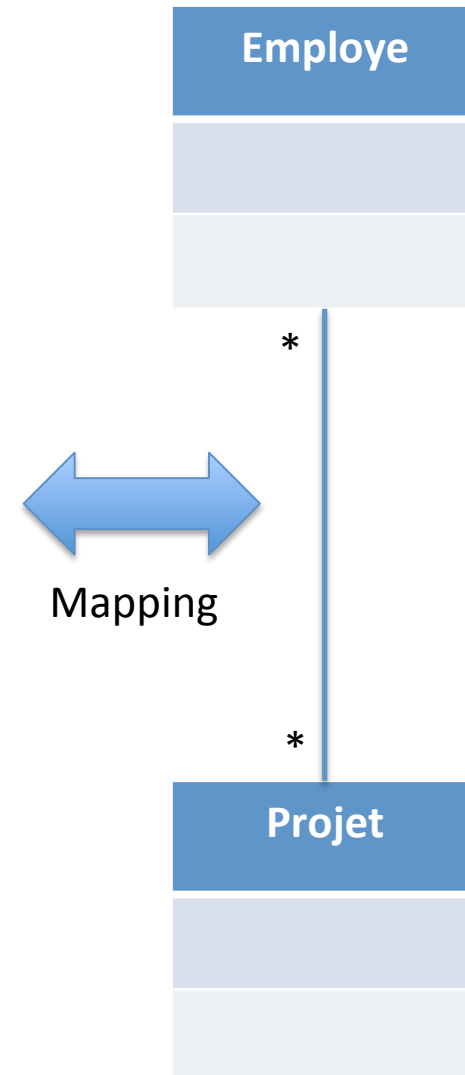
```
public class Compte {  
    public Employe employe;  
    public void setEmploye (Employe employe){  
        this.employe = employe;  
    }  
    public Employe getEmploye (){  
        return employe;  
    }  
}
```



# Les associations (3/3)

```
public class Employe {  
    public Projet[] projets;  
    public void setProjet (Projet[] projets){  
        this.projet = projet;  
    }  
    public Projet[] getProjet (){  
        return projets;  
    }  
}
```

```
public class Projet{  
    public Employe[] employes;  
    public void setEmploye (Employe[] employes){  
        this.employes = employes;  
    }  
    public Employe [] getEmploye (){  
        return employes;  
    }  
}
```



# Bilan

- Notions de base de la POO
  - Classe et Objet
  - Types de données en Java
    - Type Objet
    - Type primitifs
  - Constructeur / constructeur surchargé
  - Méthode surchargée
  - Mot clé « this »
  - Classe principale
  - Règles de nommage (Java)
- Associations et classes

# Chapitre 3 : Encapsulation

# Objectifs

- Maitriser le concept d'encapsulation en Java
  - Encapsulation des classes
  - Encapsulation des attributs
  - Encapsulation des méthodes
- Maitriser les méthodes getter et setter

# Plan

- Définition d'encapsulation
- Les packages
- Encapsulation des classes
- Encapsulation des attributs
- Encapsulation des méthodes
- Les méthodes setter et getter
- Exemple

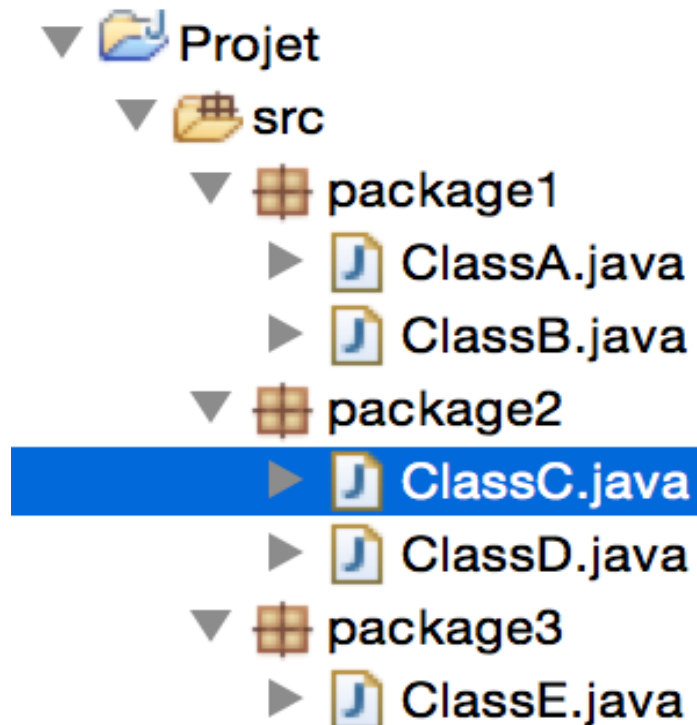
# Définition d'encapsulation

L'encapsulation décrit le degré de visibilité des classes, attributs et méthodes. Il s'agit de définir les règles d'exposition de la représentation interne d'un objet au monde extérieur.

<b>Représentation interne d'un objet</b>	<b>Monde extérieur</b>
<ul style="list-style-type: none"><li>• Attributs</li><li>• Méthodes</li></ul>	<p>Les autres classes</p> <ul style="list-style-type: none"><li>• dans le même package</li><li>• Dans des packages différents</li></ul>

# Les packages

- Package = répertoire
- Les classes Java peuvent être regroupées dans des packages





# Les packages

- Déclaration d'un package

```
package package2;
```

- Utilisation d'un package

- Nom du package suivi de nom de la classe

```
java.util.Date now = new java.util.Date() ;  
System.out.println(now) ;
```

- Import de la classe du package

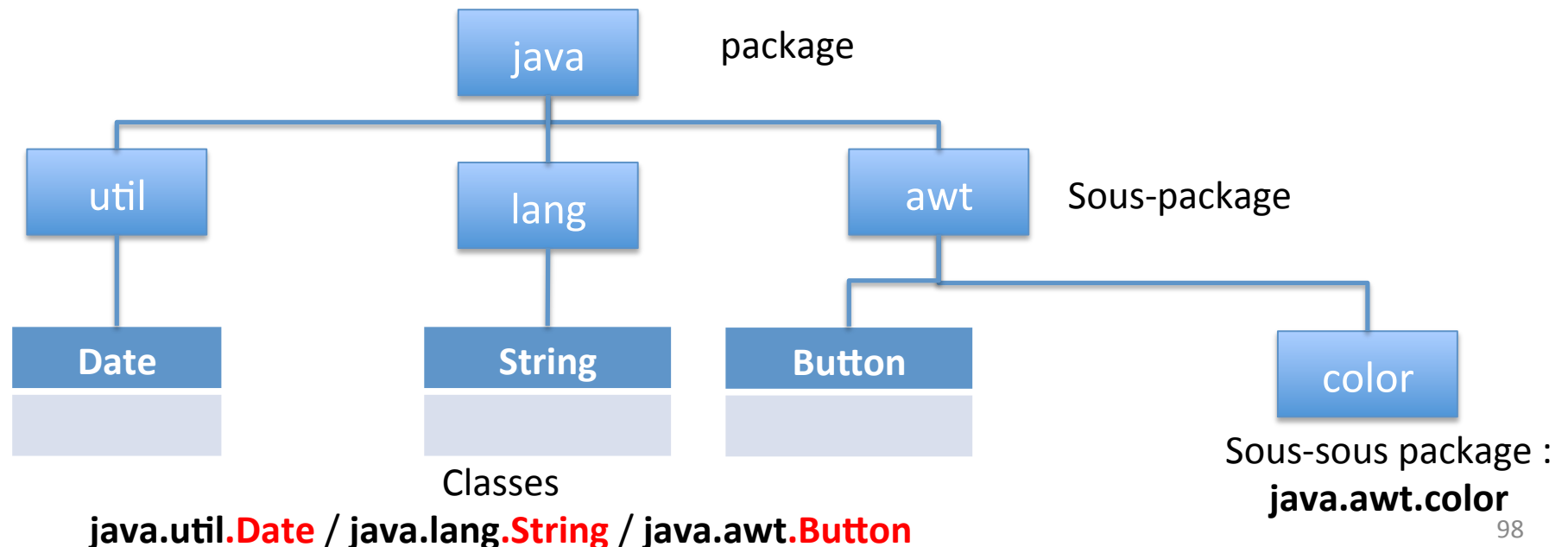
```
import java.util.Date ;  
...  
Date now = new Date() ; System.out.println(now) ;
```

- Import de tout le package

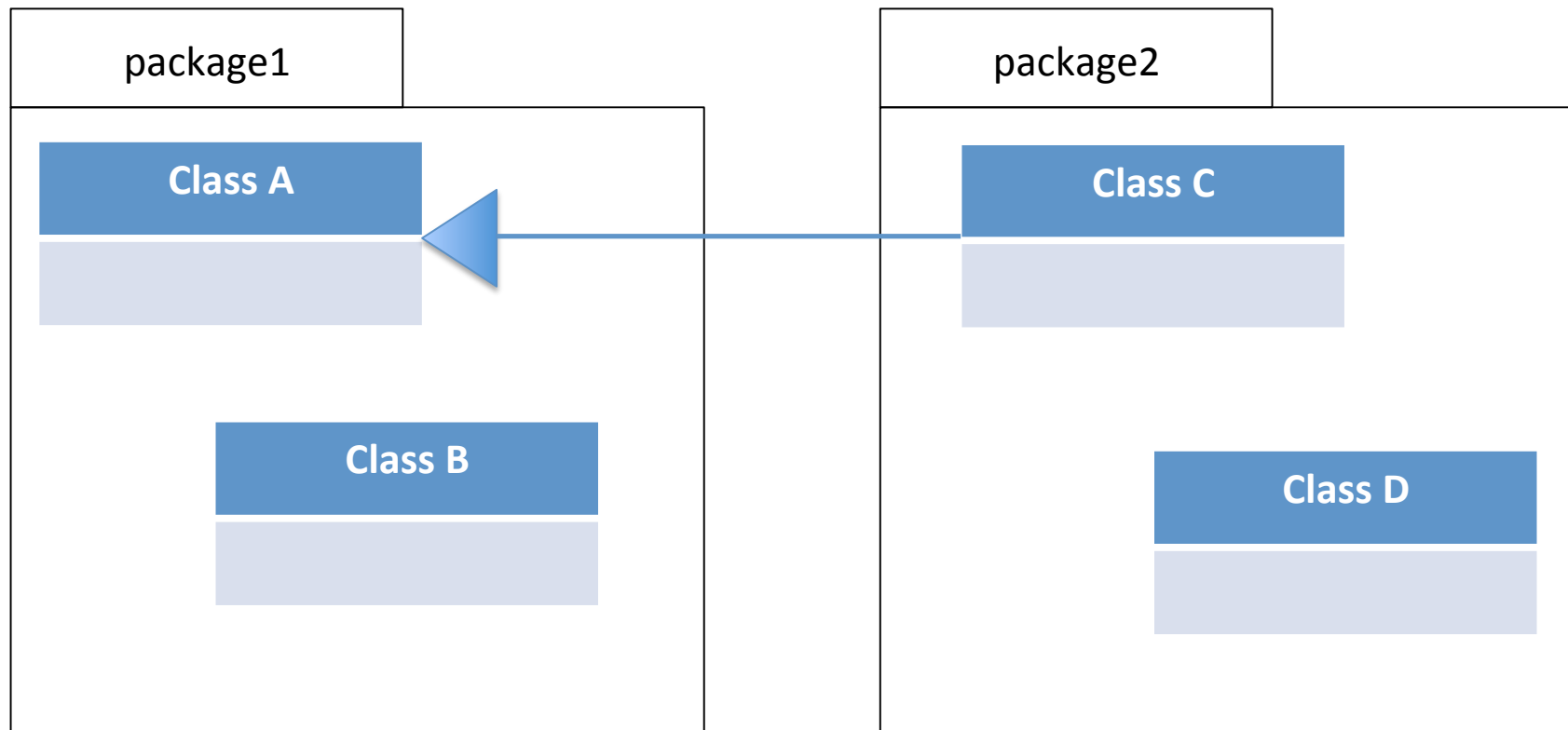
```
import java.util.* ;  
...  
Date now = new Date() ; System.out.println(now) ;
```

# Les packages

- Les packages sont organisés d'une manière hiérarchique
- Un package peut contenir des sous-packages, des sous-sous-packages, ...
- Exemple :
  - le package java contient les sous-packages « util », « lang » et « awt »
  - Le sous-package « awt » contient le sous-sous package « event »



