IUT 2 Grenoble Université Pierre-Mendès-France Informatique

EF1

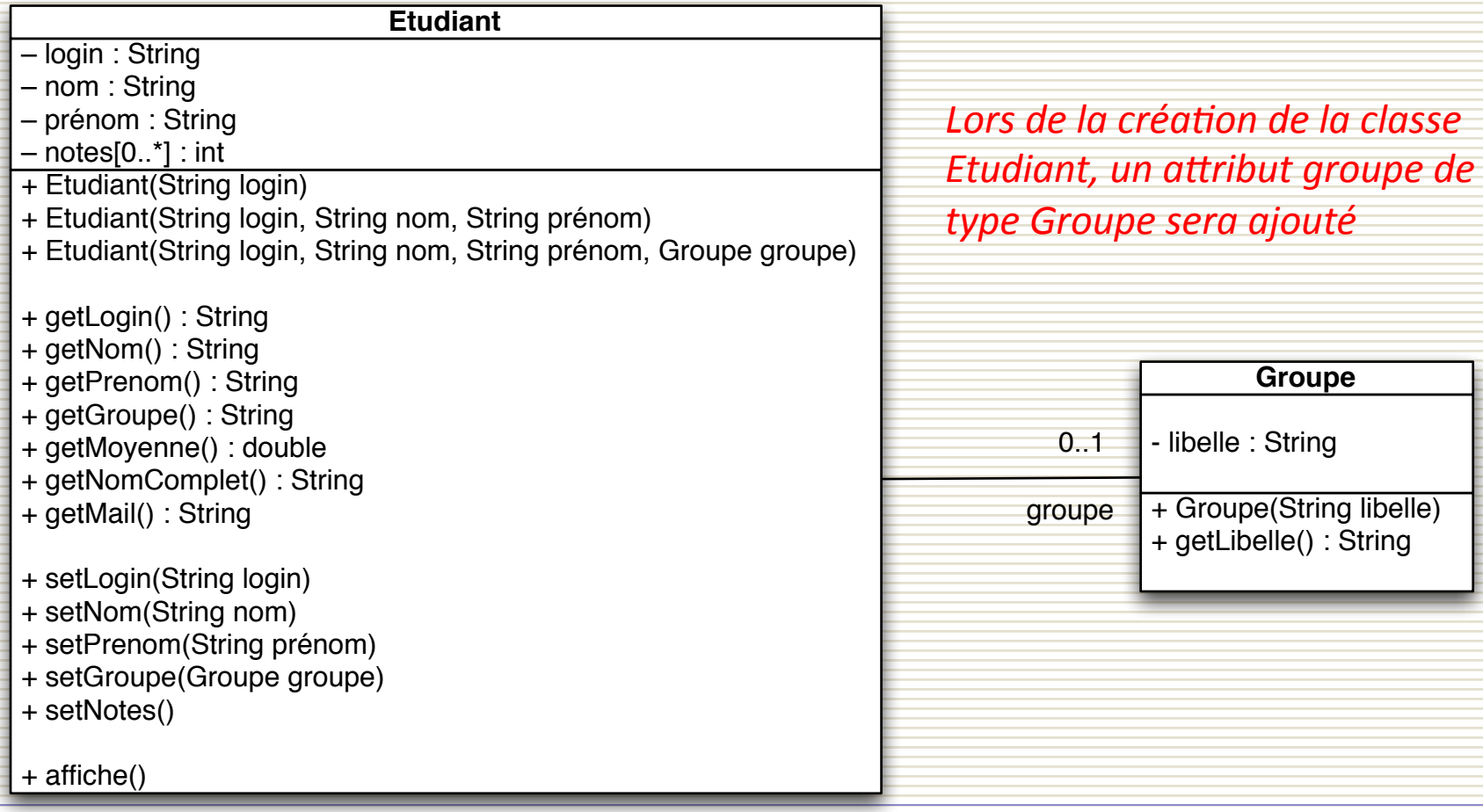
Bases de la Programmation Orientée Objet

- Association entre classes
- Collections en Java
- Notion d'héritage

ASSOCIATION ENTRE CLASSES

Association (1/4)

- ✓ Une classe « ***utilise*** » une autre classe
- ✓ Exemple : un étudiant appartient ou non à un groupe.



