

R2-01-03

DÉVELOPPEMENT ORIENTÉ OBJETS

QUALITÉ DE DÉVELOPPEMENT

Semaine 2

- Héritage
- Classes abstraites




Francis Brunet-Manquat

Université Grenoble Alpes



IUT 2 – Département Informatique

Points abordés

Exercice 1 : héritage

-  Classe mère et classes filles
-  Surcharge et redéfinition
-  Classe abstraite

Exercice 2 : création et association

-  Rappel de la première semaine
-  GIT

Exercice « fil rouge » : la bataille de Faërun

Mécanisme d'héritage (1/3)

 L'héritage est un mécanisme permettant à des classes *filles* d'hériter des caractéristiques de classe(s) *mère(s)*.

 En java, **héritage simple**

 Une ou plusieurs classes *filles* héritent d'une classe *mère*

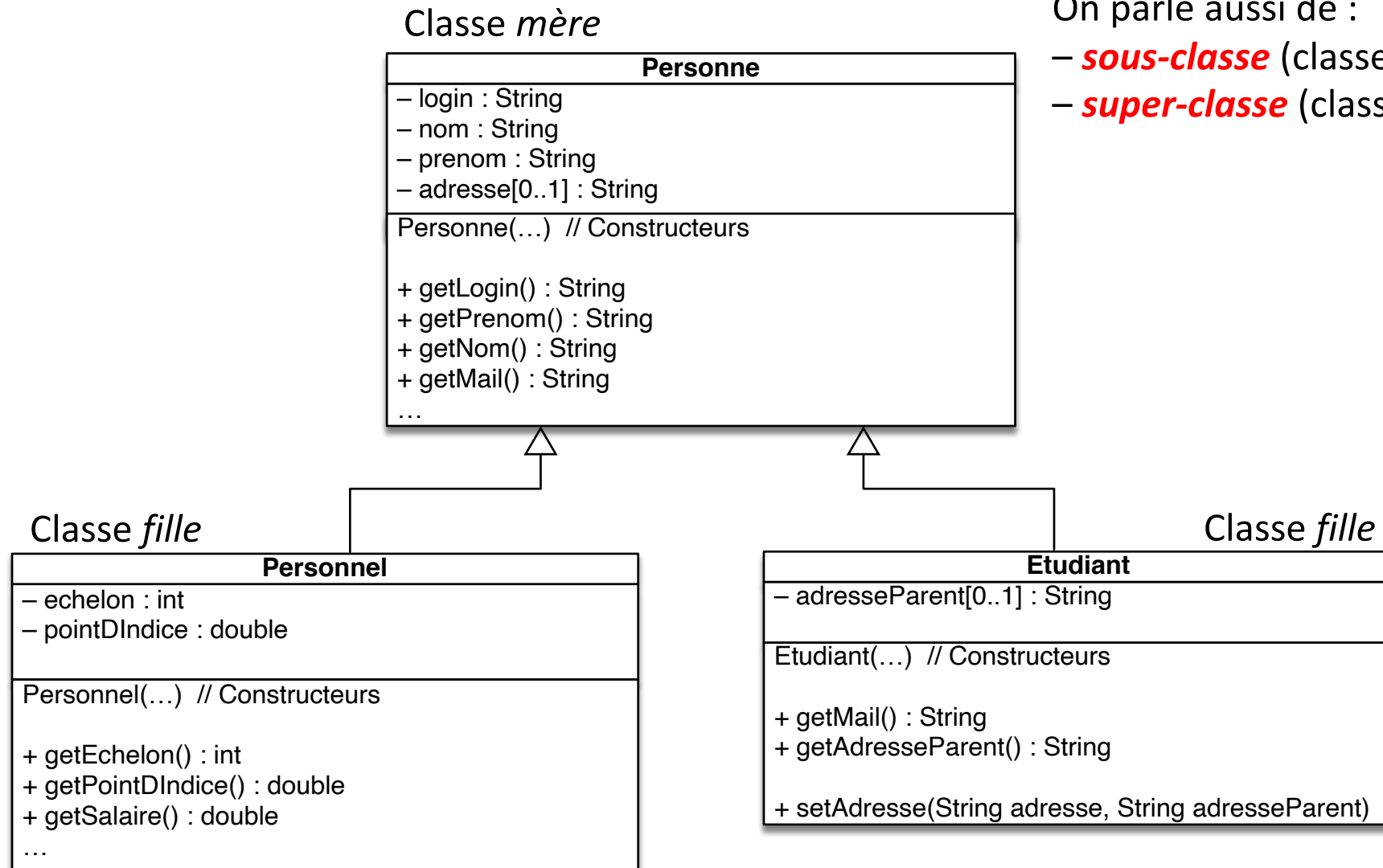
 En C++, héritage simple et multiple

 Une ou plusieurs classes *filles* héritent d'une ou plusieurs classes *mères*

Mécanisme d'héritage (2/3)


On parle aussi de :

- **sous-classe** (classe fille)
- **super-classe** (classe mère)



Note : pour l'Etudiant, on ne tient pas ici compte des notes

Mécanisme d'héritage (3/3)

 **Héritage** dans une classe *fil*le de tous les attributs et de toutes les méthodes de sa classe *m*ère

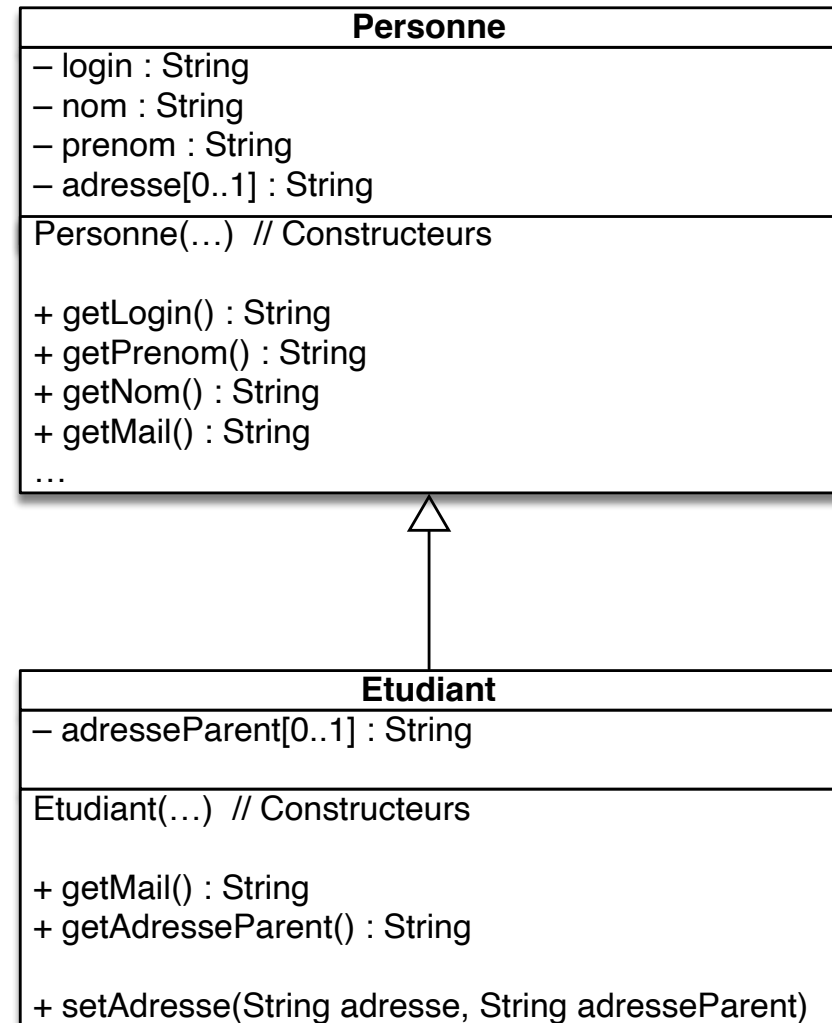
Etudiant hérite des attributs :