# Encapsulation des classes



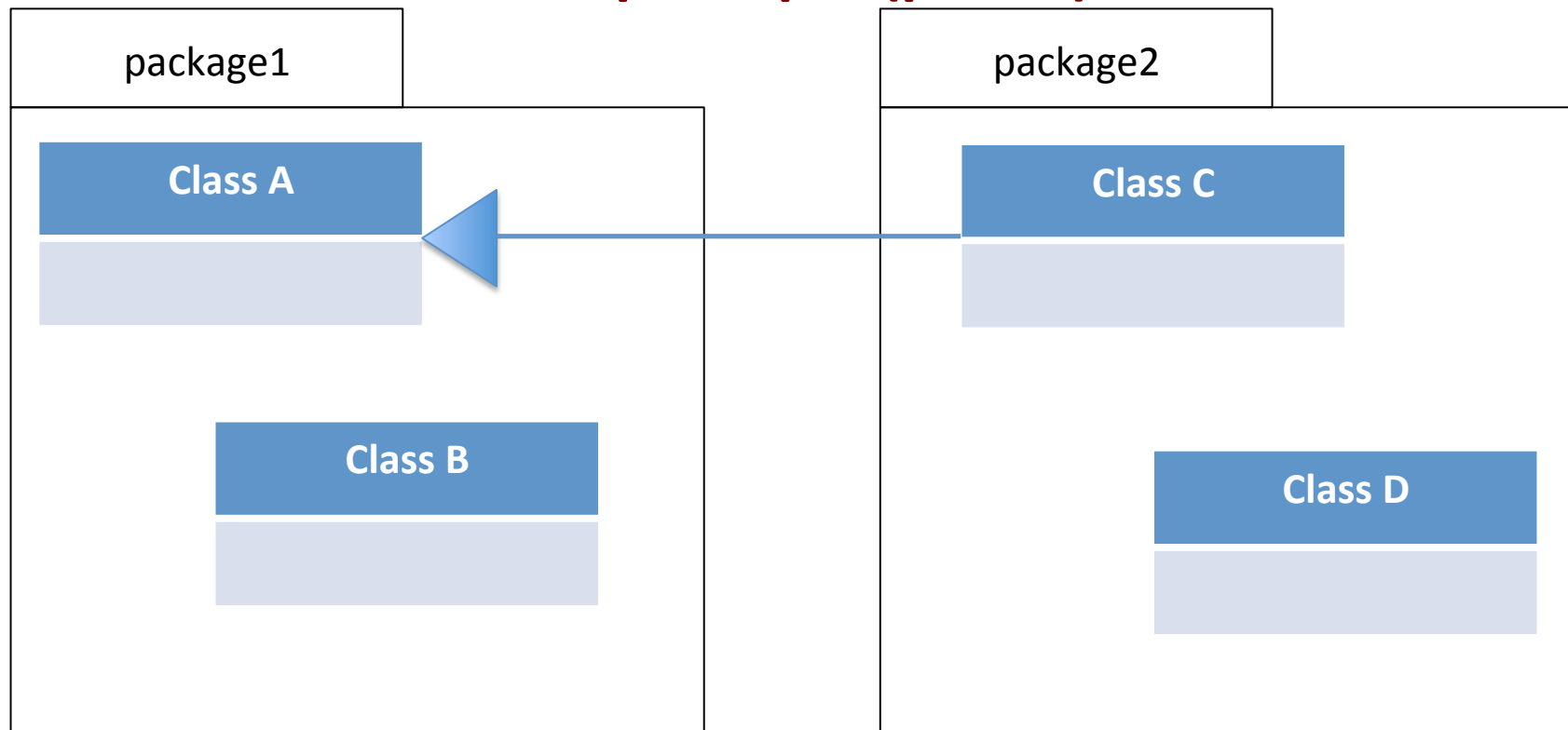
**Class C hérite de Class A**

**Class C : classe fille**

**Class A : classe mère**

# Encapsulation des classes

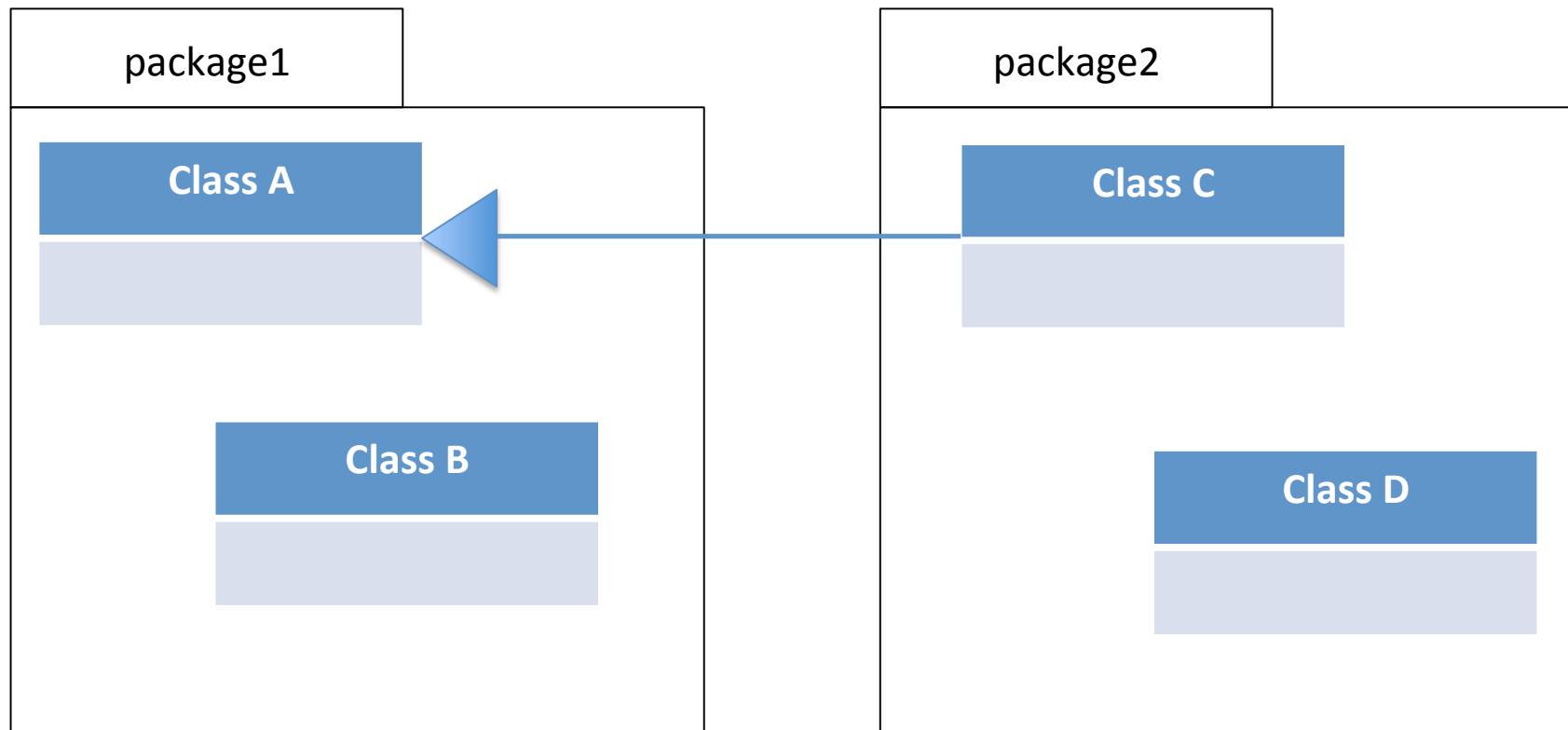
## La classe publique (public)



**public class A** → La classe public est visible/accessible depuis n'importe quelle autre classe (Class B, Class C et Class D)

