PROGRAMMATION ORIENTÉE OBJET JAVA

LICENCE 2 INGÉNIERIE - INFORMATIQUE

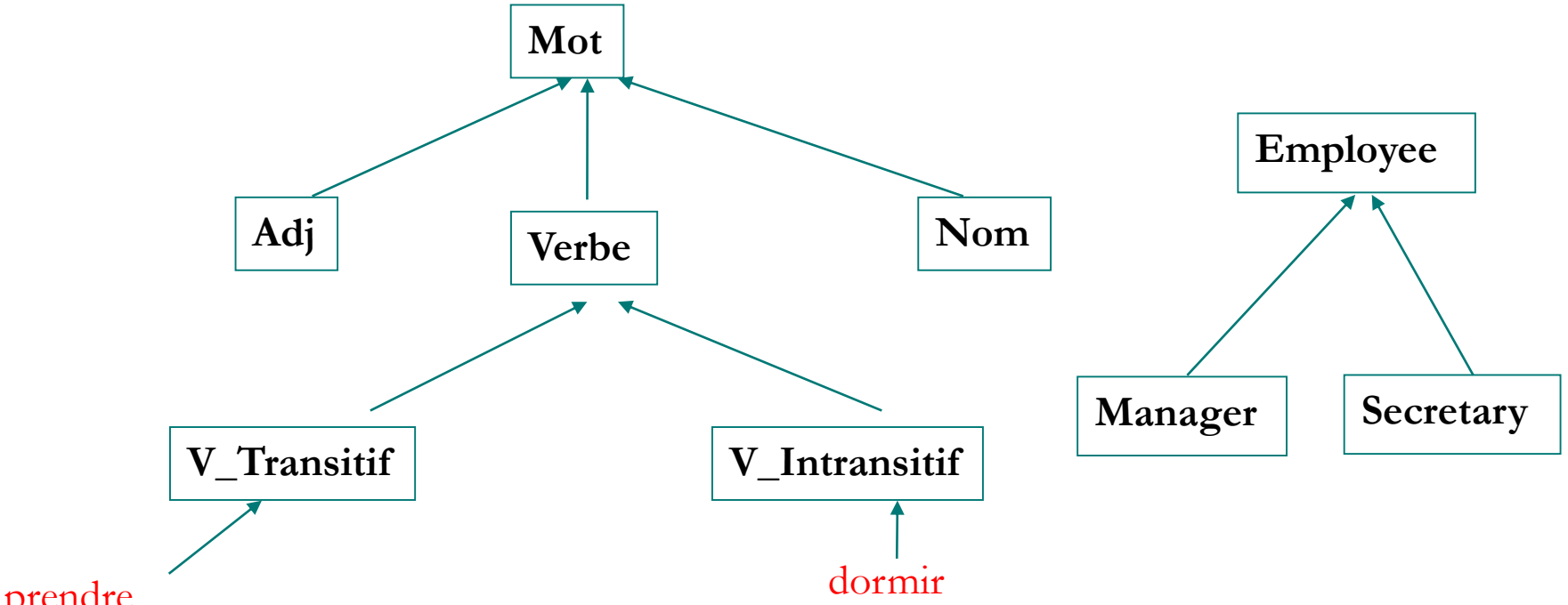
2020– 2021

Marie NDIAYE

HÉRITAGE

OBSERVATION

- On peut souvent classer les objets dans des arborescences de types de plus en plus génériques:

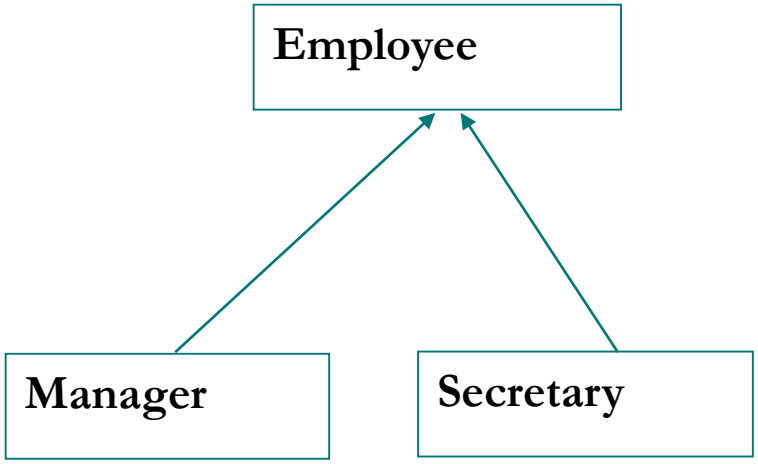


EXEMPLE : BESOIN

- La classe **Employee** représente toutes sortes d'employés.
- On pourrait définir un manager comme un employé.
- Mais un manager a ses propres spécificités.
- On pourrait créer une classe **Manager** qui ressemble à la classe **Employee**.
- Mais on ne veut pas réécrire, ou à chaque fois mettre à jour tout ce qu'elles ont en commun.