- login
- nom
- prenom
- adresse


Etudiant hérite des méthodes :


- getNomComplet()
- getMail()
- getLogin()

IMPORTANT : Penser toujours que la classe *fil*le (Etudiant) est une sorte de classe *m*ère (Personne)



Héritage et visibilité

 Tout **membre Public (+)** ou **Protected (#)** d'une classe *mère* est **HÉRITÉ** et **DIRECTEMENT ACCESSIBLE** dans sa classe *fille*

 Tout **membre Private (-)** d'une classe *mère* est **HÉRITÉ** mais **N'EST PAS DIRECTEMENT ACCESSIBLE** dans sa classe *fille*

 **Note** : le terme **membre** désigne ici un attribut ou une méthode.

Exemple d'héritage (1/4) : classe mère

```
public class Personne {  
    private String login;  
    private String nom;  
    private String prenom;  
    private String adresse;
```

```
    Personne(String login, String nom,  
              String prenom, String adresse) {  
        setLogin(login);  
        setNom(nom);  
        setPrenom(prenom);  
        setAdresse(adresse);
```

```
}
```

Constructeur

Exemple d'héritage (2/4) : classe fille

Le mot clé **extends** indique que **Etudiant** est fille de **Personne** (**Etudiant** hérite des membres de la classe mère **Personne**)

```
public class Etudiant extends Personne {  
    private String adresseParents;
```

```
    Etudiant(String login, String nom,  
              String prenom,  
              String adresseEtudiant,  
              String adresseParents) {  
        super(login, nom, prenom, adresseEtudiant);  
        setAdresseParents(adresseParents);  
    }  
}
```

Constructeur

...



le mot clé **super** dans la classe fille **Etudiant** réalise l'appel du constructeur de la classe mère **Personne**
IMPORTANT : il faut construire la **Personne** « contenue » dans l'instance de l'**Etudiant**

Exemple d'héritage (3/4) : membres hérités

Utilisation à l'intérieur de la classe fille

```
public class Etudiant extends Personne {  
    ...  
    public void affiche() {  
        System.out.print("Etudiant - Login : " + getLogin());  
        System.out.print(" - nom complet : " + getNomComplet());  
    }  
}
```

Note



-  **getLogin()** & **getNomComplet()** sont des méthodes publiques héritées qui se comportent comme des méthodes de la classe **Etudiant**
-  à l'intérieur de la classe **Etudiant** on les utilise donc comme toute méthode de la classe

Exemple d'héritage (4/4) : membres hérités

Utilisation à l'extérieur de la classe fille

```
Etudiant et1 = new Etudiant("blanchonp", "blanchon", "phil");  
System.out.println("Login : " + et1.getLogin());  
System.out.println("Nom complet : " + et1.getNomComplet());
```