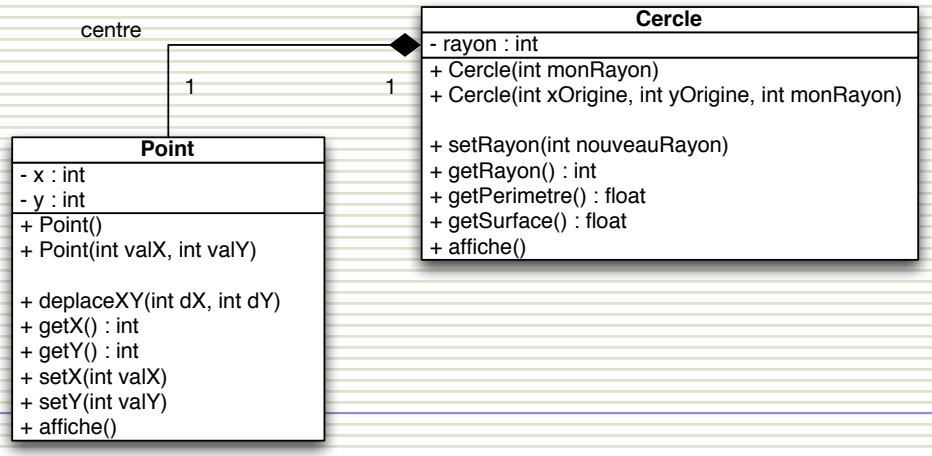
Lors de la création de la classe Etudiant, un attribut groupe de type Groupe sera ajouté

Association (2/4)

- ✓ Attention au **lien** entre les objets

  - Soit l'objet en attribut est crée à l'**extérieur** de la classe
    - Etudiant lié à un Groupe : un Groupe pour plusieurs Etudiants

  - Soit l'objet en attribut est crée à l'**intérieur** de la classe
    - Cercle lié à un Point : un Point par Cercle (TD1)
    - Appelée **composition**



Association : agrégation (3/4)

- Soit l'objet en attribut est crée à l'**extérieur** de la classe
  - Etudiant lié à un Groupe : un Groupe pour plusieurs Etudiants

Création

```
public Etudiant(String login, String nom, String prenom, Groupe groupe) {
        this(login, nom, prenom);      // appel du contructeur à 3 paramètres
        setGroupe(groupe);
 }
```

Utilisation

```
Groupe groupeA = new Groupe("A");
Etudiant et1 = new Etudiant("blanchonp", "blanchon", "philippe », groupeA);
```

Association : composition (4/4)

- Soit l'objet en attribut est crée à l'**intérieur** de la classe
  - Forme lié à un Point : un Point par Forme

Création

```
public Cercle(int x, int y) {
        centre = new Point(x,y);
}
```

Vocabulaire

- **Objet / classe**
  - Un objet est un élément d'une et une seule classe.
    - Un objet est une **instance** d'une classe.
  - Une classe décrit la structure et le comportement que partagent des objets de même nature
    - Une classe est la déclaration d'un type d'objet

- **Lien / association**
  - Un lien est **instance** d'une association.

COLLECTIONS EN JAVA

Collections (1/2)

- ✓ Permettent de stocker un **nombre variable d'éléments** de types identiques ou disparates.
- ✓ Se démarquent des tableaux dans la mesure où elles ont la **capacité de grandir et de diminuer** au gré, respectivement, des ajouts et des suppressions d'éléments.
- ✓ Dans le package *java.util*

Collections (2/2)

- ✓ Deux grands types de collection
  - **Collection<E>** : **collection d'éléments**
  - **Map<K,V>** : **collection de couples clé-valeur**.
    - à partir d'une clé **K**, on obtient la valeur **V** associée à cette clé.
    - Notes : les clés sont uniques, mais la même valeur peut-être associée à plusieurs clés.

*Collection<E> et Map<K,V> sont des **interfaces***

C'est quoi une interface ?

