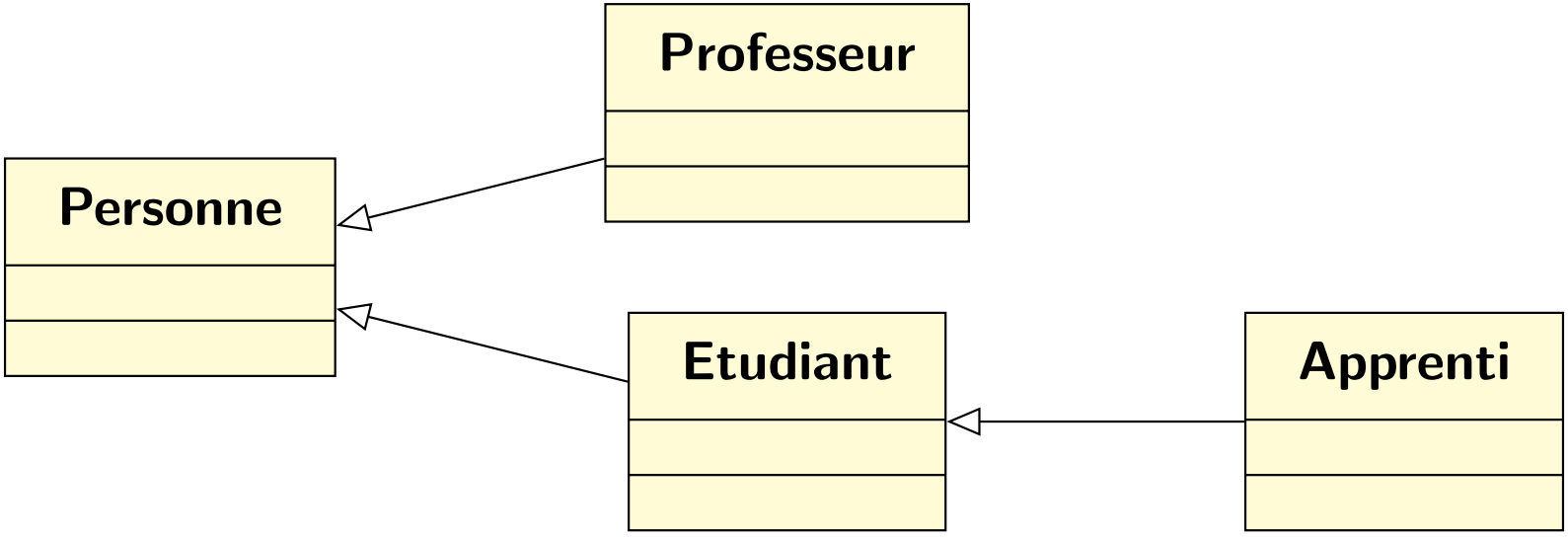
Héritage (II)

Hiérarchie de classes

La relation d'héritage établit une hiérarchie entre les classes :



La classe `Apprenti` hérite aussi de la classe `Personne` : la classe `Apprenti` est une sous-classe de la classe `Personne` .

> **Attention :** La classe `Personne` n'est pas la super-classe de la classe `Apprenti`

Hiérarchie de classes

En résumé, dire qu'une classe B est une sous-classe de A c'est dire que la classe B hérite de la classe A.

En revanche, cela ne dit pas que la super-classe de la classe B est la classe A !

