

# La Programmation Orientée Objet (POO) avec Java



Emmanuel Fernandez  
Avril 2022

# Procédural vs Objet : le monde réel

- Le monde réel est constitué d'objets aux caractéristiques et comportements divers qui sont en relation entre eux.



# Le monde procédural et le monde réel

- Traduire le monde réel en variables, fonctions, procédures, programmes, etc.
- Traduction coûteuse en temps.
- Traduction limitante.



# Le monde objet et le monde réel

- Traduire le monde réel en objets.
- Traduction peu coûteuse en temps, les deux mondes sont très proches.
- Traduction très précise.



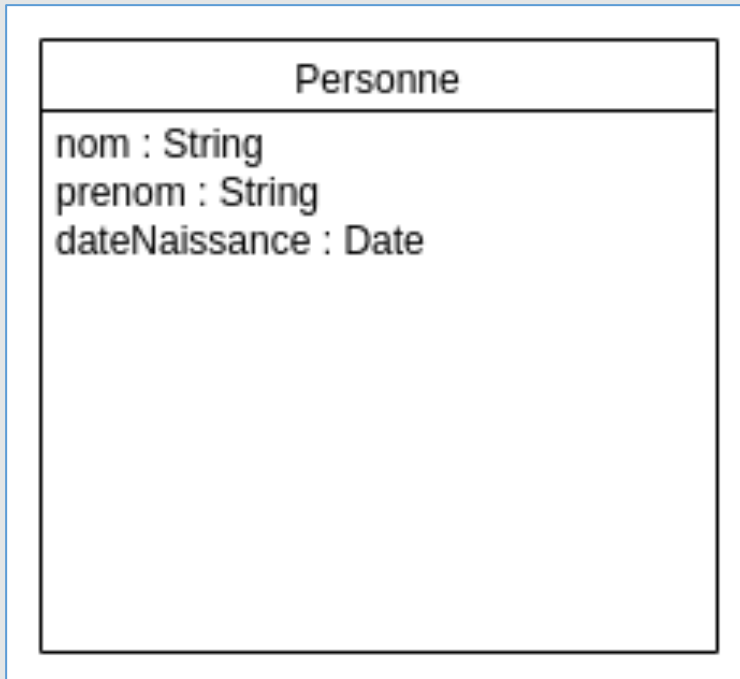
Les classes

# Définition

- Une classe est une structure de code permettant de définir les caractéristiques et les comportements d'objets en mémoire.

Les classes

# Les attributs



Maillot  
Paul  
Né le 23/09/1988

Les classes

# Les méthodes

Personne
nom : String prenom : String dateNaissance : Date
getPresentation() : String



Bonjour je m'appelle  
Paul Maillot, j'ai 30 ans

# Exemple de code

```
package demo;

import java.util.Date;

public class Personne {
    // ATTRIBUTS
    public String nom;
    public String prenom;
    public Date dateNaissance;

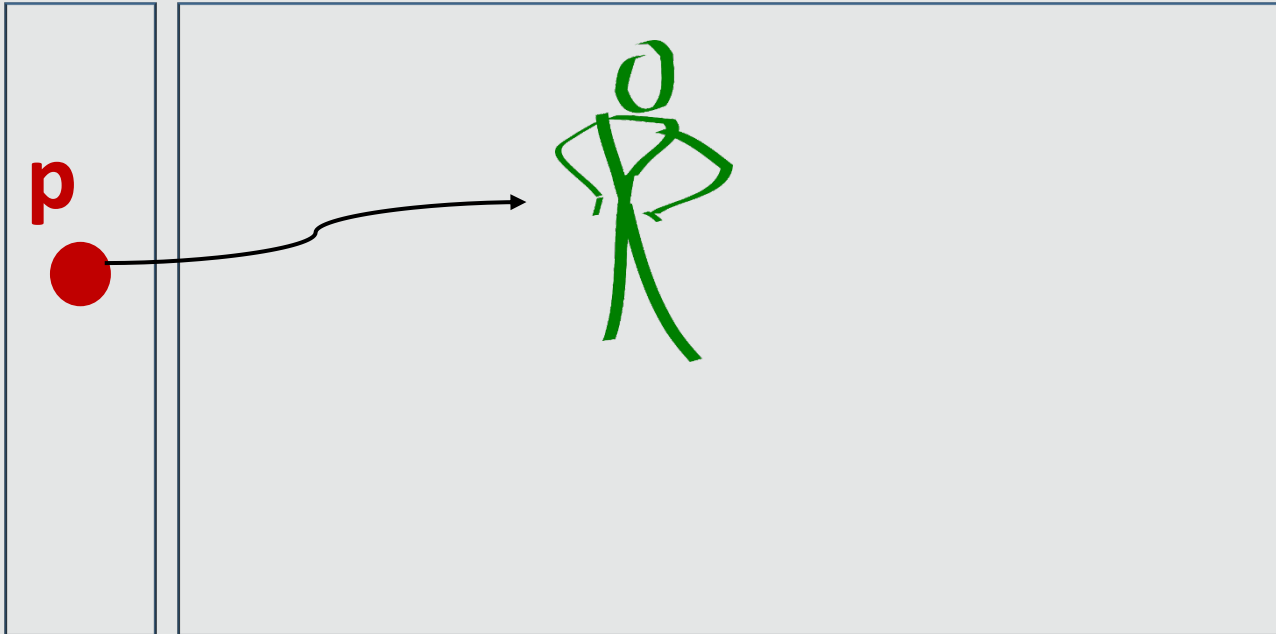
    /**
     * La personne se présente
     * @return le texte de la présentation
     */
    public String getPresentation() {
        // calcul de l'age
        long age = (new Date().getTime() - dateNaissance.getTime()) / 1000 / 60 / 60 / 24 / 365 ;

        return "Bonjour je m'appelle " + prenom + " " + nom + ", j'ai " + age + " ans";
    }
}
```



# Parlons mémoire

```
Personne p = new Personne() ;
```



En rouge : la **déclaration** d'une référence vers une zone mémoire qui sera occupée par une `Personne`.

En vert : **l'instanciation** (la création) de cette `Personne` en mémoire.

En noir : le pointage de la référence `p` vers la `Personne` en mémoire. C'est une **affectation**.

# L'objet courant : this

```
return "Bonjour je m'appelle " + this.prenom + " " + this.nom + ", j'ai " + age + " ans";
```

Le mot-clé **this** représente l'objet courant au sein de la classe. Ce mot-clé n'est pas obligatoire, mais il permet dans certains cas de différencier l'attribut d'une classe d'une variable portant le même nom.

# Les constructeurs

```
/**
 * Constructeur sans argument
 */
public Personne(){
}

/**
 * Constructeur avec arguments
 */
public Personne(String nom, String prenom, Date dateNaissance) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    this.dateNaissance = dateNaissance;
}
```

- Les constructeurs sont optionnels et permettent de définir ce qui sera exécuté à l'instanciation de l'objet en mémoire.
- Par défaut il y a toujours un constructeur sans arguments, même si celui-ci n'existe pas dans le code.
- Attention cependant à ce constructeur par défaut qui disparaît si un autre constructeur est défini.

# La visibilité

- La visibilité d'un attribut ou d'une variable peut être :

**privée** : accessible uniquement de la classe

**private** String nom;

**protégée** : accessible de la classe et des classes qui en héritent

**protected** String nom;

**amie** (friendly) : de l'ensemble des classes du même package

String nom;

**publique** : de l'ensemble de l'application

**public** String nom;

# L'encapsulation

```
public String getNom() {  
    return nom;  
}  
  
public void setNom(String nom) {  
    this.nom = nom;  
}  
  
public String getPrenom() {  
    return prenom;  
}  
  
public void setPrenom(String prenom) {  
    this.prenom = prenom;  
}  
  
public Date getDateNaissance() {  
    return dateNaissance;  
}  
  
public void setDateNaissance(Date dateNaissance) {  
    this.dateNaissance = dateNaissance;  
}
```

- **L'encapsulation** consiste à mettre les attributs d'une classe avec une visibilité privée.
- L'accès aux attributs se fait alors par des méthodes dédiées que l'on nomme **accesseurs** ou **getters** et **setters**.

# Exemple de code utilisant la classe Personne

```
package demo;
import java.text.ParseException;
import java.text.SimpleDateFormat;

public class Exec1 {

    public static void main(String[] args) throws ParseException {
        Personne p = new Personne();
        p.setNom("Maillot");
        p.setPrenom("Paul");
        p.setDateNaissance(new SimpleDateFormat("dd/MM/yyyy").parse("23/09/1988"));

        System.out.println(p.getPresentation());
    }
}
```

Les classes

# Le mot-clé static

- Des attributs ou des méthodes peuvent être liés à la classe elle-même et non à l'objet généré.
- Dans ce cas, ces attributs et ces méthodes sont précédés par le mot-clé **static**.

# Exemple de code

```
package lestatic;  
  
public class Exec {  
    public static void main(String[] args) {  
        Util.ecrire("Bonjour");  
    }  
}
```

```
package lestatic;  
  
public class Util {  
    public static void ecrire(String message){  
        System.out.println(message);  
    }  
}
```

La méthode **ecrire** est précédée de **static**.

Notez le mot-clé **static** dans la signature de la méthode.



