

LOG100 / GTI100

Programmation en génie logiciel et des TI

Automne 2024

Cours 3 : Héritage & exceptions

Chargé de cours: Anes Abdennebi

Crédits à: Ali Ouni, PhD

Plan

- Héritage
 - ▣ Polymorphisme
 - ▣ Encapsulation
- Exceptions
- Exercices
- Le Quiz: le 19 Novembre

HÉRITAGE

Polymorphisme et encapsulation

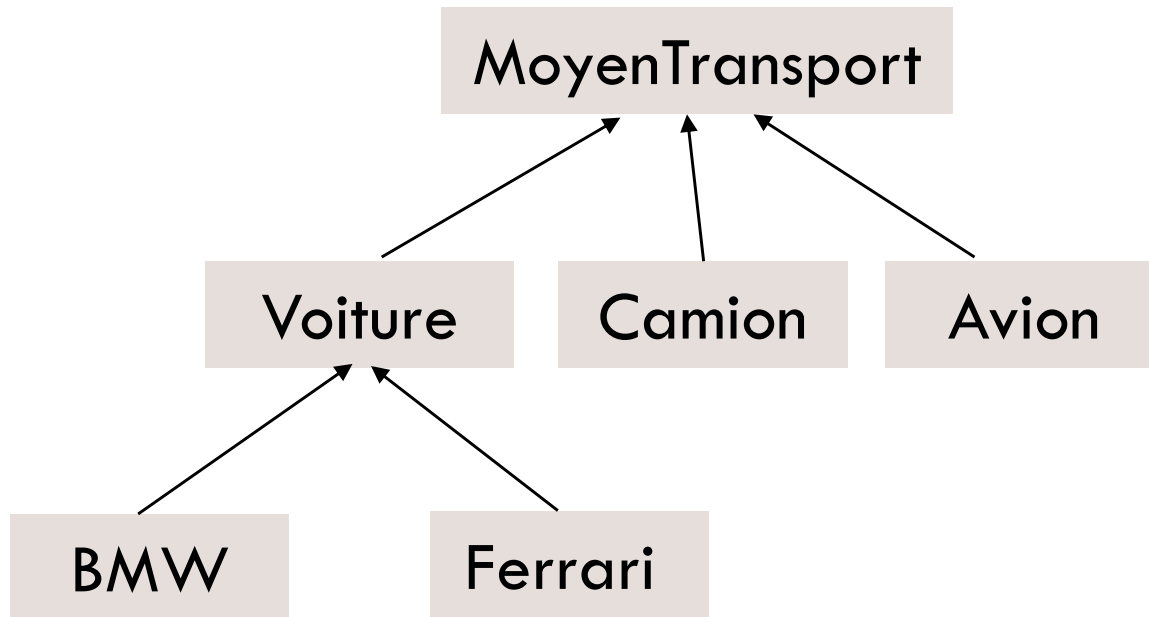
Héritage en Java

- POO: la modélisation objet résulte en une représentation abstraite du monde réelle sous forme d'objets.
- La modélisation nécessite une **classification** des objets.
- Classification : **hiérarchie** de classes.
- Exemples :
 - classification des produits, véhicules, etc.
 - classification des publications.

- C'est un mécanisme qui permet la **réutilisation** de la structure et du comportement d'une classe **générale** par une classe plus **spécialisée**
 - La classe générale définit un ensemble de propriétés **communes** à des classes plus spécialisées
 - La classe plus spécialisée peut définir des propriétés **additionnelles** qui lui sont propres