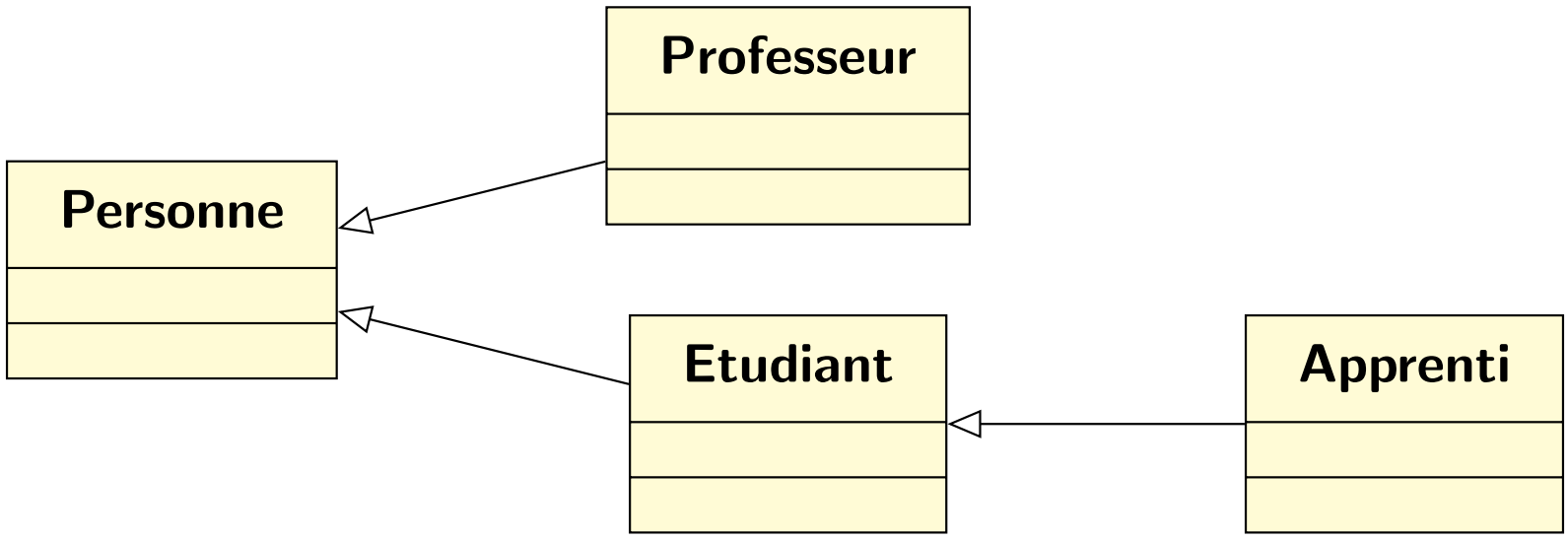
Exemple

- ☞ Les classes `Professeur`, `Etudiant` et `Apprenti` sont des sous-classes de la classe `Personne`.
- ☞ La super-classe de la classe `Apprenti` est la classe `Etudiant`.
- ☞ La super-classe des classes `Etudiant` et `Professeur` est la classe `Personne`.

Hiérarchie de classes

On utilisera parfois une formulation plus concise pour indiquer que la classe `Apprenti` est une sous-classe de la classe `Etudiant` :

« Un `Apprenti` est un `Etudiant` »



- Un `Professeur` est une `Personne`
- Un `Etudiant` est une `Personne`
- Un `Apprenti` est une `Personne`