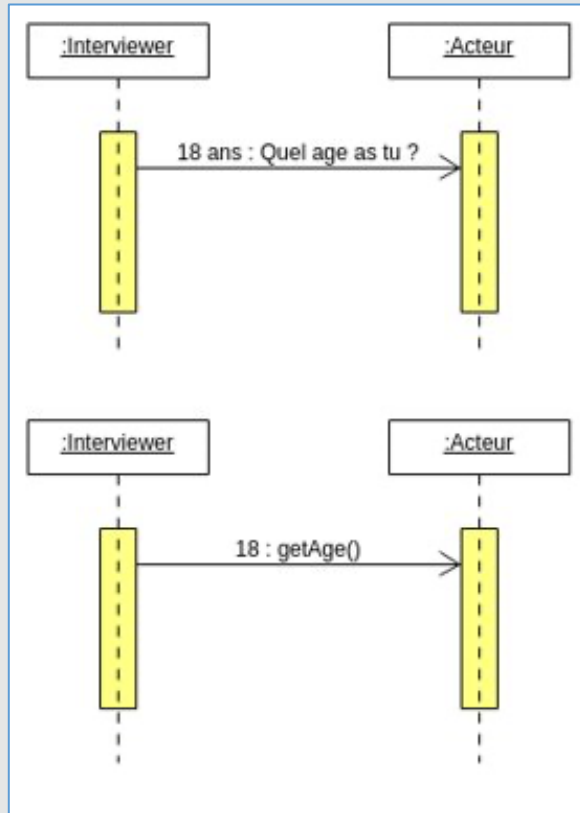
Les classes

# Les packages

- Les classes sont organisées au sein de packages et sous-packages.
- Objectif : garantir l'unicité d'une classe dans un écosystème.  
Exemple : *fr.formation.java.projetpoo*

La communication entre objets

# Diagramme de séquences de communication entre deux objets



Un objet **Interviewer** pose la question "*Quel âge as-tu ?*" à un objet **Acteur**.

Celui-ci lui répond "*18 ans*"

Cela peut correspondre à la ligne de code suivante dans la classe **Interviewer** :

```
Integer age = monActeur.getAge();
```

# La communication entre objets

## L'objet appelant

```
package demo;

import java.text.ParseException;

public class Interviewer {
    public static void main(String[] args) throws ParseException {
        // création de l'acteur
        Acteur george = new Acteur("George Clooney", "06/05/1961");

        // Interview de George Clooney
        System.out.println("Bonjour George, quel age as tu ?");
        Integer age = george.getAge();
        System.out.println("J'ai " + age + " ans");
    }
}
```

```
<terminated> Interviewer [Java Ap
Bonjour George, quel age as tu ?
J'ai 56 ans
```

La communication entre objets

# L'objet appelé

```
import java.text.ParseException;

public class Acteur {
    public String nom;
    public Date ddn;

    public Acteur() {
    }

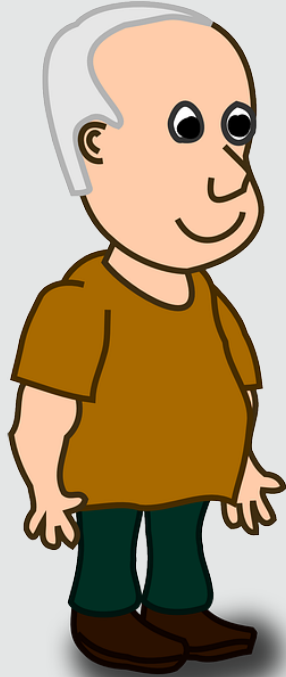
    public Acteur(String nom, String ddnInString) throws ParseException {
        super();
        this.nom = nom;
        this.ddn = new SimpleDateFormat("dd/MM/yyyy").parse(ddnInString);
    }

    public Integer getAge(){
        long age = (new Date().getTime() - ddn.getTime()) / 1000 / 60 / 60 / 24 / 365 ;
        return (int)age;
    }
}
```

L'héritage

# La problématique

Durant  
Rémi  
Né le 21/12/1972  
Spécialité: Java



**Formateur**

Personne
nom : String
prenom : String
dateNaissance : Date
getPresentation() : String

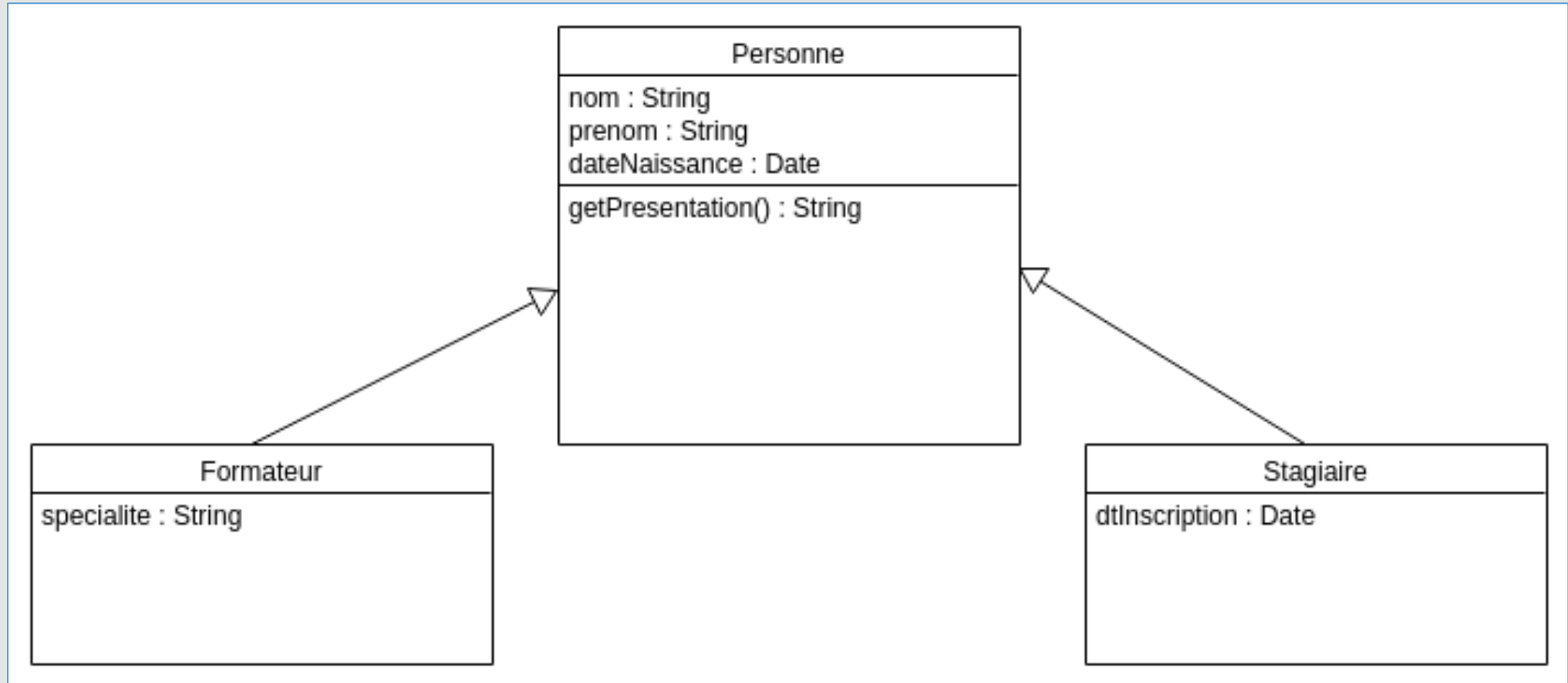


Maillot  
Paul  
Né le 23/09/1988  
Date d'inscription :  
12/09/2017

**Stagiaire**

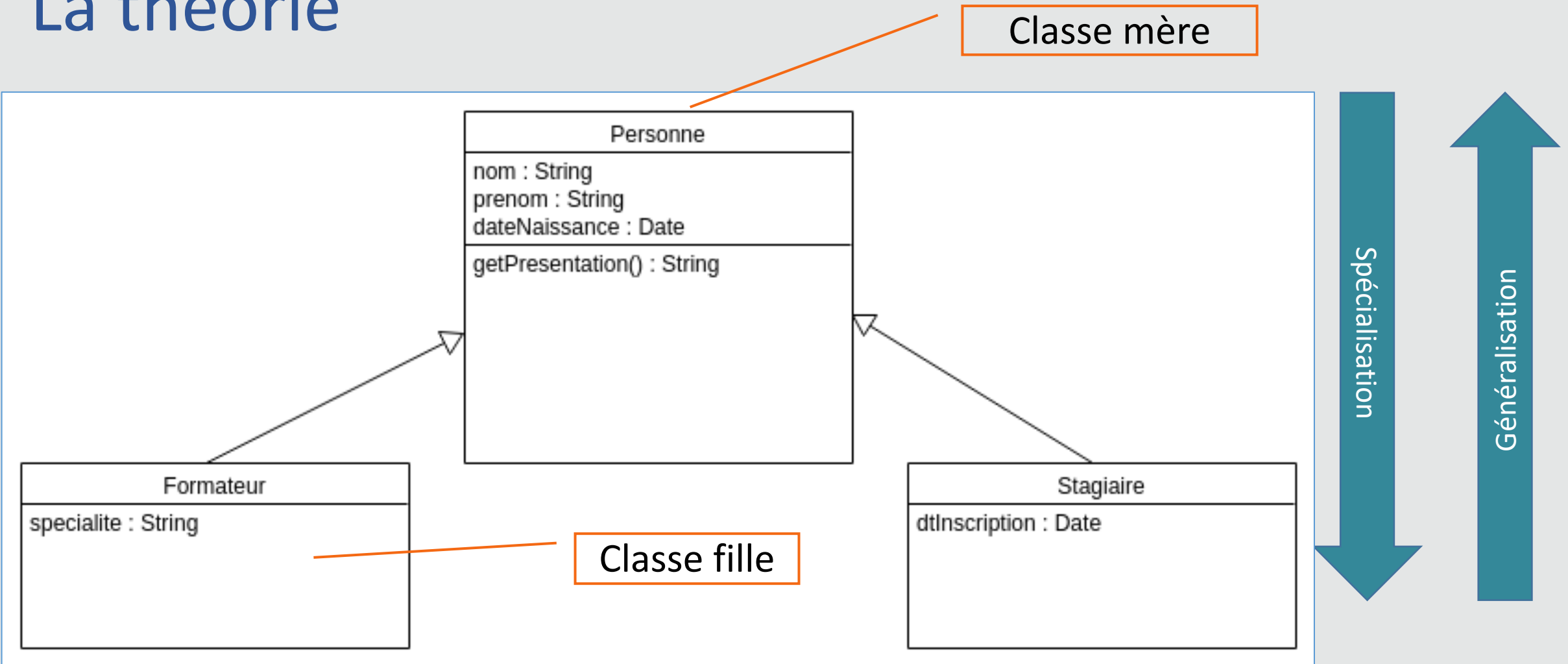
L'héritage

# La solution



# L'héritage

## La théorie



L'héritage

# L'héritage dans le code

```
package demo;

public class Formateur extends Personne {
    protected String specialite;

    public Formateur() {
        super();
    }

    public String getSpecialite() {
        return specialite;
    }

    public void setSpecialite(String specialite) {
        this.specialite = specialite;
    }
}
```