Héritage en Java

Hiérarchie des classes



classes générales ← classes spécifiques

Héritage

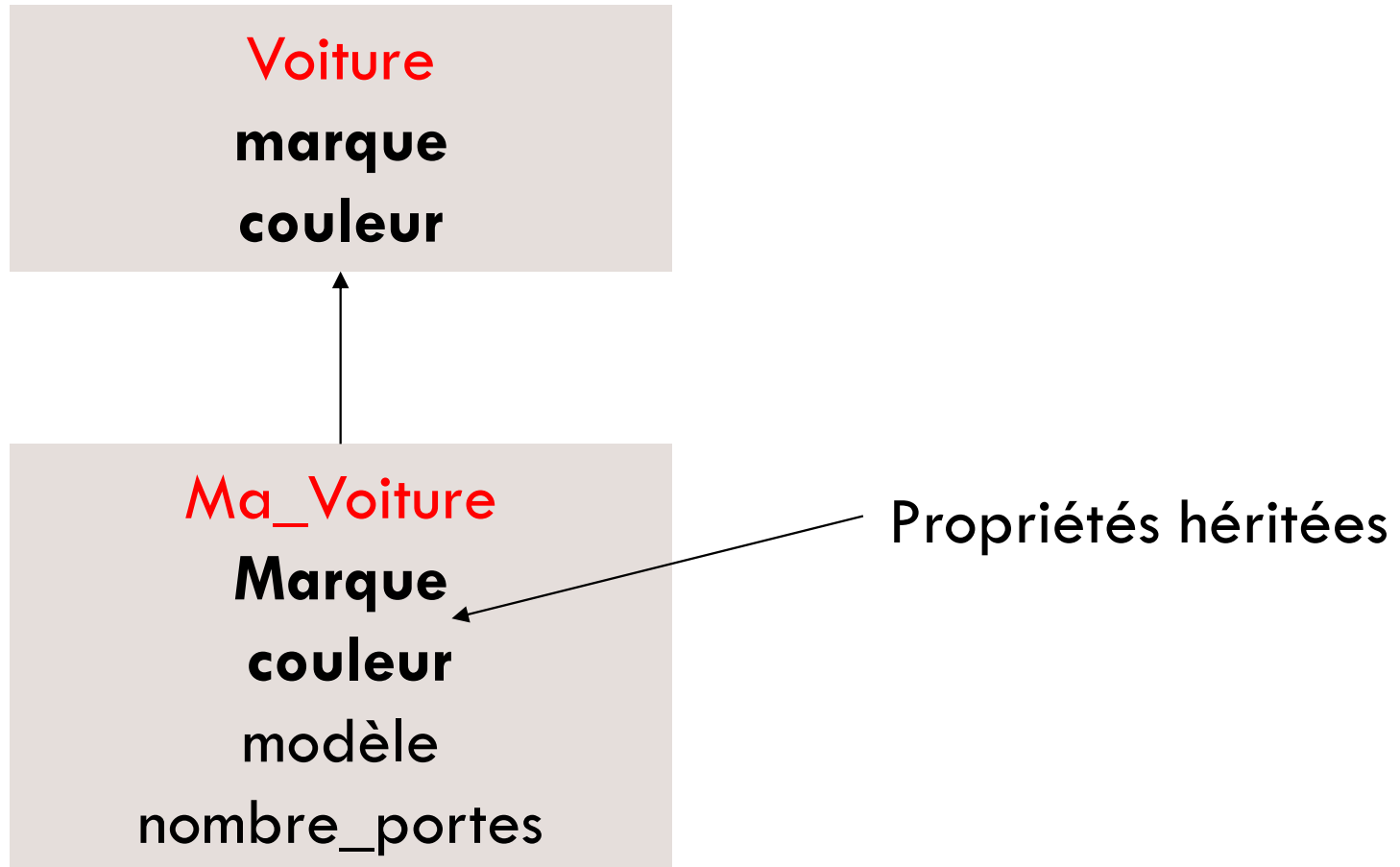
7

- La classe **dérivée** possède (hérite):
 - ▣ Tous les **attributs** de la classe mère
 - ▣ Toutes les **méthodes** de la classe mère

- Exemple :
 - ▣ MoyenDeTransport : consommation, vitesse max, nombre de places, ...
 - ▣ MoyenDeTransport : avancer(), arrêter(), ...

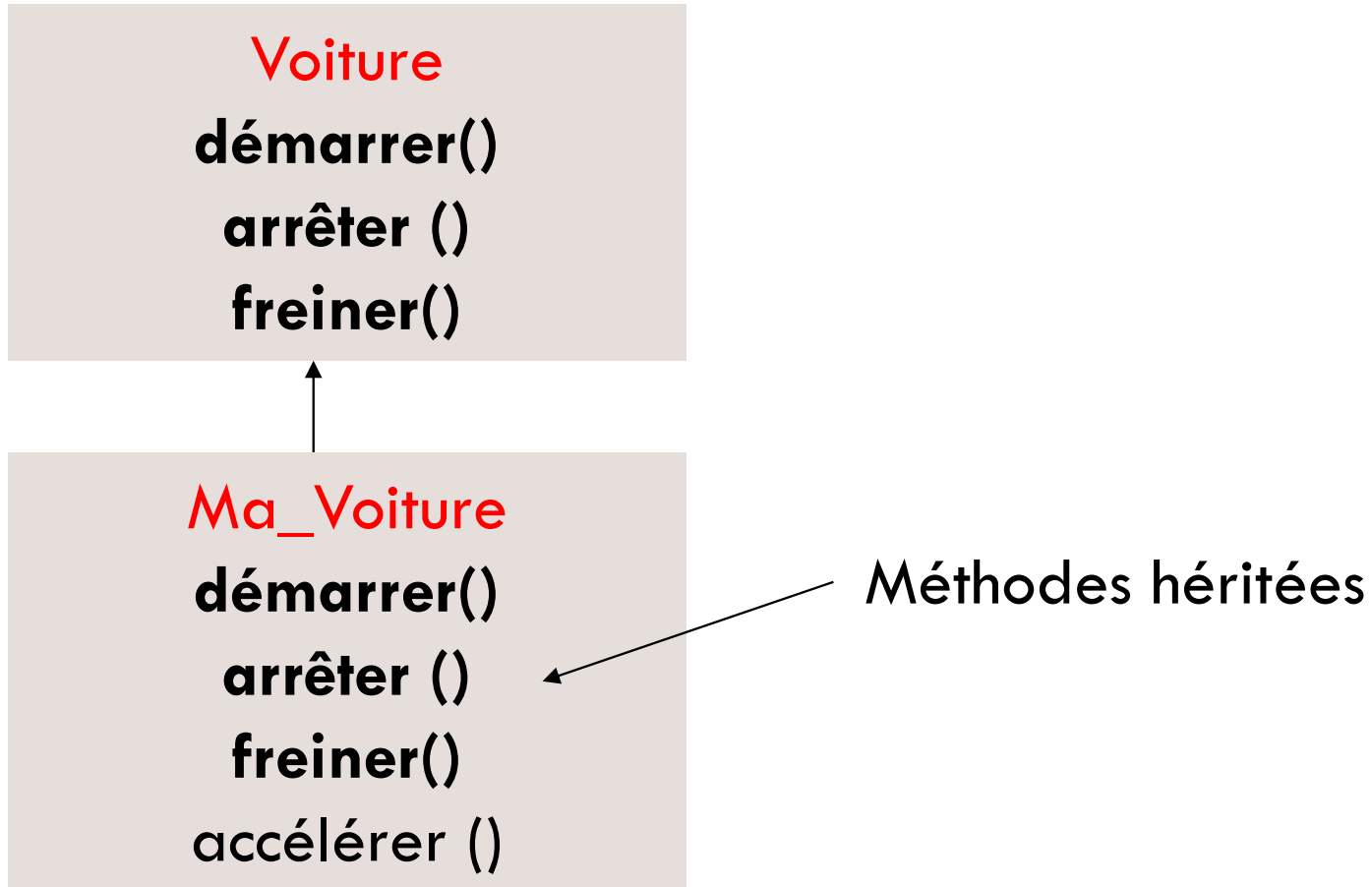
Héritage en Java

❑ Exemple (1) :