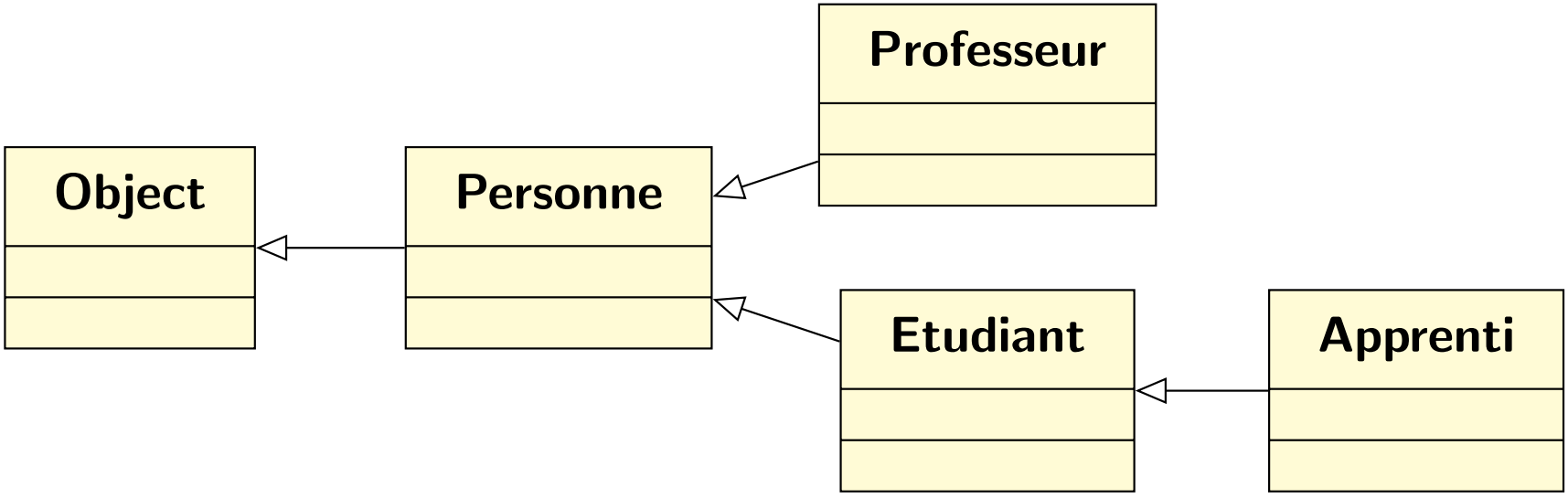
Hiérarchie de classes

En résumé,

- Une sous-classe de A est une classe qui hérite de A
- La super-classe d'une classe est la classe dont elle hérite **directement**.
- Un A est un B : la classe A est une sous-classe de la classe B

Hiérarchie de classes

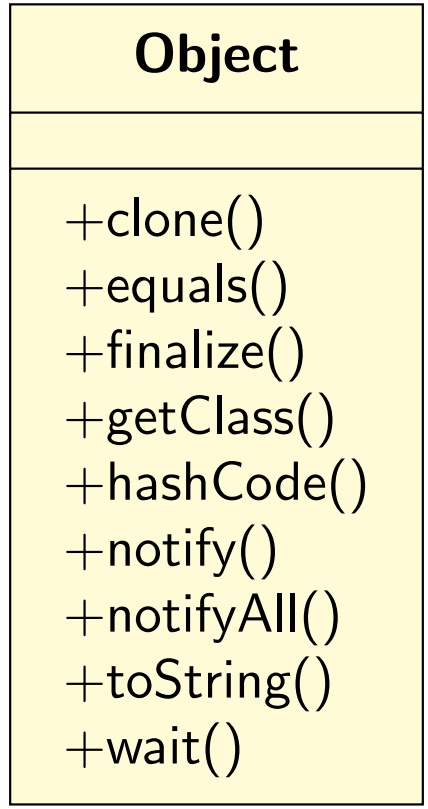
Le langage Java est un langage objet pur. Mais il possède une autre caractéritique très importante : toutes les classes en Java sont des sous-classes d'une classe particulière : la classe `Object`.



Par conséquent, toutes les classes possèdent donc toutes les méthodes de la classe `Object`.

Hiérarchie de classes

La classe `Object` ne possède que le constructeur sans argument `Object` et les méthodes :

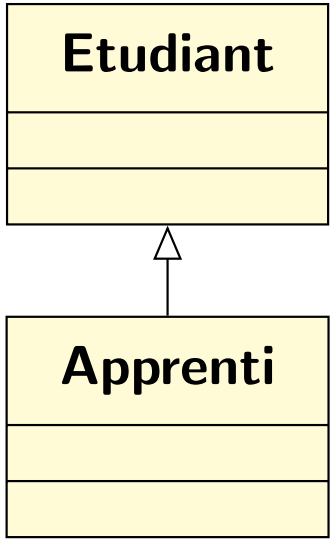


Nous n'en dirons pas plus sur cette classe pour l'instant.

Hiérarchie de classes

La réutilisation du code est un aspect important de l'héritage mais pas le seul !

Le deuxième point **fondamental** est la relation qui relie une classe à sa superclasse :



Un objet `Apprenti` possède tous les attributs et toutes les méthodes de la classe `Etudiant` .

Les objets `Apprenti` peuvent donc aussi être vus comme des « objets `Etudiant` ».

Cette vision des objets d'une sous-classe d'une classe est supportée par le langage Java grâce au **surclassement**.