EXEMPLE : SOLUTION

- La classe **Employee** contient tout ce qu'il y a de commun aux employés.
- **Manager** ne contient que ce qu'il y a de spécifique aux managers.



HIÉRARCHISATION DE CLASSES

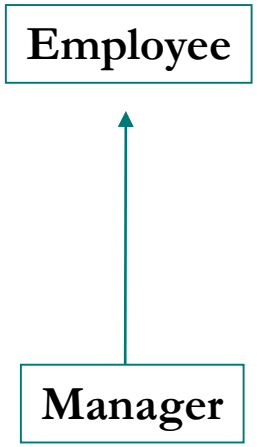
- Un des concepts fondamentaux de la programmation orientée objets est la possibilité de définir une hiérarchie de types/classes.

```
class Y extends X { ..... }
```

- La classe X est supposée définie
  - Y est une sous-classe ou classe fille (directe) de X.
  - X est la super-classe ou la classe mère de Y.
- **Java** : héritage simple
  - Pas d'héritage multiple comme en C++ (mais peut être atteint en utilisant les **interfaces**)

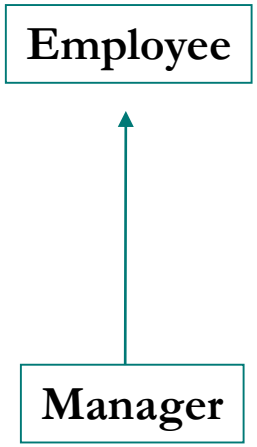
EXEMPLE : CLASSE EMPLOYEE

```
public class Employee {

    private String name;
    private double salary;

    public Employee (String n, double s){
        name = n;
        salary = s; }
    public String getName(){
        return name; }
    public double getSalary() {
        return salary; }
}
```



EXEMPLE : CLASSE MANAGER

```
public class Manager extends Employee {
    private double bonus;

    public Manager (String n, double s){
        super(n, s);
        bonus = 0; }
    public void setBonus (double b){
        bonus = b; }
    //Redéfinition de la méthode getSalary
    public double getSalary (){
        double baseSalary = super.getSalary();
        return baseSalary + bonus; }
}
```



LA MÉTHODE **SUPER()**

- **super()** permet d'appeler le constructeur de la classe mère.
- **Première chose à faire** dans la construction d'une sous-classe (pour l'initialisation des arguments de la classe mère).
- On passe les paramètres nécessaires.
- Par défaut : Le constructeur de la classe mère est appelé.

```
public Manager (String n, double s){
    super(n, s);
    bonus = 0;
}
```

LE MOT CLÉ SUPER **SUPER**

- **super** peut être vu comme une référence à la super-classe.
- Mot clé spécial qui demande au compilateur d'invoquer la méthode de la super-classe.
- A différencier de **this** qui fait référence à l'objet courant.

```
public double getSalary {
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
}
```

EN RÉSUMÉ …

- On souhaite ne décrire qu'une seule fois le même traitement lorsqu'il s'applique à plusieurs classes.
- Une classe spécifique **hérite** des méthodes et des attributs **public**, **protected** et **private** de sa classe mère.
- La classe fille ne peut pas accéder aux membres déclarés **private** dans la classe mère.
- On peut cependant **redéfinir** une méthode de la classe mère dans la classe fille (**de même signature**).

POLYMORPHISME

C'EST QUOI LE POLYMORPHISME ?

- Le terme **polymorphisme** décrit la caractéristique d'un élément qui peut prendre **plusieurs formes**, comme l'eau qui se trouve à l'état solide, liquide ou gazeux.
- En programmation Objet, on appelle polymorphisme
  - Le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe. **Surcharge**
  - Le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie. **Redéfinition**

SURCHARGE DE MÉTHODE

- La surcharge de méthode consiste à définir plusieurs fonctions avec le **même nom**, mais avec une **signature (nom de la méthode, nombre et types de paramètres) différente** dans une **même classe**.

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

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

REDÉFINITION DE MÉTHODE

- Usage de méthode avec le **même nom** et la **même signature** dans **une classe et ses classes filles**.
- Distinction sur l'objet réel dont la méthode est appelée.

```
public double getSalary (){//redéfinition
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
}
```

REDÉFINITION DE MÉTHODE : LIAISON DYNAMIQUE

- Le **type déclaré** est le type "compris" par le compilateur.
- Le **type réel** est le type "considéré" par la machine virtuelle Java à l'exécution.
- Lorsqu'une méthode d'un objet est accédée au travers d'une référence "surclassée", c'est la méthode telle qu'elle est définie au niveau de la classe effective de l'objet qui est en fait invoquée et exécutée.