Héritage en Java

❑ Exemple (2) :



Héritage

10

- Permet la **réutilisation**
- On crée un **sous-type**
- En Java : héritage **simple**

```
class ChildClass extends BaseClass {  
    // ...  
}
```

Héritage : exemple

11

```
public class Figure {  
  
    public double longueur;  
    public double hauteur;  
  
    public void afficher() {  
        System.out.println("Longueur : " + longueur);  
        System.out.println("Hauteur : " + hauteur);  
    }  
  
}
```

Héritage : exemple

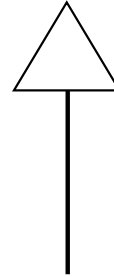
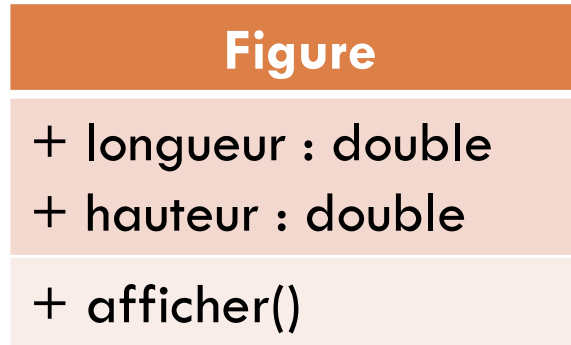
12

```
public class Triangle extends Figure {  
    public String nom;  
  
    public double calculerAire() {  
        double resultat = (longueur * hauteur) / 2;  
        return resultat;  
    }  
  
    public void afficherNom() {  
        System.out.println("Nom : " + nom);  
    }  
  
}
```

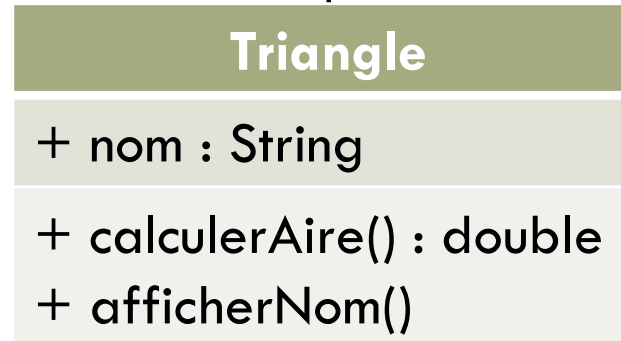
Héritage : exemple

13

Super-classe :



Sous-classe :



Héritage : exemple

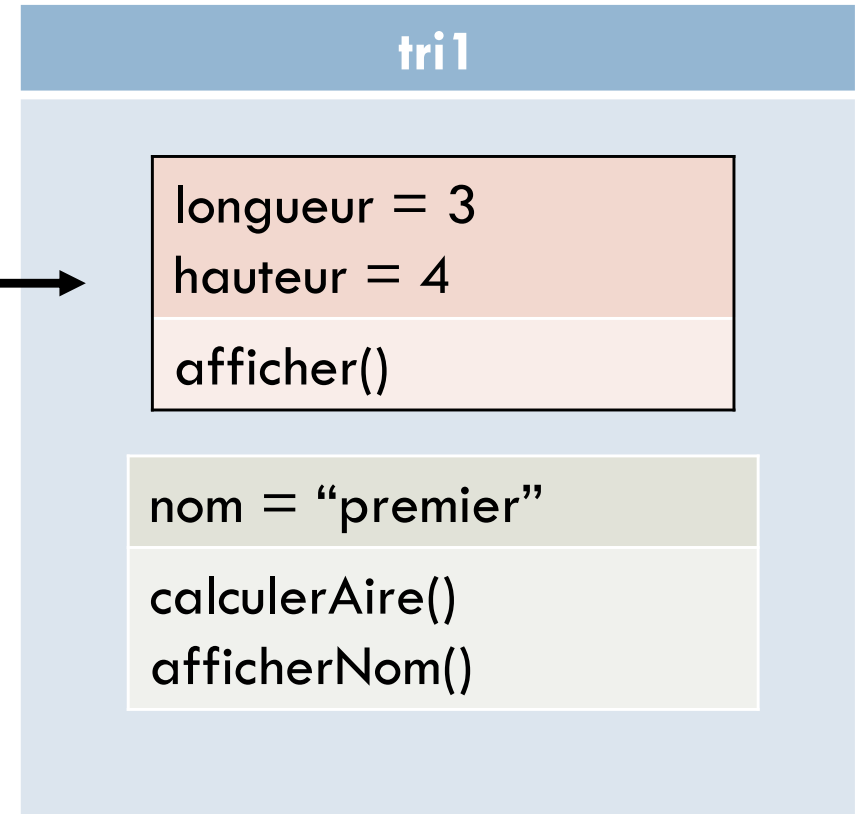
14

```
public class Exemple {  
  
    public static void main(String[] args) {  
        Triangle tri1 = new Triangle();  
        tri1.longueur = 3;  
        tri1.hauteur = 4;  
        tri1.nom = "premier";  
        System.out.print("Triangle : " + tri1.nom);  
        System.out.println(", aire : " + tri1.calculerAire());  
    }  
  
}
```