Surclassement

Le **surclassement** désigne la possibilité de **manipuler** un objet `Apprenti` comme si c'était un objet `Etudiant`.
Concrètement, cela se traduit en la possibilité d'enregistrer dans une référence de type `Etudiant` l'identité d'un objet `Apprentui` :

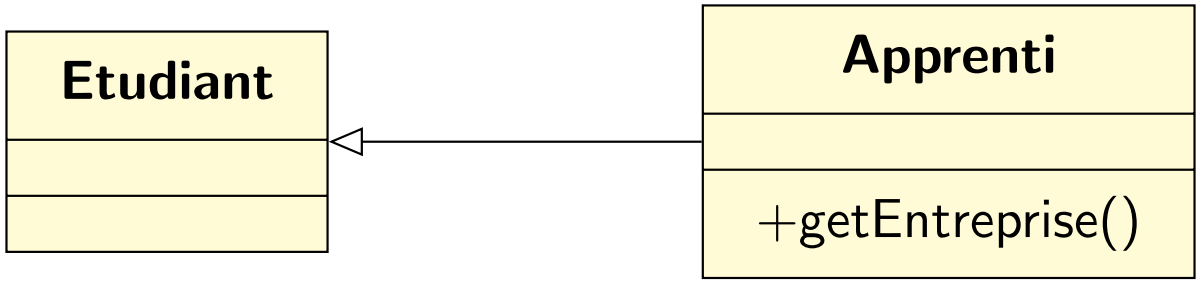
```
Etudiant p = new Apprenti();
```

Dans cette écriture, il est très important de comprendre que :

- La **référence** `p` est de type `Etudiant`.
- L'**objet** `p` est de type `Apprenti`.

Surclassement

> **Attention :** Lorsqu'un objet est "surclassé" il est vu comme un objet du type de la référence utilisée pour le désigner. **A travers la référence `p`, on ne peut appeler que les méthodes de la classe `Etudiant`.**




```
Etudiant p = new Entreprise();

p.getEntreprise(); // INTERDIT
```

La méthode `getEntreprise()` n'est pas une méthode de la classe `Etudiant` !

Surclassement

En conclusion, il est très important d'avoir à l'esprit qu'une référence ne contient pas forcément l'identité d'un objet du même type que la référence.

Le langage java fournit un opérateur **instanceof** qui permet de savoir si le type de l'objet `p` est une sous-classe d'un classe donnée :

```
Etudiant p = Apprenti();

System.out.println(p instanceof Apprenti); // affiche true
System.out.println(p instanceof Etudiant); // affiche true
```

> **Attention :** L'instruction **instanceof** ne permet donc pas de savoir si le **type réel** de l'objet `p` est `Apprenti`.

Surclassement et tableaux

Un autre intérêt du surclassement est qu'il permet d'enregistrer dans un tableau les identité d'objets de différentes classes pourvu qu'elles appartiennent à la même hiérarchie de classes :

```
Personne[] t = new Personne[3];

t[0] = new Personne("James␣Gosling");
t[1] = new Professeur("Patrick␣Naughton","Java");
t[2] = new Etudiant("Ada␣Lovelace","L3␣MIAGE");
```

On ne pourra enregistrer dans le tableau `t` que des objets `Personne` ou d'une de ses sous-classes.

Redéfinir une méthode héritée

Une autre possibilité très importante en programmation orientée objet est la possibilité de pouvoir réécrire le code d'une méthode héritée :

```
class Personne {
  public void bonjour() {
    System.out.println("Bonjour, je m'appelle " +
  nom;);
  }

class Professeur extends Personne {
  public void bonjour() {
    System.out.println("Bonjour, je suis
  professeur et mon nom est " + getNom(););
  }
```

On dit que la classe `Professeur` **redéfinit** le comportement de la la méthode héritée `bonjour()` .

Rédefinir une méthode héritée

Le comportement de la méthode `bonjour()` est alors différent selon que l'objet soit une `Personne` ou un `Professeur` !

```
Personne p = new Personne("James Gosling");
```

L'appel `p.bonjour()` affiche dans la console « `Bonjour, je m'appelle James Gosling` »

```
Professeur p = new Professeur("Patrick Naughton");
```

L'appel `p.bonjour()` affiche dans la console « `Bonjour, je suis professeur et mon nom est Patrick Naughton` »

Rédéfinir une méthode héritée

Pourquoi redéfinir une méthode héritée ?