REDÉFINITION DE MÉTHODE : LIAISON DYNAMIQUE – EXEMPLE

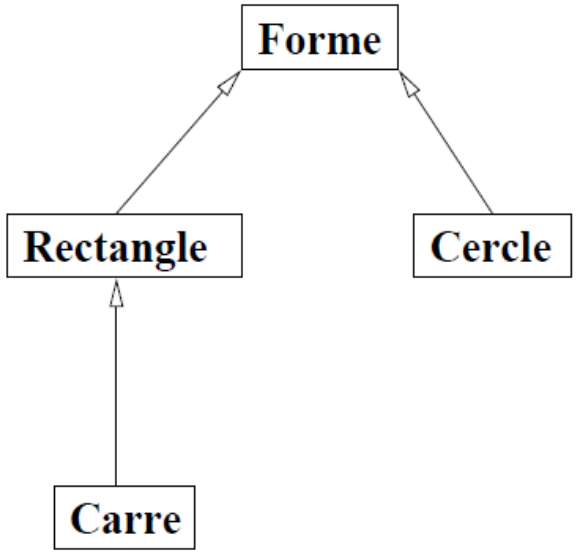
```
Manager manager = new Manager("Joseph SAGNA", 5000);
Employee employe = manager; // sur-classement de manager
employe.setBonus(2000); // ☹ pas accepté par le compilateur
double salaire = employe.getSalary ();
/* C'est la méthode getSalary() de la classe Manager qui va être invoquée.*/
```

EN RÉSUMÉ …

- En utilisant le polymorphisme …
  - Plus besoin de distinguer différents cas en fonction de la classe des objets.
  - Possible de définir de nouvelles fonctionnalités en héritant de nouveaux types de données à partir d'une classe de base commune sans avoir besoin de modifier le code qui manipule l'interface de la classe de base.
- Développement plus rapide
- Programmes plus facilement extensibles
- Maintenance du code plus aisée

EXERCICE D'APPLICATION 1/4

- **Une hiérarchie de classes**
  - Considérons les classes Forme, Carre, Rectangle et Cercle. L'objectif de cet exercice est d'organiser ces classes de sorte que Carre hérite de Rectangle qui hérite, ainsi que Cercle, d'une classe Forme.
  - Pour le moment, nous considérerons la classe Forme comme vide (c'est-à-dire sans aucun attribut ni méthode) et nous nous intéressons plus particulièrement aux classes Rectangle et Carre.



EXERCICE D'APPLICATION 2/4

- **La classe Rectangle**
  - La classe Rectangle héritant d'une classe vide, elle ne peut profiter d'aucun de ses attributs et doit définir toutes ses variables et méthodes.
  - Définir la classe Rectangle. Elle doit posséder :
    - Deux attributs privés **longueur** et **largeur** de type int.
    - Un constructeur qui prend en argument deux entiers et initialise la longueur et la largeur.
    - Des getters pour la longueur et la largeur.
    - Une méthode public int surface() qui renvoie la surface du rectangle.
    - Une méthode publique public String toString() qui retourne une description du rectangle (elle doit préciser qu'il s'agit d'un rectangle ainsi que sa longueur et sa largeur).

EXERCICE D'APPLICATION 3/4

- **La classe Carre**
  - La classe Carre peut bénéficier de la classe Rectangle et ne nécessitera pas la réécriture de ces méthodes si celles-ci conviennent à la sous-classe. Toutes les méthodes et variables de la classe Rectangle ne sont néanmoins pas accessibles dans la classe Carre. Pour qu'un attribut puisse être utilisé dans une sous-classe, il faut que son type d'accès soit public ou protected, ou, si les deux classes sont situées dans le même package, qu'il utilise le **type d'accès par défaut**.
  - Définir la classe Carre qui hérite de la classe Rectangle. Elle doit posséder :
  - Un constructeur qui prend en argument un entier et initialise la longueur et la largeur. Ce constructeur devra faire appel celui de sa classe mère (Rectangle) en utilisant la méthode super().

EXERCICE D'APPLICATION 4/4

- **La classe Carre**
  - **Remarque**
    - L'appel au constructeur d'une classe supérieure doit toujours se situer dans un constructeur et toujours en tant que première instruction ;
    - Si aucun appel à un constructeur d'une classe supérieure n'est fait, le constructeur fait appel implicitement à un constructeur vide de la classe supérieure (comme si la ligne super()était présente). Si aucun constructeur vide n'est accessible dans la classe supérieure, une erreur se produit lors de la compilation.
- Une redéfinition de la méthode public String toString() qui retourne une description du carré (elle doit préciser qu'il s'agit d'un carré ainsi que son côté qui est sa longueur ou sa largeur).