Langage de programmation orientée objet - Série 4

Objectifs

- Manipuler les notions élémentaires : classe et objet, constructeur et méthode, surcharge de constructeur et de méthode, méthode toString, Information Hiding
- Etablir des liens entre objets par l'intermédiaire de variables d'instance
- Manipuler les notions liées à **l'héritage** et comprendre le **polymorphisme**
- Par convention, toutes les variables d'instance doivent être privées

Introduction à l'héritage

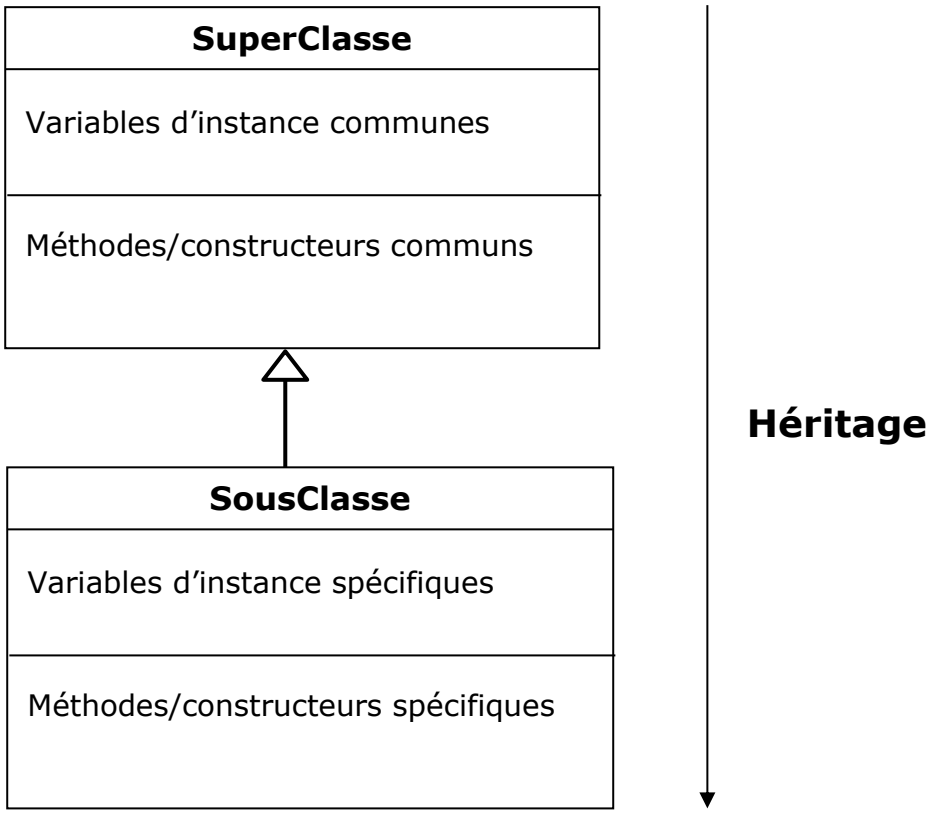
Principe de généralisation

Si plusieurs classes partagent des fonctionnalités similaires (variables d'instance ou méthodes), on crée une classe qui encapsule ces fonctionnalités communes (classe mère ou super-classe). On construit ensuite de nouvelles classes (classes enfants ou sous-classes) à partir de cette classe mère en ajoutant des fonctionnalités spécifiques.

Héritage

Les caractéristiques (variables d'instance ou méthodes) de la super-classe sont héritées dans les sous-classes.
Tout objet de la sous-classe aura donc en mémoire une valeur pour chaque variable d'instance de la super-classe en plus des valeurs des variables d'instance propres à la sous-classe.
De même, toute méthode de la super-classe pourra être appelée sur n'importe quel objet de la sous-classe.

Représentation de l'héritage dans un diagramme de classes UML



Découvrez l'héritage :

Lisez les sections 7.1, 7.2 et 7.3 du syllabus.

Synthèse

① Pour réutiliser une classe existante et l'adapter, il suffit de créer une nouvelle classe en la déclarant sous-classe de la classe réutilisée. La classe réutilisée jouera le rôle de super-classe. La déclaration de la sous-classe est :

class SousClasse ***extends*** *SuperClasse*

② Une sous-classe hérite des variables d'instance de la super-classe. Toute occurrence de la sous-classe possède en mémoire une copie des variables d'instance héritées.

③ Une sous-classe hérite des méthodes de la super-classe. Toute méthode de la super-classe peut être appelée sur toute occurrence (objet) de la sous-classe.

④ La sous-classe peut contenir ses propres variables d'instance.