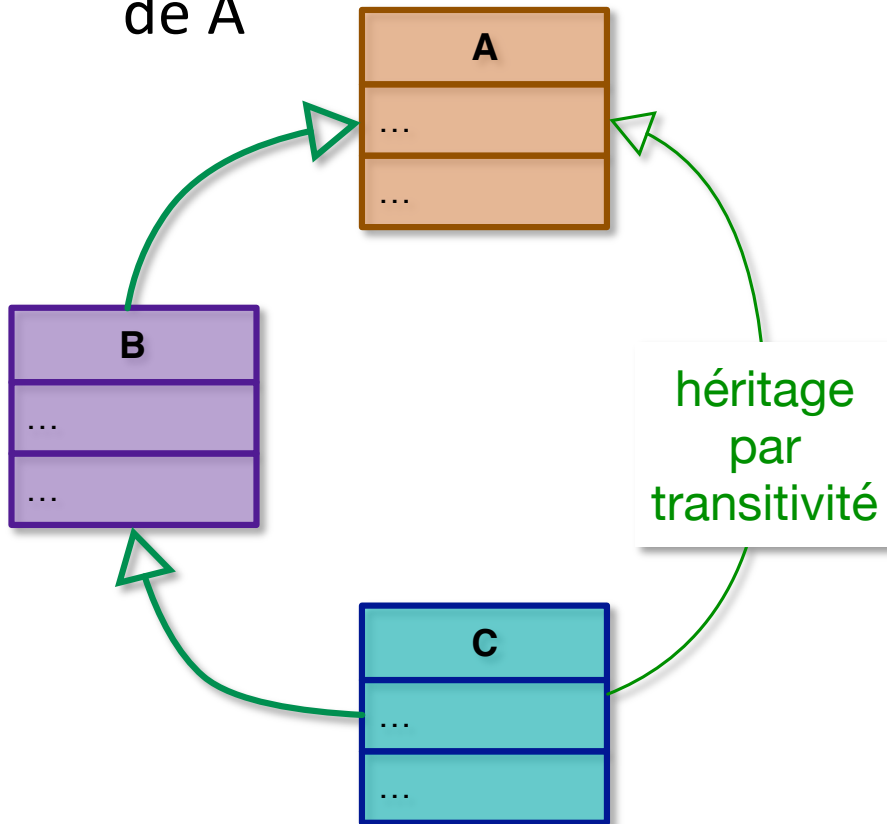
Note

-  **getLogin()** & **getNomComplet()** sont des méthodes publiques héritées qui s'utilisent comme des méthodes de la classe **Etudiant**
-  à l'extérieur de la classe **Etudiant** on les utilise donc comme toute méthode de la classe avec la notation pointée '**.**'

Propriétés de la relation d'héritage (1/2)

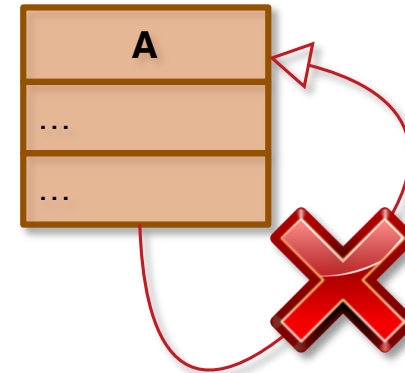
Transitive

■ si B hérite de A et si C hérite de B alors C hérite de A



Non réflexive

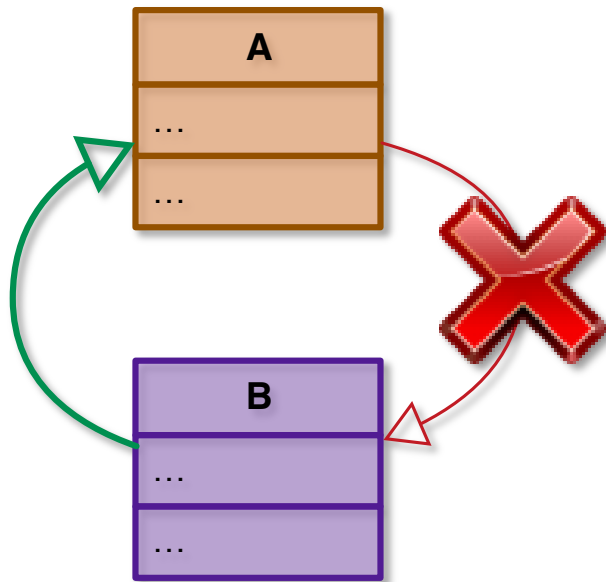
■ une classe ne peut hériter d'elle même



Propriétés de la relation d'héritage (2/2)