# Encapsulation des classes

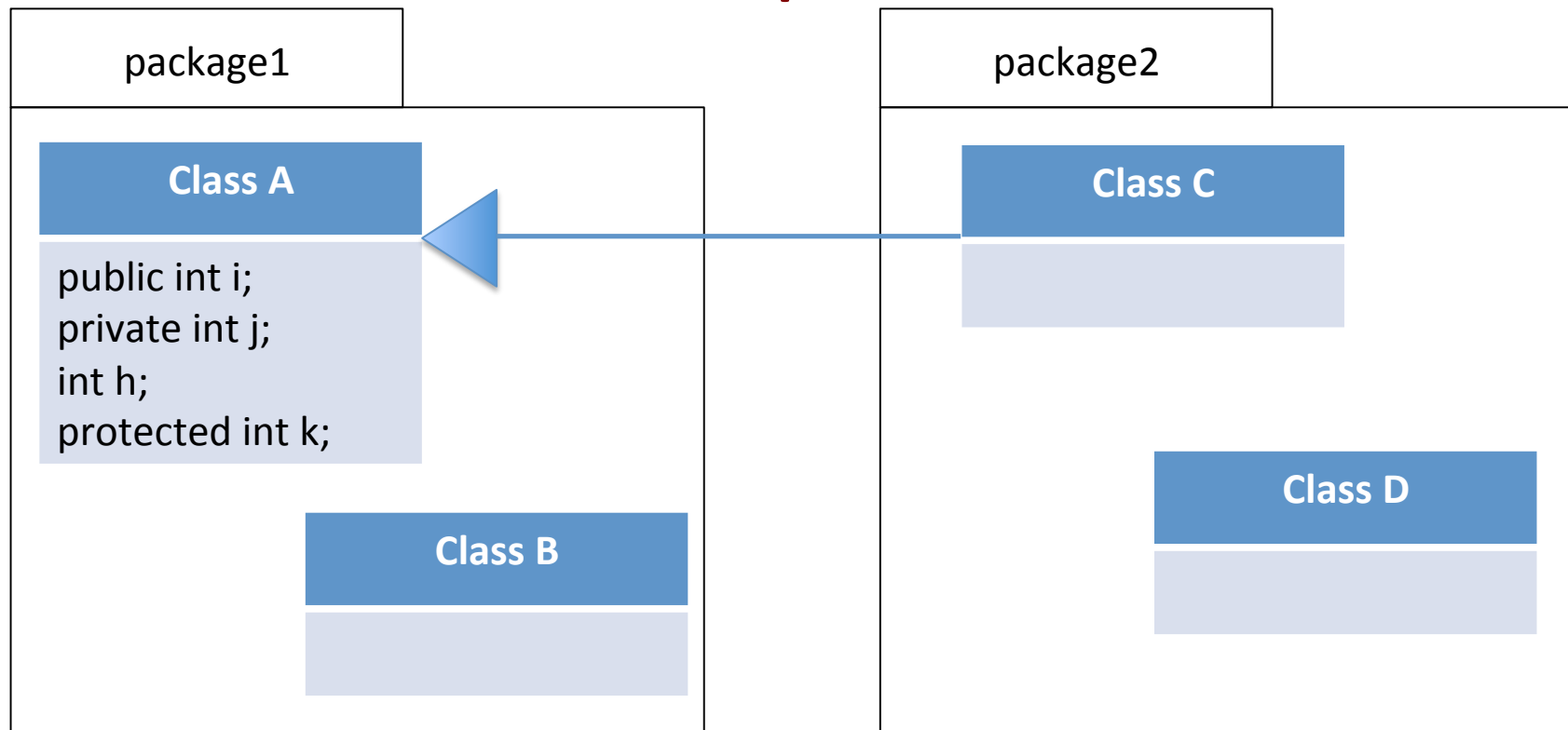
## La classe default



**class A** → La classe default est visible uniquement par les classes appartenant à son package (Class B)

# Encapsulation des attributs

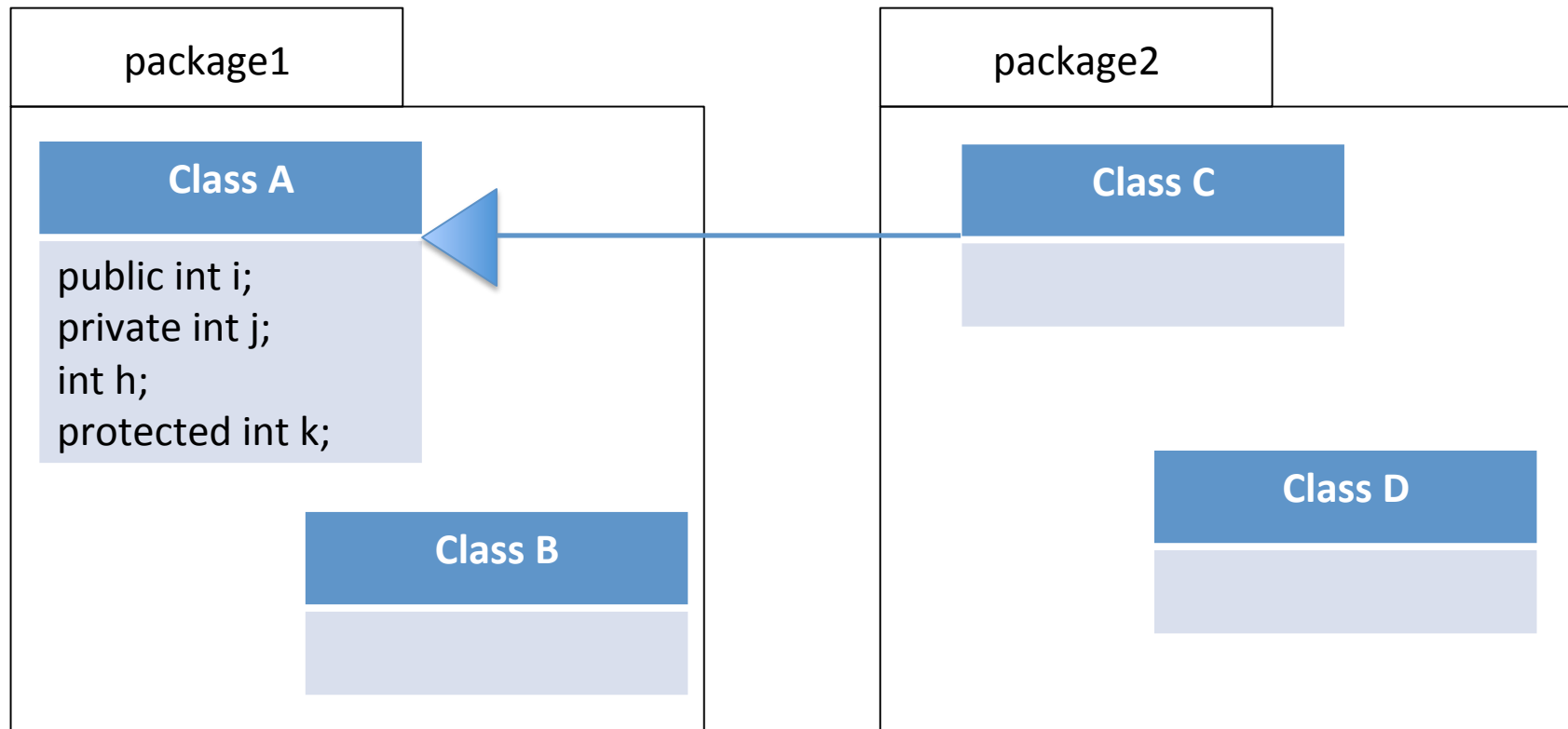
## L'attribut public



**public int i** → L'attribut public est visible par toutes les autres classes (Class A, Class B, Class C et Class D)

# Encapsulation des attributs

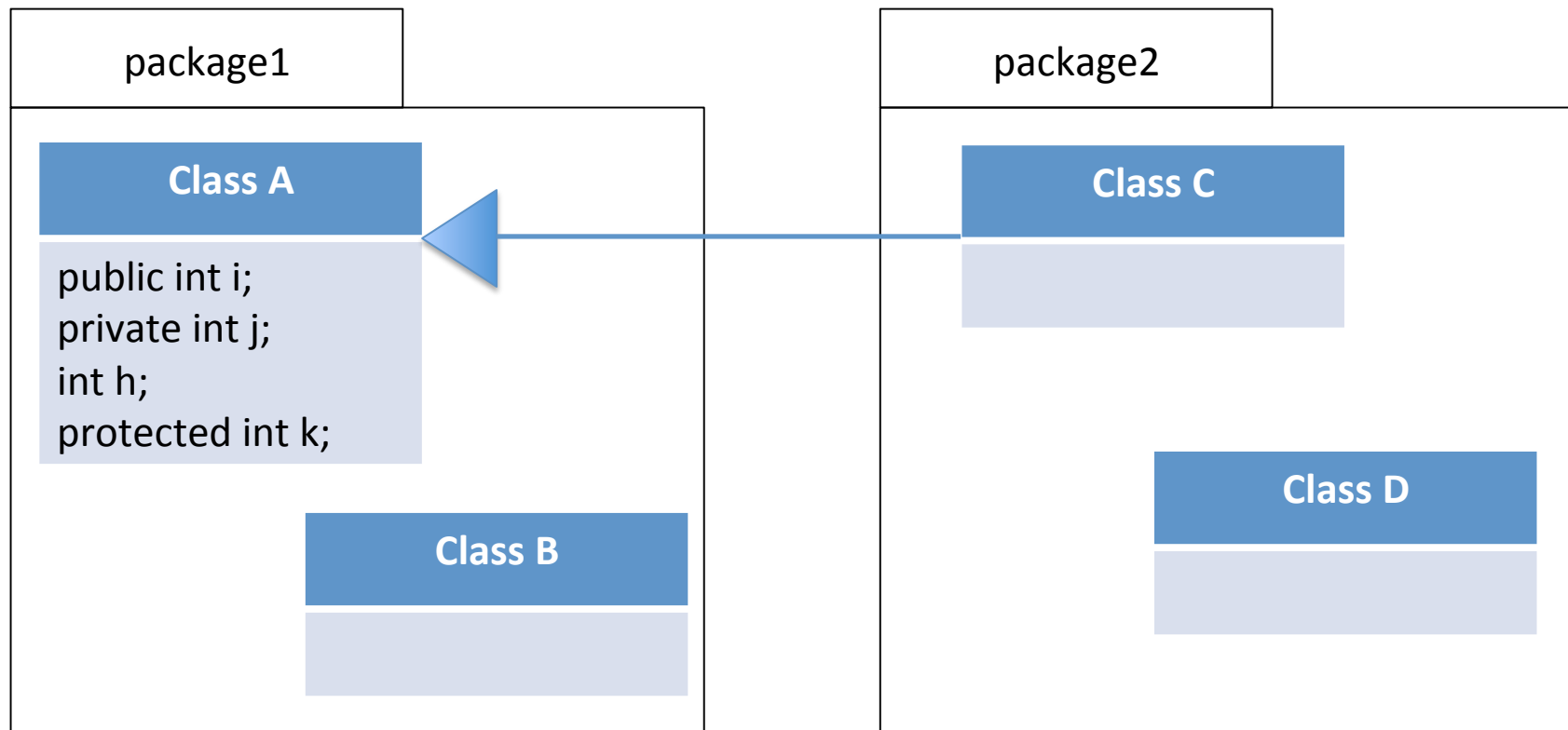
## L'attribut private



**private int j** → L'attribut private n'est visible qu'à l'intérieur de sa classe (Class A)

# Encapsulation des attributs

## L'attribut default

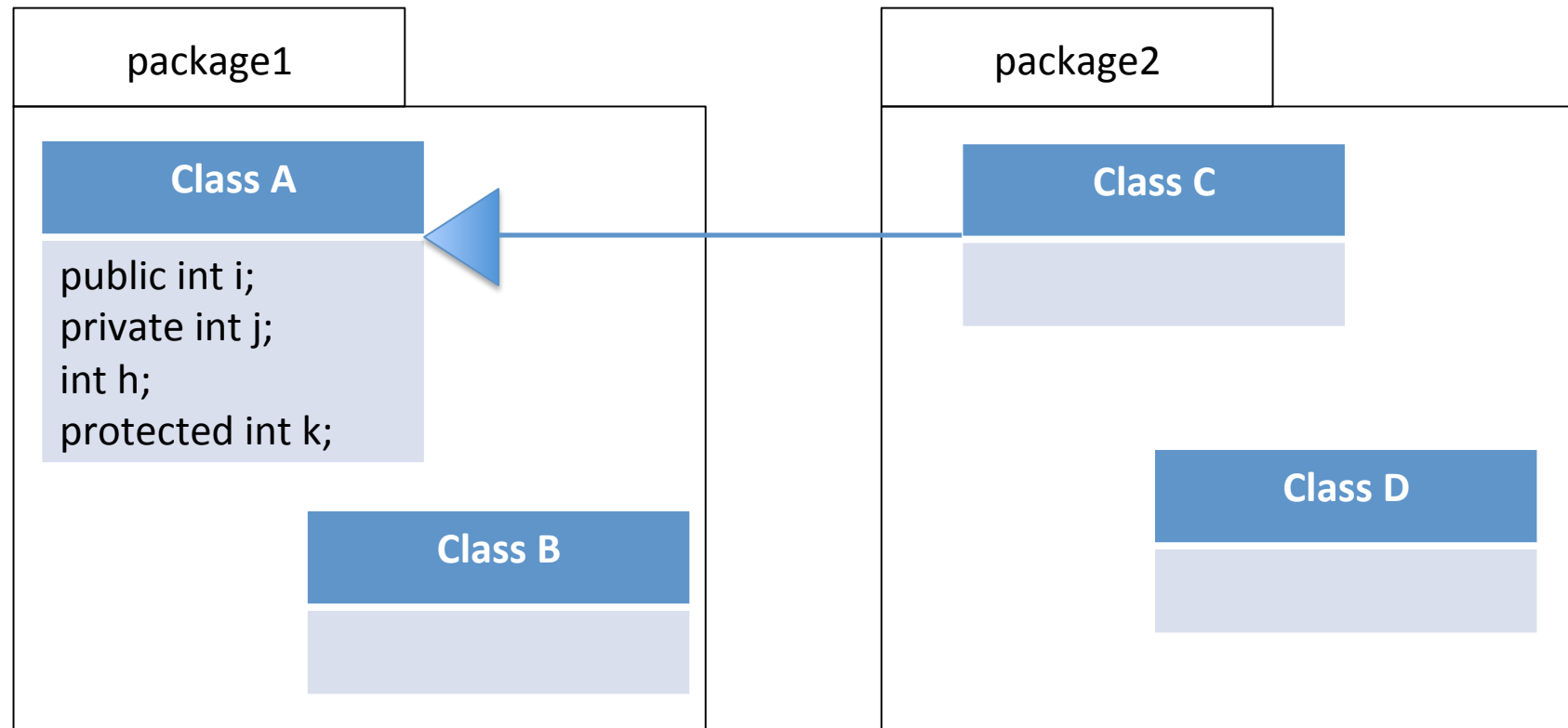


**int h** → L'attribut default est visible au niveau des classes appartenant au même package (Class A et Class B)



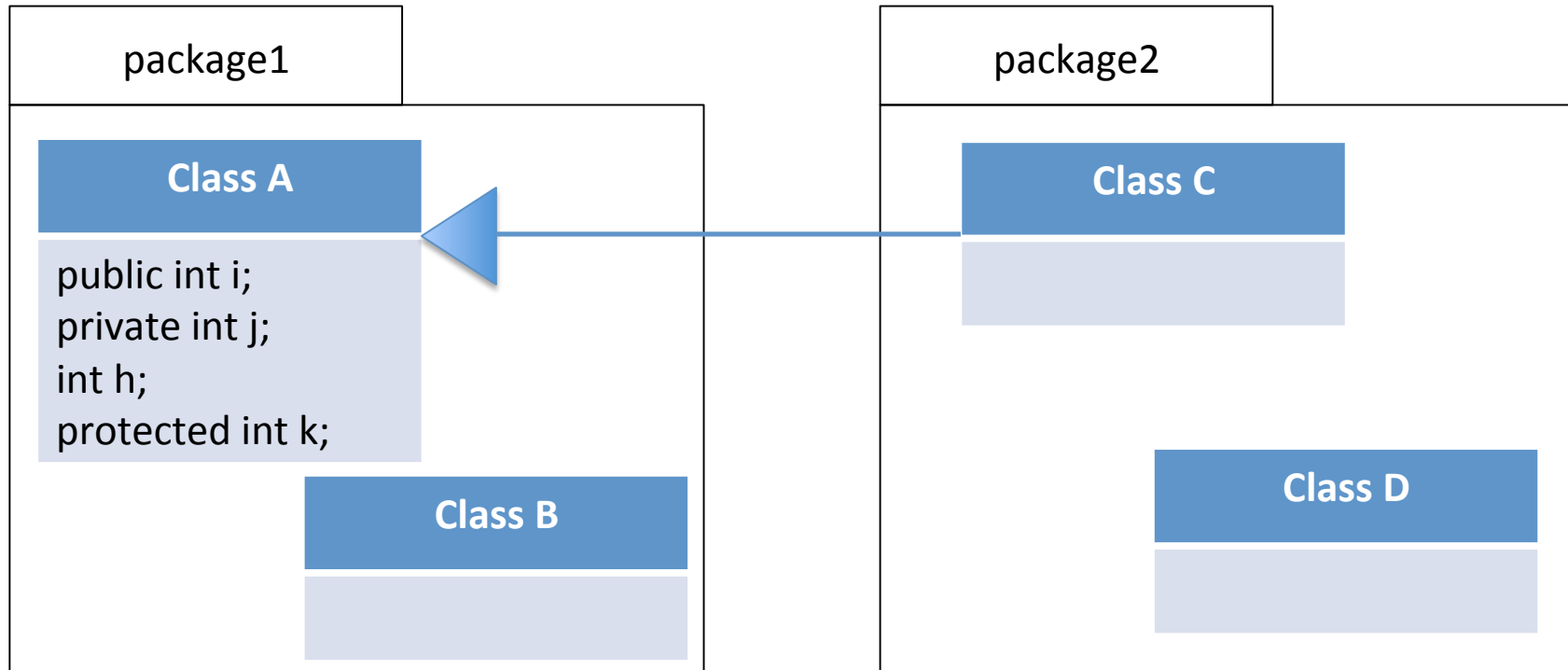
# Encapsulation des attributs

## L'attribut protected



**Protected int w** → L'attribut protected est accessible au niveau des classes du même package et à ses classes filles (même si elles n'appartiennent pas au même package) : Class A, Class B et Class C

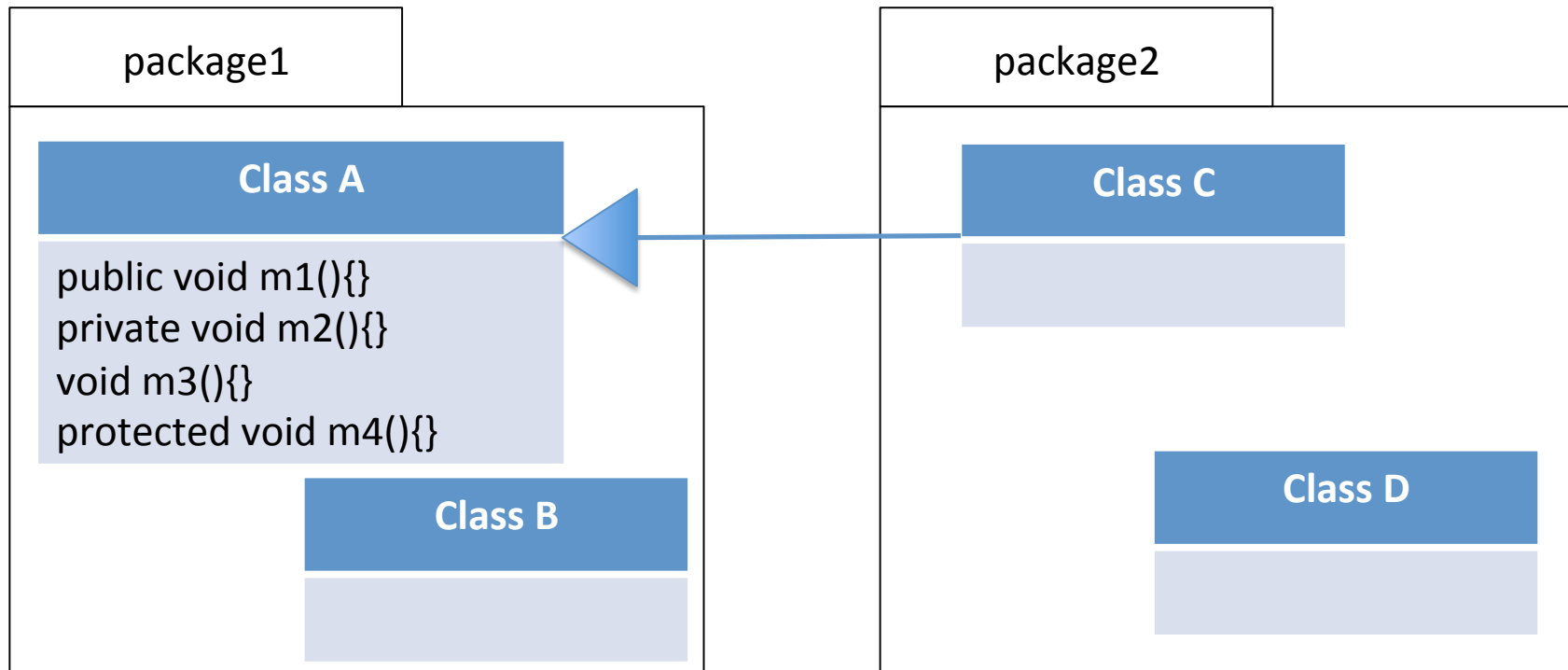
# Encapsulation des attributs



Visibilité à partir de	Class A	Class B	Class C	Tout le reste (Class D)
public int i;				
private int j;				
int h;				
protected int k;				

# Encapsulation des méthodes

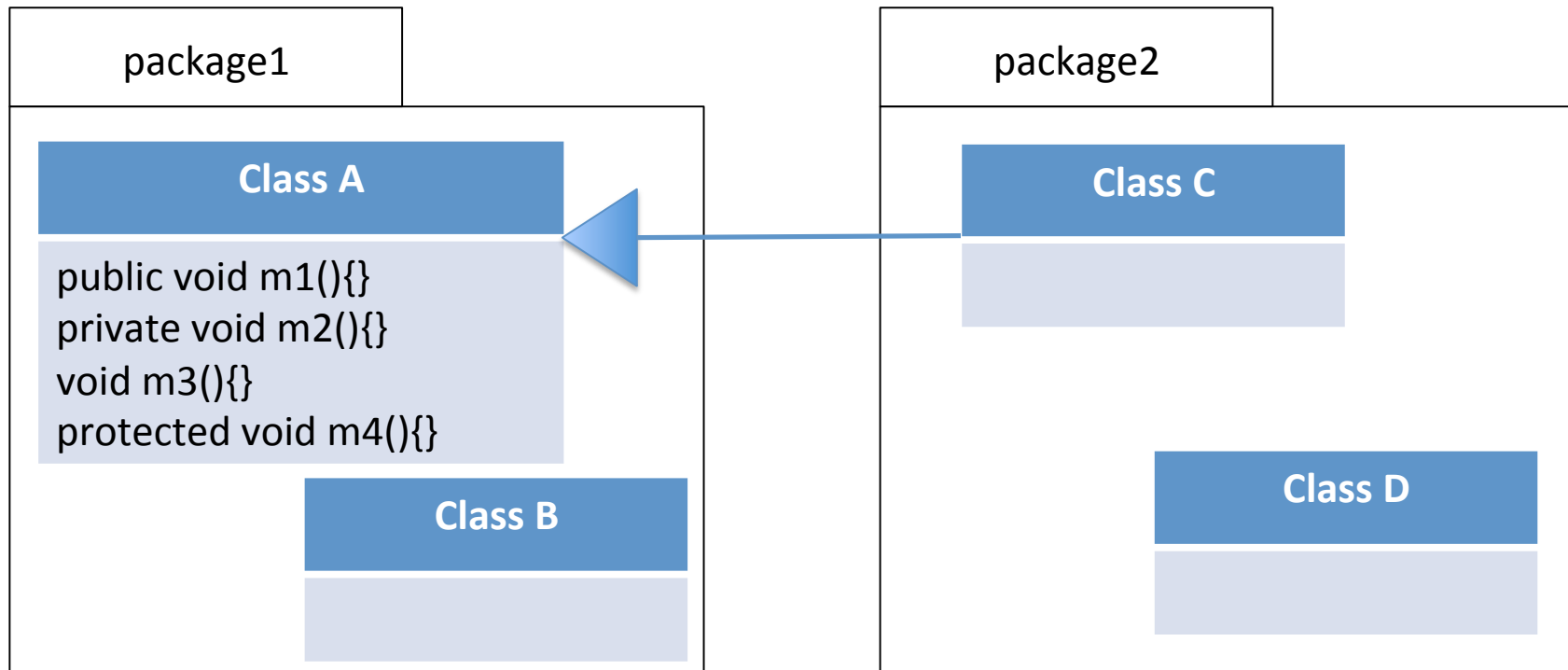
## La méthode public



**public void m1()** → La méthode public est accessible par toutes les autres classes (Class A, Class B, Class C et Class D)

# Encapsulation des méthodes

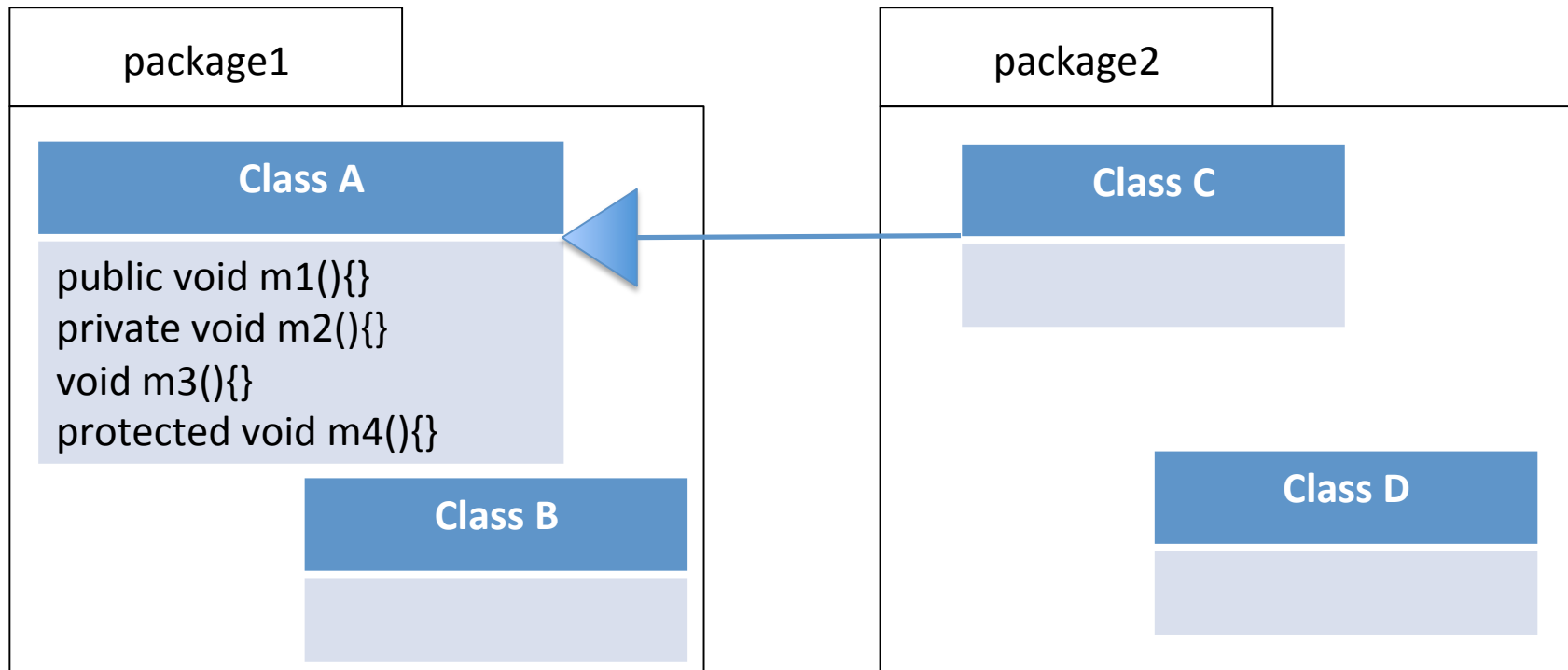
## La méthode private



**private void m2()** → La méthode private n'est accessible que depuis l'intérieur de la même classe (Class A uniquement)