- ✓ C'est une collection de méthodes utilisées pour spécifier un **service** offert par une classe
- ✓ Elle est **totalement abstraite** et vouée à être implémentée par d'autres classes
- ✓ Elle est **non-instanciable**, mais peut être utilisée comme un **type**
- ➲ Une classe peut **implémenter une ou plusieurs** interfaces

L'interface Collection<E>

Quelques méthodes et description :

boolean add(E e)

Ensures that this collection contains the specified element **e** (optional operation).

boolean addAll(Collection<? extends E> c)

Adds all of the elements in the specified collection **c** to this collection (optional operation).

void clear()

Removes all of the elements from this collection (optional operation).

boolean contains(Object o)

Returns true if this collection contains the specified element **o**.

boolean containsAll(Collection<?> c)

Returns true if this collection contains all of the elements in the specified collection .

Quelques collections

- ✓ **Classes implémentant Collection<E> :**

ArrayList<E>	représente un **tableau dynamique** dont la taille peut varier. Implémente aussi Iterable<E>, List<E>
LinkedList<E>	Représente **une pile ou une file d'attente**. Implémente aussi Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

D'autres classes implémentant Collection<E> :

AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedDeque, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, ...

Quelques collections

- ✓ **Classes implémentant Map<K,V> :**

HashMap<K,V>	représente un tableau associatif dont une clé et des valeurs peuvent être nulles.
TreeMap<K,V>	représente un tableau associatif dont les clés sont classées en ordre croissant.

D'autres classes implémentant Map<K,V> :

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

Choix de la collection

- ✓ En fonction des besoins de l'application !
  - fréquences des lectures
  - fréquences des ajouts ou suppression (au début, à la fin ou au milieu de la collection)
  - nécessité ou non de recherche indexée…
- ✓ Tout dépend des cas d'utilisation !

- ✓ Exemple Billetterie
  - Quel objet utiliser pour enregistrer des trajets et les rechercher ensuite par ville de départ ?
- ✓ Exemple Gestion des étudiants :
  - Quel objet utiliser pour enregistrer des étudiants ?

Exemple : la classe ArrayList (1/3)

- ✓ La classe **ArrayList** implémente un tableau d'objets qui peut grandir ou rétrécir à la demande : tableau « dynamique »

Documentation

http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

Exemple : la classe ArrayList (2/3)

- ✓ Création d'un tableau d'objets de type String

```
ArrayList<String> chaines= new ArrayList<String>();
ArrayList<String> chaines= new ArrayList<>(); (depuis java 1.7)
```

- ✓ Quelques méthodes :
  - Ajouter un objet `chaines.add("test1")`
  - Donner la taille du tableau `chaines.size()`
  - Donner l'objet à l'indice i `chaines.get(i)`
  - Supprimer l'objet à l'indice i `chaines.remove(i)`

Exemple : la classe ArrayList (3/3)

- ✓ Création d'un tableau d'objets de type Integer

```
ArrayList<Integer> entiers= new ArrayList<>();
```

- ✓ Ajout d'un objet

```
Integer entier = new Integer(10);
entiers.add(entier);
entiers.add(12); // facilité d'écriture
```

- ✓ Itérations

```
for(int i = 0; i < entiers.size(); i++) {
        …
}

for(Integer tmp : entiers) { // simplification d'écriture
        …
}
```