Le mot-clé **extends** définit la relation d'héritage

**super()** appelle le constructeur de la classe mère



## L'héritage

# La surcharge

```
/**
 * La personne se présente
 * @return le texte de la présentation
 */
public String getPresentation() {
    // calcul de l'age
    long age = (new Date().getTime() - dateNaissance.getTime()) / 1000 / 60 / 60 / 24 / 365 ;

    return "Bonjour je m'appelle " + this.prenom + " " + this.nom + ", j'ai " + age + " ans";
}

/**
 * methode surchargée permettant de gérer la langue
 * @param langue
 * @return le texte de présentation
 */
public String getPresentation(String langue) {
    // calcul de l'age
    long age = (new Date().getTime() - dateNaissance.getTime()) / 1000 / 60 / 60 / 24 / 365 ;

    if("anglais".equals(langue)){
        return "Hello my name is " + this.prenom + " " + this.nom + ", I am " + age + " years old";
    }
    return getPresentation();
}
```

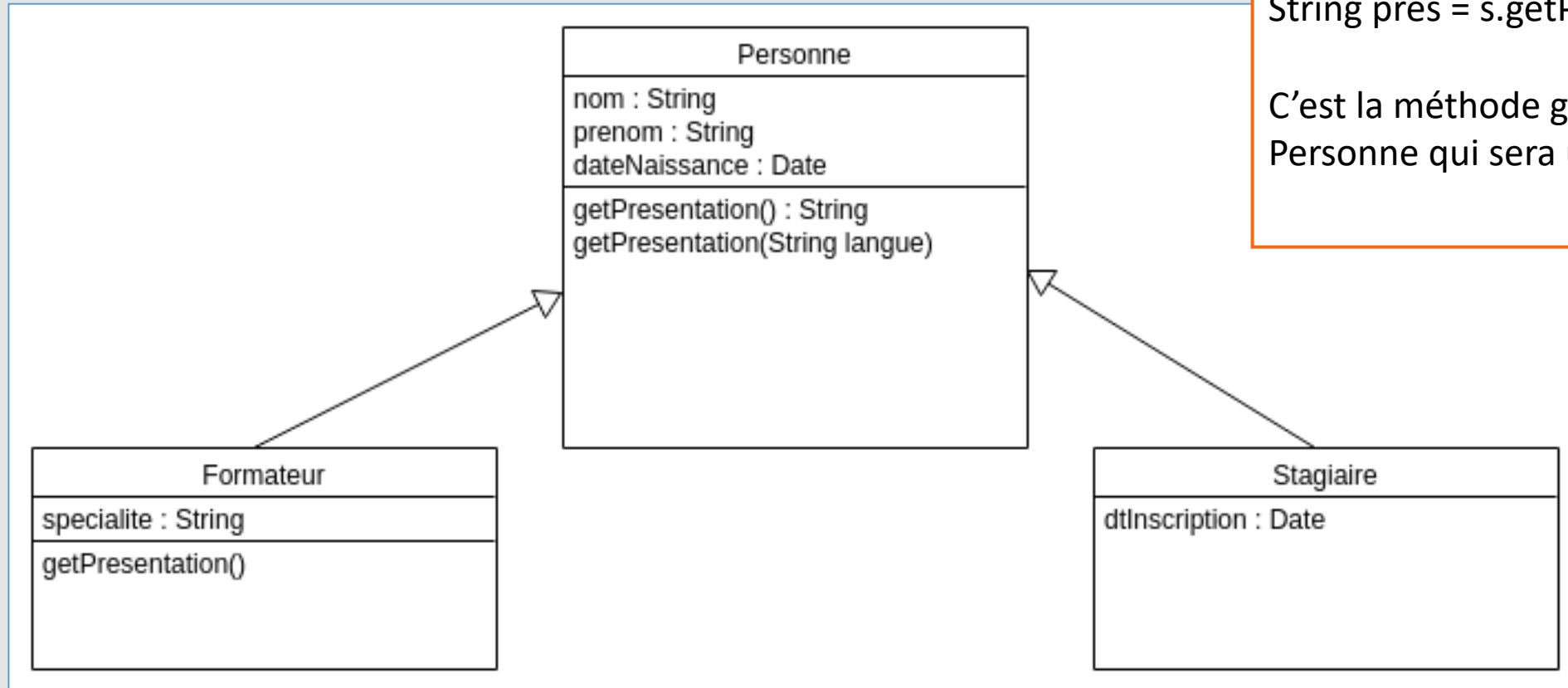
- Plusieurs méthodes peuvent porter le même nom si leurs types d'arguments sont différents. On dit alors que leurs signatures sont différentes.

L'héritage

# Le polymorphisme

```
Stagiaire s = new Stagiaire();  
String pres = s.getPresentation()
```

C'est la méthode `getPresentation()` de `Personne` qui sera utilisée



# Parlons de types

```
Formateur s = new Formateur();
```

```
(s instanceof Formateur) ➔ Vrai
```

```
(s instanceof Personne) ➔ Vrai
```

```
(s instanceof Stagiaire) ➔ Faux
```

- En POO une variable peut avoir plusieurs types.
- La variable p est de type Formateur, mais aussi de type Personne.
- L'opérateur **instanceof** permet de tester si une instance est d'un type défini.

## Le cast

Personne p = new Formateur();

p.specialite → Impossible

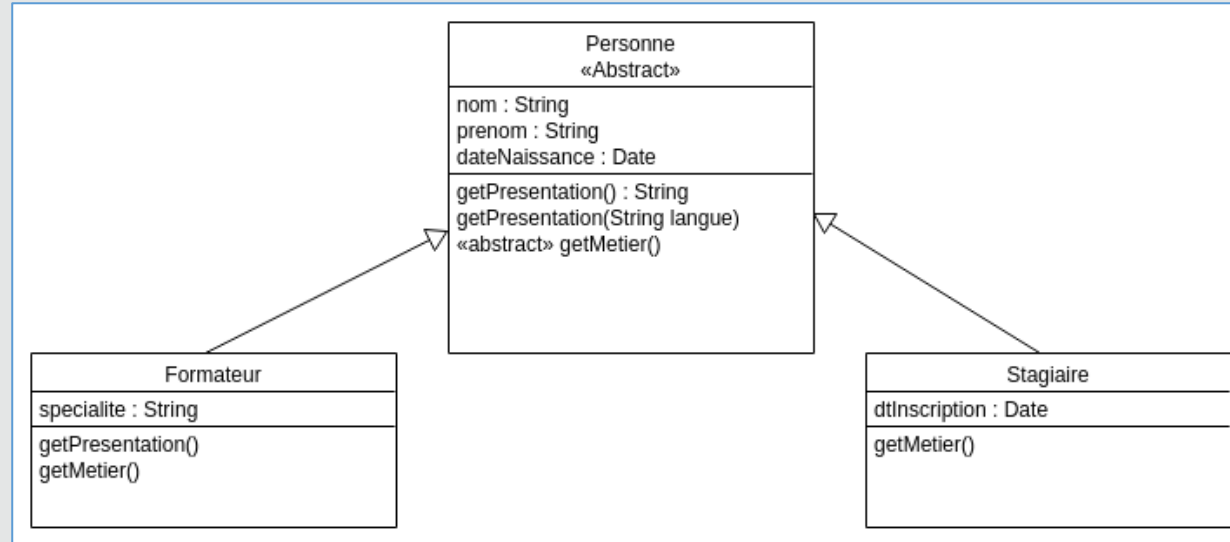
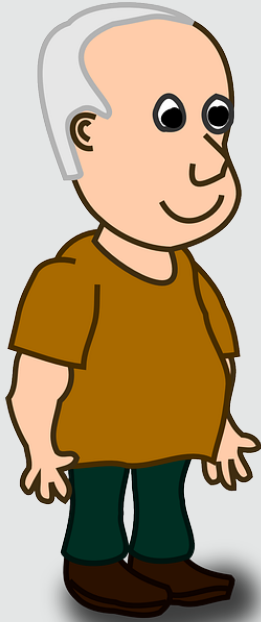
((Formateur)p).specialite → Possible

- p est déclaré en tant que Personne, mais instancié en temps que Formateur.
- C'est donc un Formateur mais pour l'utiliser en tant que Formateur il faudra faire un **cast** vers le type Formateur.