# Encapsulation des méthodes

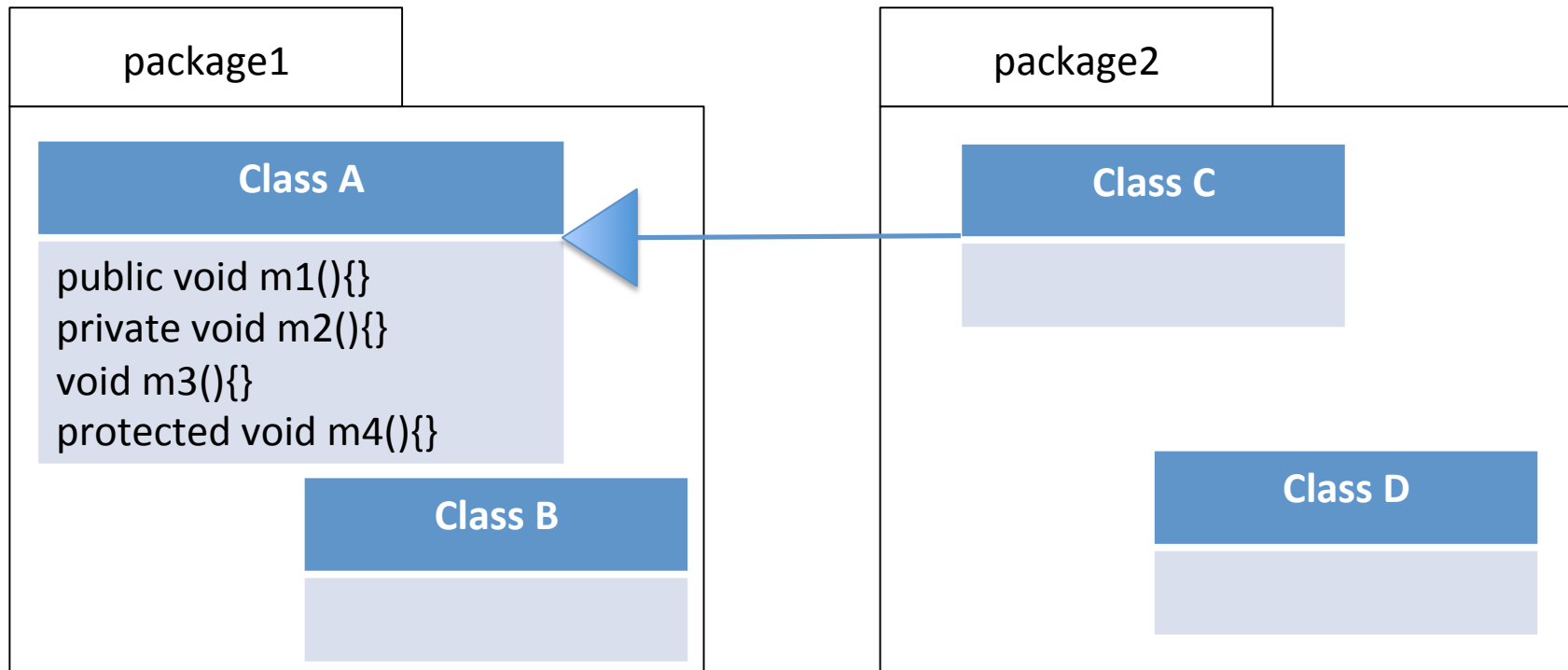
## La méthode default



**void m3()** → La méthode default n'est accessible que depuis les classes appartenant au même package (Class A et Class B)

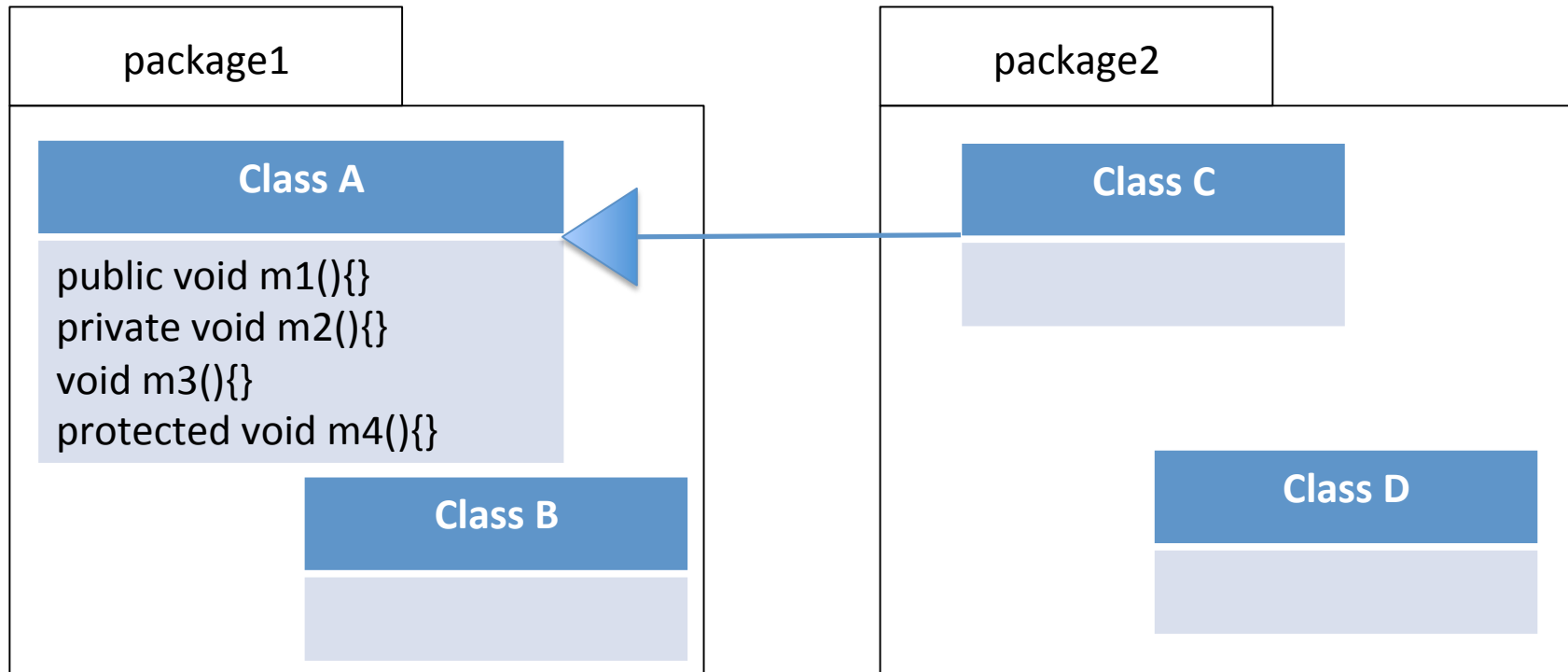
# Encapsulation des méthodes

## La méthode **protected**



**protected void m4()** → La méthode **protected** est visible au niveau des classes de son package et de ses sous-classes (même si elles sont définies dans un package différent) : Class A, Class B et Class C

# Encapsulation des méthodes



Visibilité à partir de	Class A	Class B	Class C	Tout le reste (Class D)
public void m1(){} 				
private void m2(){} 				
void m3(){} 				
protected void m4(){} 				

# Les méthodes setter et getter

- Pour manipuler des attributs « private », on utilise:
  - Le setter (un mutateur) : une méthode qui permet de définir la valeur d'un attribut particulier
  - Le getter (un accesseur) : une méthode qui permet d'obtenir la valeur d'un attribut particulier
- Le setter et le getter doivent être déclarés public

```
private float note;  
public void setNote(float note)  
{ this.note=note; }  
public float getNote()  
{ return note; }
```



# Exemple (1/2)

```
public class Cercle{
    private int rayon;
    Cercle(){
        rayon= 0;
    }
    void allongerCercle(int lg){
        rayon+=lg;}
    void augmenterCercle(){
        allongerCercle(2);}
    void affiche(){
        System.out.println("Rayon = "+rayon);
    }
    int getRayon(){
        return rayon;}
}
```

## Exemple (2/2)

```
public class TestCercle {  
    public static void main(String[] args) {  
        Cercle c1 = new Cercle();  
        c1.affiche();  
        c1.rayon = 10; //instruction incorrecte car l'attribut "rayon" est private  
        Cercle c2 = new Cercle(10);  
        c2.affiche();  
        c2.augmenterCercle();  
        System.out.println(c2.rayon); //instruction incorrecte car l'attribut "rayon" est private  
        System.out.println(c2.getRayon()); //On utilise le getter pour accéder à l'attribut private  
        c1.affiche();  
    }  
}
```

# Bilan

- Encapsulation des classes
- Encapsulation des attributs
- Encapsulation des méthodes
- Méthodes getter et setter

# Chapitre 4 : Héritage et Polymorphisme

# Objectifs

- Maitriser le concept d'héritage en Java
  - Super Classe et Sous Classe
  - Le mot clé super
  - Héritage et constructeurs
  - Re-définition des méthodes
  - Surcharge des méthodes
  - Le mot clé « final »
  - La classe « Object » et la méthode « equals »
- Maitriser le concept de polymorphisme en Java
  - Méthode polymorphe
  - Tableau polymorphe
  - Argument polymorphe
  - Sur-classement et substitution
  - Mécanisme de liaison retardée

# Plan

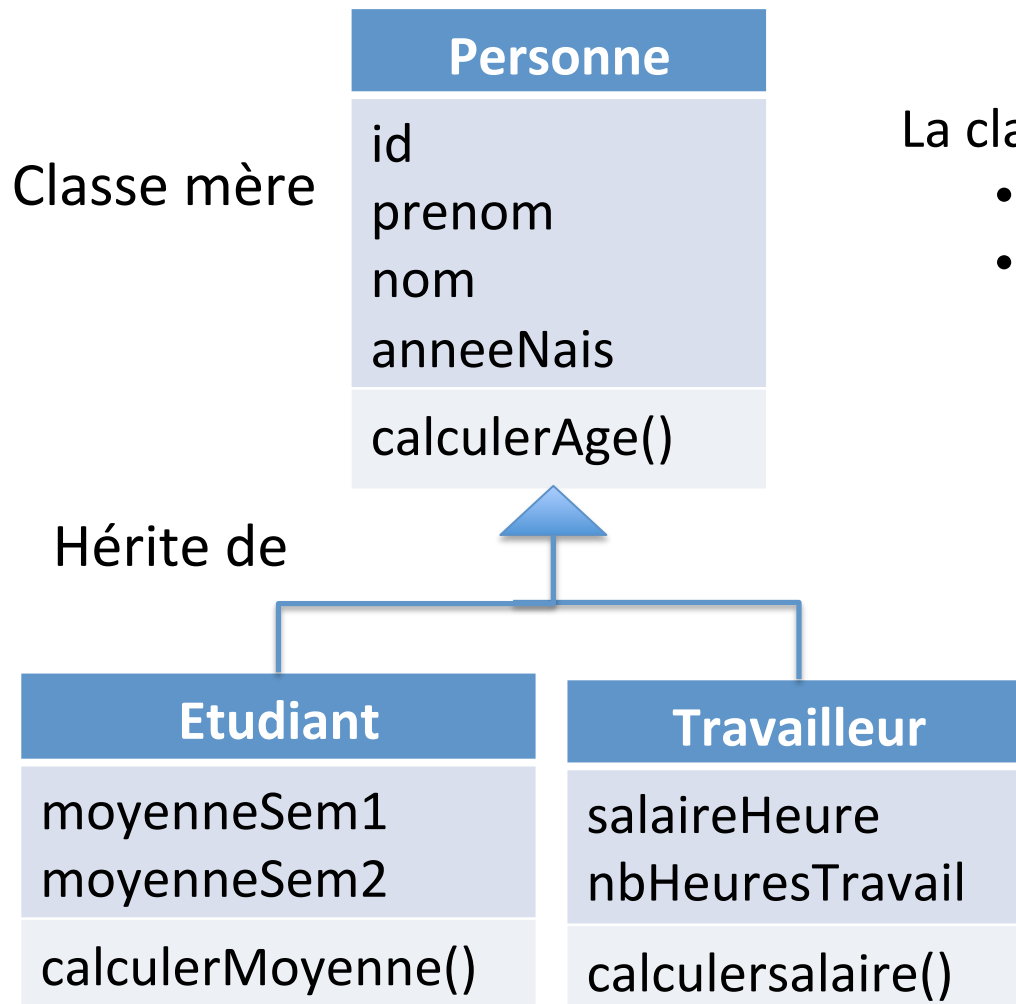
- Héritage
  - Introduction à l'héritage
  - Héritage
  - Héritage en Java
  - Le mot clé super
  - Héritage et encapsulation
  - Héritage et constructeurs
  - Redéfinition des méthodes
  - Surcharge des méthodes
  - Surcharge Vs. Redéfinition
  - Le mot clé final
  - La classe Object
  - La méthode « equals »
- Polymorphisme
  - Les méthodes polymorphes
  - Définition de polymorphisme
  - Tableau polymorphe
  - Argument polymorphe
  - Sur-classement et substitution
  - Mécanisme de polymorphisme
  - Interêt de polymorphisme

# Introduction à l'héritage (1/3)

Personne	Etudiant	Travailleur
id prenom nom anneeNais	id prenom nom anneeNais	id prenom nom anneeNais
calculerAge()	moyenneSem1 moyenneSem2	salaireHeure nbHeuresTravail
	calculerAge() calculerMoyenne()	calculerAge() calculersalaire()

- ☹ Des lignes de code qui se répètent → Duplication de code !
- ☹ Maintenance et évolution complexe. Par exemple, la modification de l'attribut « id » doit se faire 3 fois !

# Introduction à l'héritage (2/3)



La classe mère (super-classe) inclut :

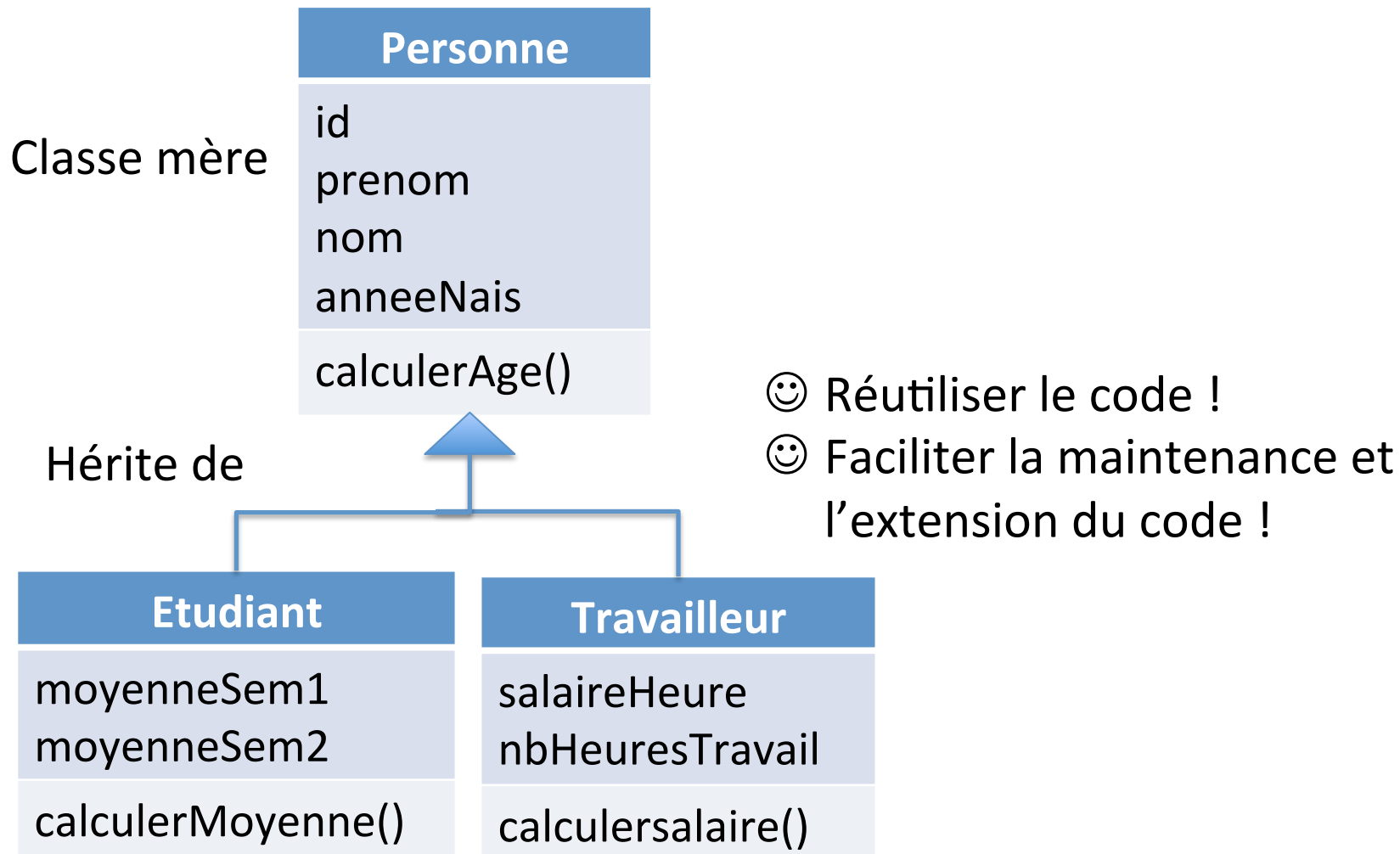
- tous les attributs communs
- toutes les méthodes communes

