Conception Orienté Objet et Programmation Java

Chapitre 5: Héritage

Objectifs du chapitre

- Savoir identifier le lien entre les classes
- Introduire la technique d'héritage : intérêt et notation .
- Introduire les droits d'accès d'une classe dérivée aux membres de la classe de base.
- Comprendre la construction d'un objet dérivé
- Maîtriser la notion de redéfinition.
- Comprendre la notion des classes abstraites

Héritage

Héritage: Définition

- L’héritage est l'un des mécanismes les plus puissants de programmation orientée objet qui permet à une classe d'hériter les **propriétés** et les **comportements** d'une autre classe.

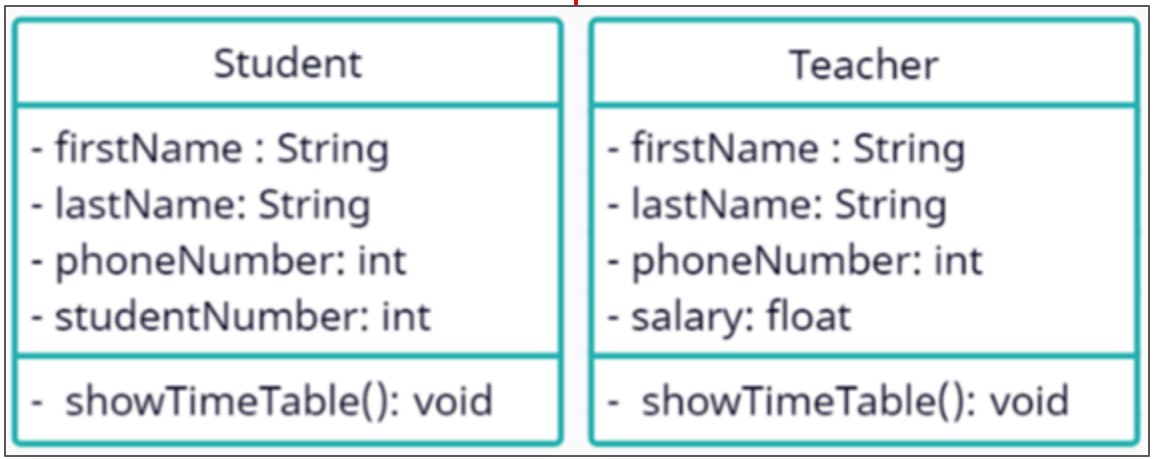
- La classe qui hérite est appelée **sous-classe** ou **classe dérivée** ou **classe fille**, tandis que la classe dont elle hérite est appelée **classe de base** ou **classe parent** ou **super classe** ou même **classe mère**.

Héritage: Intérêts

- **Spécialisation**: une nouvelle classe réutilise les attributs et les opérations d 'une classe en y ajoutant et/ou des opérations particulières à la nouvelle classe
- **Redéfinition**: une nouvelle classe redéfinit les attributs et opérations d'une classe de manière à en changer le sens et/ou le comportement pour le cas particulier défini par la nouvelle classe
- **Réutilisation**: évite de réécrire du code existant et parfois on ne possède pas les sources de la classe à hériter

Héritage: Exemple

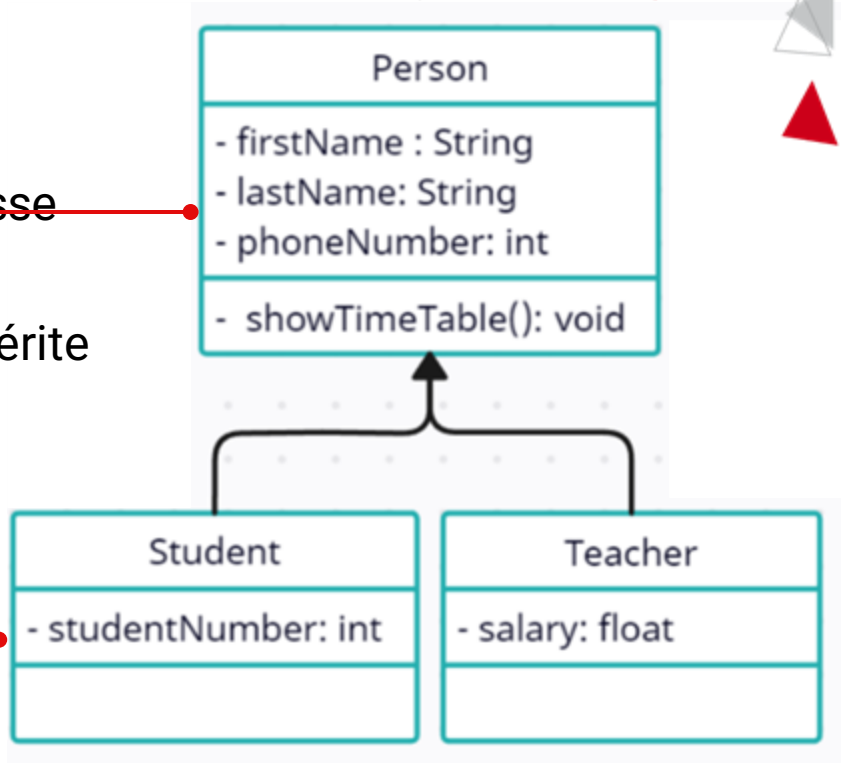
- Si une application a 2 types d'utilisateurs : Student et Teacher.
  - On arrive donc au design suivant:



On peut remarquer qu'il y a la **méthode** et beaucoup **d'attributs** qui sont en commun.

Héritage: Exemple

- Une sous-classe hérite d'une super-classe
- La super-classe est la classe dont on hérite
- La sous-classe est la classe qui hérite



Héritage: terminologie

- La classe **Student** hérite de la classe **Person**.
- **Person** est la classe mère et **Student** la classe fille.
- **Person** est la super-classe de la classe **Student**.
- **Student** est une sous-classe de **Person**.

➔ **Un objet de la classe Student ou Teacher est forcément un objet de la classe Person**

➔ **Un objet de la classe Person n'est pas forcément un objet de la classe Student ou Teacher**

Héritage: Exemple

- Le mot clef **extends** indique que la classe Student et la classe Teacher héritent de la classe Person

```
1  public class Person {
2      protected String firstName, lastName;
3      protected int phoneNumber;
4      public void showTimeTable(){ }
5  }
6  class Student extends Person {
7      private int studentNumber;
8  }
9  class Teacher extends Person {
10     private float salary;
11 }
```

Héritage: Notons que…

- Toutes les classes héritent de la super classe « **Object** »

Ex:

```
public class Person {
        ...
}
```

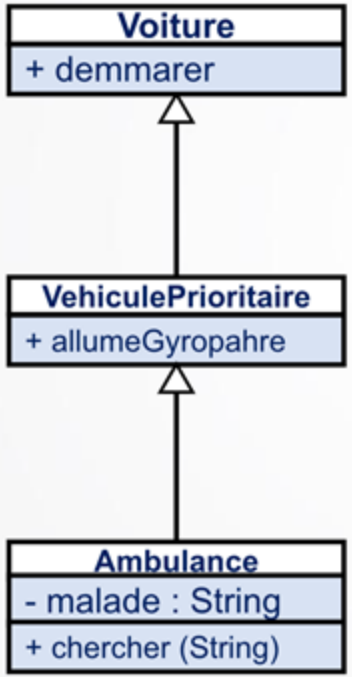
```
public class Person extends Object {
        ... cela signifie :
}
```

- Une classe ne peut étendre qu'une seule classe : **Pas d'héritage multiple**.
- Une classe déclarée **final** ne peut pas être étendue.

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

⇒ La classe A ne peut pas être étendue.

Héritage: Héritage à plusieurs niveaux




```
public class Voiture {
        ...
        public void demarre() {
                ...
        }
}
```

```
public class VehiculePrioritaire  extends
Voiture {
        ...
    public void allumeGyrophare()
    {
                ...
    }
}
```

```
public class Ambulance  extends
VehiculePrioritaire {
   private String malade;
        ...
     public void chercher(String ma) {
                ...
        }
}
```

```
Ambulance am =  new
Ambulance(...);
am.demarre();
am.allumeGyrophare();
am.chercher("Raoul");
```

Chaînage des constructeurs

Héritage: Chaînage des constructeurs

- Tout constructeur, sauf celui de la classe java.lang.Object, fait appel à un autre constructeur qui est :
  - Un constructeur de sa superclasse (appelé par **super(...)**) ;
  - Un autre constructeur de la même classe (appelé par **this(...)**).
- Cet appel est mis nécessairement **en première ligne du constructeur**.
- En cas d'absence de cet appel, le compilateur **ajoute super(); en première ligne** du constructeur.