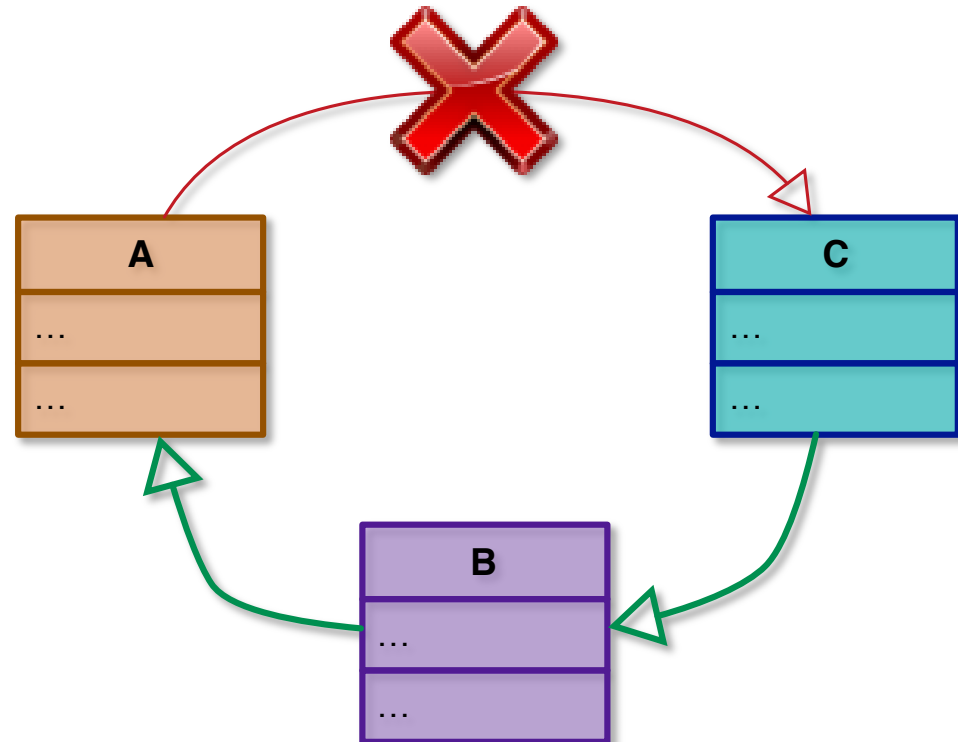
Non symétrique

■ si A hérite de B, B n'hérite pas de A



Sans cycle

■ Il n'est pas possible que B hérite de A, C hérite de B et que A hérite de C



Comment enrichir les classes filles ? (1/3)