Il arrive dans certains cas que le comportement d'une méthode héritée ne soit plus adaptée à la classe que l'on est en train de définir. On est donc dans ce cas amené à redéfinir la méthode pour l'adapter à la classe.

Donc, quand on définit une classe héritant d'une autre : pour chaque méthode héritée, il faut choisir entre

- ☞ ne rien faire, c'est-à-dire considérer que le comportement de la méthode héritée convient,
- ☞ et la redéfinir, c'est-à-dire modifier son comportement pour l'adapter à la classe.

Rédéfinir une méthode héritée

Lorsqu'on redéfinit une méthode, une bonne pratique est d'ajouter l'annotation `@Override` à la définition de la méthode (même si cela ne soit pas obligatoire) :

```
class Professeur extends Personne {
   @Override
   public void bonjour() {
     System.out.println("Bonjour, je suis
   professeur et mon nom est " + getNom());
   }
```

Pourquoi ?

- ☞ Vous indiquez au compilateur que vous redéfinissez une méthode : le compilateur vérifiera la signature de la méthode et indiquera une erreur si vous vous trompez dans la déclaration de la méthode.
- ☞ Cela améliore la lisibilité de votre code en indiquant que la méthode est une redéfinition une méthode héritée.

Rédéfinition d'une méthode et surclassement : Lien dynamique

Si on "surclasse" un objet, que se passe-t-il lors de l'appel d'une méthode redéfinie ?

```
Personne p = new Professeur("Patrick Naughton");
```

- ☞ L'appel `p.bonjour()` est **autorisé** car la méthode `bonjour()` est une méthode de la classe `Personne`.
- ☞ L'exécution de l'appel `p.bonjour()` affichera dans la console « `Bonjour, je suis professeur et mon non est Patrick Naughton` », c'est-à-dire, c'est la méthode `bonjour()` de la classe `Professeur` qui est appelée et non la méthode `bonjour()` de la classe `Personne`.

Rédéfinition d'une méthode et surclassement : Liaison dynamique

Pourquoi ?
Dans le cas d'une référence surclassée,

- A la compilation, le compilateur vérifie que la classe de la référence possède la méthode appelée.
- A l'exécution, la méthode appelé sera la méthode de l'objet. On parle de **liaison dynamique** (en anglais, dynamic binding).

Rédéfinition d'une méthode : réutiliser le code hérité

Il est possible de réutiliser le code hérité dans la redéfinition d'une méthode à l'aide du mot-clef **super** :

```
class Personne {
  private String nom;

  public void bonjour() {
    System.out.println("Bonjour, je m'appelle " +
  nom;);
  }

  class Professeur extends Personne {
    private String cours;
  @Override
  public void bonjour() {
    super.bonjour(); // appel de la méthode
   bonjour() de la classe Personne
    System.out.println("et j'enseigne" + cours;);
  }
```

Redéfinition d'une méthode : réutiliser le code hérité

```
Professeur p = new Professeur("Patrick␣Naughton","
    Java");

p.bonjour();
```

Ce programme affichera dans la console :

```
Bonjour, je m'appelle Patrick Naughton
et j'enseigne Java
```

Polymorphisme

Le terme « polymorphisme » n'a pas de définition partagée à ce jour. Mais si vous vous demandez ce que recouvre ce terme, voici des éléments de réponse :

- il recouvre le fait qu'un objet puisse être manipulé comme s'il appartenait à une autre classe (surclassement)
- ainsi que le fait que la même opération puisse avoir des comportements différents pour différentes classes d'une hiérarchie de classes (redéfinition de méthodes).

Sous-classement

Dans le cas d'un objet « surclassée », est-t-il possible d'enregistrer son identité dans une référence de type de l'objet ?

Non cela n'est pas possible **directement** :

```
Personne p = new Professeur("Patrick␣Naughton","
    Java");
Professeur r = p; // INTERDIT
```

On ne peut affecter à une référence que la valeur d'une référence du **même type** !

Sous-classement

Le sous-classement (ou transtypage) permet de convertir une référence en une référence d'un autre type :

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

> **Attention :** La conversion de la référence `p` de type `Personne` en une référence de type `Professeur` est possible car l'objet référencé par `p` est un objet de `Professeur`. Autrement, la conversion provoquera une erreur à la compilation.