Héritage: Chaînage des constructeurs

Si:

```
public class A {
        public A () { }
}
```

```
public class A {
        public A () {
                super();
        }
}
```
cela signifie :

Si:

```
public class A {
        public A (int x) {
                super();
                this() ;
        }
}
```

INTERDIT

Il n'est pas possible d'utiliser à la fois un autre constructeur de la même classe et un constructeur de sa classe mère dans la définition d'un de ses constructeurs.

Héritage: Chaînage des constructeurs

1

```
public class A {
	public A() {

		System.out.println("Constructor A");
	}
}
```

2

```
public class B extends A {
	public B() {

		System.out.println("Constructor B");
	}

	public B(int n) {
		this();
				System.out.println("2nd
Constructor B");
	}
}
```

3

```
public class C extends B {
	public C() {
		super(1);

		System.out.println("Constructor C");
	}
```

4

```
public class Test {
	public static void main(String[] args) {
		new C();
	}
}
```

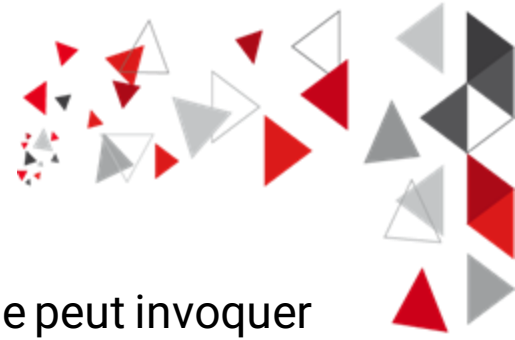
Output:

```
Constructor A
Constructor B
2nd Constructor B
Constructor C
```

Explication:

- Peut-être avez-vous oublié constructeur de A, si vous n'avez plus pensé que l'instruction super(); est ajoutée en première ligne du constructeur sans paramètre de la classe B.
- L'instruction super(); est aussi ajoutée en première ligne du constructeur de la classe A, faisant ainsi appel au constructeur sans paramètre de la classe Object, mais ce constructeur ne fait rien.

Héritage: Chaînage des constructeurs



- Pour initialiser les attributs hérités, le constructeur d'une classe peut invoquer un des constructeurs de la classe mère à l'aide du mot-clé **super()**.

```
public Student(String firstName, String lastName, int phoneNumber, int studentNumber) {
    super(firstName, lastName, phoneNumber);
    this.studentNumber = studentNumber;
}
```

super doit être la première instruction

```
public Person(String firstName, String lastName, int phoneNumber) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.phoneNumber = phoneNumber;
}
```

Si on ne fait pas d'appel explicite au constructeur de la superclasse, c'est le constructeur par défaut de la superclasse qui est appelé implicitement.

Héritage: Chaînage des constructeurs

Constructeur implicite: erreur fréquente

1

```
public class A {
        public int x;
}

public class B extends A {
        public int y;

        public B (int x, int y) {
                this.x = x;
                this.y = y;
        }
}
```

2

```
public class A {
        public int x;

        public A (int x) {
                this.x = x;
        }
}

public class B extends A {
        public int y;

        public B (int x, int y) {
                this.x = x;
                this.y = y;
        }
}
```

ERROR

Héritage: Chaînage des constructeurs

Solution 1

1

```
public class A {
        public int x;
        public A(){}
        public A (int x) {
                this.x = x;
        }
}


public class B extends A {
        public int y;

        public B (int x, int y) {
                this.x = x;
                this.y = y;
        }
}
```

Solution 2

2

```
public class A {
        public int x;

        public A (int x) {
                this.x = x;
        }
}


public class B extends A {
        public int y;

        public B (int x, int y) {
                super(x);
                this.y = y;
        }
}
```

Surcharge & Redéfinition

Surcharge (overloading)

- DEF 1: Un même nom de fonction pour plusieurs fonctions qui sont distinguées par leur signature (**Nom de la méthode** et **la liste des paramètres**)
- DEF 2: Avoir une même méthode qui possède des paramètres de nature différentes...

```
public void showMessage(String a)          { }
public void showMessage(int a)             { }
public void showMessage(String a, String b) { }
```

Signature de la méthode

Redéfinition (override)

- Permet à une sous-classe de fournir une définition spécifique d'une méthode déjà définie dans l'une de ses superclasses.
- La version de la méthode de la super classe peut être invoquée à partir du code de la sous-classe en utilisant le mot clé super (exemple : super.doCallOverridenMethod()).
- Redéfinition est un concept qui s'applique uniquement aux méthodes et non pas aux variables.
- La redéfinition est la possibilité d'utiliser exactement la même signature pour définir un service dans un type et dans un sous type. Le type de retour du service doit être le même, mais la visibilité peut changer.