Héritage : exemple

15

Attributs & méthodes hérités →



Redéfinition de méthode

16

- Quand une sous-classe a une méthode avec la même **signature** (*method overriding*)
 - ▣ On peut appeler la méthode redéfinie avec **super** :

```
class Triangle extends Figure {  
    @Override  
    public void afficher() {  
        super.afficher();  
        System.out.println("Nom : " + nom);  
    }  
}
```


Représenter la spécialisation

17

Considérons une classe *Employee* :

```
public class Employee {  
    private String name;  
    private double salary;  
    public Employee(String aName) {  
        name = aName; }  
    public void setSalary(double aSalary) {  
        salary = aSalary; }  
    public String getName() {  
        return name; }  
    public double getSalary() {  
        return salary; }  
}
```

Représenter la spécialisation

18

- `Manager` est une sous-classe de `Employee`
- La classe `Manager` définit une nouvelle méthode `setBonus`
- La classe `Manager` **redéfinit** (overrides) la méthode `getSalary`
 - ▣ Elle additionne le salaire et le bonus

Représenter la spécialisation

19

```
public class Manager extends Employee {  
    // Nouveau attribut :  
    private double bonus;  
  
    public Manager(String aName) { ... }  
    // Nouvelle méthode :  
    public void setBonus(double aBonus) { bonus = aBonus; }  
  
    // Redéfinition de la méthode de la classe Employee :  
    public double getSalary() { ... }  
  
}
```

La terminologie « super » / « sous »

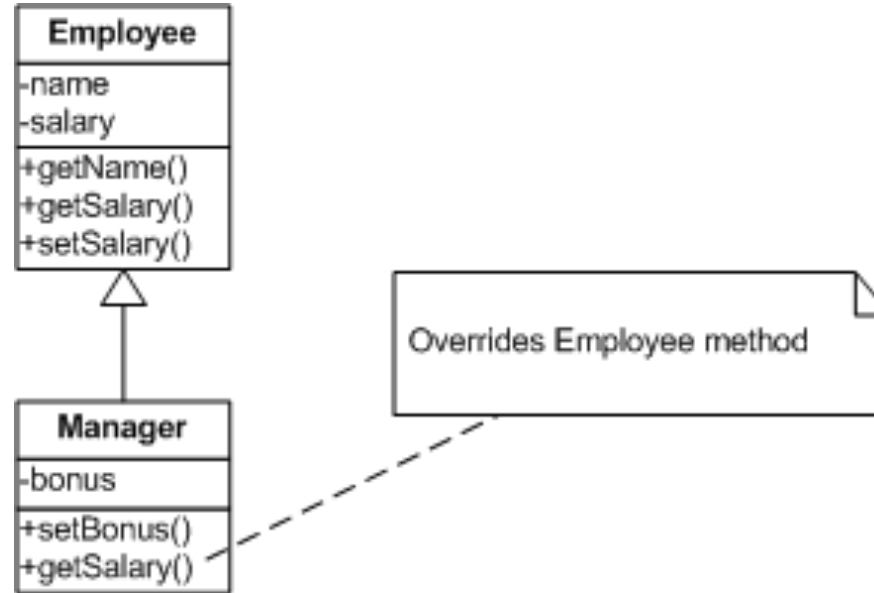
20

Pourquoi la classe `Manager` est-elle une **sous-classe** de `Employee`?

- L'ensemble des managers est un **SOUS-** ensemble de l'ensemble des employés
- Un *Manager* **EST** un *Employee*

Représenter la spécialisation

21



- Dans la sous-classe, on représente :
 - Les éléments **additionnels** : méthodes et attributs propres à la sous-classe
 - Les méthodes **redéfinies**

Représenter la spécialisation

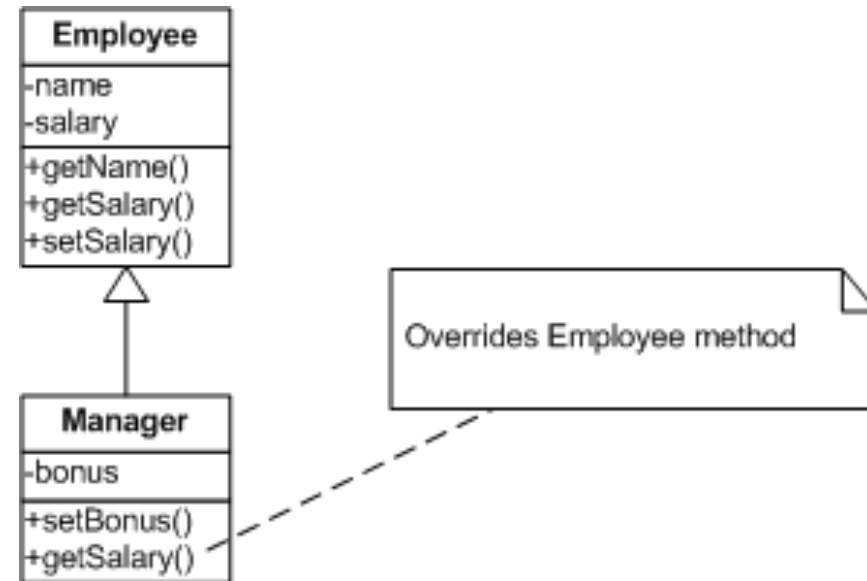
22

□ Attributs de *Manager* :

- hérités de *Employee* : *name* et *salary*
- propres à *Manager* : *bonus*

□ Méthodes de *Manager*

- héritées de *Employee* : *setSalary*, *getName*
- redéfinies par *Manager* : *getSalary*
- propres à *Manager* : *setBonus*



Représenter la spécialisation

23

- Attention à **final**: il empêche l'héritage, et la redéfinition des méthodes.