⑤ La sous-classe peut contenir ses propres méthodes.

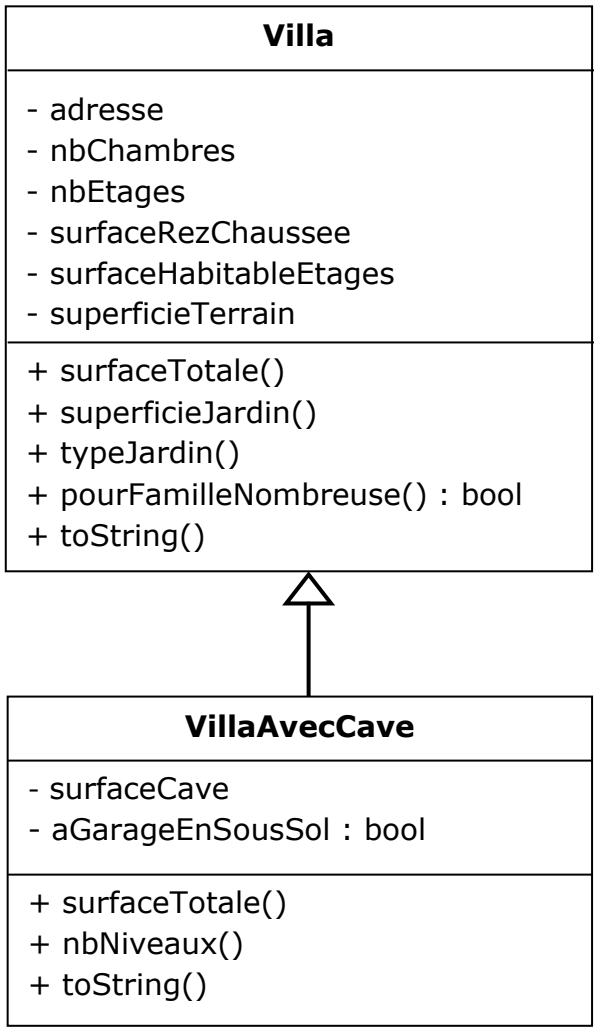
⑥ Un objet d'une sous-classe peut être affecté à un objet déclaré de type d'une super-classe.

⑦ Une variable d'instance, constructeur ou méthode déclaré *private* dans une super-classe n'est pas accessible au sein de la sous-classe.

⑧ Pour accéder en lecture à une variable d'instance privée héritée, il faut utiliser le getter public de la super-classe.

Exercice 1 : Sweet Home

Soient les classes `Villa` et `VillaAvecCave` qui permettent de gérer des villas sans cave ou avec caves pour l'agence immobilière *Sweet Home*.



Étape 1 : Super-classe Villa

Créez un nouveau projet et créez-y un package intitulé `sweetHome`.

Créez-y la classe `Villa`. Une villa est décrite par l'adresse, le nombre de chambres, le nombre d'étages, la surface du rez-de-chaussée (en m^2), la surface habitable pour l'ensemble des étages (en m^2) et la superficie du terrain (en m^2).

Prévoyez un constructeur permettant d'initialiser toutes les variables d'instance. Assurez-vous que s'il n'y a pas d'étage, la surface habitable aux étages soit égale à 0.

Surchargez le constructeur en prévoyant un second constructeur qui permet de créer des villas sans étage. Réfléchissez aux arguments strictement nécessaires. Essayez donc de fournir à ce constructeur le moins possible d'arguments.

Ajoutez les méthodes ci-dessous.

- La méthode `surfaceTotale` totalise la surface du rez-de-chaussée et des étages.
- La méthode `superficieJardin` retourne la superficie du jardin calculée en ares en soustrayant la surface du rez-de-chaussée de la superficie du terrain.
- La méthode `typeJardin` retourne, sur base de la superficie calculée du jardin,
    - soit ***sans jardin***
    - soit ***avec jardin de … ares*** s'il compte moins de 50 ares
    - soit ***avec parc de … ares*** s'il compte au moins 50 ares
- La méthode `pourFamilleNombreuse` retourne `true` si la villa compte au moins 4 chambres, `false` dans le cas contraire.
- La méthode `toString` retourne la chaîne de caractères décrivant une villa en respectant la structure ci-dessous :

  Surface totale ⬎
  La villa située au (adresse) d'une surface habitable de … m2
  … convient pour une famille nombreuse.
  ⬑ *Appel à la méthode typeJardin*

  Faites-en sorte que soit remplacé "*convient pour une famille nombreuse*" par "***ne*** *convient* ***pas*** *pour une famille nombreuse*" s'il y a lieu !