- Les classes filles héritent toutes les caractéristiques (attributs et méthodes) de la classe mère
- Les classes filles n'incluent que les caractéristiques (attributs ou méthodes) spécifiques

Classes filles



# Introduction à l'héritage (3/3)



Classes filles

# Héritage en Java

```
import java.util.Calendar;
public class Personne {
    int id;
    String nom, prenom;
    int anneeNais;
    int calculerAge(){
        Calendar calendrier;
        calendrier = Calendar.getInstance();
        int anneeEnCours =
        calendrier.get(Calendar.YEAR);
        return anneeEnCours-anneeNais;
    }
}
```

```
public class Etudiant extends Personne{
    float moyenneSem1;
    float moyenneSem2;
    String section;
    float calculerMoyenne(){
        return (moyenneSem1 +
        moyenneSem2*2)/3;
    }
}
```

Le mot clé « extends » indique que la classe « Etudiant » hérite de la classe « Personne »

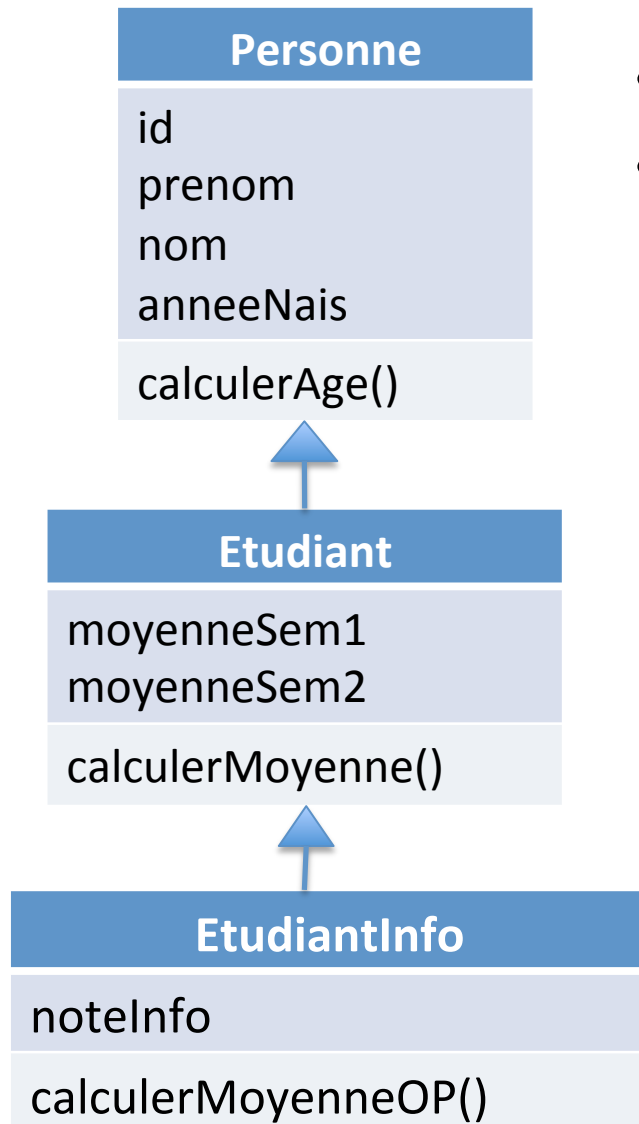
# Héritage en Java

- Une classe mère (super-classe) peut avoir plusieurs classes filles (sous-classes)
- Une classe fille **ne peut pas** avoir plusieurs classes mère

→ Héritage multiple n'est pas permis en Java !

→ ~~Class Etudiant extends Personne, Humain~~

# Héritage en cascade (1/2)

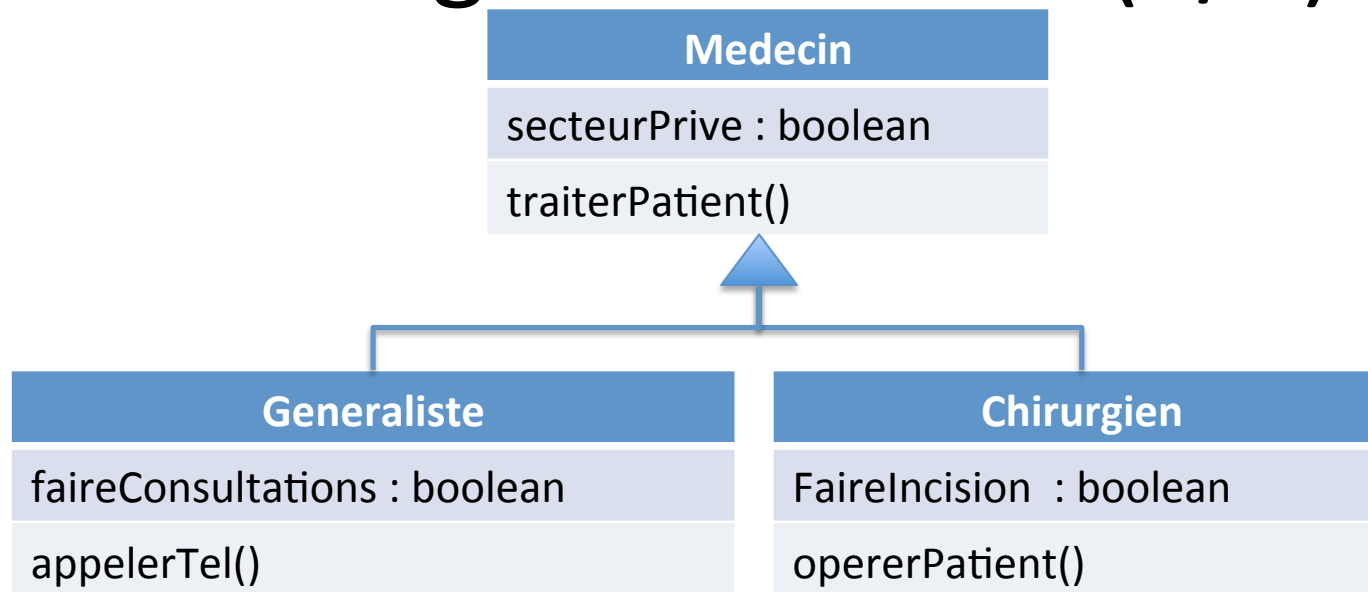


- « Etudiant » hérite de « Personne »
  - « EtudiantInfo » hérite de « Etudiant »
- « EtudiantInfo » hérite de « Personne »

```
Etudiant E1 = new Etudiant();    →OK
E1.calculerAge();                 →OK
E1.calculerMoyenne();            →OK
E1.calculerMoyenneOP();          →KO
```

```
Etudiant Einfo = new EtudiantInfo(); →OK
Einfo.calculerAge();                 →OK
Einfo.calculerMoyenne();             →OK
Einfo.calculerMoyenneOP();           →OK
```

# Héritage en cascade (2/2)



- Nombre d'attributs de la classe « Medecin » ??
- Nombre d'attributs de la classe « Generaliste » ??
- Nombre d'attributs de la classe « Chirurgien » ??
- Nombre de méthodes de la classe « Medecin » ??
- Nombre de méthodes de la classe « Generaliste » ??
- Nombre de méthodes de la classe « Chirurgien » ??
- Est ce qu'une instance de la classe « Chirurgien » peut invoquer la méthode « `traiterPatient()` » ?
- Est ce qu'une instance de la classe « Chirurgien » peut invoquer la méthode « `appelerTel()` » ?

# « super »

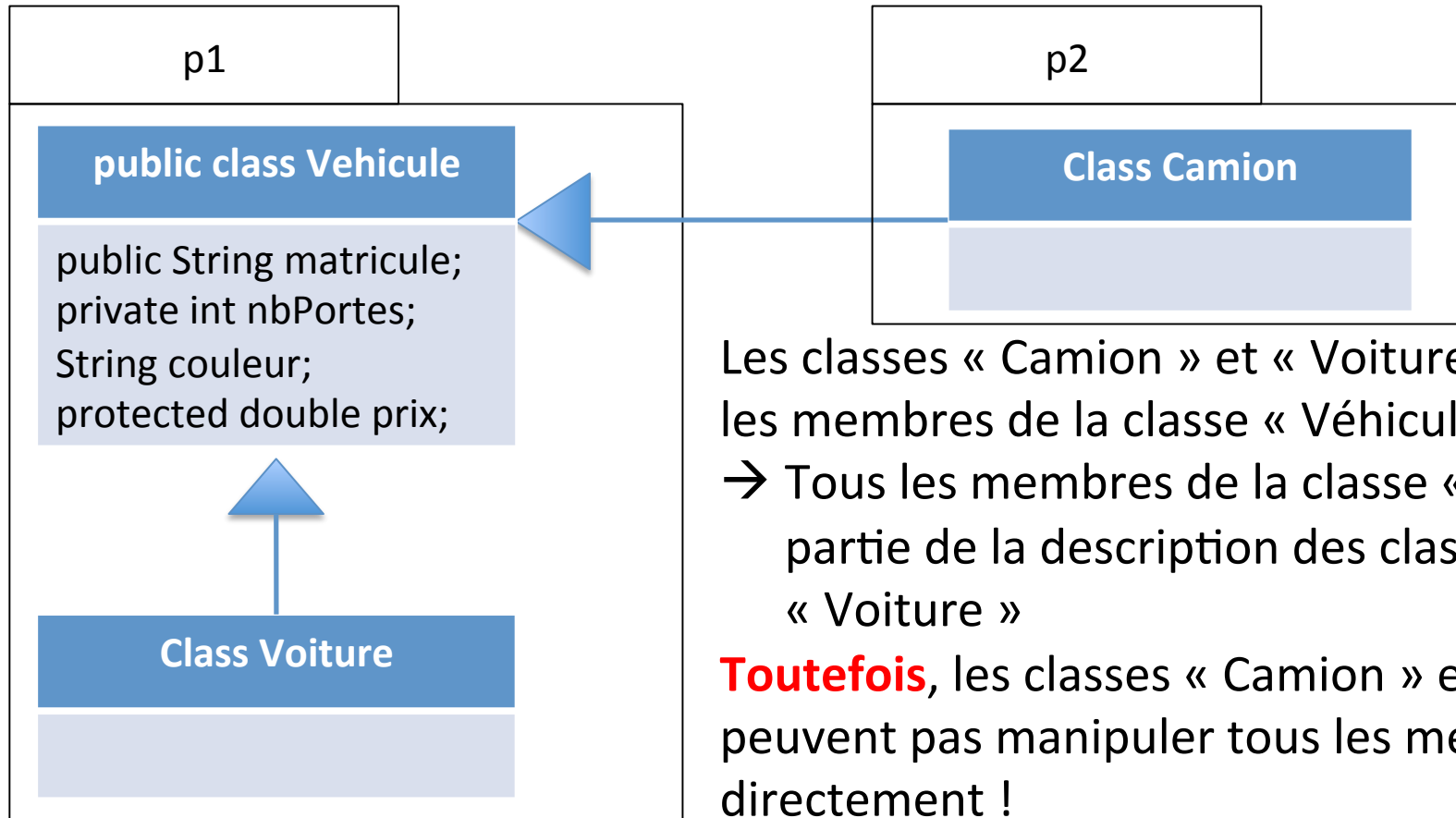
- « **super** » désigne la classe mère (**super-classe**)
  - « **super** » permet d'
    - invoquer les méthodes et constructeurs de la classe mère
    - Accéder aux attributs de la classe mère
- Exemples :
  - Pour accéder à l'attribut « id » de la classe mère, on utilise « **super.id** »
  - Pour invoquer la méthode « calculerAge() » de la classe mère, on utilise « **super.calculerAge()** »
  - Pour appeler le constructeur par défaut de la classe mère, on utilise **super()**
  - Pour appeler le constructeur surchargé de la classe mère, on utilise **super(nom, prenom)**

# Héritage et encapsulation

Classe mère	Classe fille
Membre (Attributs / méthodes) <b>private</b>	ne peut pas manipuler directement les membres <b>private</b>
Membre (Attributs / méthodes) <b>default</b>	peut manipuler les membres <b>default</b> si la classe mère existe dans le même package que la classe fille
Membre (Attributs / méthodes) <b>protected</b>	peut manipuler les membres <b>protected</b>
Membre (Attributs / méthodes) <b>public</b>	peut toujours manipuler les membres <b>public</b>

Un membre hérité est désigné par **super.membre**, si l'accès est autorisé

# Héritage et encapsulation



Voiture accède directement à :

- matricule
- couleur
- prix

Camion accède directement à :

- matricule
- prix



# Héritage et encapsulation : Exemple

```
package ch3.p1;

public class Vehicule {
    public String matricule;
    private int nbPortes;
    String couleur;
    protected double prix;
    public int getNbPortes() {
        return nbPortes;
    }
    public void setNbPortes(int nbPortes) {
        this.nbPortes = nbPortes;
    }
    public String getCouleur() {
        return couleur;
    }
    public void setCouleur(String couleur) {
        this.couleur = couleur;
    }
    public double getPrix() {
        return prix;
    }
    public void setPrix(double prix) {
        this.prix = prix;
    }
}
```

```
package ch3.p1;

public class Voiture extends Vehicule{
    public void afficher(){
        System.out.println("Camion :
"+super.matricule+
    "de couleur"+super.couleur+
    " ayant "+super.getNbPortes()+
    " portes, coûte "+super.prix);}}
```

```
package ch3.p2;

import ch3.p1.Vehicule;

public class Camion extends Vehicule{
    public void afficher(){
        System.out.println("Camion : "+
super.matricule+
    "de couleur"+super.getCouleur()+
    " ayant "+super.getNbPortes()+
    " portes, coûte "+super.prix);}}
```

# Héritage et constructeurs : constructeur par défaut

```
public class Voiture {  
    String matricule;  
    int nbPortes;  
    String couleur;  
    public Voiture() {  
        matricule = "124Tunis1999";  
        nbPortes = 4;  
        couleur = "vert";  
    }  
}
```

```
public class VoitureLocation extends Voiture{  
    int nbHeuresLocation;  
    double prixHeureLocation;  
    public VoitureLocation() {  
        super();  
        nbHeuresLocation = 10;  
        prixHeureLocation = 120;  
    }  
}
```

« **super()** » invoque le **constructeur par défaut** de la classe Voiture (**classe mère**)

# Héritage et constructeurs : constructeur surchargé

```
public class Voiture {  
    String matricule;  
    int nbPortes;  
    String couleur;  
    public Voiture(String matricule,  
int nbPortes, String couleur) {  
        this.matricule = matricule;  
        this.nbPortes = nbPortes;  
        this.couleur = couleur;  
    }  
}
```

```
public class VoitureLocation extends Voiture{  
    int nbHeuresLocation;  
    double prixHeureLocation;  
    public VoitureLocation(String matricule,  
int nbPortes, String couleur,  
int nbHeuresLocation,  
double prixHeureLocation) {  
        super(matricule, nbPortes, couleur);  
        nbHeuresLocation = 10;  
        prixHeureLocation = 120;  
    }  
}
```

« **super(matricule, nbPortes, couleur)** » invoque le **constructeur surchargé** de la classe Voiture (**classe mère**)

# Constructeur de la classe fille

- 1<sup>ère</sup> instruction du constructeur de la classe fille = invocation du constructeur de la classe mère, en utilisant « **super** »
- Si on n'appelle pas explicitement un constructeur de la classe mère, le constructeur par défaut de la classe mère est invoqué implicitement via « **super()** »

# Constructeur de la classe fille : Exemple (1/3)

```
public class Forme {  
    public int nbCotes;  
    public Forme(int nbCotes) {  
        this.nbCotes = nbCotes;  
    }  
}
```

Constructeur surchargé →  
Pas de constructeur par  
défaut !


```
public class Triangle extends Forme{  
    public Triangle() {  
        super();  
    }  
}
```

Appel implicite au  
constructeur par défaut la  
classe mère **qui n'existe  
pas** → Erreur de  
compilation !

→ Solution ???