Les interfaces

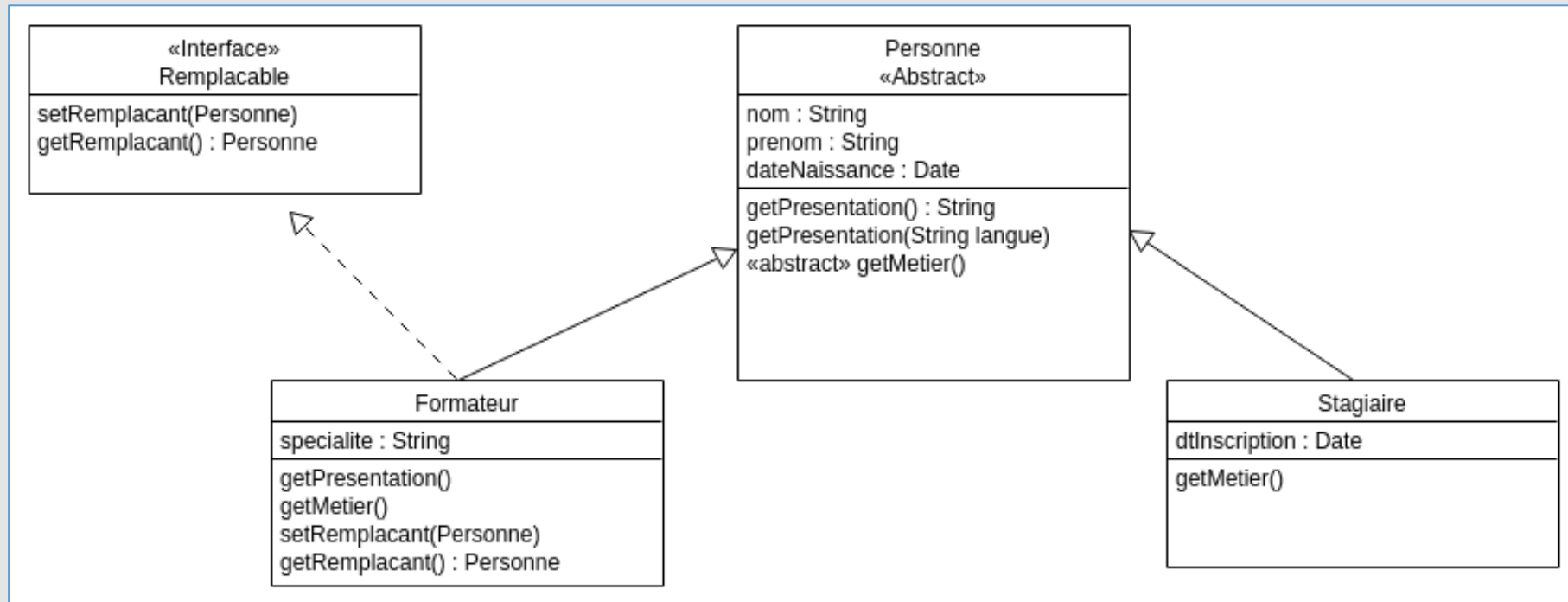
# La problématique

Peut être remplacé



- Il est parfois nécessaire de donner des **comportements** particuliers à certaines classes autrement que par la hiérarchisation fournie par l'héritage.

# La solution




- Une interface permet d'imposer un ou des **contrats** à la classe qui l'utilise.

## Les interfaces

# Dans le code

```
package demoInterface;

public interface Remplacable {
    public void setRemplacant(Personne p);
    public Personne getRemplacant();
}
```



```
public class Formateur extends Personne implements Remplacable {
    protected String specialite;

    protected Personne remplacant;

    @Override
    public void setRemplacant(Personne p) {
        remplacant = p;
    }

    @Override
    public Personne getRemplacant() {
        return remplacant;
    }
}
```

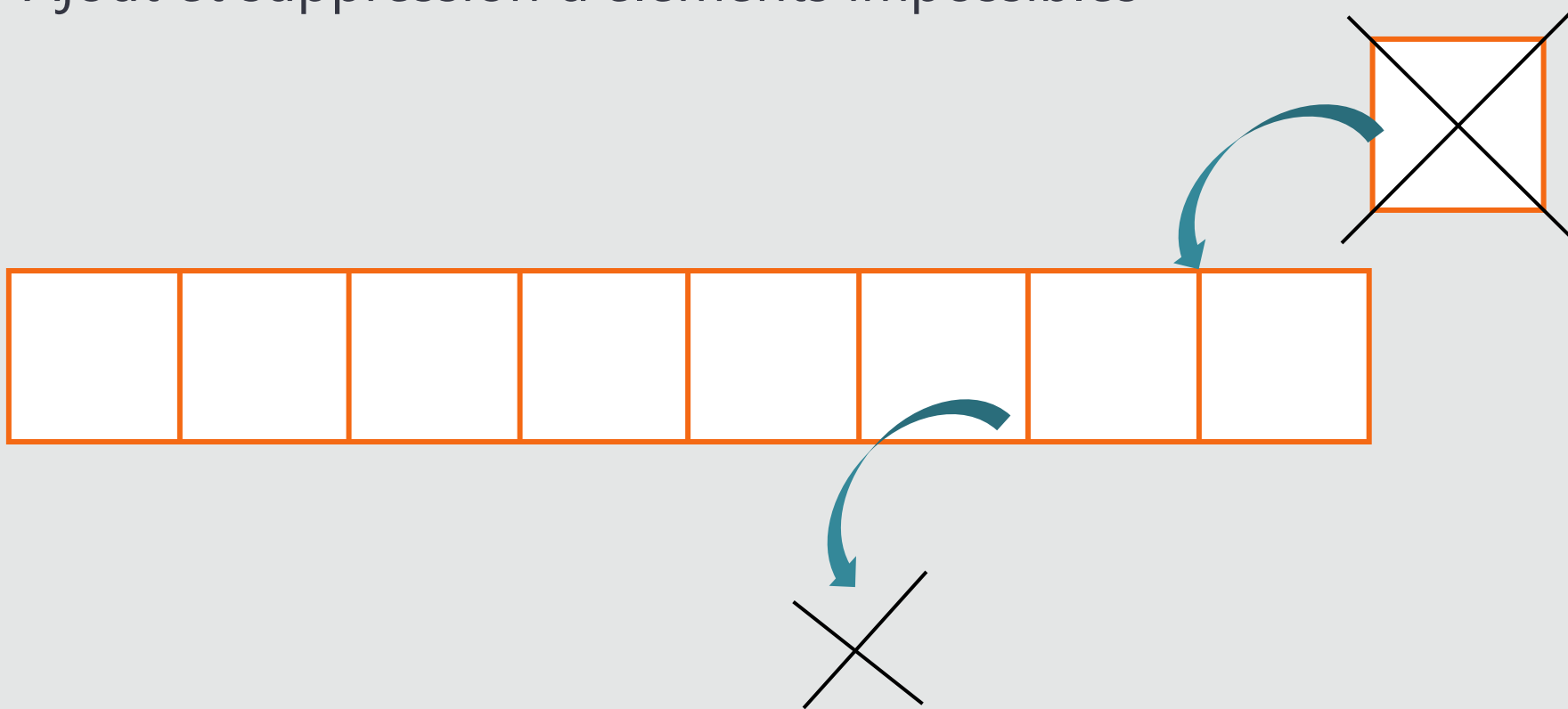
# Dans le code

```
public class Exec {  
    public static void main(String[] args) throws ParseException {  
        Formateur louis = new Formateur();  
        louis.setNom("Duval");  
        louis.setPrenom("Louis");  
        louis.setDateNaissance(new SimpleDateFormat("dd/MM/yyyy").parse("28/05/1973"));  
        louis.setSpecialite("Java");  
  
        Formateur michel = new Formateur();  
        michel.setNom("Durant");  
        michel.setPrenom("Michel");  
        michel.setDateNaissance(new SimpleDateFormat("dd/MM/yyyy").parse("01/01/1980"));  
        michel.setSpecialite("Java");  
  
        Stagiaire kevin = new Stagiaire();  
        kevin.setNom("Martel");  
        kevin.setPrenom("kevin");  
        kevin.setDateNaissance(new SimpleDateFormat("dd/MM/yyyy").parse("20/12/1998"));  
  
        Stagiaire leo = new Stagiaire();  
        leo.setNom("Martin");  
        leo.setPrenom("Léo");  
        leo.setDateNaissance(new SimpleDateFormat("dd/MM/yyyy").parse("2/11/1996"));  
  
        // attribution des remplaçants  
        louis.setRemplacant(michel);  
        michel.setRemplacant(louis);  
  
        // leo.setRemplacant(louis); ==> impossible  
  
    }  
}
```