Ajout d'attributs

 **adresseParent**

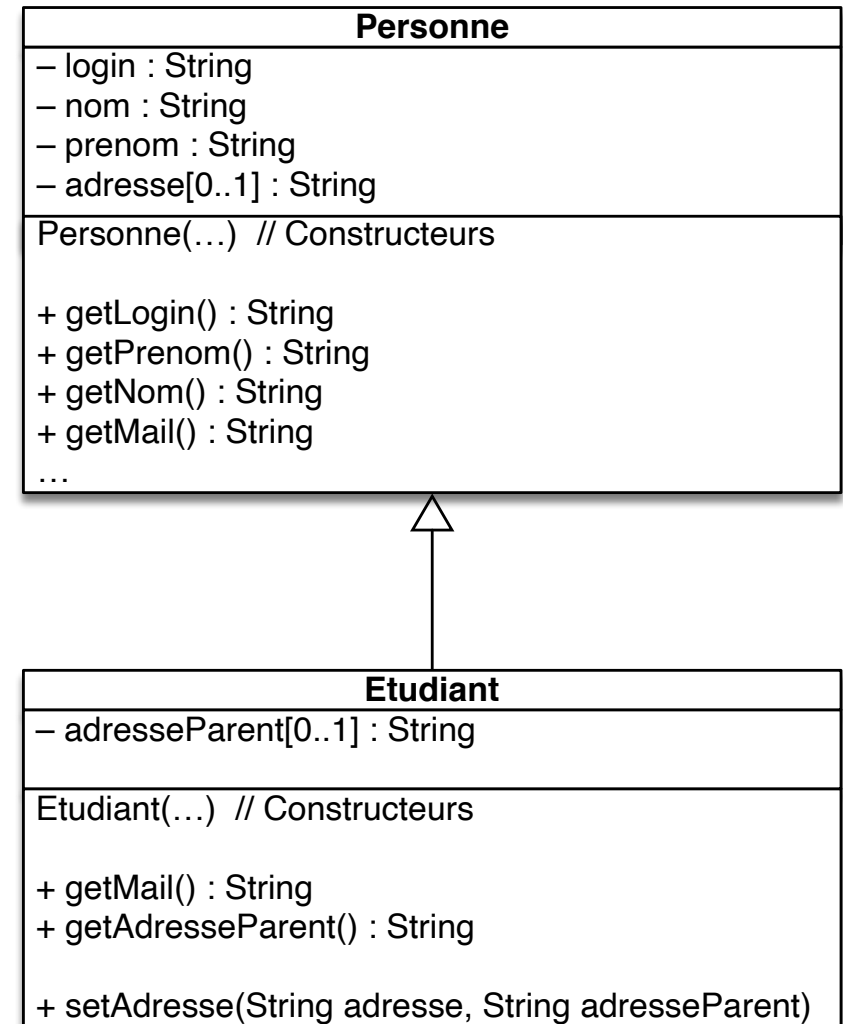
Ajout de méthodes

 Surcharge de méthodes

 *setAdresse(...)*


 Redéfinition de méthodes

 *getMail()*



Comment enrichir les classes filles ? (2/3)

La surcharge de méthode

 **méthode sémantiquement similaire** (nom identique & signature différente) **qui cohabite avec celle de la classe mère**

 *(signature : type et nombre des paramètres)*

 **setAdresse(String adresse)** surchargées dans Etudiant par

 **setAdresse(string adresse, String adresseParent)**

Comment enrichir les classes filles ? (2/3)

La **redéfinition** de méthode

 **méthode avec même nom et signature qui se substitue à celle de la classe mère**

 **getMail()** les Etudiants n'ont pas le même mail que les Personnes

 **Personne** : nom.prenom@univ-grenoble-alpes.fr

 **Etudiant** : nom.prenom@etu.univ-grenoble-alpes.fr

Exemples

```
public class Etudiant extends Personne {  
    private String adresseParent; // nouvel attribut
```

@Override

```
public String getMail() {  
    return getPrenom() + "." + getNom() + "@etu.univ-grenoble-alpes.fr";  
}
```

redéfinition

```
public void setAdresse(String adresse, String adresseParent) {  
    setAdresse(adresse); // méthode de Personne  
    this.adresseParent = adresseParent;  
}
```

surcharge

...

Modèles de surcharge & de redéfinition

```
public class MereClasse {  
    ...  
    public void maMethode (int i)  
    {...}  
}
```

surcharge

```
public class FilleClasse1 extends MereClasse {  
    ...  
    public void maMethode (double i) {...}  
}
```

redéfinition

```
public class FilleClasse2 extends MereClasse {  
    ...  
    @Override  
    public void maMethode (int i) {...}  
}
```



