R2-01-03 : TP 2

Développement orienté objets & Qualité de développement

Pour commencer

Pour la suite de ce module et l'utilisation de GIT, vous devez vous connecter au moins une fois au site https://gricad-gitlab.univ-grenoble-alpes.fr/. Ce site permettra d'héberger vos dépôts GIT dans le cadre de vos projet universitaire.

Continuer dans le projet crée pour le TP1, pour se faire **créer** un package tp2.

Continuer à utiliser les outils mis à votre disposition vu en R1.01 et au début de ce module : les mécanismes automatiques dans l'IDE, le deboggueur, etc.

Exercice 1 : Gestion d'une université

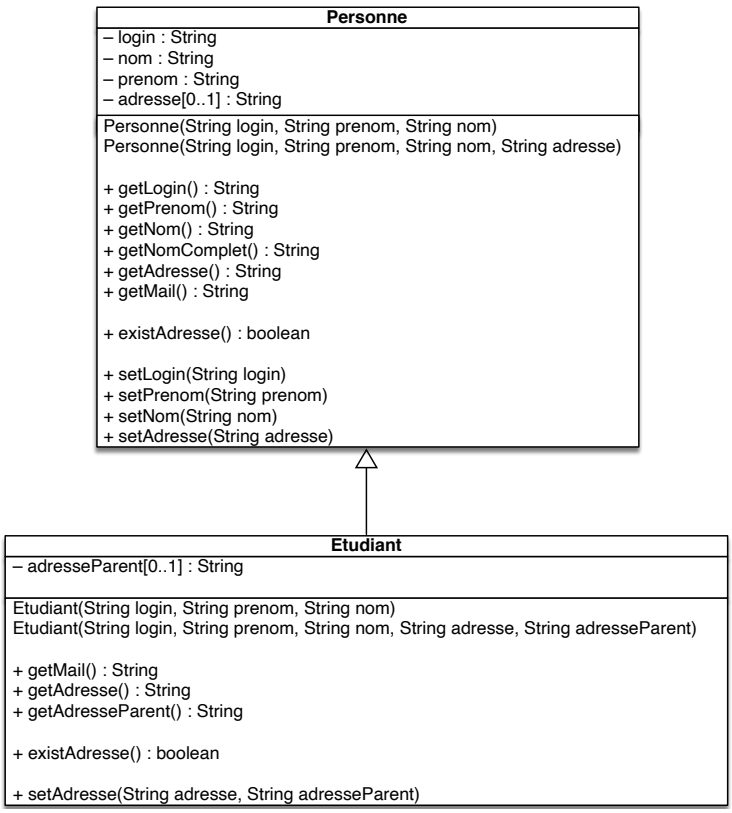
Objectifs R2-01 : héritage, créer une classe mère et une classe fille associée, créer une méthode redéfinie et créer une méthode surchargée.
Objectifs R2-03 : comprendre l'utilisation des redéfinitions de méthodes de leurs champs d'action.

Dans cet exercice, vous allez reprendre le sujet vu dans le TP1 : la gestion d'étudiants en l'étoffant. Vous allez ajouter la notion de personne, de personnelle et de notes contraintes.

ATTENTION vous allez recréer de nouvelles classes pour cet exercice pour conserver vos classes du package tp1 !

Exercice 1.1 : Héritage entre la classe Personne et Etudiant

On propose le modèle UML ci-dessous avec une classe **`Etudiant`** qui hérite d'une classe **`Personne`** (symbolisé par la flèche ⟶▷). La classe **`Personne`** est utilisée pour représenter une personne de l'université (étudiant, personnel, invité, etc.). La classe **`Etudiant`** est donc une classe fille de **`Personne`**. Un étudiant (objet de type **`Etudiant`**) est une personne avec « des spécificités ». Dans notre cas et pour le moment, un étudiant a une adresse en plus (celle de ses parents) et un mail différent d'une personne de l'université.



Explication de certaines méthodes du diagramme UML POUR LA CLASSE PERSONNE :

- `getMail()` : retourne une chaîne de caractères correspondant à « prenom.nom@univ-grenoble-alpes.fr »
- les autres attributs/méthodes correspondent aux caractéristiques d'étudiant dans le TP1. Les contraintes aussi !!! login en minuscule, prénom et nom avec une majuscule au début, etc.

Explication de certaines méthodes du diagramme UML POUR LA CLASSE ETUDIANT :

- `setAdresse(adresseEtudiant, adresseParent)` : C'est une surcharge de la méthode `setAdresse(…)` de la classe `Personne`. Un étudiant à maintenant deux adresses, la sienne personnelle (héritée de `Personne`) et celle de ses parents (spécifié dans `Etudiant`). Cette méthode permet de mettre à jour les deux informations.

Aide : une classe fille n'a pas accès aux attributs privées de la classe mère mais elle a accès à ses méthodes `public`.