Redéfinition (override)

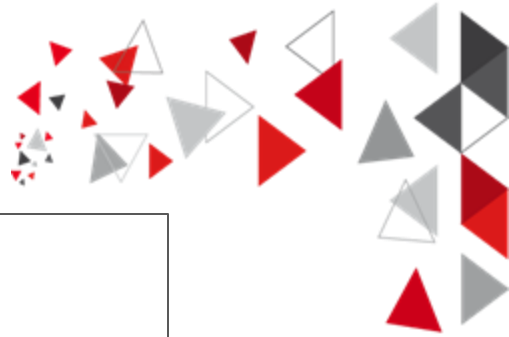
- La redéfinition des méthodes est possible uniquement pour les méthodes qui sont héritables.

  ⚠ **Une méthode marquée private n'est pas héritable.**

  On peut fournir une implémentation de la même méthode avec le même nom, la même signature, et le même type de retour dans la sous-classe, c'est comme si on a créé une nouvelle méthode qui n'a absolument rien avoir avec la méthode de superclasse.

- Une sous-classe dans un package différent de celui de la super classe peut redéfinir toutes les méthodes de cette classe qui sont marquées **public** ou **protected**.
- On ne peut pas redéfinir une méthode marquée **final**.

Surcharge & Redéfinition : Exemple



```java
public class A {
  public void saySomething() {
    System.out.println("Hello ?");
  }
}

public class B extends A {
  public void saySomething() {
    System.out.println("Hello from the other side!");
  }
  public void saySomething(String msg) {
    System.out.println("Hello" + msg + "!");
  }
}
```

Classe abstraite & Classe scellées

Classes et méthodes abstraites: Définition

- Une classe abstraite (**abstract class** en anglais) est une classe qui **ne peut pas être instanciée directement**. Elle sert de modèle pour les sous-classes.
- Les classes abstraites sont déclarées à l'aide du mot-clé **abstract**.
- Les classes abstraites **peuvent** contenir des **méthodes abstraites** (0 ou plusieurs).
  - Une méthode abstraite est une méthode déclarée sans implémentation dans la classe abstraite. Toutes les sous-classes de la classe abstraite doivent fournir une implémentation concrète de toutes les méthodes abstraites héritées.

Classe Mére

```
public abstract void sayHello();
```

```
public void sayHello(){
        System.out.println("Hell
o");
```

Classe Fille

Classes et méthodes abstraites: Utilisation

- L'utilisation de classes abstraites est courante lorsque vous souhaitez définir une structure de base commune pour un groupe de classes apparentées tout en forçant les sous-classes à implémenter certaines méthodes spécifiques à leur contexte.

```
public abstract class Person {
        public abstract void calculateSalary();
}

public class Developer extends Person {
        public void calculateSalary () { … }
}

public class Manager extends Person { }
```

Cette classe générera une **erreur de compilation** car elle n'a pas redéfinie la méthode calculateSalary()

Classes scellées: Définition et Utilisation

- Une classe scellée (ou "**sealed class**" en anglais) est une classe qui autorise l'héritage à certaines classes en utilisant le mot clé **permits**.
- Cependant, toutes les sous-classes doivent être déclarés avec le mot clé **final** pour **interdire** l'héritage davantage ou **non-sealed** pour **permettre** l'héritage d'autres classes.

```
sealed class Shape permits Circle, Rectangle, Triangle { }
non-sealed class Circle extends Shape {}
final class Rectangle extends Shape {}
class Triangle extends Shape {}
class OtherShape extends Shape {}
```

Cette classe générera une **erreur de compilation** car elle doit être une classe **final** ou **non-sealed**.

Cette classe générera une **erreur de compilation** car elle n'est pas autorisée à étendre Shape.

Classes scellées : Résumé

- La déclaration de la classe mère comme scellée principalement **limite l'héritage** pour les sous-classes **autorisées**.
- Le choix entre final et non scellée dépend de votre intention quant à la possibilité d'extension de la **sous-classe**.
  - ⚠ Utilisez **final** lorsque vous voulez **interdire** toute extension supplémentaire.
  - ⚠ Utilisez **non-sealed** lorsque vous voulez **permettre** l'extension à d'autres classes.

Merci pour votre attention