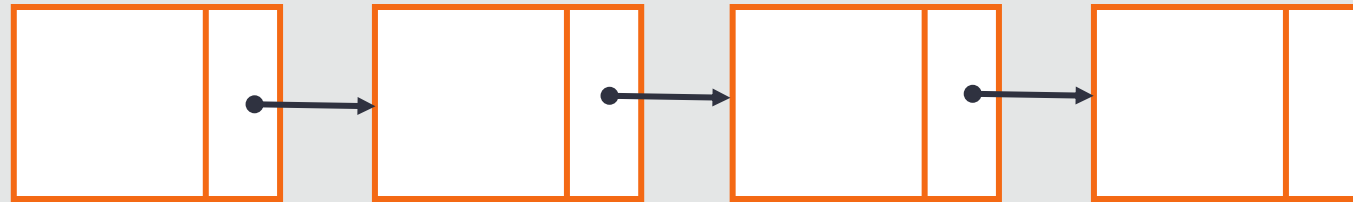
# La problématique

- Ajout et suppression d'éléments impossibles

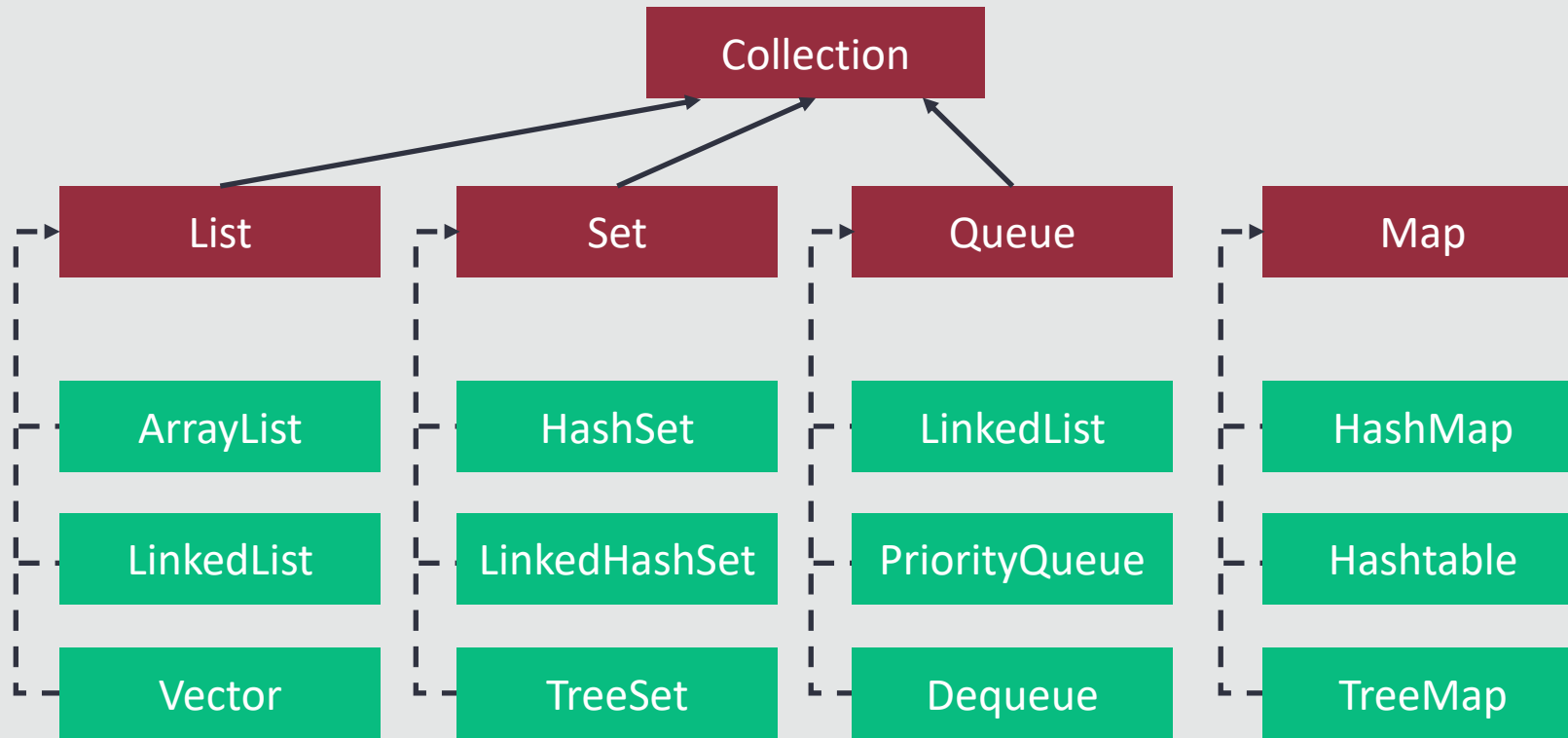


Les collections

# La solution : la collection



# La théorie



Exemples de classes permettant de gérer des problématiques particulières :

- **List**
- **Set**
- **Map**
- **Queue**

# Exemple d'une liste

```
package demo;

import java.util.ArrayList;
import java.util.List;

public class Exec2 {
    public static void main(String[] args) {
        // Création de la liste. Ici on choisit l'implémentation ArrayList
        // c'est une liste de Personne. Elle stockera donc des personnes, mais aussi des formateurs
        // ou des stagiaires (qui sont aussi des personnes).
        List<Personne> lst = new ArrayList<Personne>();

        Personne p = new Personne();
        p.setNom("Dupont");
        p.setPrenom("Jean");

        Personne p2 = new Personne();
        p2.setNom("Durant");
        p2.setPrenom("Pierre");

        // ajout d'une personne à la liste
        lst.add(p);
        // ajout d'une autre personne
        lst.add(p2);
        // affichage du premier élément
        System.out.println(lst.get(0)); // on commence par l'index 0
        // retrait du premier élément
        lst.remove(0);
    }
}
```

# Exemple d'une Map

```
package demo;

import java.util.HashMap;
import java.util.Map;

public class Exec3 {
    public static void main(String[] args) {
        // Création de la map. Ici on choisit l'implémentation HashMap.
        // c'est une Map de Personne. Elle stockera donc des personnes, mais aussi des formateurs
        // ou des stagiaires (qui sont aussi des personnes).
        // La référence se fera par une clé de type String
        Map<String, Personne> map = new HashMap<String, Personne>();

        Personne p = new Personne();
        p.setNom("Dupont");
        p.setPrenom("Jean");

        Personne p2 = new Personne();
        p2.setNom("Durant");
        p2.setPrenom("Pierre");

        // ajout de personnes à la map
        map.put("Jannot", p);
        map.put("Pierrot", p2);

        // récupération d'une personne de la map
        System.out.println(map.get("Jannot"));

        // retrait d'un élément de la map
        map.remove("Jannot");
    }
}
```

# Le foreach

```
package demo;

import java.util.ArrayList;

public class Exec4 {
    public static void main(String[] args) {
        List<Personne> lst = new ArrayList<Personne>();

        Personne p = new Personne();
        p.setNom("Dupont");
        p.setPrenom("Jean");

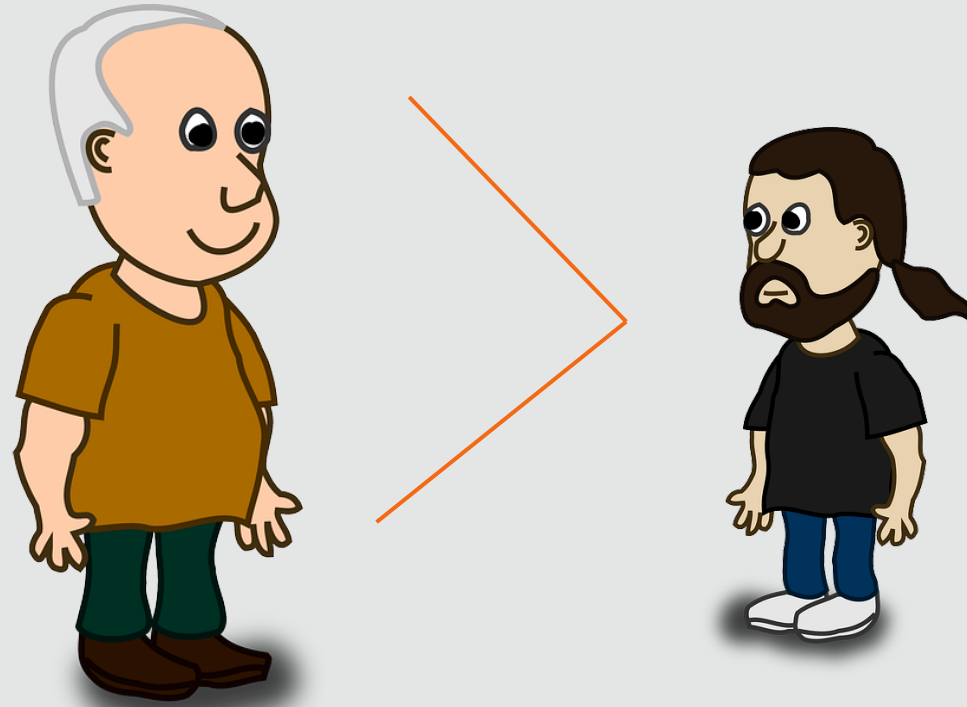
        Personne p2 = new Personne();
        p2.setNom("Durant");
        p2.setPrenom("Pierre");

        lst.add(p);
        lst.add(p2);

        // A chaque tour de boucle la personne suivante de la liste lst est positionnée dans la variable 'personne'
        for (Personne personne : lst) {
            System.out.println(personne);
        }

        // Cette façon de parcourir une liste est à prohiber.
        // En effet à chaque tour de boucle la méthode get refait un parcours de la liste jusqu'à l'élément à trouver.
        for (int i = 0; i < lst.size(); i++) {
            System.out.println(lst.get(i));
        }
    }
}
```

# Le tri



- Quel que soit l'algorithme de tri ou les éléments à trier, une seule chose est absolument indispensable : pouvoir comparer deux éléments

# Le tri (interface Comparable)

```
public class Personne implements Comparable<Personne>{
```

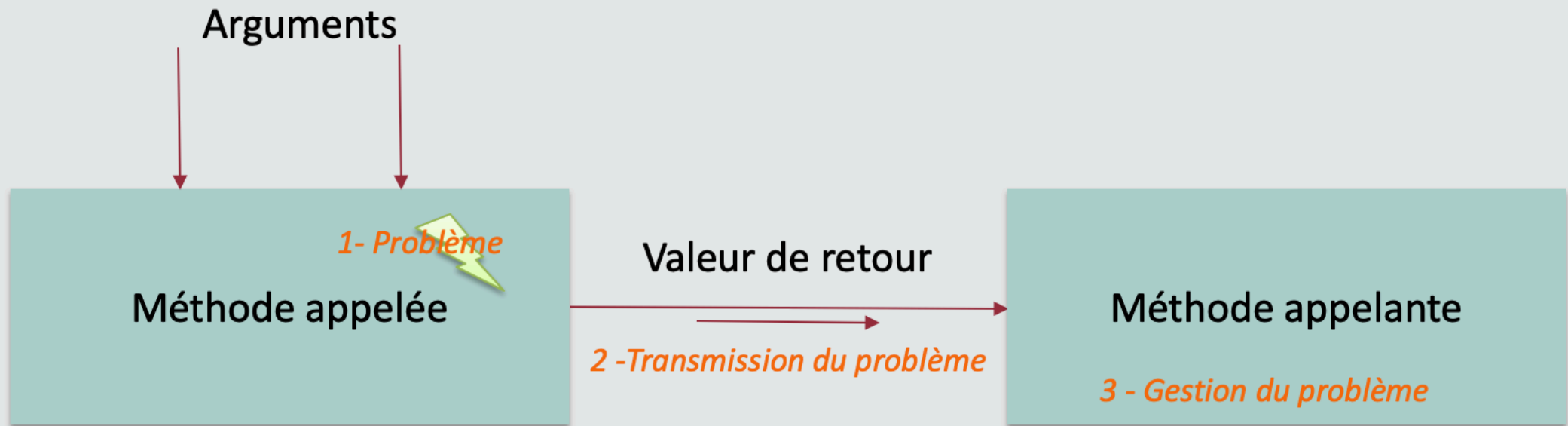
```
@Override  
public int compareTo(Personne o) {  
    return this.nom.compareTo(o.nom);  
}
```

```
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.List;  
  
public class Exec5 {  
    public static void main(String[] args) {  
        List<Personne> lst = new ArrayList<Personne>();  
  
        Personne p = new Personne();  
        p.setNom("Dupont");  
        p.setPrenom("Jean");  
  
        Personne p2 = new Personne();  
        p2.setNom("Durant");  
        p2.setPrenom("Pierre");  
  
        Personne p3 = new Personne();  
        p3.setNom("Bern");  
        p3.setPrenom("Karine");  
  
        lst.add(p);    lst.add(p2);    lst.add(p3);  
  
        System.out.println(lst);  
  
        Collections.sort(lst);  
  
        System.out.println(lst);  
  
    }  
}
```