pour un objet de type
FilleClasse1



2 méthodes **maMethode**



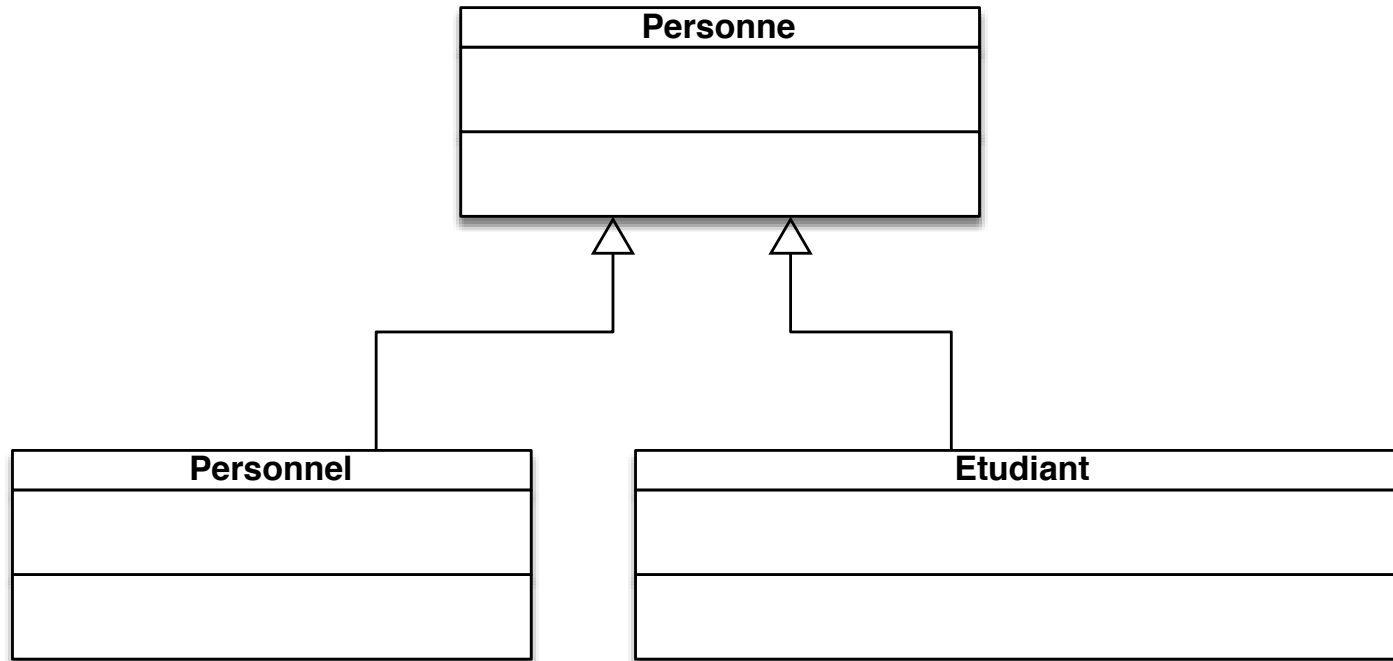
pour un objet de type
FilleClasse2



1 méthode **maMethode**




Classe abstraite (1/3)