Variable final → pour créer une variable constante

Méthode final → pour empêcher la redéfinition

Classe final → pour empêcher l'héritage

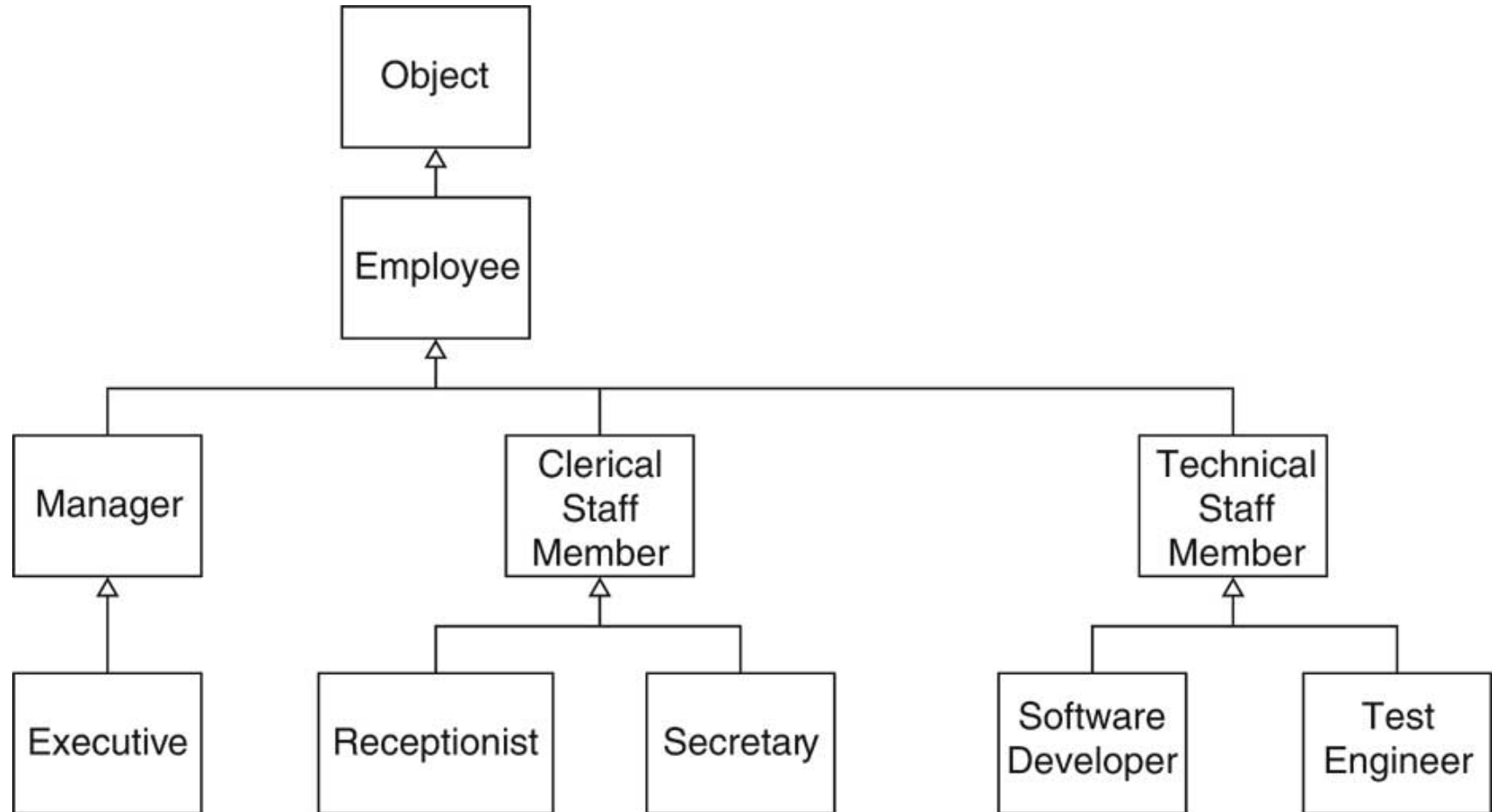
Hiérarchies d'héritage

24

- Dans le monde réel :
 - On classe les concepts en **hiérarchies**
 - Les hiérarchies sont représentées par des **arbres**
 - Le concept général est la **racine** de l'arbre
 - Les concepts plus spécifiques sont les **enfants**
- En orienté objet :
 - On groupe les classes en **hiérarchies d'héritage**
 - La **super-classe générale** est la racine de l'arbre
 - Les **sous-classes plus spécifiques** sont les enfants

Hiérarchies d'héritage

25



Classe *Object*

26

- La classe **Object** est la racine de la hiérarchie des classes
- Toutes les classes héritent de la classe **Object** par défaut
- La classe **Object** définit plusieurs méthodes qui sont souvent redéfinies
 - ▣ La méthode pour afficher l'information d'un objet (appelée avec `System.out.print`) :

```
public String toString()
```
- La méthode pour évaluer si deux objets sont égaux ou non :

```
public boolean equals(Object obj)
```



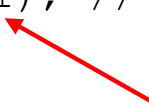
Classe Object

27

❑ Exemple de redéfinition de la méthode *toString* :

```
public class Triangle extends Figure {  
    // ...  
    public String toString() {  
        String res = "Triangle " + nom + " (" + longueur + " X " + hauteur + ")";  
        return res;  
    }  
    // ...  
}
```

```
Triangle t1 = new Triangle(3, 4, "Premier");  
System.out.println("T1 " + t1); // Affiche "T1 Triangle Premier (3.0 X 4.0)"
```