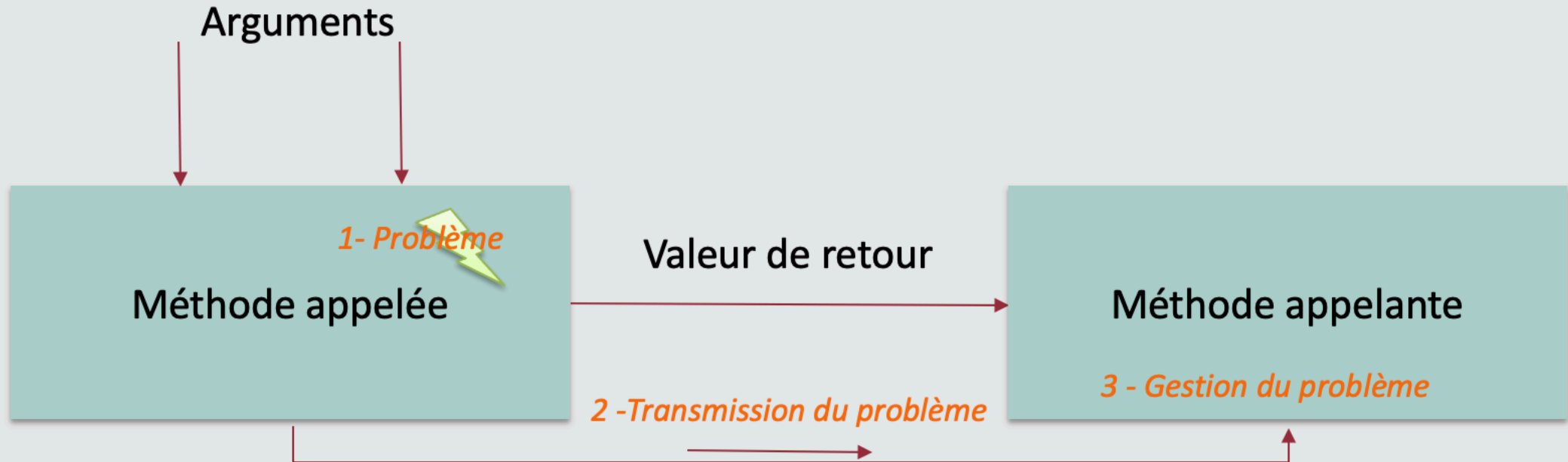
Les exceptions

# La problématique



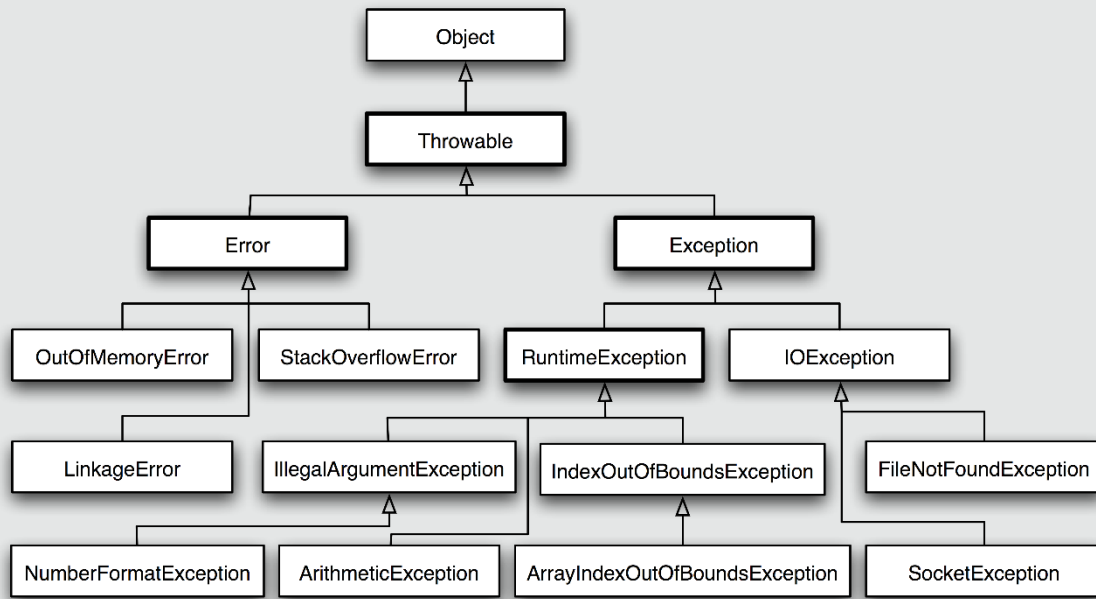
Les exceptions

# La solution



## Les exceptions

# Hiérarchie des exceptions



- Chaque exception représente un problème particulier.
- Une exception est une **classe**.
- La hiérarchie peut être enrichie avec d'autres classes d'exception.

## Les exceptions

# Gérer une exception

```
try {  
    Instruction à surveiller  
}  
catch (TypeException e) {  
    Code à exécuter en cas d'exception  
}
```

```
public class Exec6 {  
  
    /**  
     * Méthode appelée  
     * Ici la méthode gère l'exception  
     * @param date  
     * @return  
     */  
    private static Integer calculAge(String date) {  
        Integer age = 0;  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");  
        try {  
            Date dt = sdf.parse(date);  
            // la méthode parse déclenche une exception  
            // le flux d'exécution est alors redirigé vers le catch  
            age = (int)((new Date().getTime() - dt.getTime()) / 1000 / 60 / 60 / 24 / 365); // non exécuté  
        } catch (ParseException e) {  
            // le catch est exécuté et affiche le message de l'exception qui s'est produite  
            System.out.println(e.getMessage());  
        }  
        return age; // l'age retourné est donc de 0 (comme initialisé)  
    }  
  
    /**  
     * Méthode appelante  
     * @param args  
     */  
    public static void main(String[] args) {  
        Integer age = calculAge("28 mai 1998"); // la date est mal formatée par rapport à ce que fait la méthode  
        System.out.println("votre age est de " + age + " ans");  
    }  
}
```

## Les exceptions

# Transmettre une exception

**throws** : si elle est détectée, l'exception est transmise à la méthode appelante...

.. qui la traite à l'aide d'un try catch

```
public class Exec7 {  
  
    /**  
     * Méthode appelée Ici la méthode transmet l'exception  
     * @param date  
     * @return  
     * @throws ParseException  
     */  
    private static Integer calculAge(String date) throws ParseException { // L'exception est transmise  
        Integer age = 0;  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");  
        Date dt = sdf.parse(date);  
        age = (int) ((new Date().getTime() - dt.getTime()) / 1000 / 60 / 60 / 24 / 365); // non  
        return age;  
    }  
  
    /**  
     * Méthode appelante  
     * @param args  
     */  
    public static void main(String[] args) {  
        Integer age=0;  
        try {  
            // la date est mal formatée par rapport à ce que fait la méthode  
            // la méthode appelée transmet à la méthode appelante l'exception qu'elle doit au final gérer  
            age = calculAge("28 mai 1998");  
        } catch (ParseException e) {  
            System.out.println(e.getMessage());  
        }  
        System.out.println("votre age est de " + age + " ans");  
    }  
}
```

## Les exceptions

# Les exceptions personnalisées

```
package demo;

/**
 * Cette classe est une Exception personnalisée qui sera
 * utilisée pour signaler un problème de date. Elle
 * hérite de Exception.
 */
public class PersoDateException extends Exception {

    /**
     * La surcharge du constructeur permet d'attribuer un
     * message particulier à l'exception
     * @param message
     */
    public PersoDateException(String message) {
        super(message);
    }
}
```

Capture et transformation de l'exception  
*ParseException* en nouvelle exception personnalisée  
*PersoDateException* par un **throw**.  
L'exception *PersoDateException* est transmise à la  
méthode appelante par un **throws** ...

... la méthode appelante traite cette nouvelle  
exception.