# Constructeur de la classe fille : Exemple (2/3)

```
public class Forme {  
    public int nbCotes;  
  
    public Forme(){}  
  
    public Forme(int nbCotes) {  
        this.nbCotes = nbCotes;  
    }  
}
```




Définition explicite du  
constructeur par défaut  
au niveau de la classe  
mère

```
public class Triangle extends Forme{  
  
}
```

# Constructeur de la classe fille : Exemple (3/3)

```
public class Forme {  
    public int nbCotes;  
    public Forme(int nbCotes) {  
        this.nbCotes = nbCotes;  
    }  
}
```

```
public class Triangle extends Forme{  
    public Triangle() {  
        super(3);  
    }  
}
```



Invocation explicite au  
constructeur surchargé  
de la super-mère

# Héritage : Enchaînement des constructeurs

```
public class Medecin {  
    public Medecin(){  
        System.out.println("Je suis un médecin");  
    }  
}
```

```
public class MedecinGeneraliste extends Medecin{  
    public MedecinGeneraliste() {  
        System.out.println("Je suis un médecin généraliste");  
    }  
}
```

```
public class MedecinGeneralisteHopital extends MedecinGeneraliste{  
    public MedecinGeneralisteHopital() {  
        System.out.println("Je suis un médecin généraliste et je travaille à l'hopital");  
    }  
}
```

```
MedecinGeneralisteHopital m = new MedecinGeneralisteHopital();
```

???



# Redéfinition de méthodes

```
public class Voiture {  
    public void afficher(){  
        System.out.print("C'est une voiture");  
    }  
}
```

```
public class VoitureLocation extends Voiture{  
    public void afficher(){  
        System.out.println("C'est une voiture de location");  
    }  
}
```

Redéfinition de méthodes :

- Même signature :  
**public void afficher()**
- Changer le corps de la méthode

- La classe « VoitureLocation » hérite de la classe « Voiture » la méthode « void afficher() »
- Mais, on souhaite changer les fonctionnalités de la méthode « afficher() »  
→ **Redéfinir la méthode « afficher() »**

# Redéfinition de méthodes

```
public class Voiture {  
    public void afficher(){  
        System.out.print("C'est une voiture");  
    }  
}
```

```
public class VoitureLocation extends Voiture{  
    public void afficher(){  
        super.afficher();  
        System.out.println(" de location");  
    }  
}
```

Invocation de la méthode  
afficher() de la classe mère  
→ Réutiliser le code  
développé au niveau de la  
classe mère !

# Surcharge de méthodes

```
public class Voiture {  
    public void afficher(){  
        System.out.print("C'est une voiture");  
    }  
}
```

```
public class VoitureLocation extends Voiture{  
    //méthode surchargée  
    public void afficher(int prixLocationHeure){  
        super.afficher();  
        System.out.println(" de location pour "+prixLocationHeure+" DT par heure");  
    }  
}
```

## Surcharge de méthode :

- **même** nom de la méthode : **public void afficher**
- **Changer** le nombre / type d'argument d'entrées (input) de la méthode :  
**public void afficher(int prixLocationHeure)**

# Surcharge vs. Redéfinition

```
public class Voiture {  
    public void afficher(){  
        System.out.print("C'est une voiture");  
    }  
}
```

Redéfinition

Surcharge

```
public class VoitureLocation extends Voiture  
{  
    public void afficher(){  
        super.afficher();  
        System.out.println(" de location");  
    }  
}
```

La classe « VoitureLocation » contient une seule méthode « **public void afficher()** »

```
public class VoitureLocation extends Voiture  
{  
    public void afficher(int prixLocationHeure){  
        super.afficher();  
        System.out.println(" de location pour "+  
            prixLocationHeure+" DT par heure");  
    }  
}
```

La classe « VoitureLocation » contient 2 méthodes

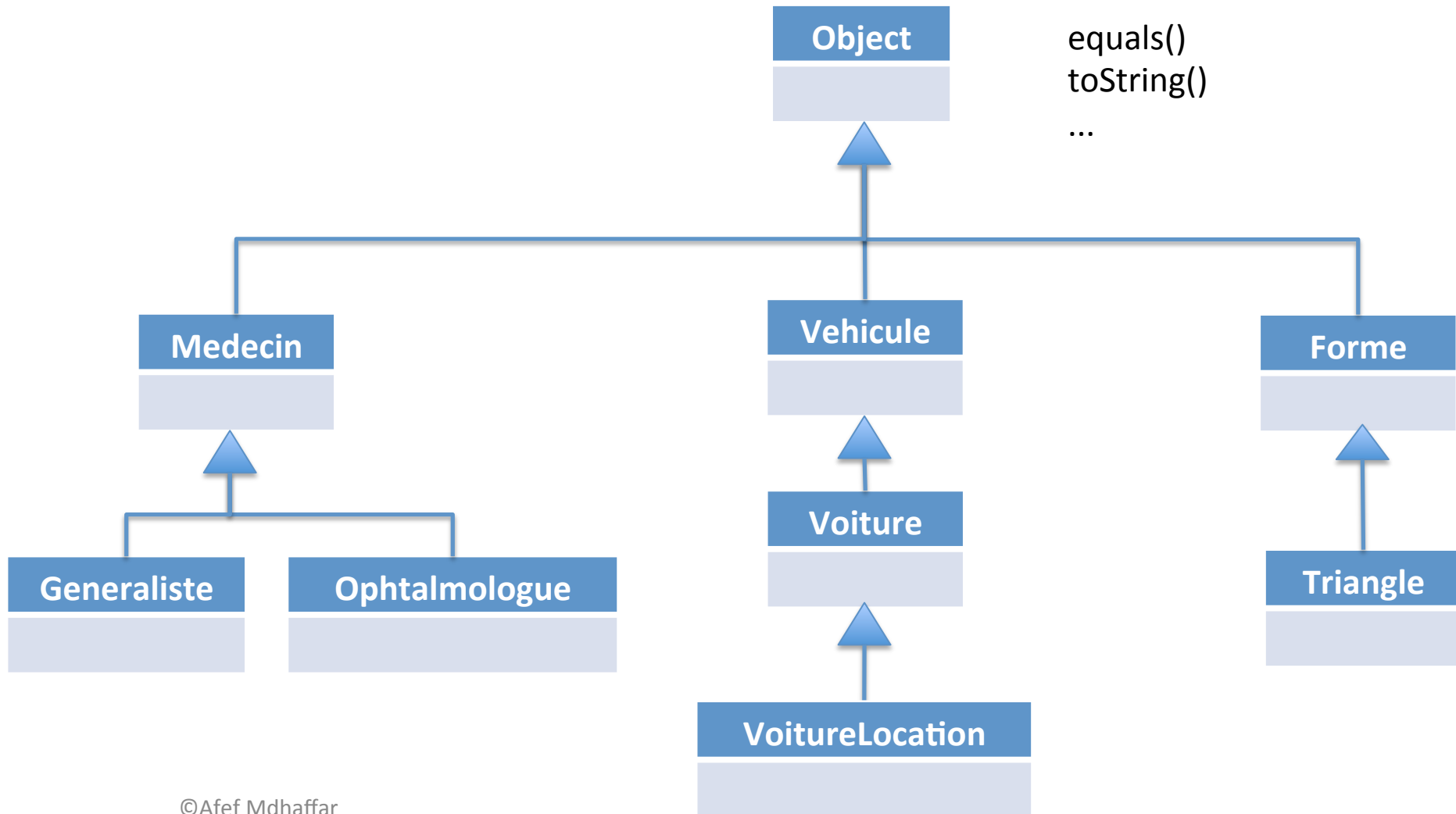
- **public void afficher()**
- **public void afficher(int prixLocationHeure)**

# « final »

- Pour empêcher la redéfinition de méthodes et l'héritage de classes, on utilise le mot clé « final »
  - **final** void afficher() → cette méthode ne peut pas être redéfinie
  - **final** class Voiture { } → cette classe ne peut pas être une classe mère. On ne peut pas hériter de la classe Voiture
  - Une classe déclarée « **final** » ne comprend que des méthodes « **final** »

# La classe « Object »

- La classe « Object » est la classe mère (ancêtre) de toutes les classes Java
- Toute classe Java hérite **implicitement** de la classe « Object »



# La méthode « equals() » (1/2)

**public boolean equals(Object obj)**

- Définie initialement au niveau de la classe Object
- Comparer les références de 2 objets
  - Retourne « true » si les 2 références sont identiques
  - Retourne « false » sinon
- Peut être redéfinie aux niveaux des classes filles (toutes les classes Java) pour comparer 2 objets

# La méthode « equals() » (2/2)

- Redéfinir la méthode equals() pour la classe Voiture
- Deux voitures sont égales s'elles ont la même matricule et la même couleur

```
public boolean equals(Object obj){  
    if(obj instanceof Voiture){  
        Voiture v = (Voiture) obj;  
        if(v.matricule.equals(matricule) &&  
            (v.couleur.equals(couleur)))  
            return true;  
        }  
        return false;  
    }  
}
```



# Polymorphisme

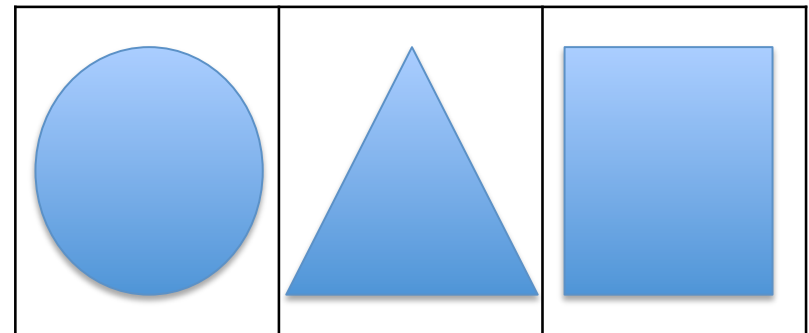
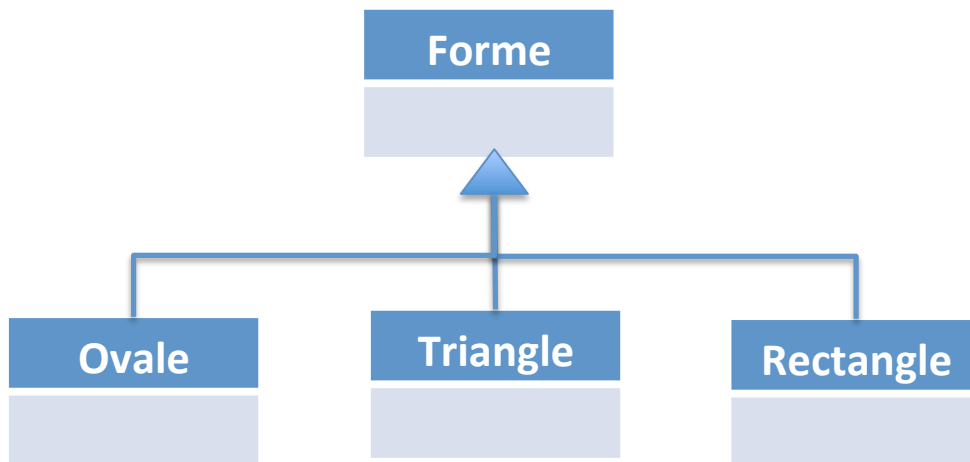
- C'est un mécanisme permettant d'invoquer la « même » méthode sur des objets de nature « différentes » donc retourner des résultats / actions différent(e)s
- Basé sur l'utilisation des méthodes polymorphes
- Méthode polymorphe ?

# Méthode polymorphe

- C'est une méthode qui a été déclarée dans la classe mère et redéfinie dans une (ou plusieurs) classes filles
  - Les méthodes polymorphes, redéfinies dans plusieurs classes filles, portent la même signature et prévoient des actions similaires
- Une même écriture (instruction) retourne des résultats différents donc correspond à des actions différentes
- Le résultat retourné dépend de la nature de l'objet qu'invoque la méthode polymorphe
  - L'objet correspondant peut être un objet de la classe fille ou de la classe mère
  - Une méthode finale ne peut être une méthode polymorphe

# Tableau polymorphe (1/3)

- C'est un tableau dont le type de référence et le type de la classe mère, comportant des objets de nature différentes, instances de classes filles différentes



```
Forme [] formes = new Forme[3];  
formes[0] = new Ovale();  
formes[1] = new Triangle();  
formes[2] = new Rectangle();
```

## Tableau polymorphe (2/3)

```
public class Forme {  
    public void afficherDetails(){  
        System.out.println("C'est une forme géométrique");  
    }  
}
```

```
public class Ovale extends Forme{  
    public void afficherDetails(){  
        System.out.println("C'est un oval");  
    }  
}
```

```
public class Triangle extends Forme{  
    public void afficherDetails(){  
        System.out.println("C'est un triangle");  
    }  
}
```

```
public class Rectangle extends Forme{  
    public void afficherDetails(){  
        System.out.println("C'est un rectangle");  
    }  
}
```

# Tableau polymorphe (3/3)

```
public class TestForme {  
    public static void main(String[] args){  
        Forme[] formes = new Forme[3];  
        formes[0] = new Ovale();  
        formes[1] = new Triangle();  
        formes[2] = new Rectangle();  
        for(int i=0;i<formes.length;i++)  
            formes[i].afficherDetails();  
    }  
}
```

La même écriture, lancée  
sur des objets de natures  
différentes, provoque  
des actions différentes

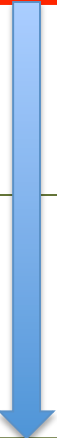


Console X @ Javadoc Declaration

<terminated> TestForme [Java Application] /Library/  
C'est un oval  
C'est un triangle  
C'est un rectangle  
|



# Argument polymorphe

```
public class Dessin {  
    public void dessiner(Forme f){  
        f.afficherDetails();  
    }  
}
```



Forme peut être un rectangle,  
triangle ou oval (n'importe quel  
classe fille) → **argument  
polymorphe**

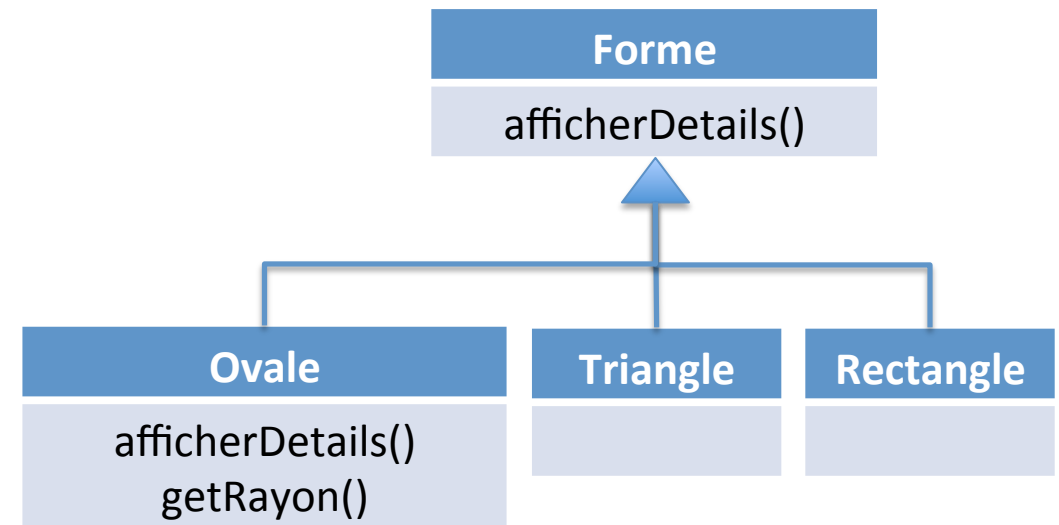
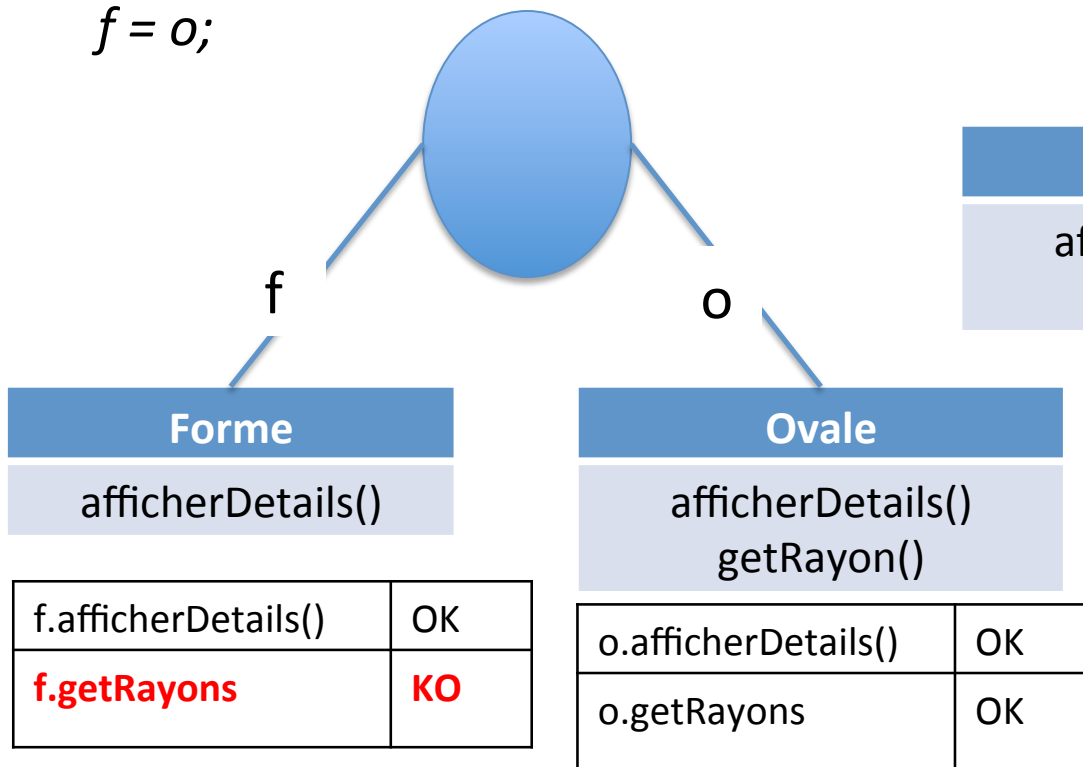
```
public class TestForme {  
    public static void main(String[] args){  
        Ovale o = new Ovale();  
        Rectangle r = new Rectangle();  
        Triangle t = new Triangle();  
        Dessin d = new Dessin();  
        d.dessiner(o);  
        d.dessiner(t);  
    }  
}
```



C'est un oval  
C'est un triangle

# Sur-classement et substitution

```
Forme f = new Forme();
Ovale o = new Ovale();
f = o;
```



- L'objet o (Ovale) est « surclassé ». Il est vu de type Forme de la référence déclarée (Forme)
- Les fonctionnalités de o sont limitées à celles de Forme.