Equivalent à `t1.toString()`

Le principe de substitution

28

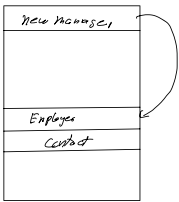
- Ce principe a été formulé par Barbara Liskov :
« *On peut utiliser une sous-classe partout où une super-classe est attendue* »

```
Employee e;  
...  
e = new Manager("Bernie Smith");  
System.out.println("Name = " + e.getName());  
System.out.println("Salary = " + e.getSalary());
```

- ▣ La méthode `getSalary` de `Manager` sera invoquée



```
employee c = new Manager("c");
```



Le principe de substitution

29

- **Polymorphisme** : Sélectionner la méthode appropriée selon le type de l'objet recevant l'appel

« Poly- » : plusieurs

« -morphisme » : formes

Invoquer les méthodes d'une super-classe

30

- ❑ Et si on invoquait la méthode publique `getSalary` de la super-classe?

```
public double getSalary() {  
    return getSalary() + bonus; // ERREUR : appel récursif infini  
}
```

- ❑ Il faut utiliser le mot réservé « `super` » :

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

- ❑ « `super` » n'est pas une référence
- ❑ « `super` » supporte l'appel polymorphique

Invoquer le constructeur d'une super-classe

31

- Utiliser le mot réservé « *super* » dans le constructeur de la sous-classe

```
public Manager(String aName) {  
    super(aName); // Appelle le constructeur de la superclasse pour initialiser ses champs privés  
    bonus = 0;  
}
```

- L'appel au constructeur de la super-classe doit être la **1^{ère} instruction** dans le constructeur de la sous-classe
- Si le constructeur de la sous-classe n'appelle pas « *super* », la super-classe doit avoir un **constructeur sans paramètre**, qui sera alors appelé
 - Les constructeurs ne sont pas hérités

Ignore ce slide

Invoquer le constructeur d'une super-classe

32

- Si le constructeur de la sous-classe n'appelle pas « *super* », la super-classe doit avoir un **constructeur sans paramètre**, qui sera alors appelé (qui est automatiquement déclaré si l'utilisateur ne l'a pas déclaré).

```
public class institution{  
  
}  
  
public class universite extends institution{  
    public universite() {  
        super();  
    }  
}
```

Accès aux attributs d'une super-classe

33

- On ne **peut pas** accéder aux attributs **privés** d'une super-classe

```
public class Manager extends Employee {  
    public double getSalary() {  
        return salary + bonus; // ERREUR : variable privée  
    }  
    ...  
}
```

*↳ déclarer comme privé
alors final*

- Accès aux membres d'une classe par d'autres classes :

Visibilité	Classes dans le même package	Ses sous-classes	Autres classes
public	Oui	Oui	Oui
protected	Oui	Oui	Non
private	Non	Non	Non

Accès aux attributs d'une super-classe

34

- *Cacher des informations*
 - ▣ Cacher ou isoler l'implémentation d'une classe
 - ▣ Gérer l'accès

- Limitez l'accès aux attributs **et aux méthodes**
 - ▣ *private* pour les attributs
 - ▣ Choisir au plus restreint selon l'objectif des méthodes
 - ▣ La redéfinition des méthodes privées d'une super-classe n'est pas possible dans ses sous-classes.

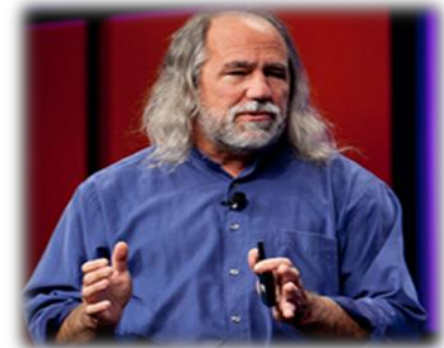
Encapsulation

35

Objet = une **boîte noire** pour ses utilisateurs

- **Cacher** les détails d'implémentation aux classes utilisant l'objet
 - ▣ Le concepteur peut changer d'implémentation sans que cela impacte les utilisateurs
 - ▣ Un code maintenable à un moindre coût
- **L'interaction** avec l'objet se fait via une **interface publique**

« *Encapsulation serves to separate the contractual interface of an abstraction and its implementation* » — Grady Booch



Encapsulation

36

- Fournir des méthodes d'**accès** (accessors, « getters ») et des **mutateurs** (modifiers, « setters »)
 - ▣ **Accesseur** : lit l'**état** de l'objet sans le modifier
 - ▣ **Mutateur** : **change** l'état de l'objet
 - ▣ Mais toujours en utilisant **.clone()** *(type) obj.clone();*
- Les classes sans mutateur sont **immuables** :
 - ▣ Leur état ne peut pas être modifié
 - ▣ String et Integer sont immuables
 - ▣ java.util.Date est mutable



Encapsulation

37

- ❑ Les références aux objets immuables peuvent être partagées
- ❑ Ne pas partager les références muables

```
public final class Personne {  
    private final String nom;  
    private final List<String> livresPreferes; // variable muable  
  
    public String getNom() { return nom; }  
    public List<String> getLivresPreferes() {return livresPreferes;  
}  
}
```

**List est
muable !**

*On peut changer la liste, mais pas ce qui est
déjà présent*

Le Problème

Encapsulation

38

❑ Ne pas partager les références muables (suite)

▣ Le danger : `getFavoriteBooks()` brise l'encapsulation

```
Personne person = new Personne("Alice", livres);  
List<String> extlivres = person.getLivresPreferes();  
extlivres.add(«Brave New World»); // Change l'état de  
person !
```

▣ Remède : utiliser un clone ou instancier une copie manuellement avec un constructeur

```
public List<String> getLivresPreferes() {  
    return (List<String>) livresPreferes.clone();  
}
```

Encapsulation

39

- Variables d'instance définies comme **final**
 - ▣ Variable d'instance **immuable** déclarée finale : ne changera plus une fois l'objet construit

```
private final String name;  
private final double salary;
```