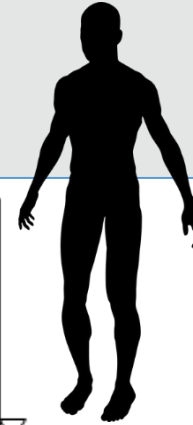
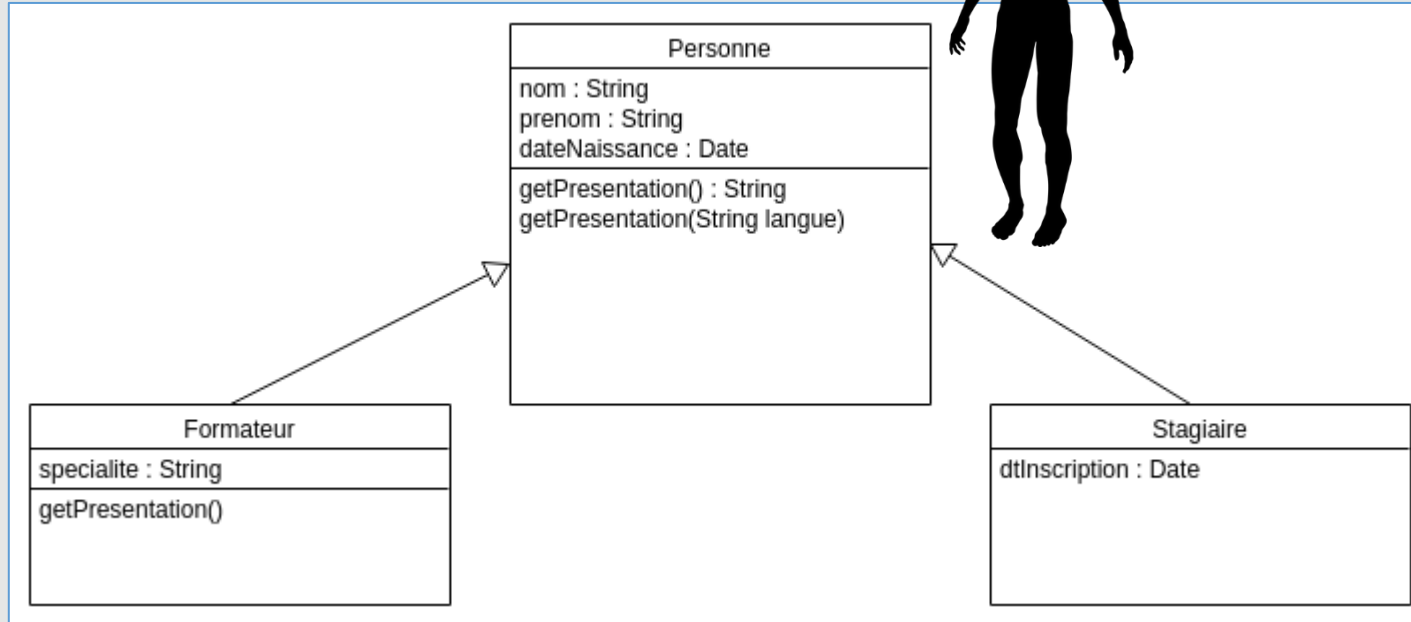
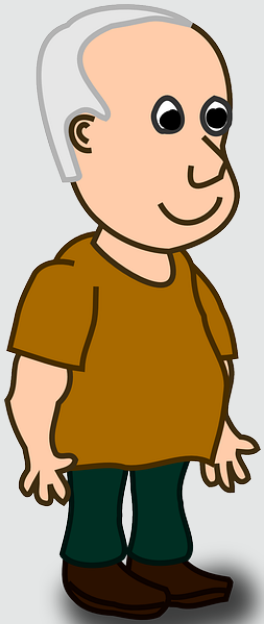
```
public class Exec8 {

    /**
     * Méthode appelée
     * C'est dorénavant une exception personnalisée que transmettra la méthode
     * Cela permet d'encapsuler les exceptions de bas niveau pour les remplacer
     * par des exceptions de plus haut niveau
     * @param date
     * @return
     * @throws PersoDateException
     */
    private static Integer calculAge(String date) throws PersoDateException{
        Integer age = 0;
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        Date dt;
        try {
            dt = sdf.parse(date);
        } catch (ParseException e) {
            throw new PersoDateException("La date n'est pas du bon format");
        }
        age = (int) ((new Date().getTime() - dt.getTime()) / 1000 / 60 / 60 / 24 / 365);
        return age;
    }

    public static void main(String[] args) {
        Integer age=0;
        try {
            age = calculAge("28 mai 1998");
        } catch (PersoDateException e) {
            // C'est l'exception personnalisée qui est traitée ici
            System.out.println(e.getMessage());
        }
        System.out.println("votre age est de " + age + " ans");
    }
}
```

Les classes abstraites

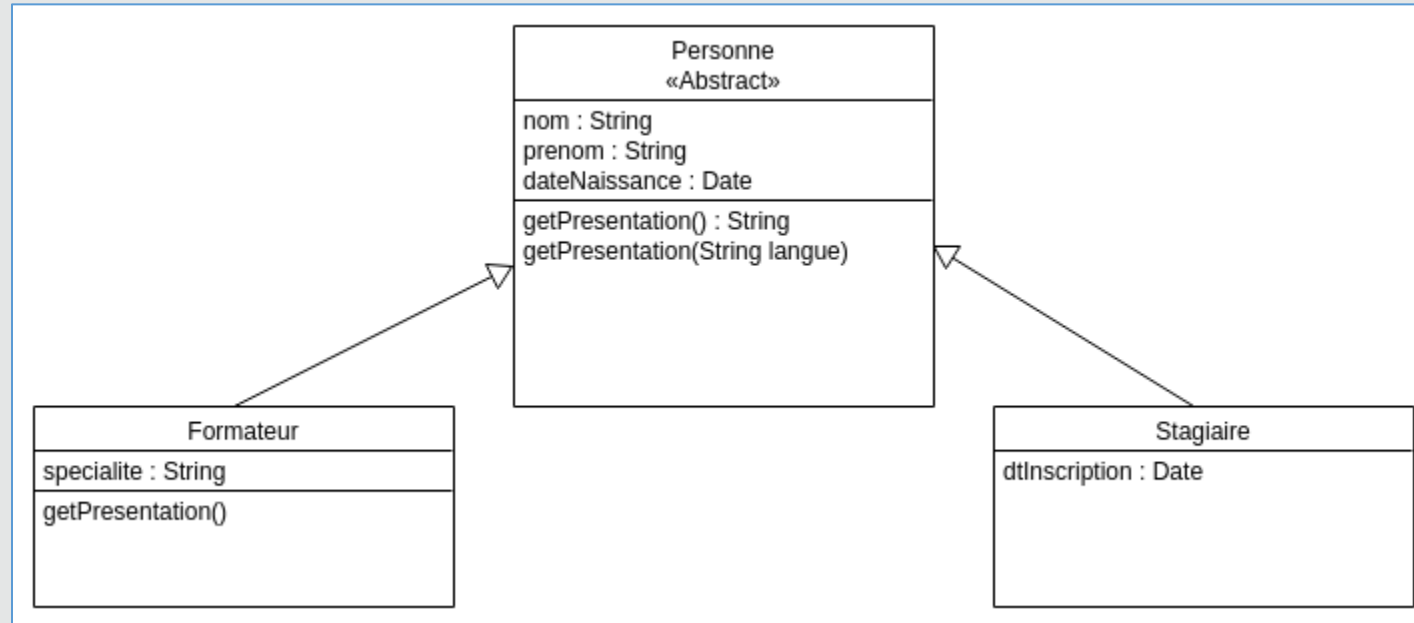
# La problématique



- On peut créer des Formateurs, des Stagiaires mais nous ne souhaitons pas pouvoir créer des Personnes (qui ne sont ni Stagiaires, ni Formateurs),

Les classes abstraites

# La solution



- Il faut donc rendre la classe **Personne** **Abstraite**.



# La théorie

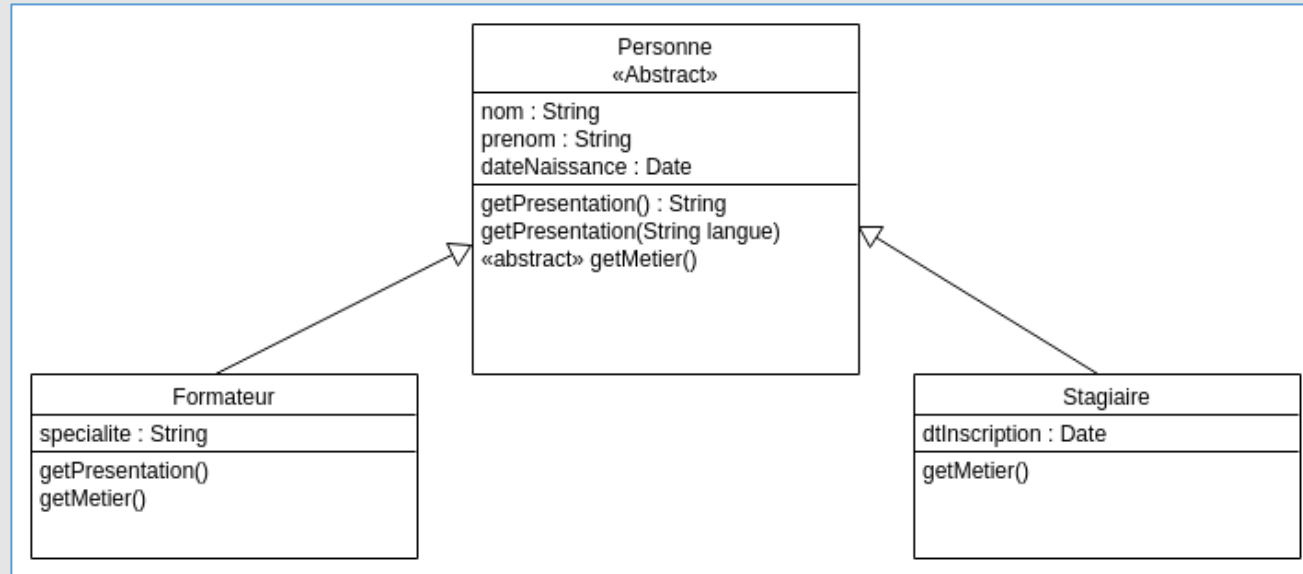
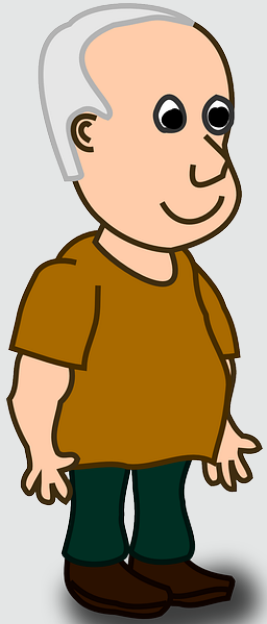
- Une classe abstraite est une classe qui ne peut pas être instanciée.

**Personne p = new Personne()**



Les classes abstraites

# Les méthodes abstraites

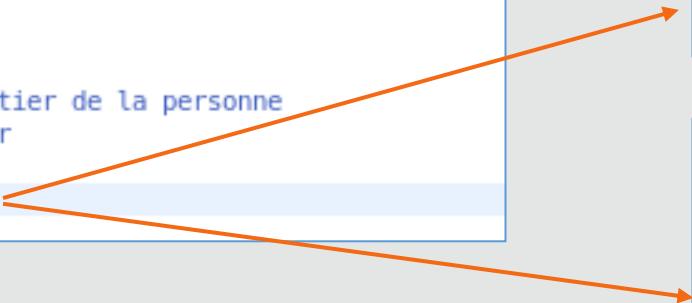


- Une méthode abstraite ne peut être déclarée que dans une classe abstraite.
- Cette méthode n'est qu'une signature, elle n'a pas de code.
- Cela impose aux classes filles d'implémenter cette méthode.