L'HÉRITAGE

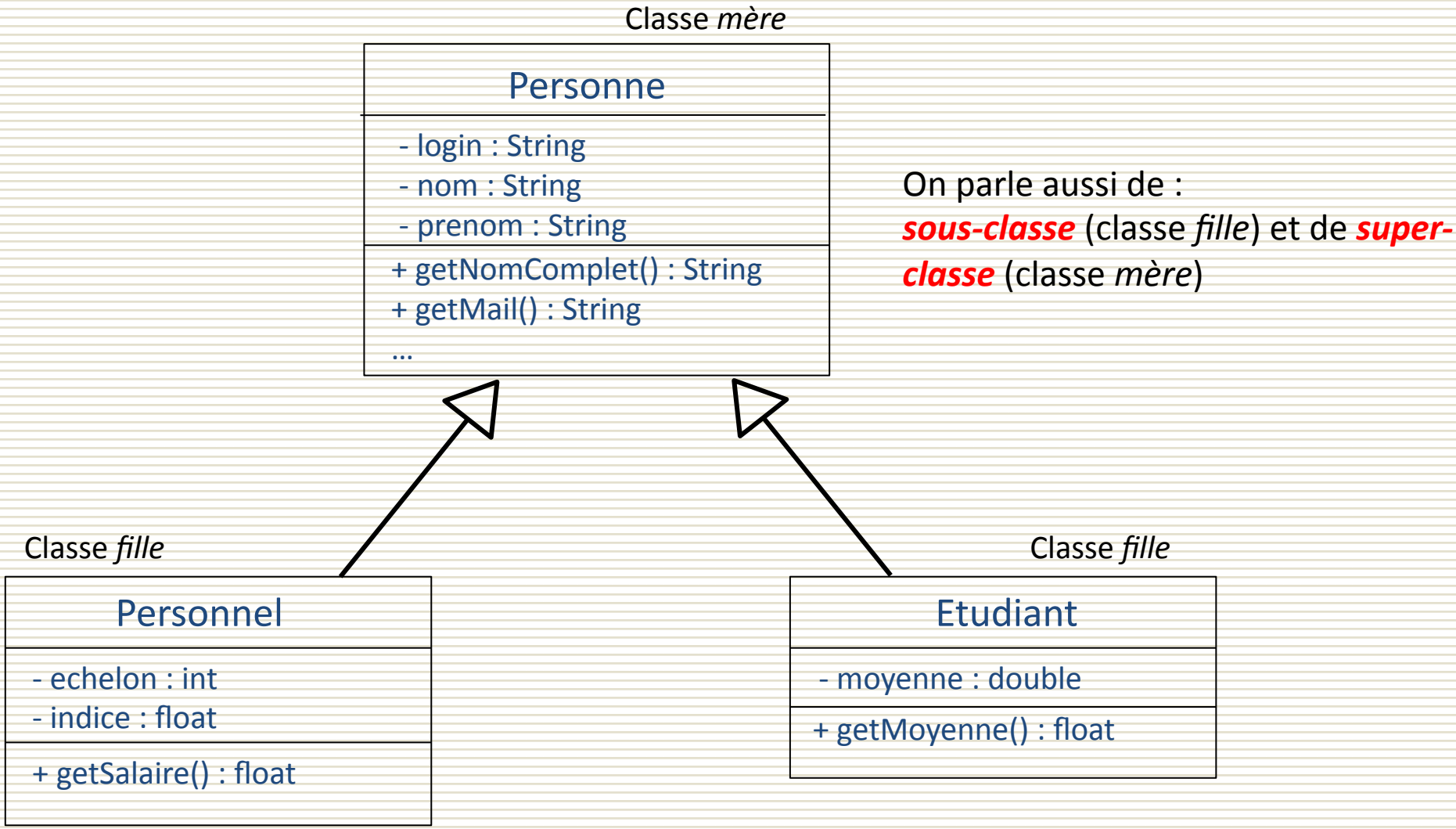
Mécanisme d'héritage (1/3)

- ✓ 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)



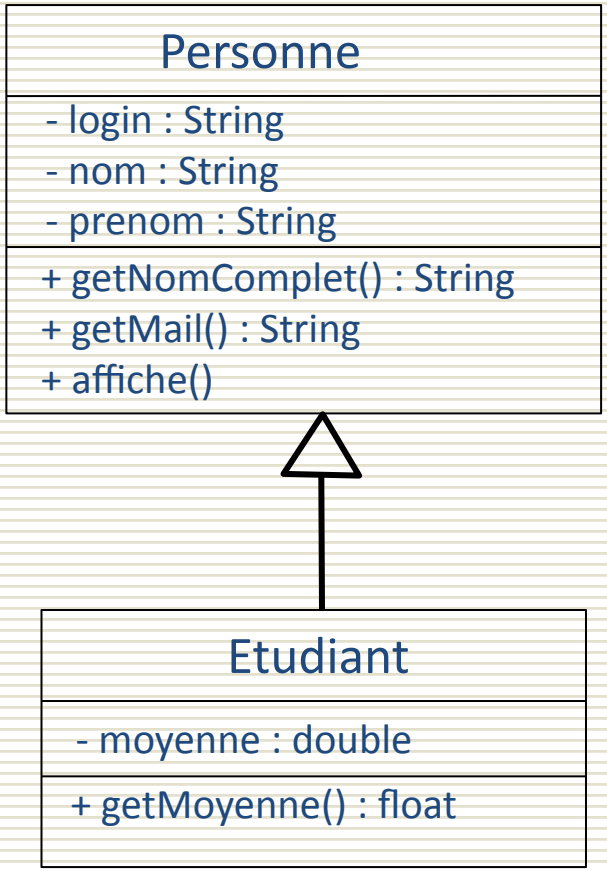
On parle aussi de :
sous-classe (classe *fille*) et de ***super-classe*** (classe *mère*)