Créez la classe `Principale` dans le package `sweetHome`.

Créez dans la méthode `main` de la classe `Principale` un objet de type `Villa` appelé `bungalow` (sans étage ni jardin) et un autre objet appelé `maisonCampagne` avec 2 étages et une superficie de terrain de 30 ares.

Affichez à la console la description de ces deux objets.

Étape 2 : Sous-classe VillaAvecCave

Dans le package `sweetHome`, créez la classe `VillaAvecCave`. Une villa avec cave **est une villa** avec comme propriétés supplémentaires : la surface des caves (en m^2) et un booléen précisant s'il y a des garages en sous-sol.

Prévoyez un constructeur permettant d'initialiser toutes les variables d'instance.

> **Constructeur d'une sous-classe**
>
> Vous devez donc prévoir un constructeur qui prendra **8 arguments** : 6 arguments permettant d'initialiser les 6 variables d'instance héritées de la classe `Villa` et 2 arguments permettant d'initialiser les variables d'instance spécifiques à la classe `VillaAvecCave`.
>
> Dans ce constructeur, faites appel au constructeur de la super classe (via `super(…)`). Attention, cet appel doit obligatoirement être placé en première position dans le constructeur (cf. point 7.1 du syllabus).

Ajoutez la méthode `nbNiveaux` qui retourne le nombre de niveaux calculé à partir du nombre d'étages augmenté de 2, à savoir, le rez-de-chaussée et le niveau des caves.

> **Note**
>
> Voyez comment accéder à la variable d'instance `nbEtages` qui est héritée de la classe `Villa`. En effet, celle-ci est déclarée `private` dans la classe `Villa`.

Dans la méthode `main` de la classe `Principale`, créez au moins un objet de type `VillaAvecCave` appelé `manoir` avec 4 étages et une superficie de terrain de 80 ares.

Affichez à la console le nombre de niveaux de cet objet `manoir`.

Étape 3 : Redéfinition de méthode héritée

Redéfinition de méthode

Il est possible d'adapter une méthode héritée de la super-classe qui ne conviendrait pas parfaitement dans la sous-classe. Il suffit de réécrire cette méthode dans la sous-classe en gardant la **même signature** que la méthode héritée.

Appréhendez la redéfinition de méthode :

Lisez les sections 7.4 et 7.5 du syllabus.

Synthèse

① Une sous-classe peut adapter une méthode héritée qui ne conviendrait pas parfaitement. On parle alors de **redéfinition de méthode**. Pour redéfinir une méthode héritée, il suffit de prévoir dans la sous-classe, une méthode possédant la **même signature** que la méthode héritée à redéfinir.

② Une méthode peut être redéfinie de deux façons :

- Soit en **remplaçant** purement et simplement le code de la méthode héritée ;

- Soit en **récupérant** et en **étendant** le code de la méthode héritée : l'instruction pour appeler la méthode héritée est : ***super.methodeHeritee*** *(…)*

③ Une méthode héritée redéfinie sera annotée **@Override** par le compilateur.

Redéfinissez le méthode `surfaceTotale` de la classe `VillaAvecCave` pour qu'elle prenne en compte la surface des caves.

Note

Faites impérativement appel à la méthode `surfaceTotale` héritée de la classe `Villa` en utilisant l'instruction qui débute par `super.` (cf. point 7.5 du syllabus).

Redéfinissez ensuite la méthode `toString` qui retourne la chaîne de caractères décrivant une villa avec caves en respectant la structure ci-dessous :

... ⇨ *Appel à la méthode toString héritée*
Elle comporte ... niveaux et contient des caves de ... m2 avec garage(s) en sous-sol.

Complétez le nombre de niveaux ainsi que la surface des caves et remplacez "avec garage(s)" par "sans garage" s'il y a lieu

Affichez à la console la description de l'objet `manoir`.

Étape 4 : Polymorphisme

Principe du polymorphisme

Si une méthode héritée de la super-classe peut être redéfinie dans une sous-classe, la même méthode (même signature) peut donc être présente dans plusieurs classes de la même hiérarchie.

Lors de l'appel d'une telle méthode sur un objet, quelle méthode sera exécutée ? La méthode à exécuter sera choisie en **fonction du type de l'objet** sur lequel est appelée la méthode.

Tentez de maîtriser le polymorphisme :

Lisez la section 7.6 du syllabus.