- ▣ Attention : une variable déclarée finale peut encore référer à un objet muable :

```
private final ArrayList elements;
```

- elements ne peut pas référer à une autre liste
- Mais le contenu de la liste peut changer

Quiz (Héritage et Encapsulation)

40



EXCEPTIONS

Gestion des Erreurs

Exceptions – idées principales

42

- ❑ Les exceptions sont utilisées pour signaler les erreurs pendant l'exécution du code
- ❑ Quand on trouve une erreur, **on lance** une exception

```
try {  
    { catch (type e) { }  
}
```
- ❑ Une exception doit **être attrapée** par un **exception handler** (traitement d'exceptions) à un certain moment pendant l'exécution
- ❑ Pourquoi sont-elles importantes?

```
public class Aire {  
  
    public double calculerAire(double base, double hauteur) {  
        double resultat = (base * hauteur) / 2.0;  
        return resultat;  
    }  
  
}
```

Exemple d'une exception

44

```
public class ExempleException {  
  
    public static void test() {  
        Aire exemple = new Aire();  
        double aire1 = exemple.calculerAire(3.0, 4.0);  
        System.out.println("Aire1 = " + aire1);  
    }  
    ...  
}
```

Tout est bien mais ...

Exemple d'une exception

45

- Et si on a :

```
double aire2 = exemple.calculerAire(-2.5, 3.0);  
System.out.println("Aire2 = " + aire2);
```

- La syntaxe est correcte mais **pas la signification**
- On doit signaler les erreurs :
 - ▣ base et hauteur ne peuvent pas avoir des **valeurs négatives**
 - ▣ base ou hauteur ne peuvent pas être **zéro**

Lancement d'une exception

46

- Classe **Exception** est la racine de toutes les exceptions en Java
 - Hérite de la classe **Throwable**
- Une méthode, comme `calculerAire` peut indiquer les types d'exceptions (erreurs) qu'elle peut lancer (**throw**) pendant l'exécution

- Le mot-clé **throws** est utilisé après la déclaration des paramètres, suivi par les noms des types des exceptions

```
public double computeArea(double base, double height) throws MyException {  
    // ...  
}
```

Lancer les exceptions

47

- Pour **lancer** une exception, on crée un objet d'une sous-classe de la classe `Exception`
- Quand une exception est lancée, le contrôle du programme saute au code qui **attrape** (*catch*) l'exception
- Syntaxe :
 - ▣ `throw <instance d'exception>;`
- Exemple :
 - ▣ `throw new MyException("Erreur : base est zéro");`

Objet

Trouble



Exception



myexception

myexception

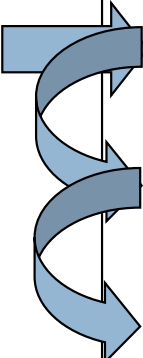
héritage



Exception

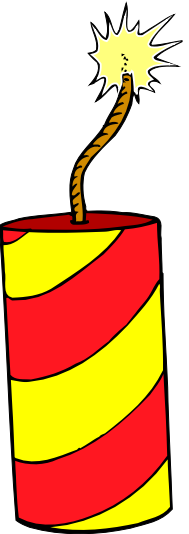
Exemple de *MyException*

48



```
public double calculerAire(double base, double hauteur)
    throws MyException {
    if (base == 0)
        throw new MyException("Erreur: Base est zéro");
    if (hauteur == 0)
        throw new MyException("Erreur: Hauteur est zéro");
    if (base < 0)
        throw new MyException("Erreur: Base est négative");
    if (hauteur < 0)
        throw new MyException("Erreur: Hauteur est négative");
    double resultat = (base * hauteur) / 2.0;
    return resultat;
}
```

base = - 2.5, hauteur = 3.0



Attraper les exceptions

49

- Le code où on peut lancer une exception est défini avec le mot-clé **try**

- Syntaxe :

```
try {  
    ... séquence d'instructions où peut lancer une exception ...  
}
```

- Pour attraper une exception, on utilise le mot-clé **catch**

- ▣ Le paramètre est l'objet crée quand l'exception est lancée

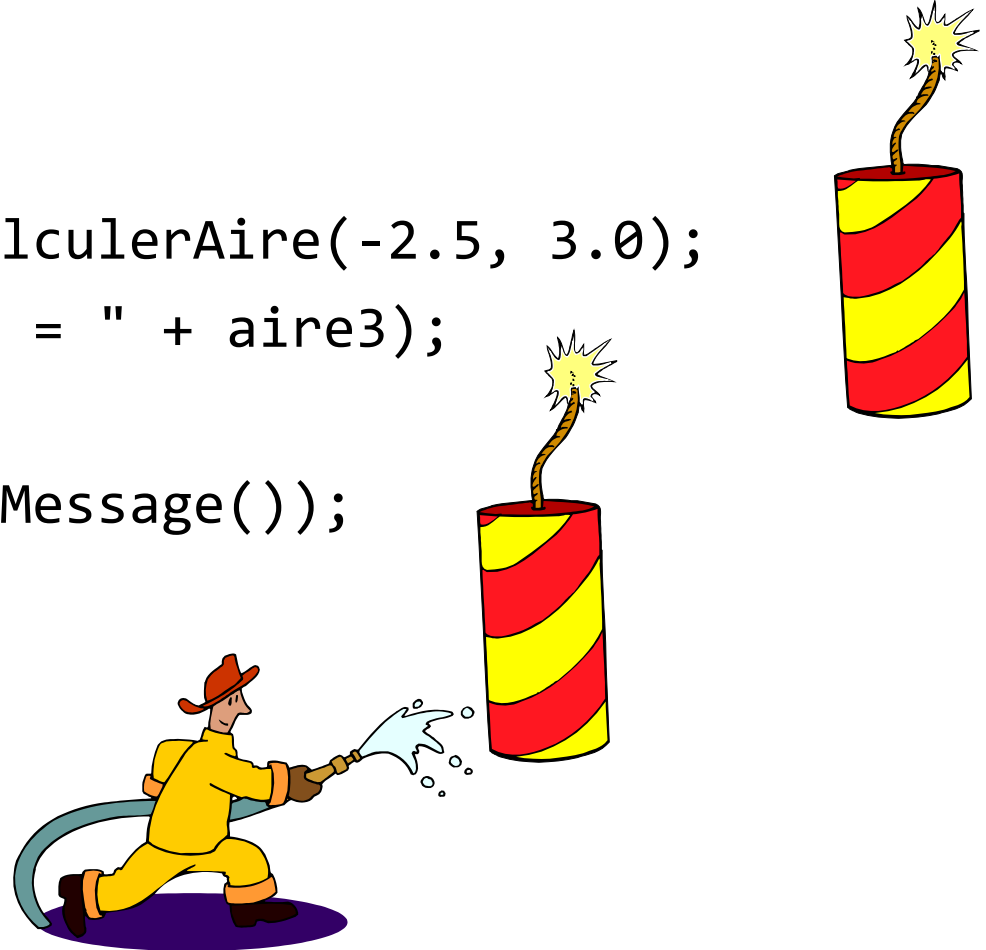
- Syntaxe :

```
catch (TypeException e) {  
    ... séquence d'instructions ...  
}
```