Mécanisme d'héritage (3/3)

- ✓ **Héritage** dans une classe *fille* de tous les attributs et de toutes les méthodes de sa classe *mère*

Etudiant hérite des méthodes:

- getNomComplet()
- getMail()
- affiche()



IMPORTANT : Penser toujours que la classe *fille* (Etudiant) est une sorte de classe mère (*Personne*)

Héritage et visibilité

- ✓ Tout membre (attribut ou méthode) **public** ou **protégé** est hérité dans sa classe *fille*

- ✓ Tout membre **privé** d'une classe *mère* n'est pas accessible dans sa classe *fille*

Exemple d'héritage (1/3) : classe mère

```
public class Personne {
    private String login;
    private String nom;
    private String prenom;
    public Personne(String login, String nom, String prenom) {
        setLogin(login);
        setNom(nom);
        setPrenom(prenom);
    }
    public String getLogin() {
        return login;
    }
...
}
```

Exemple d'héritage (2/3) : classe fille

```
public class Etudiant extends Personne {

    private ArrayList<EntierContraint> notes;
    private Groupe groupe;
    public Etudiant(String login, String nom,
                    String prenom, Groupe groupe) {
        super(login, nom, prenom);
        setGroupe(groupe);
        … ;
    }
...
```

Le mot clé **extends** permet l'héritage des membres **public** ou **protected** de la classe *mère* Personne

Le mot clé **super** permet l'appelle du constructeur de la classe *mère* Personne

IMPORTANT : il faut construire la partie personne contenu dans l'étudiant

Exemple d'héritage (3/3)

- ✓ Utilisation dans la classe

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

- ✓ Utilisation en dehors de la classe

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

- ✓ Les membres hérités s'utilisent comme les membres normaux !

Propriétés de la relation d'héritage

- ✓ **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 ;
- ✓ **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

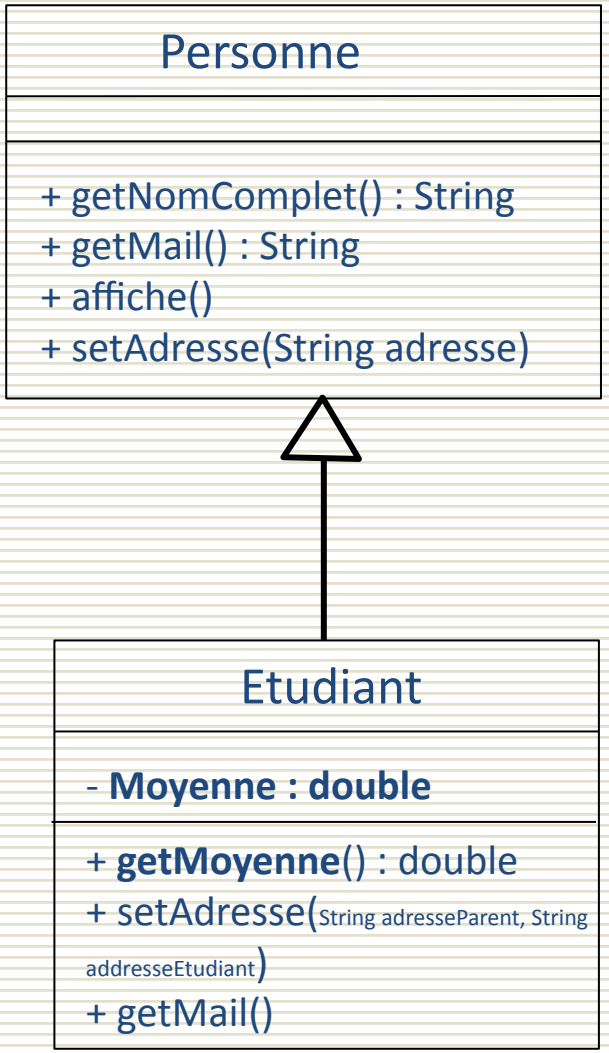
Visibilité : petit résumé

Accès depuis … sur un membre de la classe A	private	rien	protected	public
la même classe A	OUI	OUI	OUI	OUI
une classe *fille* de A du même package	NON	OUI	OUI	OUI
une classe B du même package	NON	OUI	OUI	OUI
une classe *fille* de A d'un package différent	NON	NON	OUI	OUI
une classe C d'un package différent	NON	NON	NON	OUI