Synthèse

① Quand il y a appel d'une méthode sur un objet, il y a recherche de cette méthode d'abord dans la classe correspondant à l'objet. Si elle ne s'y trouve pas, on remonte la hiérarchie d'héritage et la première méthode trouvée est exécutée. Si la méthode a été redéfinie dans plusieurs classes, ce principe assure que c'est la méthode **la plus spécifique** qui sera exécutée (c'est-à-dire celle qui se trouve le plus bas possible dans la hiérarchie d'héritage).

② On parle de polymorphisme lorsque ce n'est qu'**à l'exécution** qu'on sait déterminer **quelle méthode exécuter** si celle-ci (même signature) existe dans plusieurs classes. Le choix de la méthode à exécuter est déterminé en fonction du type de l'objet sur lequel la méthode est appelée.

Défi :
Détectez où le polymorphisme est appliqué dans le package `sweetHome`.

Où le polymorphisme intervient-il ici ?

Vérifiez que c'est la bonne méthode `surfaceTotale` qui a été appelée sur l'objet `manoir` !

Reprenons :

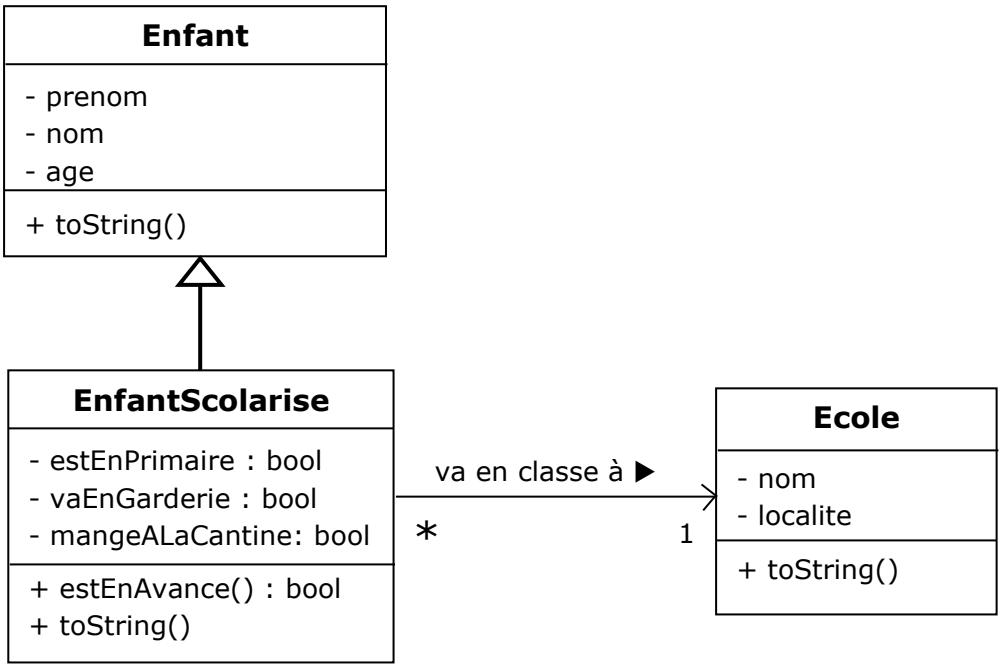
La méthode `toString` de la classe `VillaAvecCave` appelle la méthode `toString` héritée de la classe `Villa`. Le code de cette méthode fait appel à la méthode `surfaceTotale`, qui est définie à la fois dans `Villa` et `VillaAvecCave`. Laquelle de ces deux méthodes sera exécutée ?

Le polymorphisme assure que c'est toujours la méthode la plus spécifique qui sera choisie, en l'occurrence celle de la classe correspondant à l'objet sur lequel on appelle la méthode ; ici, il s'agit de l'objet `manoir` de type `VillaAvecCave`. C'est donc la méthode `surfaceTotale` de la classe `VillaAvecCave` qui est exécutée. La surface de la cave est donc ainsi bien prise en compte dans le calcul de la surface totale !

Exercice 2 : Petit écolier

Soient les classes `Enfant` et `EnfantScolarise` qui permettent de gérer des enfants respectivement non scolarisés et scolarisés.

Diagramme de classes



Étape 1 : Classe Enfant

Créez un package intitulé `petitEcolier`.

Créez-y la classe `Enfant`. Un enfant est décrit par son prénom, son nom et son âge.

Prévoyez un constructeur permettant d'initialiser toutes les variables d'instance.

Prévoyez un **filtre sur l'âge**.

- Si on tente d'attribuer un âge < 1, affectez 1 à la variable d'instance `age`.
- Si on tente d'attribuer un âge > 12, affectez 12 à la variable d'instance `age`.