→ Substituer Forme (f) par Ovale : **((Ovale) f).getRayon()**

# Polymorphisme : Liaison retardée

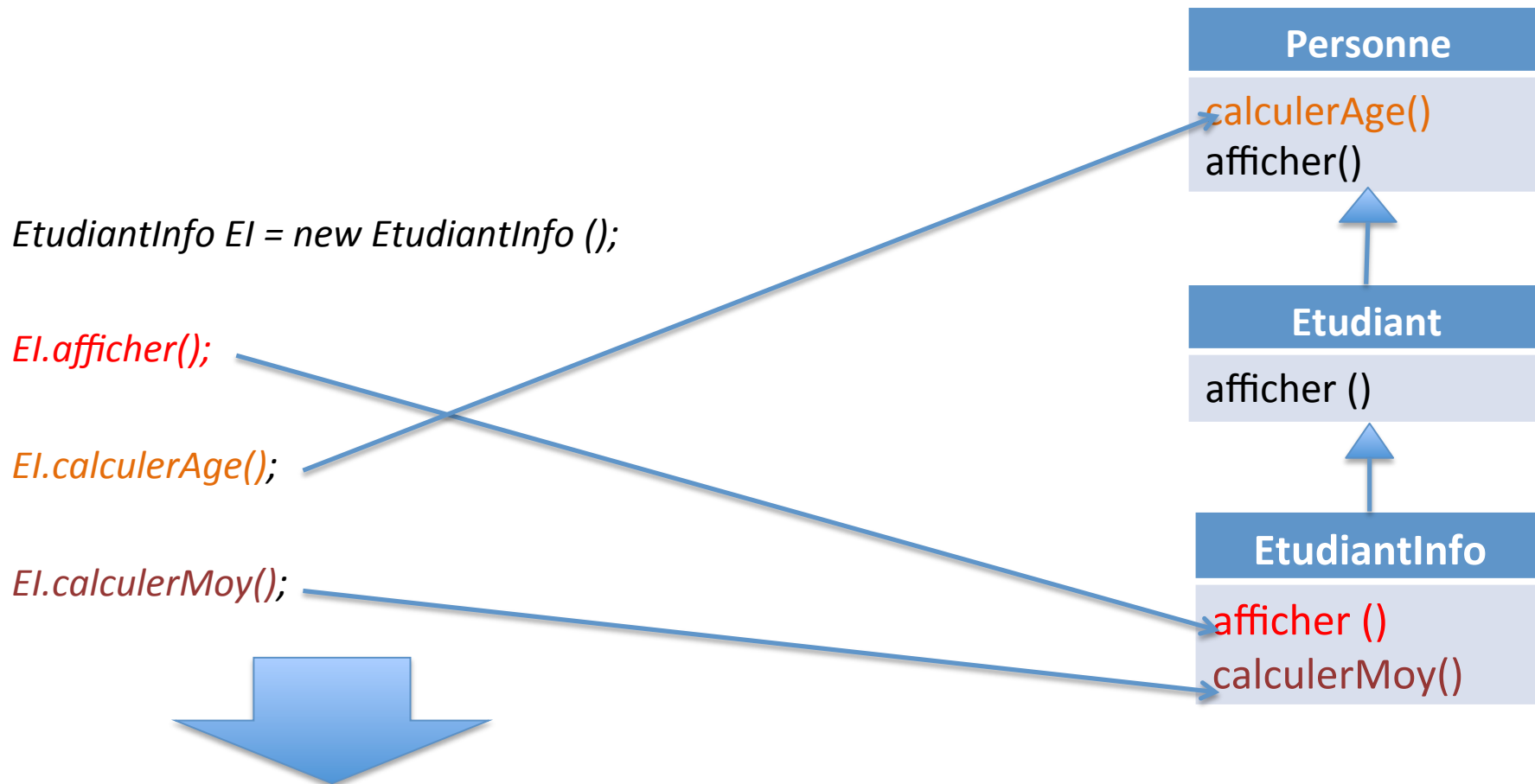
- Le polymorphisme est basé sur le mécanisme dynamique de liaison retardée (late binding)
- On détermine la méthode à exécuter :
  - Au moment de l'exécution
  - En se basant sur le type réel de l'objet qu'a invoqué la méthode et non pas le type de la référence de l'objet



# Liaison retardée

- Soit l'objet *o*, une instance de la classe *C*. Soit la méthode *meth()*. On lance l'instruction « ***o.meth()*** »
  - Si la classe *C* inclut la définition / redéfinition de *meth()*, *meth()* de la classe *C* sera invoquée
  - Sinon, le mécanisme de liaison retardée continue à chercher *meth()* dans la super-classe de *C*, ensuite dans la classe mère de cette super-classe, et ainsi de suite, jusqu'à trouver la définition de *meth()* qui sera appelée

# Liaison retardée : exemple



La JVM commence la recherche de la méthode dans la classe elle-même (***EtudiantInfo***) et ensuite elle remonte dans la hiérarchie (de la classe fille vers la classe mère), jusqu'à trouver la bonne méthode

# Avantages du polymorphisme

- Le polymorphisme réduit le nombre de lignes de code
- Le polymorphisme permet l'extension faciles des programmes
- Grâce au polymorphisme, il est possible de créer des classes filles dont les instances peuvent être traitées sans modifier le code déjà développé

# Exemple

```
class Volaille{
    public void quiEtesVous()
    {System.out.println("Volaille");}
}

class Caille extends Volaille{
    public void quiEtesVous()
    {System.out.println("Caille");}
}

class Pigeon extends Volaille{
    public void quiEtesVous()
    {System.out.println("Pigeon");}
}
```

```
public class PolymorphismeExemple{
    public static void main(String [] args){
        Volaille v;
        int x;
        System.out.println("Taper 1,2 ou 3");
        x=Clavier.lireInt();
        if (x==1)
            v = new Volaille();
        else
            if (x == 2)
                v = new Caille();
            else
                v = new Pigeon();
        //appel polymorphe
        v.quiEtesVous();
    }}
}
```

# Bilan

- Super Classe et Sous Classe
- Le mot clé super
- Héritage et constructeurs
- Re-définition des méthodes
- Surcharge des méthodes
- Le mot clé « final »
- La classe « Object » et la méthode « equals »
- Méthode polymorphe
- Tableau polymorphe
- Argument polymorphe
- Sur-classement et substitution
- Mécanisme de liaison retardée

# Chapitre 5 : Classes Abstraites et Interfaces

# Objectifs

- Maitriser les classes abstraites
- Maitriser les interfaces
- Maitriser les différences entre les classes abstraites et les interfaces

# Plan

- Classes abstraites
- Interfaces
- Classes abstraites vs. Interfaces



# Classes abstraites (1/2)

- Méthode abstraite :
    - Déclarée avec le mot clé « **abstract** »
    - Son corps est remplacé par « ; »
    - Exemple : **abstract** void afficher();
  - Classe abstraite
    - contient une (au moins) ou plusieurs méthode(s) abstraite(s)
    - peut inclure des constructeurs mais **ne peux JAMAIS être instanciée**
- une classe qui inclut une méthode abstraite est une classe abstraite

# Classes abstraites (2/2)

```
public abstract class Forme {  
    int x;  
    int y;  
    Forme(int x,int y){  
        this.x = x; this.y = y; }  
    abstract void afficherDetails();  
    void redimensionner(int d){ x = x+d; y = y+d;}}
```

```
public class Cercle extends Forme{  
    float rayon;  
    Cercle(int x, int y, float rayon){  
        super(x,y); this.rayon = rayon;}  
    void afficherDetails(){  
        System.out.println("C'est un cercle de rayon "+rayon+  
". Les coordonnées du centre sont ("+x+", "+y+")");}}
```

```
public class TestForme {  
    public static void main(String[] args) {  
        Cercle c = new Cercle(5,5,10);  
        c.afficherDetails();  
        c.redimensionner(1);  
        c.afficherDetails();}}
```

# Interfaces (1/2)

- Définition
  - Une interface inclut
    - La déclaration des constantes,
    - La déclaration de méthodes abstraites.
  - Une classe peut implémenter (hérite de) beaucoup d'interfaces en même temps.
  - Les interfaces permettent de remplacer l'héritage multiple, qui n'est pas permis en JAVA !

# Interfaces (2/2)

```
public interface Affichage {  
    public void afficher();  
}
```

```
public class Point implements Affichage{  
    public void afficher() {  
        System.out.println("C'est un point");  
    }  
}
```

```
public class Losange implements Affichage{  
    public void afficher() {  
        System.out.println("C'est un losange");  
    }  
}
```

# Classes abstraites Vs. Interfaces

- Classe abstraite
    - peut inclure des attributs (<> constantes)
    - peut inclure des méthodes qui possèdent une implémentation
    - Mais, il est nécessaire d'utiliser la notion d'héritage
  - Interface
    - Permet au développeur d'implémenter (hériter de) beaucoup d'interfaces à la fois
    - Une interface n'inclut ni attributs (sauf constantes) ni des méthodes implémentées
- Le choix dépend de votre cas / problème !
- Faites un bon choix 😊

# Bilan

- Classes abstraites
- Interfaces
- Différences entre classes abstraites et interfaces

# Chapitre 6 : Les collections

# Objectifs

- Maitriser les collections
- Maitriser ArrayList et HashMap



# Plan

- Introduction
- Types de collections
- Classes étudiées
- ArrayList
- HashMap

# Introduction

- Collection = objet rassemblant plusieurs éléments dans une entité comme le tableau.
- L'objet Collection permet de :
  - Stocker des données
  - Traiter des données
- Le JDK offre plusieurs types de collections qui se présentent généralement sous forme de classes et interfaces
- Les classes et interfaces de Collection appartiennent au package `java.util`
- Avant l'apparition du JDK 5.0, les objets appartenant à un objet Collection sont déclarés en tant que « Object »
- Depuis l'apparition du JDK 5.0, on peut préciser le type des objets inclus dans une collection

# Types de collections

Il existe 2 hiérarchies principales :

- Collection<E>
- Map<K,V>
- **Collection<E>** : Il s'agit des interfaces des collections « rigoureusement dites »
- **Map<K,V>** :
  - Il s'agit de collections dont les éléments ont la forme (clé , valeur)
  - Chaque valeur **V** est indexées par une clé **K**
  - Ainsi, un élément de type **V** d'une Map peut être rapidement trouvé par sa clé de type **K**

# Classes étudiées

Dans ce module, nous nous intéressons principalement aux classes ***ArrayList*** (implantation de ***Collection***) et ***HashMap*** (implantation de ***Map***)

- Pour utiliser ***ArrayList***, on doit l'importer à travers  
« ***import java.util.ArrayList;*** »
- Pour utiliser ***HashMap***, on doit l'importer à travers  
« ***import java.util.HashMap;*** »

# ArrayList (1/2)

```
package chcollections;
import java.util.ArrayList;
import java.util.Arrays;
public class ArrayListTestLFSI2 {
    public static void main(String[] args) {
        ArrayList<Personne> listpersonnes = new ArrayList<Personne>();
        //add (ajouter un élément à la liste)
        Personne p1 = new Personne("Mdhaaffar", "Afef");
        Personne p2 = new Personne("Kolsi", "Ali");
        Personne p3 = new Personne("Hafedh", "Abdelhalim");
        Personne p4 = new Personne("Klai", "Mohamed Ali");
        listpersonnes.add(p1);
        listpersonnes.add(p2);
        listpersonnes.add(p3);

        //contains (tester si la liste contient un élément)
        listpersonnes.contains(p1);
        System.out.println("Est ce que la liste contient 'Afef Mdhaaffar' :"+listpersonnes.contains(p1));
        System.out.println("Est ce que la liste contient 'Mohamed Ali Klai' :"+listpersonnes.contains(p4));

        //size (retourner la taille de la liste)
        System.out.println("la taille de l'ArrayList est : " + listpersonnes.size(););
    }
}
```

# ArrayList (2/2)

//remplacer un objet de la liste par un autre

```
System.out.println("avant l'update : " + listpersonnes);
```

```
listpersonnes.set(2, p4);
```

```
System.out.println("après l'update : "+ listpersonnes);
```

//tester si ArrayList est vide (empty)

```
System.out.println("liste vide ? " + listpersonnes.isEmpty());
```

// supprimer un objet de l'ArrayList

```
System.out.println("avant la suppression : "+ listpersonnes);
```

```
listpersonnes.remove(2); //removing third object
```

```
System.out.println("après la suppression : " + listpersonnes);
```

// trouver l'indice (index) d'un objet dans la liste

```
System.out.println("Quel est l'indice (index) de 'Afef Mdhaffar' : " + listpersonnes.indexOf(p1));
```

// convertir la liste en un Array

```
Personne[] array = listpersonnes.toArray(new Personne[1]);
```

```
System.out.println("Array from ArrayList : " + Arrays.toString(array));
```

// supprimer tous les elements de la liste

```
listpersonnes.clear();
```

```
System.out.println("taille de l'ArrayList après clear : " + listpersonnes.size());}}
```

# Classe Personne

```
package chcollections;

/**
 * @author mdhaffar
 */
public class Personne {
    String nom;
    String prenom;
    Personne(String nom, String prenom){
        this.nom = nom;
        this.prenom = prenom;
    }
    public String toString(){
        return(nom+" "+prenom);
    }
}
```

# Outputs

Est ce que la liste contient 'Afef Mdhaffar' :true

Est ce que la liste contient 'Mohamed Ali Klai' :false

la taille de l'ArrayList est : 3

avant l'update : [Mdhaffar Afef, Kolsi Ali, Hafedh Abdelhalim]

après l'update : [Mdhaffar Afef, Kolsi Ali, Klai Mohamed Ali]

liste vide ? false

avant la suppression : [Mdhaffar Afef, Kolsi Ali, Klai Mohamed Ali]

après la suppression : [Mdhaffar Afef, Kolsi Ali]

Quel est l'indice (index) de 'Afef Mdhaffar' : 0

Array from ArrayList : [Mdhaffar Afef, Kolsi Ali]

taille de l'ArrayList après clear : 0



# HashMap(1/2)

```
package chcollections;
import java.util.HashMap;
import java.util.Iterator;
public class HashMapTestLFSI2{
    public static void main(String args[]){
        HashMap<String, Personne> hashMaptest = new HashMap<String, Personne>();
        // ajouter put(key,value) au HashMap
        Personne p1 = new Personne("Mdhaftar", "Afef");
        Personne p2 = new Personne("Kolsi", "Ali");
        Personne p3 = new Personne("Hafedh", "Abdelhalim");
        Personne p4 = new Personne("Klai", "Mohamed Ali");
        hashMaptest.put("Ali", p2);
        hashMaptest.put("Halim", p3);
        hashMaptest.put("Klai", p4);
        //size
        System.out.println("HashMap contient "+ hashMaptest.size()+" éléments");
        //containsValue
        if(hashMaptest.containsValue(p1))
            System.out.println("HashMap contient 'Afef Mdhaftar' comme value");
        else
            System.out.println("HashMap ne contient pas 'Afef Mdhaftar' comme value");
    }
}
```

# HashMap(2/2)

## //containsKey

```
if( hashMaptest.containsKey("Ali") )  
    System.out.println("HashMap contient Ali comme key");  
else  
    System.out.println("HashMap ne contient pas Ali comme key");
```

## //get(key)

```
Personne ali = (Personne) hashMaptest.get("Ali");  
System.out.println("Value mapped avec key \"Ali\" est " + ali);
```

## //Iterator pour les clés (keySet)

```
System.out.println("Les clés (keys) du HashMap sont : ");  
Iterator iterator = hashMaptest.keySet().iterator();  
while(iterator.hasNext())  
    System.out.println(iterator.next());
```

## //Iterator pour les valeurs (entrySet)

```
System.out.println("Les valeurs (values) du HashMap sont : ");  
iterator = hashMaptest.entrySet().iterator();  
while(iterator.hasNext())  
    System.out.println(iterator.next());
```

## //remove

```
System.out.println(hashMaptest.remove("Halim") + " n'est plus dans HashMap.");
```

## //Affichage après la suppression

```
iterator = hashMaptest.entrySet().iterator();  
while(iterator.hasNext())  
    System.out.println(iterator.next()); } }  
©Afef Mdhaffar
```

# Outputs

HashMap contient 3 éléments

HashMap ne contient pas 'Afef Mdhaftar' comme value

HashMap contient Ali comme key

Value mapped avec key "Ali" est Kolsi Ali

Les clés (keys) du HashMap sont :

Halim

Klai

Ali

Les valeurs (values) du HashMap sont :

Halim=Hafedh Abdelhalim

Klai=Klai Mohamed Ali

Ali=Kolsi Ali

Hafedh Abdelhalim n'est plus dans HashMap.