Membre = attribut ou méthode

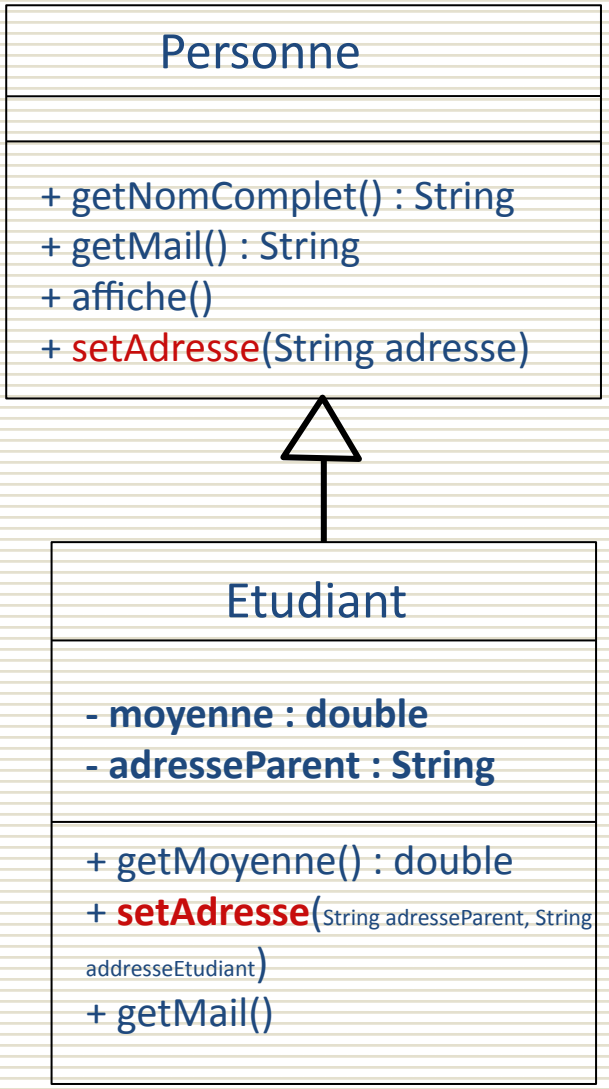
Comment enrichir les classes *filles* ? (1/3)

- ✓ Ajouter de nouveaux attributs
  - `moyenne`
- ✓ Ajouter de nouvelles méthodes
  - `getMoyenne()`



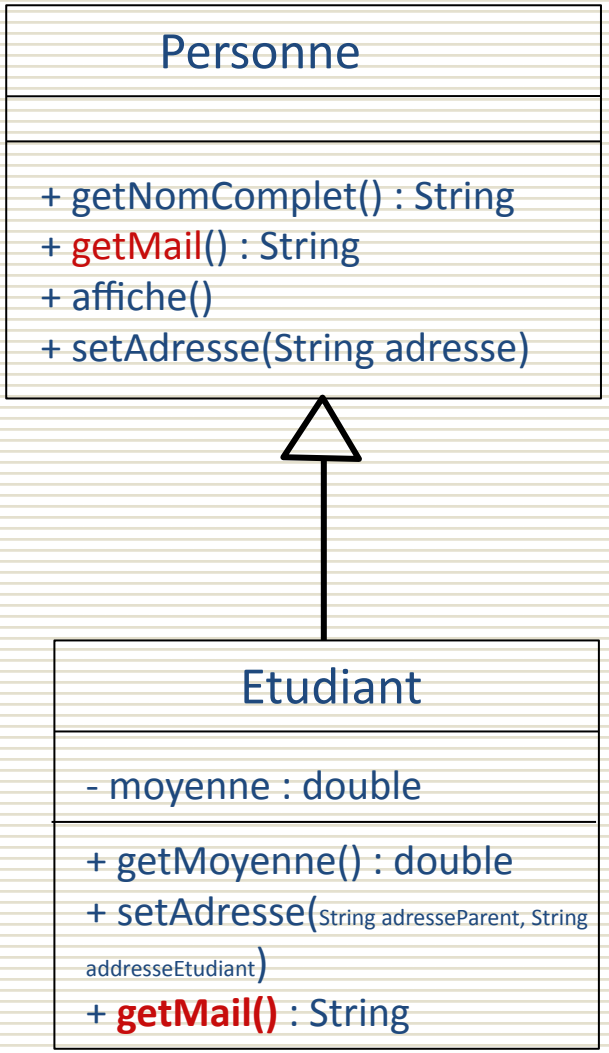
Comment enrichir les classes *filles* ? (2/3)