Ajoutez la méthode `toString` qui retourne la chaîne de caractères décrivant un enfant en respectant la structure ci-dessous :

Pour un enfant âgé d'un an :

L'enfant (prenom) (nom) d'un an

Pour un enfant plus âgé :

L'enfant (prenom) (nom) de (age) ans

Créez la classe `Principale` dans le package `petitEcolier`.

Créez dans la méthode `main` de la classe `Principale` 3 objets de type `Enfant` en tentant de leur attribuer respectivement un âge de 8, 15 et -2 ans.
Affichez à la console la description de ces 3 objets et interprétez les résultats.

Étape 2 : Classe Ecole

Dans le package `petitEcolier`, créez la classe `Ecole`. Une école est décrite par son nom et sa localité.

Prévoyez un constructeur permettant d'initialiser toutes les variables d'instance.

Ajoutez la méthode `toString` qui retourne la chaîne de caractères décrivant une école en respectant la structure ci-dessous :

l'école (nom) (localité : (localité))

Étape 3 : Classe EnfantScolarise

Dans le package `petitEcolier`, créez la classe `EnfantScolarise` qui permet de gérer des enfants scolarisés. Un enfant scolarisé **est un enfant** pour lequel on précise, outre son école, s'il est inscrit ou non en primaire, s'il reste ou non à la garderie et s'il mange ou non à la cantine.

Prévoyez un constructeur permettant d'initialiser toutes les variables d'instance.

Contrainte

Faites appel au constructeur de la super classe (via `super(…)`).

Ajoutez la méthode `estEnAvance` qui retourne un booléen. Un enfant est en avance s'il est inscrit en primaire alors qu'il n'a pas encore 6 ans.

Note

Soyez attentifs, vous devez accéder à la variable d'instance `age` héritée mais déclarée `private` dans la super-classe.

Redéfinissez la méthode `toString` qui retourne la chaîne de caractères décrivant un enfant scolarisé en respectant la structure ci-dessous :

… ⇨ *description héritée de l'enfant*
est scolarisé en primaire à … . ⇨ *description de l'école*
(prenom) va à la garderie et mange à la cantine.

Faites en sorte que soient remplacés "en primaire" par "en maternelle", "va à la garderie" par "ne va pas à la garderie" et "mange à la cantine" par "ne mange pas à la cantine" s'il y a lieu

Dans la méthode `main` de la classe `Principale`, créez un objet de type `EnfantScolarise`.

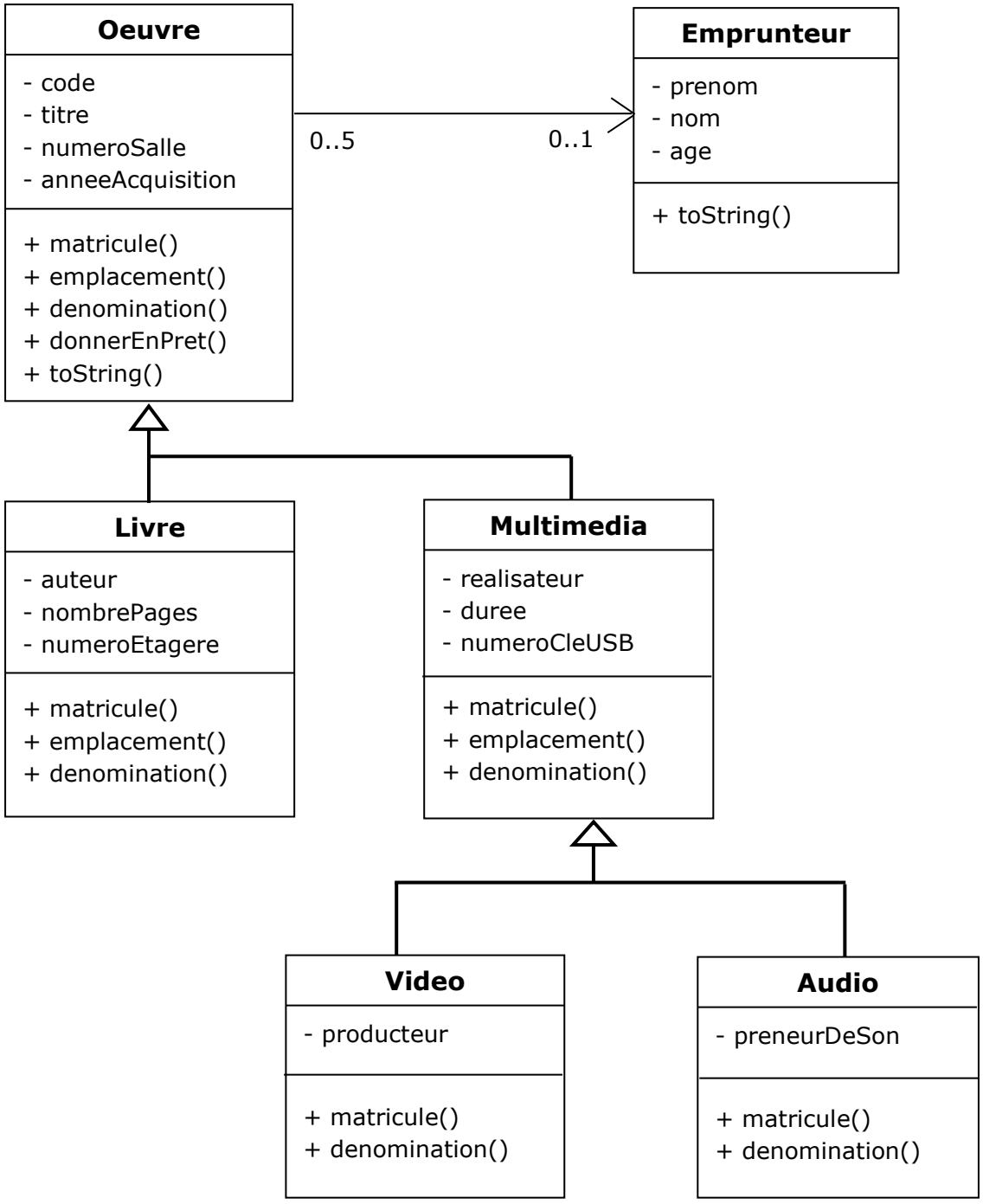
En fonction de l'état d'avancement de scolarité de cet enfant, affichez à la console : ***(prenom) est en avance***
ou
(prenom) n'est pas en avance