Klai=Klai Mohamed Ali

Ali=Kolsi Ali

# Bilan

- Collections
- ArrayList
- HashMap

# Chapitre 7 : Les exceptions

# Objectif : Maitriser les exceptions en Java

## Plan

- Notion d'exceptions
- Les exceptions
- Arbre des exceptions
- Nature des exceptions
- Quelques exceptions prédéfinies en Java
- Capture d'une exception
- Interception Vs. Propagation des exceptions
- Le mot clé « throws »

# Notion d'exception

- Les erreurs, qui se déclenchent lors de l'exécution d'un programme Java, prennent la forme d'exceptions.
- Une exception :
  - est une instance (objet) d'une classe d'exception
  - provoque l'arrêt du programme
  - est un type d'erreur
  - inclut des détails sur l'erreur produite

# Exceptions

- Exception : situation exceptionnelle provoquant l'arrêt du programme (impossibilité, erreur)
- Un objet de type **Exception** est une « bulle » logicielle, qui se produit dans une situation particulière
- Une fois un Objet **Exception** est produit, il remonte la pile d'exécution jusqu'à trouver une portion de code permettant de le traiter ou arrêter le programme
- Le programme s'arrête généralement suite à une exception s'il n'existe pas une portion de code permettant de traiter cette exception
- En cas de rupture du programme, les détails de l'exception seront affichés



# Terminologie

- Une exception correspond à une alerte indiquant qu'une situation exceptionnelle est survenue au moment de l'exécution

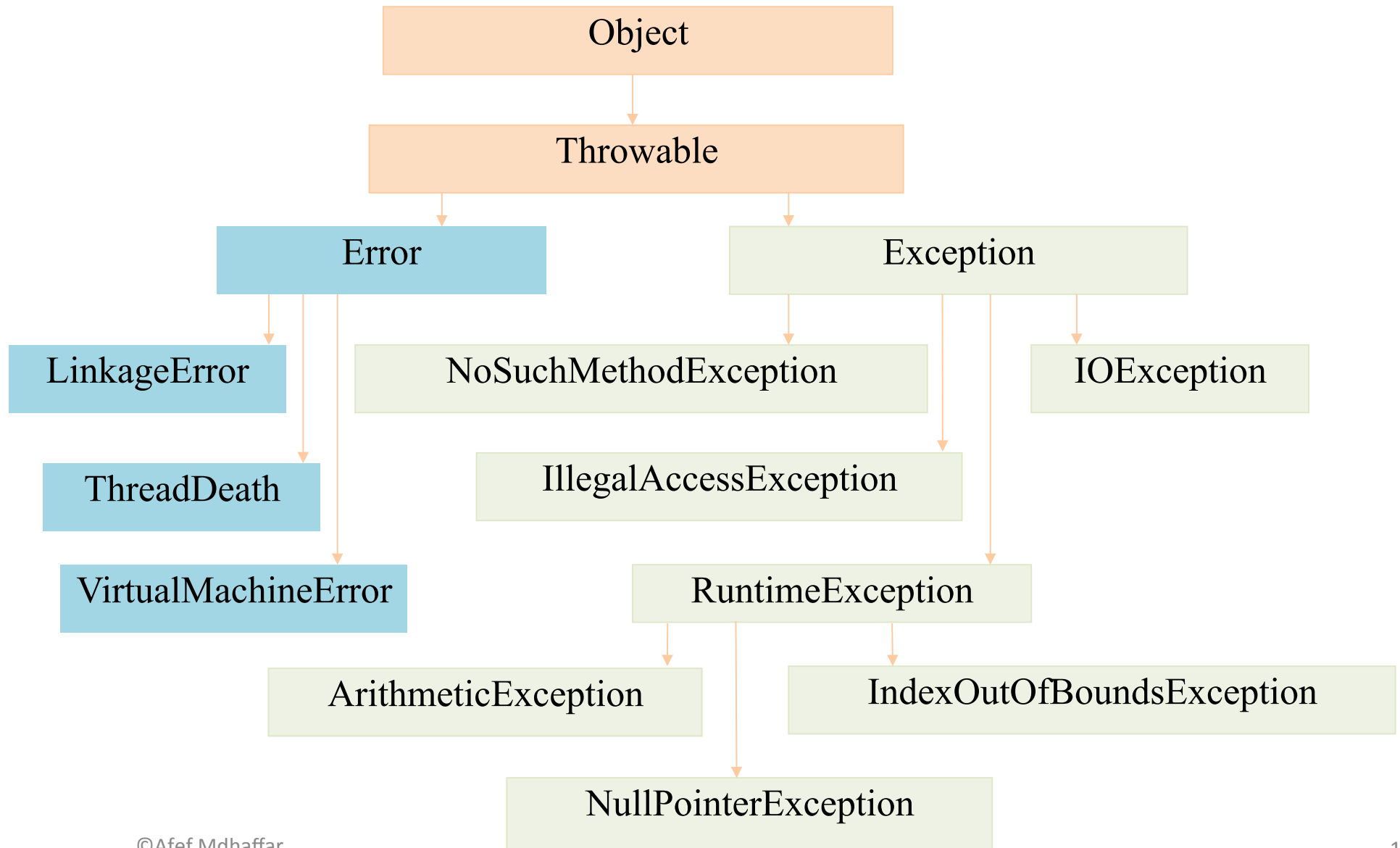
→ 2 solutions

→ laisser le programme s'arrêter avec une erreur

→ essayer de ne pas interrompre le programme et assurer son exécution normale, malgré l'exception

- **Lever une exception** : Il s'agit de signaler l'erreur (exception)
- **Capter l'exception** : Il s'agit de traiter l'exception

# Arbre des exceptions (1/2)



# Arbre des exceptions (2/2)

- Throwable : C'est la super-classe de de toutes les exceptions
- Error : Elle est responsable de la gestion des erreurs de la machine virtuelle, comme LinkageError et ThreadDeath
- Exception : C'est la classe mère de toutes les exceptions gérées par le développeur comme ArithmeticException
- RuntimeException : C'est la classe qui rassemble les erreurs de base comme Arithmetic
- IOException : C'est la classe qui rassemble les erreurs Inputs/Outputs (entrées/sorties)

# Nature des exceptions

- Les classes exceptions « prédéfinies » et « développées par l'utilisateur » héritent de la classe *Exception*
- *RuntimeException, Error* sont des exceptions (erreurs) prédéfinies et gérées par JVM

# Exceptions prédéfinies en Java

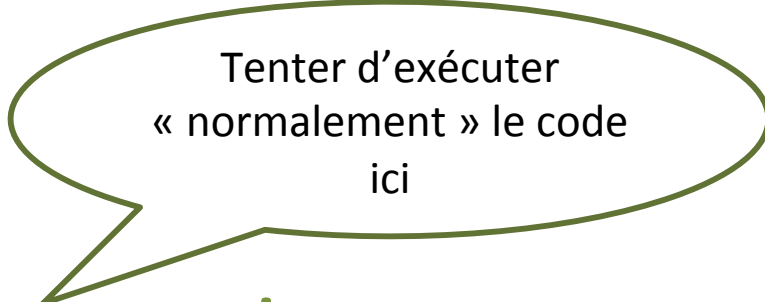
- **ArithmeticException** : gère les erreurs arithmétiques liés à la division par zéro (cas des entiers)
- **ArrayIndexOutOfBoundsException** : Cette exception est déclenchée si la taille limite d'un tableau a été dépassée
- **NegativeArraySizeException** : Tentative de création d'un tableau ayant une taille (longueur) négative
- **NullPointerException** : Cette exception est déclenchée en cas de référence nulle
- **ClassCastException** : Cette exception est déclenchée si une tentative de forçage de type illégale a été détectée

# Capture d'une exception


- Pour capturer une exception, on utilise les blocs `try {}` et `catch {}` dans une méthode. Ceci permettra d'attraper la bulle logicielle de l'exception.

- Exemple :

```
public void methode(arguments) {  
    try{  
        //code susceptible de lancer une exception  
    }  
    catch {  
        //code permettant de gérer l'exception  
    }  
}
```



Tenter d'exécuter  
« normalement » le code  
ici



Attraper l'exception ici

# try / catch / finally

```
try
{
    //code
}
catch (<exception_number1>)
{
    //code
}
catch (<exception_number1>)
{
    //code
}
```

**//On peut mettre autant de blocs catch que l'on souhaite**

```
finally
{
    //Bloc finally est facultatif
}
```

# Traitement des exceptions (1/2)

- Les instructions de la section **try {}** s'exécutent jusqu'à ce qu'il se terminent avec succès ou bien qu'une exception soit déclenchée
- Si une exception est déclenchée, les clauses **catch {}** sont examinées l'une après l'autre pour trouver une clause **catch {}** traitant la classe (ou super-classe) de l'exception déclenchée
- La JVM commence par examiner les clauses **catch {}** des exceptions les plus spécifiques tout en allant aux exceptions les plus générales
- Si la clause **catch {}** adéquate à cette exception a été retrouvée et que son bloc a été exécuté, le programme reprend son cours



# Traitement des exceptions (2/2)

- Si les exceptions déclenchées ne sont pas capturées « immédiatement » par une section **catch{}**, elles se propagent tout en remontant la pile d'appels (calls) des méthodes, jusqu'à être gérées
- Si une exception n'a jamais été capturée, elle se propage jusqu'au programme principal (méthode **main()**), ceci incite l'interpréteur Java à arrêter l'exécution du programme tout en affichant un message d'erreur
- L'interpréteur Java affiche un message identifiant :
  - l'exception,
  - la méthode qui a provoqué cette exception,
  - Le numéro de la ligne correspondante dans le fichier.

## Section « finally »

- Le bloc **finally** permet aux développeurs de définir un ensemble d'instructions qui s'exécutera dans tous les cas, que l'exception soit déclenchée ou pas, capturée ou pas
- L'unique instruction (commande) qui peut entraîner que les instructions d'un bloc **finally** ne soient pas exécutées est **System.exit()**

# Propagation vs. Interception

Si une méthode peut déclencher une exception  
(ou appelle une autre qui déclenche une autre), il faut :

➤ **propager** l'exception (la méthode doit déclarer  
l'exception)

**Ou**

➤ **intercepter** et traiter l'exception

# Propager une exception : exemple

```
public int soustraire(int x, String str) throws NumberFormatException
{
    int y = Integer.parseInt(str);
    x = x - y;
    return x;
}
```

# Interceptor une exception : exemple

```
public int soustraire(int x, String str)
{
    try {
        int y = Integer.parseInt(str);
        x = x - y;
    }
    catch (NumberFormatException e1)
    {
        System.out.println(e1.getMessage());
    }
    return x;
}
```

# Checked /unchecked exceptions

- **Error et RuntimeException** font partie de la catégorie «**unchecked**». Ainsi, il n'est pas nécessaire de mettre le mot clé « throws » à côté de la méthode qui déclenche une exception faisant partie de cette catégorie
- Toutes les autres exceptions (exceptions prédéfinies et celles développées par le développeur) font partie de la catégorie "**checked**". Ainsi, l'utilisation de "throws" est exigée dans ce cas

# Les objets **Exception**

- La classe **Exception** étend la classe **Throwable**
- La classe **Throwable** déclare un message (attribut) de type **String**, qui va être hérité par toutes les autres classes d'exception
- Cet attribut sert à stocker le **message** qui décrit l'exception
- Le constructeur de la classe **Exception** permet de définir le contenu de l'attribut « **message** »
- L'attribut « **message** » peut être récupéré en utilisant la méthode « **String getMessage()** »

# Ma première classe Exception !

```
package chexceptions;  
public class MyFirstException extends Exception {  
    public MyFirstException(){  
        super();  
    }  
    public MyFirstException(String s)  
    {  
        super(s);  
    }  
}
```



# Lever des exceptions

- Le développeur a la possibilité de lever ses propres exceptions en utilisant mot clé **throw**.
- **throw** prend comme paramètre une instance de la classe **Throwable** ou d'une de ses classes filles
- Les objets de type « Exception » sont généralement créés au niveau de la même instruction qui assure leur lancement.
- Exemple :  
**throw new MyFirstExecption( »Une exception a eu lieu !!!");**

# Emission d'une exception

- On utilise le mot clé « **throw** » pour lever une exception
- On ajoute le mot clé « throws » suivi du nom de la classe Exception à la signature d'une méthode qui est susceptible de lever une exception
- exemple :

```
public void openFile(String fileName) throws MyFirstException{  
    if (fileName==null)  
        throw new MyFirstException("nom du fichier invalide");  
    else{  
        //code qui doit s'exécuter dans le cas habituel  
    }  
}
```

# Le mot clé throws (1/2)

- Le mot clé « throws » est utilisé à fin de « laisser remonter » une exception, qu'il ne veut pas traiter, à la méthode appelante
- Une même méthode peut "laisser remonter" plusieurs types d'exceptions (séparés par des virgules « , »).

```
public void uneMethode() throws IOException, MyFirstException
{
    // ne gère pas l'exception IOException
    // mais peut la provoquer
}
```

# throws (2/2)

- Les développeurs doivent connaître les exceptions qu'une méthode peut lever pour planifier des blocs **Try{} / catch()** lors de l'appel de cette méthode
- Une méthode doit traiter ou "laisser remonter" toutes les exceptions qui peuvent être générées dans les méthodes qu'elle appelle
- Si une méthode lève une exception, elle doit capturer cette Classe d'Exception ou bien une des super-classes de la classe Exception correspondante ou la classe mère de toutes les exception (**Exception**)

# Bilan

- Les exceptions permettent à Java de bien gérer les erreurs → code « robuste »
- Une méthode peut lever des exceptions au moment opportun
- Le programme capture et traite les exceptions grâce aux sections ***try{} / catch{}***
- Le développeur peut définir ses classes d'exceptions

# Références et sources Web

- JAVA Tutorial, <https://www.tutorialspoint.com/java/>, 2017
- S. Laporte, Introduction à JAVA, Lycée Louise Michel BTS IG DA, pages 1 – 7, <http://stephanie.laporte.pagesperso-orange.fr/Pdf/INTRODUCTION%20A%20JAVA.pdf>, 2017
- Anne Tasso, Le livre de JAVA, EYROLLES, pages 1 – 450, 2005
- [http://www.irisa.fr/imadoc/equipe/anquetil/images/PreSpelnitCPOOJava\\_V2.pdf](http://www.irisa.fr/imadoc/equipe/anquetil/images/PreSpelnitCPOOJava_V2.pdf)
- <https://www.ukonline.be/cours/java/apprendre-java/chapitre5-6>
- <http://docplayer.fr/52022117-Les-collections-dans-java-2.html>
- [http://www.mcours.net/cours/pdf/info/Programmation\\_avec\\_le\\_langage\\_Java.pdf](http://www.mcours.net/cours/pdf/info/Programmation_avec_le_langage_Java.pdf)
- <https://fr.slideshare.net/MohamedLahmer3/cours-java-v13>
- *François Bonneville, **Traitement des erreurs en Java***, Laboratoire d'Informatique de Besançon