# Les classes abstraites

## Dans le code

```
public abstract class Personne implements Comparable<Personne>{  
    // ATTRIBUTS  
    protected String nom;  
    protected String prenom;  
    protected Date dateNaissance;  
  
    /**  
     * Méthode abstraite qui retourne le métier de la personne  
     * @return la phrase décrivant le métier  
     */  
    public abstract String getMetier();  
}
```



```
public class Formateur extends Personne {  
    protected String specialite;  
  
    @Override  
    public String getMetier() {  
        return "je suis Formateur";  
    }  
}
```

```
public class Stagiaire extends Personne {  
    protected Date dtInscription;  
  
    @Override  
    public String getMetier() {  
        return "Je suis Stagiaire";  
    }  
}
```

# Les classes abstraites

## Dans le code

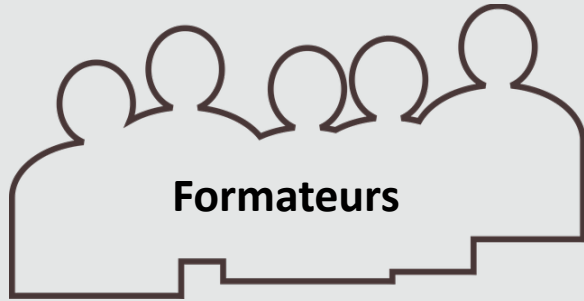
```
1 package demoAbstract;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class Exec {
7     public static void main(String[] args) {
8         // Personne p = new Personne(); impossible
9
10        // On peut créer des stagiaires et des formateurs en tant que personnes
11        Personne p1 = new Stagiaire();
12        Personne p2 = new Formateur();
13
14        // On peut gérer des listes de personnes
15        List<Personne> list = new ArrayList<Personne>();
16        list.add(p1);
17        list.add(p2);
18
19        // chaque personne pourra donner son métier
20        for (Personne personne : list) {
21            System.out.println(personne.getMetier());
22        }
23    }
24 }
25 }
26 }
```

<terminated> Exec [Java Application]

Je suis Stagiaire  
je suis Formateur

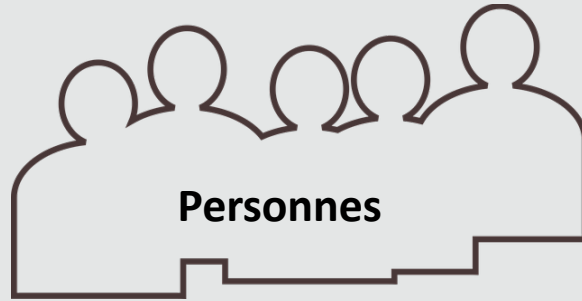
Les génériques

# La problématique



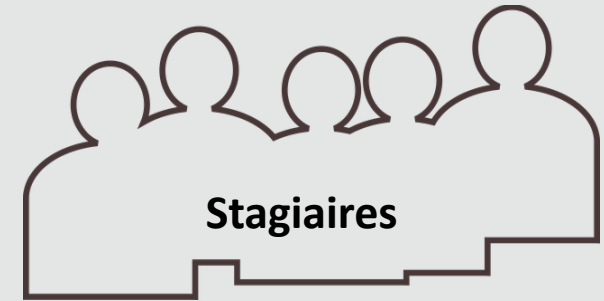
class  
CommunauteFormateurs

**Formateur** chef  
**List<Formateur>** membres



class  
CommunautePersonnes

**Personne** chef  
**List<Personne>** membres



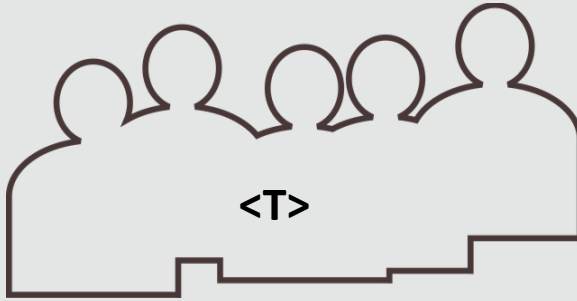
class  
CommunauteStagiaires

**Stagiaire** chef  
**List<Stagiaire>** membres

Les classes **CommunauteFormateurs**, **CommunautePersonnes** et **CommunauteStagiaires** sont identiques sauf qu'elles ne gèrent pas le même type d'élément.

Les génériques

# La solution



```
class Communaute<T>
```

T chef

**List<T>** membres

- Depuis Java 5, les classes peuvent être **génériques**.

# Les génériques

## Dans le code

```
package Generic;

import java.util.ArrayList;
import java.util.List;

public class Communaute<T extends Personne> {
    private String nom;
    private List<T> membres = new ArrayList<T>();
    private T responsable;

    public Communaute() {
    }

    public Communaute(String nom) {
        this.nom = nom;
    }

    public void addMembre(T membre){
        membres.add(membre);
    }

    public void setResponsable(T membre){
        // cas où le membre n'est pas déjà membre de la communauté
        if(!membres.contains(membre)){
            addMembre(membre);
        }
        responsable = membre;
    }

    public void presentation(){
        System.out.println("Nous sommes la communauté "+ nom);
        System.out.println("Le mot de notre responsable : "+responsable.getPresentation());
        System.out.println("voici la présentation de nos membres : ");
        for (T membre : membres) {
            System.out.println(membre.getPresentation());
        }
    }
}
```

```
public class Communaute<T extends Personne> {
```

La classe Communaute prend en paramètre un type T qui doit obligatoirement hériter de Personne ou être Personne

```
private String nom;
private List<T> membres = new ArrayList<T>();
private T responsable;
```

Les attributs de la classe peuvent utiliser le type paramètre T

# Les génériques

## Dans le code

```
package Generic;

import java.text.ParseException;

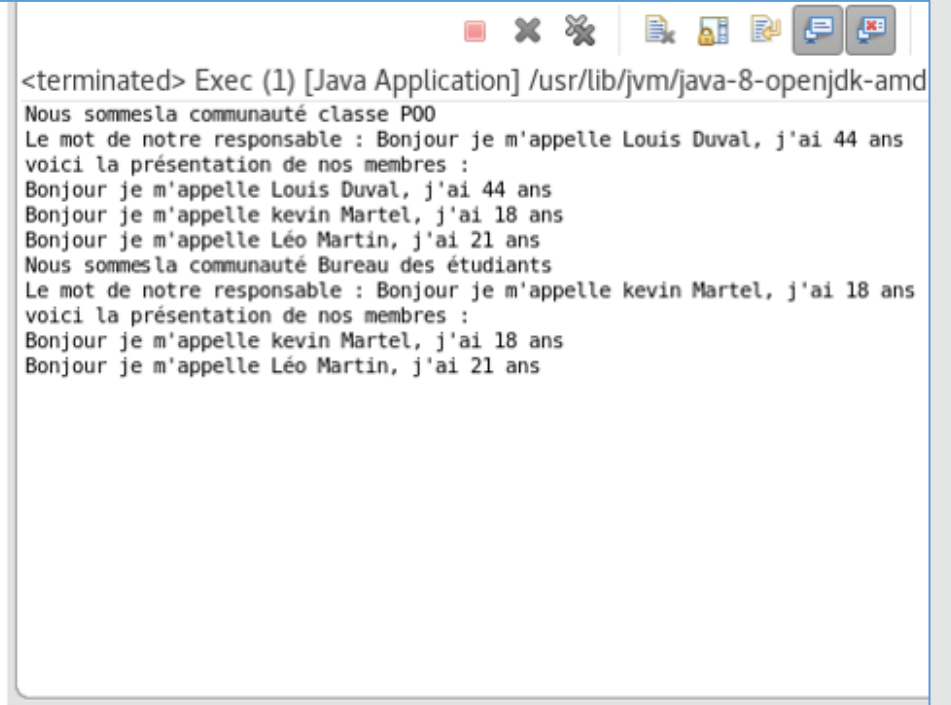
public class Exec {
    public static void main(String[] args) throws ParseException {
        Formateur louis = new Formateur();
        louis.setNom("Duval");
        louis.setPrenom("Louis");
        louis.setDateNaissance(new SimpleDateFormat("dd/MM/yyyy").parse("28/05/1973"));
        louis.setSpecialite("Java");

        Stagiaire kevin = new Stagiaire();
        kevin.setNom("Martel");
        kevin.setPrenom("kevin");
        kevin.setDateNaissance(new SimpleDateFormat("dd/MM/yyyy").parse("20/12/1998"));

        Stagiaire leo = new Stagiaire();
        leo.setNom("Martin");
        leo.setPrenom("Léo");
        leo.setDateNaissance(new SimpleDateFormat("dd/MM/yyyy").parse("2/11/1996"));

        // création d'une communauté de Personnes
        Communaute<Personne> classeJava = new Communaute<Personne>("classe P00");
        classeJava.setResponsable(louis);
        classeJava.addMembre(kevin);
        classeJava.addMembre(leo);
        classeJava.prensentation();

        // création d'une communauté de stagiaires
        Communaute<Stagiaire> bde = new Communaute<Stagiaire>("Bureau des étudiants");
        // bde.setResponsable(louis); ==> Impossible car la communauté n'accepte que des stagiaires
        bde.addMembre(kevin);
        bde.addMembre(leo);
        bde.setResponsable(kevin);
        bde.prensentation();
    }
}
```



<terminated> Exec (1) [Java Application] /usr/lib/jvm/java-8-openjdk-amd  
Nous sommes la communauté classe P00  
Le mot de notre responsable : Bonjour je m'appelle Louis Duval, j'ai 44 ans  
voici la présentation de nos membres :  
Bonjour je m'appelle Louis Duval, j'ai 44 ans  
Bonjour je m'appelle kevin Martel, j'ai 18 ans  
Bonjour je m'appelle Léo Martin, j'ai 21 ans  
Nous sommes la communauté Bureau des étudiants  
Le mot de notre responsable : Bonjour je m'appelle kevin Martel, j'ai 18 ans  
voici la présentation de nos membres :  
Bonjour je m'appelle kevin Martel, j'ai 18 ans  
Bonjour je m'appelle Léo Martin, j'ai 21 ans