Affichez à la console la description de cet objet.

Exercice 3 : Bibliothèque

Dans une bibliothèque, des personnes peuvent emprunter des livres ou des documents multimédia de différentes catégories et ce, pour une durée de 2 semaines.

Le diagramme UML ci-dessous résume la situation :



Étape 1 : Classe Emprunteur

Créez le package `bibliotheque`. Créez-y la classe `Emprunteur`.

Un emprunteur est un client de la bibliothèque décrit par un nom, un prénom et un âge.

Prévoyez la méthode `toString` qui retourne la chaîne de caractères décrivant un emprunteur : combinaison de son prénom suivi de son nom.

Créez la classe `Principale` dans le package `bibliotheque.`

Dans la méthode `main`, créez un objet de type `Emprunteur` et affichez sa description.

Étape 2 : Classe Oeuvre

Créez la classe `Oeuvre` dans le package `bibliotheque`.

Chaque oeuvre est décrite par un code (de type int), un titre, le numéro de la salle où elle se trouve, l'année d'acquisition par la bibliothèque et son emprunteur éventuel.

Prévoyez un constructeur permettant de créer une oeuvre non empruntée.

Ajoutez les méthodes suivantes :

- une méthode `matricule` qui renvoit le matricule de l'oeuvre selon le format suivant : la lettre '**O**' suivie du code de l'oeuvre (exemple : *O3011*).
- une méthode `emplacement` qui indique la salle où une oeuvre se trouve, sous la forme *: **salle numéro (numSalle)**.*
- une méthode `denomination` qui renvoie la chaine de caractères "***Oeuvre***".
- Une méthode `donnerEnPret` qui reçoit en argument un objet de type `emprunteur` et qui permet d'associer cet emprunteur à l'oeuvre qu'il a empruntée.
- une méthode `toString` qui retourne la chaîne de caractères décrivant une oeuvre en respectant la structure proposée à travers l'exemple suivant :

  ***Oeuvre** - **O**3011 (salle numéro 17) : Jumanji*
  Emprunteur : Louis Lejeune

  Remplacez la dernière partie par "*Aucun emprunteur*" le cas échéant.

Dans la méthode `main` de la classe `Principale`, créez une oeuvre non empruntée. Affichez sa description.
Faites en sorte ensuite que cette oeuvre soit empruntée par l'emprunteur que vous avez créé précédemment. Affichez de nouveau la description de l'oeuvre.

Étape 3 : Classe Livre

Créez la classe `Livre` dans le package `bibliotheque`.

Un livre **est une oeuvre** avec comme propriétés supplémentaires : le nom de l'auteur, le nombre de pages et le numéro de l'étagère où il est rangé.