Exemple de *try* / *catch*

50

```
try {  
    ➡ double aire3 = exemple.calculerAire(-2.5, 3.0);  
    System.out.println("Aire3 = " + aire3);  
} catch (MyException me) {  
    ➡ System.err.println(me.getMessage());  
}
```



Où attrape-t-on les exceptions?

51

- Toutes les exceptions doivent être attrapées à un point dans l'exécution du code
- Si on n'attrape pas l'exception dans la méthode où elle est lancée, cette méthode passe la responsabilité à la méthode qui l'a **appelée**
 - ▣ On utilise le mot-clé **throws** pour exprimer cette responsabilité
 - ▣ Si nécessaire, on répète ce processus (traverse la pile d'appels de méthodes) jusqu'au début de l'exécution (méthode *main*)
- Si le compilateur ne trouve aucun code qui attrape une exception, il signale une erreur comme :

ExceptionExample.java:43: unreported exception java.lang.Exception; must be caught

Où attrape-t-on les exceptions?

52

```
public class Example {  
    // Méthode qui lance une exception  
    public static void method1() throws MyException {  
        throw new MyException("Une erreur s'est produite dans method1."); }  
    // Méthode qui appelle method1, et propage l'exception  
    public static void method2() throws MyException {    method1(); // Appelle method1, ne capture pas l'exception }  
    // Méthode qui appelle method2, et propage l'exception  
    public static void method3() throws MyException {    method2(); // Appelle method2, ne capture pas l'exception }  
    public static void main(String[] args) {  
        try {  
            // Appel de method3, ici l'exception doit être capturée  
            method3();    }  
        catch (MyException e) {    System.out.println("Exception capturée dans main : " + e.getMessage());  
        }  
    }  
}
```

Le bloc *finally*

53

- On utilise le bloc **finally** pour définir une séquence d'instructions que l'on veut toujours exécuter
- Il est exécuté :
 - ▣ Si une exception est lancée et attrapée
 - ▣ Si une exception est lancée mais non attrapée
 - ▣ Si aucune exception n'est lancée
- Il est optionnel

□ Syntaxe :

```
try {  
    // ... potentiels exceptions  
} catch(ExcType1 e1) {  
    // ... gestion ExcType1  
} catch(ExcType2 e2) {  
    // ... gestion ExcType2  
}  
// autres blocs catch  
finally {  
    // instructions exécutées  
    // dans tous les cas  
}
```

Exceptions *checked* et *unchecked*

54

- Les exceptions vérifiées (*checked*) :
 - ▣ Elles sont vérifiées au moment de la compilation.
 - ▣ Doivent utiliser explicitement des instructions *try* / *catch* et *throws*, et lancer explicitement les exceptions

- Les exceptions non-vérifiées (*unchecked*) :
 - ▣ Ne nécessitent pas de *try* / *catch* ou *throws*
 - ▣ Sont souvent lancées automatiquement par le système d'exécution de Java
 - Ou explicitement par le programmeur
 - ▣ Toutefois, elles peuvent être gérées
 - ▣ Signalent généralement une erreur de programmation (*bug*)
 - Exemples : `IllegalArgumentException`, `Arithmetic`, `ClassCast`, `NullPointerException`, `NumberFormatException`, `IndexOutOfBoundsException`, ...

Exceptions *checked* et *unchecked*

55

```
import java.io.*;

public class CheckedExceptionExample {

    public static void readFile() throws IOException {
        // Cette méthode lance une IOException qui est vérifiée
        FileReader file = new FileReader("fichier.txt");
        BufferedReader fileInput = new BufferedReader(file);
        // Lecture du fichier (si le fichier n'existe pas, IOException sera levée)
        System.out.println(fileInput.readLine());
        fileInput.close();
    }

    public static void main(String[] args) {
        try {
            readFile(); // Appel de la méthode qui lance une exception vérifiée
        } catch (IOException e) {
            System.out.println("Exception vérifiée capturée : " + e.getMessage());
        }
    }
}
```

Quiz Exceptions

56



Exemple

57

- Un système de réservation simplifié dans une compagnie aérienne.

(Classes Vol, Réservation, RéservationPremium)