- ✓ Ajouter de nouvelles méthodes
  - **la surcharge**
    - méthodes sémantiquement similaires avec le même nom mais des prototypes ≠ *(type et nb des paramètres)*
      - **`setAdresse(String adresseParent, String adresseEtudiant)`**
    - Exemple d'adresse:
      - 1 place Doyen Gosse 38000 Grenoble



Comment enrichir les classes *filles* ? (3/3)

- ✓ Redéfinir une méthode
  - **la redéfinition**
    - Substitution d'une méthode héritée de la classe mère par une nouvelle méthode
    - **`getMail()`**
      - les étudiants n'ont pas les mêmes email que les personnes
        - personne@iut2.upmf-grenoble.fr
        - etudiant@etu.upmf-grenoble.fr
  - La signature (nom, paramètre, résultat) de la méthode redéfinie doit être identique.



Exemples

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

 @Override
 public String getMail() {
    return getPrenom() + "." + getNom() + "@etu.iut2.upmf-grenoble.fr";
 }

 public void setAdresse(String adresseParent, String adresseEtudiant) {
    setAdresse(adresseEtudiant);
    this.adresseParent = adresseParent;
 }

...
```

Redéfinition

Surcharge

Modèle surcharge et redéfinition

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

Surcharge

Redéfinition

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

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

Pour un objet de type **FilleClasse1**:

- 2 méthodes maMethode

Pour un objet de type **FilleClasse2**:

- Une seule méthode maMethode

Polymorphisme d'héritage (1/2)

- ✓ **Manipuler des objets de types différents mais qui ont une *base* commune.**

- ✓ Dans notre exemple:
  - Manipuler un tableau de personnes comprenant des étudiants et du personnel indistinctement et leur demander à tous leur email.

Polymorphisme d'héritage (2/2)

✓ Comment le polymorphisme fonctionne ?

- Un objet peut être manipulé comme s'il appartenait à une autre classe dont il hérite ➲ **surclassement**

ET

- Une méthode peut se comporter différemment sur différentes classes de la hiérarchie ➲ **redéfinition**

ET

- Le type d'un objet peut être retrouvé à l'exécution et ainsi la méthode appropriée peut être effectuée ➲ **lien dynamique**

Exemple de polymorphisme (1/2)

```
// Tableau d'objets de type Personne
ArrayList<Personne> personnes = new ArrayList<>();
// Les étudiants
Etudiant et1 = new Etudiant("blanchonp", "blanchon", "philippe", groupeA);
Etudiant et2 = new Etudiant("martinf", "martin", "francis", groupeA);
personnes.add(et1);
personnes.add(et2);
// Le personnel
Personnel per1 = new Personnel("goulianh", "goulian", "herve");
Personnel per2 = new Personnel("brunetj", "brunet", "jerome");
personnes.add(per1);
personnes.add(per2);
// afficher les email
for(Personne personne : personnes) {
            System.out.println(personne.getMail());
    }
```

Surclassement

Redéfinition + lien dynamique

Exemple de polymorphisme (2/2)

- ✓ Résultat :

```
philippe.blanchon@etu.iut2.upmf-grenoble.fr
francis.martin@etu.iut2.upmf-grenoble.fr
herve.goulian@iut2.upmf-grenoble.fr
jerome.brunet@iut2.upmf-grenoble.fr
```

- ✓ Lors de l'exécution, les objets de Type Etudiant, même s'ils ont subi un surclassement dans un tableau d'objet de type Personne, ont utilisé leur méthode redéfinie.