Prévoyez un constructeur permettant de créer un livre non emprunté (sans oublier de faire appel au constructeur de la classe mère).

Étape 4 : Classe Multimedia

Dans le package `bibliotheque`, créez la classe `Multimedia` permettant de gérer des documents multimedia (reportages, interviews...).

Un document multimedia **est une oeuvre** avec comme propriétés supplémentaires : le nom du réalisateur, la durée en secondes et le numéro de la clef USB sur laquelle elle est stockée.

Prévoyez un constructeur permettant de créer un document multimedia non emprunté (faites appel au constructeur de la classe mère).

Étape 5 : Méthodes des classes Livre et Multimedia

Dans les classes `Livre` et `Multimedia`, redéfinissez

- les méthodes `emplacement` qui renvoient la localisation des oeuvres. Le résultat attendu est le suivant : *salle numéro 17, **étagère** 5* (pour les livres) ; *salle numéro 17, **clef USB** 23* (pour les documents multimedia) (où les parties soulignées varient d'une oeuvre à l'autre).
- les méthodes `matricule` qui renvoient les matricules des oeuvres selon le format suivant :
  - pour les livres : la lettre '**L**' suivie du code du livre puis d'une barre oblique et du numéro de la salle (*par exemple : **L**1077/21 pour le code 1077 et la salle numéro 21*) ;
  - pour les documents multimedia : la lettre '**M**' suivie du code du document multimedia (*par exemple : **M**5025 pour le code 5025*).
- les méthodes `denomination` qui renvoient la chaîne de caractères "**Livre**" ou "**Document multimedia**".

Défi :

Attention, vous ne pouvez pas redéfinir de méthode `toString` dans les classes `Livre` et `Multimedia` !

Faites en sorte cependant que la méthode `toString` appelée sur un objet de type `Livre` ou `Multimedia` renvoie une chaîne de caractères dont la première ligne respecte la structure proposée à travers les exemples suivants :

pour un objet de type `Livre` :

Livre *-* ***L****1077/15 (salle numéro 15,* ***étagère*** *3) : Alice au pays des merveilles*

et pour un objet de type `Multimedia` :

Document multimedia *-* ***M****5025 (salle numéro 5,* ***clef USB*** *5) : Big Ben*

Heritage

Observez que le code permettant de renvoyer la description d'une oeuvre (`toString`) est le même pour toutes les oeuvres, qu'il s'agisse d'un livre ou d'un document multimedia. Avez-vous donc bien utilisé le concept d'héritage de méthode, en ne définissant la méthode `toString` que dans la classe `Oeuvre` ?

Polymorphisme

Cela nécessitera peut-être une révision du code que vous avez écrit dans la classe `Oeuvre`. Etes-vous certains d'avoir bien écrit le code de cette méthode `toString` ? En effet, vous devez faire en sorte que la chaîne de caractères `'Oeuvre'` devienne `'Livre'` ou `'Document multimedia'`. Il ne fallait donc pas hardcoder la chaîne de caractères `'Oeuvre'` dans `toString` de `Oeuvre`.

Indice

Pensez à remplacer cette chaîne de caractères par un appel de méthode. L'avantage est que cette méthode peut être redéfinie dans les sous-classes. Voyez si une des méthodes déjà existantes (définie dans `Oeuvre` et redéfinie dans `Livre` et `Multimedia`) ne ferait pas l'affaire.

Avantage

Le mécanisme du polymorphisme permet de ne pas redéfinir de méthode `toString` dans les classes `Livre` et `Multimedia`.

Dans la méthode `main` de la classe `Principale`, créez un livre non emprunté ainsi qu'un document multimedia non emprunté. Affichez leur description.
Assurez-vous que les affichages soient bien ceux attendus (cf. polymorphisme).

Faites en sorte ensuite que le livre soit emprunté par un emprunteur.
Affichez de nouveau la description du livre.

Étape 6 : Classe Video

Créez la classe `Video` dans le package `bibliotheque`.

Une vidéo **est un document multimedia** avec le nom du producteur comme propriété supplémentaire.

Prévoyez un constructeur permettant de créer une vidéo non empruntée (faites appel au constructeur de la classe mère).

Étape 7 : Classe Audio

Créez la classe `Audio` dans le package `bibliotheque`.

Un document audio **est un document multimedia** avec le nom du preneur de son comme propriété supplémentaire.

Prévoyez un constructeur permettant de créer un document audio non emprunté (faites appel au constructeur de la classe mère).