🧩 Gérer le personnel et des étudiants



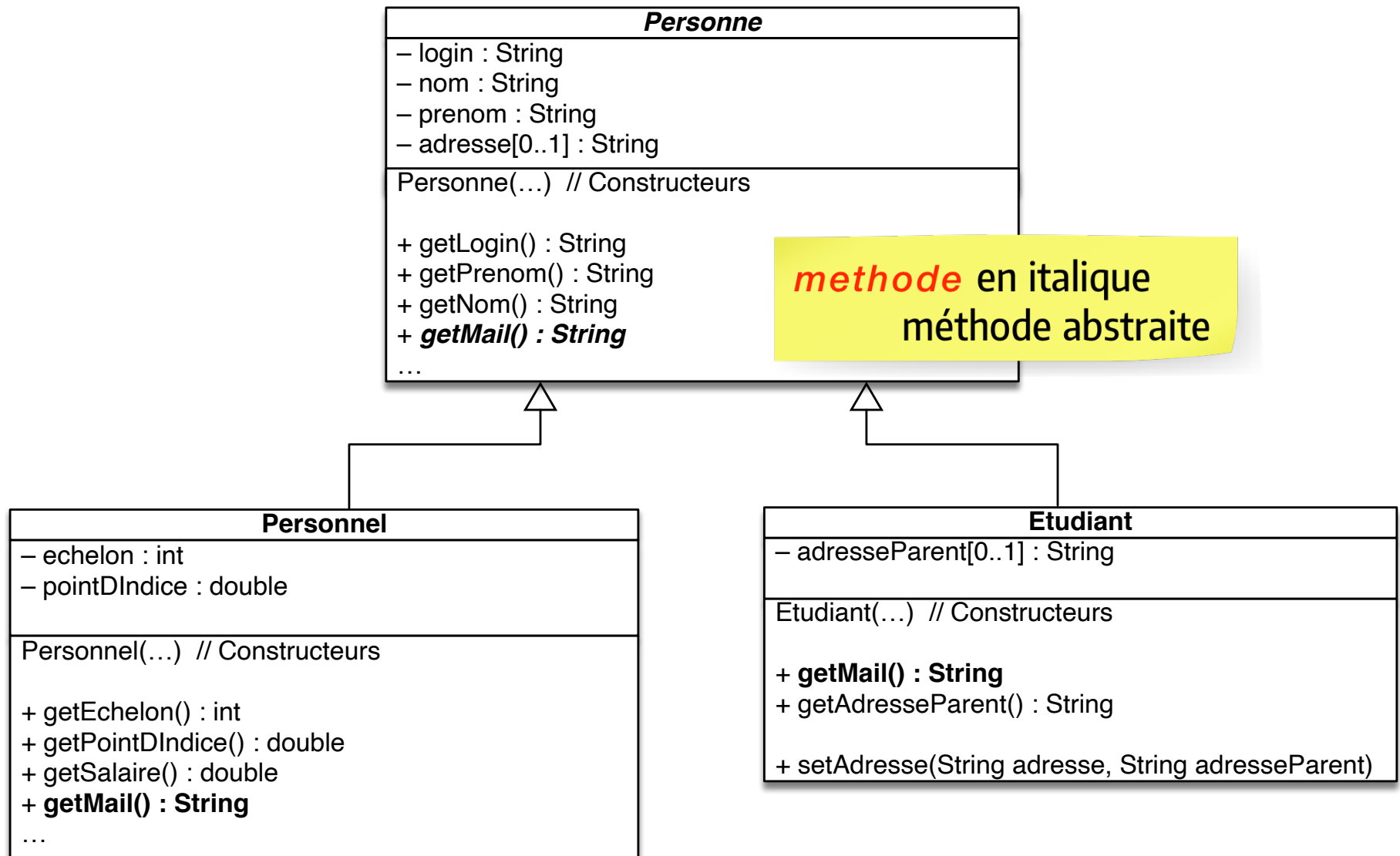
🧩 Personne sert à **factoriser** les membres des classes Personnel et Etudiant. Mais elle ne sera jamais *instanciée*.

Classe abstraite (2/3)

-  Permet de **définir des concepts incomplets** qui devront être implémentés dans les classes *filles* plus spécifiques.
-  Permet de **factoriser** les propriétés de ses classes *filles* en regroupant attributs et méthodes communes
-  Est **non-instanciable**, mais peut être utilisée comme un **type**

Classe abstraite (3/3)

Classe en italique
classe abstraite



methode en italique
méthode abstraite

Exemple d'une classe abstraite

```
public abstract class Personne {  
    private String login;  
    private String nom;  
    ...
```

le mot clé **abstract** indique si une classe ou une méthode est abstraite

```
    public Personne(String login, String nom, String prenom){  
        setLogin(login);  
        ...
```

```
    public abstract String getMail();  
        // devra être redéfinie dans les classes filles  
    ...
```

Important : une méthode abstraite doit obligatoirement être redéfinie dans une classe fille (sauf si cette classe est également abstraite)