Étape 8 : Méthodes des classes Video et Audio

Dans les classes `Video` et `Audio`, redéfinissez

- les méthodes `matricule` qui renvoient les matricules des documents de type vidéo ou audio selon le format suivant :
  - pour les documents de type vidéo : la lettre '**V**' suivie du code de la vidéo (*par exemple : **V**1077 pour le code 1077*) ;
  - pour les documents de type audio : la lettre '**D**' (pour *Digital*) suivie du code du document audio si celui-ci a été acquis avant l'année 2000 ; la lettre '**N**' (Numérique) suivie code du document audio si celui-ci a été acquis en 2000 ou par la suite.
- les méthodes `denomination` qui renvoient la chaine de caractères "**Document vidéo**" ou "**Document audio**".

Faites en sorte que la méthode `toString` appelée sur un objet de type `Audio` renvoie une chaîne de caractères dont la première ligne respecte la structure proposée à travers les exemples suivants :

pour un objet de type `Audio` :

> ***Document audio*** - ***N****8532 (salle numéro 4, clef USB 3) : Le Big Data et nous*

et pour un objet de type `Video` :

> ***Document vidéo*** - ***V****8533 (salle numéro 5, clef USB 2) : Les cinglés de l'IOT*

Dans la méthode `main` de la classe `Principale`, créez une vidéo et un document audio non empruntés. Affichez leur description.
Assurez-vous que les affichages soient bien ceux attendus (cf. polymorphisme).

Faites en sorte ensuite que le document audio soit emprunté par un emprunteur. Affichez de nouveau la description du document audio.

Étape 10 : Pour aller plus loin : gestion des dates

Adaptez la classe `Emprunteur` : remplacez l'âge par la date de naissance.

Prévoyez une variable d'instance de type `LocalDate` pour stocker la date de naissance.

LocalDate

Dans l'exercice 2 de la série 3, vous avez dû créer un objet de type `LocalDate`.

Envie d'en savoir plus sur la gestion des dates via la classe `LocalDate` ?

Rendez-vous au point 8.2.2 du syllabus.

Adaptez le constructeur permettant d'initialiser toutes les variables d'instance. Celui-ci recevra donc en argument un objet de type `LocalDate`.

Surchargez le constructeur en prévoyant un second constructeur facilitant la création de la date de naissance. Ce constructeur ne recevra pas d'argument de type `LocalDate` mais 3 arguments de type entier : l'année, le mois et le jour.

Si le mois passé en argument est <1 ou >12, le mois sera considéré comme étant le mois de janvier.

Si le jour passé en argument est :

- <1
- >31
- =31 pour les mois comptant 30 jours
- >29 pour le mois de février

alors, le jour sera considéré comme étant le premier jour du mois.

Par facilité, on ne gérera pas les cas d'erreur liés au 29 février pour les années non bissextiles.

Suggestion

Faites appel au constructeur déjà créé (via l'instruction `this(…)`), en créant dans un premier temps un emprunteur avec une date de naissance nulle :

```
this(…, null) ;
```

Surchargez le setter `setDateNaissance` : prévoyez un second setter qui reçoit en argument une année, un mois et un jour. Les tests sur l'année, le mois et le jour sont placés au sein de ce second setter.

Créez l'objet de type `LocalDate` au sein-même de ce second setter. Cet objet de type date sera créé à partir de l'année, le mois et le jour éventuellement corrigés via l'instruction `LocalDate.of(…,…,…).`

Appelez ensuite ce second setter au sein du constructeur en lui passant l'année, le mois et le jour reçus en arguments du constructeur.

Prévoyez la méthode `presentationDateNaissance` qui doit afficher une date de naissance sous le format JJ/MM/AAAA : par exemple, *26/11/1990.*

Rappel

Il suffit d'accéder en lecture via les méthodes `getYear()`, `getMonthValue()` et `getDayOfMonth()` aux différentes informations d'un objet de type `LocalDate` .

```
LocalDate date = LocalDate.of(…,…,…);
// Récupération de l'année à partir d'un objet de type LocalDate
int annee = date.getYear();
```

Dans la méthode main de la classe `Principale`, créez un objet de type `LocalDate` correspondant à la date de naissance d'un emprunteur (date valide).

Créez ensuite cet emprunteur en appellant le premier constructeur.

Affichez à la console la description de sa date de naissance.

Créez ensuite un second emprunteur en appellant le second constructeur. Tentez de lui attribuer comme date de naissance le **32 décembre** 1980. Affichez la description de sa date de naissance et interprétez le résultat.

Créez d'autres emprunteurs en tentant de leur attribuer le 13/**15**/1981 ou le **31 avril** 1955 comme date de naissance et affichez la description de leur date de naissance. Interprétez les résultats.