- `getMail()` : C'est une redéfinition de la méthode `getMail()` de la classe `Personne`, voir ci-avant. Retourne une chaîne de caractères correspondant à « prenom.nom@**etu.**univ-grenoble-alpes.fr ».
- `existAdresse()` C'est une redéfinition de la méthode `existAdresse()` de la classe `Personne`. Retourne vrai si l'étudiant à au moins une adresse.
- `getAdresse()` C'est une redéfinition de la méthode `getAdresse()` de la classe `Personne`, voir ci-avant. Retourne une chaîne de caractères correspondant à l'adresse principale suivi de l'adresse des parents. Si une seule des deux existe, la méthode ne retourne qu'une adresse soit principale soit celle des parents (voir la trace donnée ci-après).

Aide : dans une classe fille, vous pouvez utiliser les méthodes de la classe mère que vous êtes en train de redéfinir à l'aide du mot clé **`super`**. Par exemple, **`super.getAdresse()`**. Ceci est utile si vous souhaitez réutiliser ou retraiter une information de la classe mère.

Dans le package tp2, **créer** un package **`universite`** et dans ce package les classes **`Etudiant`**, **`Personne`**, **`TestUniversite`** et **`UniversiteUtilitaire`**.

ATTENTION : Dans votre projet, vous aurez maintenant 2 classes **`Etudiant`** une dans le package tp1.universite et une dans le package tp2.universite. **Faites attention aux imports lors de l'utilisation d'une classe `Etudiant` pour utiliser la bonne !** Vous remarquerez que les packages servent pour structurer et aussi pour désambiguïser les usages des classes.

Compléter la classe **`Personne`** et la classe **`Etudiant`** en respectant bien l'héritage. La classe **`Etudiant`** ne contiendra que le strict nécessaire ! Attention à bien respecter le principe d'encapsulation.

```
public class Etudiant extends Personne {

    private String adresseParent;

    public Etudiant(String login, String prenom, String nom) {

        // Appel du constructeur de la classe parent
        super(login, prenom, nom);
    }
```

Aide concernant les mots clés de l'héritage : **`extends`** pour la caractérisation de l'héritage et **`super`** pour l'appel au constructeur. **`super`** est également utilisé pour appeler une méthode de la classe mère comme vu ci-avant, par exemple **`super.getAdresse()`**

Compléter `TestUniversite` et **`UniversiteUtilitaire`** en utilisant le test et la trace ci-après. **`UniversiteUtilitaire`** contiendra la méthode d'affichage d'une personne.

TestUniversite :

```
public static void main(String[] args) {

    // Création d'une liste de personnes
    ArrayList<Personne> personnes = new ArrayList<>();

    // Création de deux personnes
    Personne personne1 = new Personne("martinp", "Philippe", "MARTIN");
    Personne personne2 = new Personne("blanchonh", "Hervé", "BLANCHON", "2 Place Doyen Gosse");

    // Ajouter les deux personnes à la liste
    personnes.add(personne1);
    personnes.add(personne2);

    // Création de trois étudiants
    Etudiant etudiant1 = new Etudiant("SANZF", "FLORIANT", "Sanz");
    Etudiant etudiant2 = new Etudiant("portepa", "Pierre-Antoine", "Porte");
    Etudiant etudiant3 = new Etudiant("burlatn", "nils", "burlat", "1 rue pas loin", "10 rue des parents");


    // Ajouter l'adresse de ses parents à l'étudiant1
    // Ajouter l'adresse personnelle de l'étudiant2
    etudiant1.setAdresse("13 rue à côté", "23 rue beaucoup plus loin");
    etudiant2.setAdresse("30 avenue Berlioz");

    // Ajouter les trois étudiants à la liste
```

```
    personnes.add(etudiant1);
    personnes.add(etudiant2);
    personnes.add(etudiant3);

    // Affichage des personnes
    // Utilisation d'une simplifacation d'écriture avec le for
    for (Personne personne : personnes) {
        UniversiteUtilitaire.affichePersonne(personne);
    }
```

CONSEIL : toujours être le plus exhaustif dans vos tests.

Exemple de trace :

```
-----------------
Login : martinp
Nom complet : Philippe Martin
Mail : Philippe.Martin@univ-grenoble-alpes.fr
Adresse : aucune

-----------------
Login : blanchonh
Nom complet : Hervé Blanchon
Mail : Hervé.Blanchon@univ-grenoble-alpes.fr
Adresse : 2 Place Doyen Gosse

-----------------
Login : sanzf
Nom complet : Floriant Sanz
Mail : Floriant.Sanz@etu.univ-grenoble-alpes.fr
Adresse : 13 rue à côté (Adresse des parents 23 rue beaucoup plus loin)

-----------------
Login : portepa
Nom complet : Pierre-antoine Porte
Mail : Pierre-antoine.Porte@etu.univ-grenoble-alpes.fr
Adresse : 30 avenue Berlioz

-----------------
Login : burlatn
Nom complet : Nils Burlat
Mail : Nils.Burlat@etu.univ-grenoble-alpes.fr
Adresse : 1 rue pas loin (Adresse des parents 10 rue des parents)
```

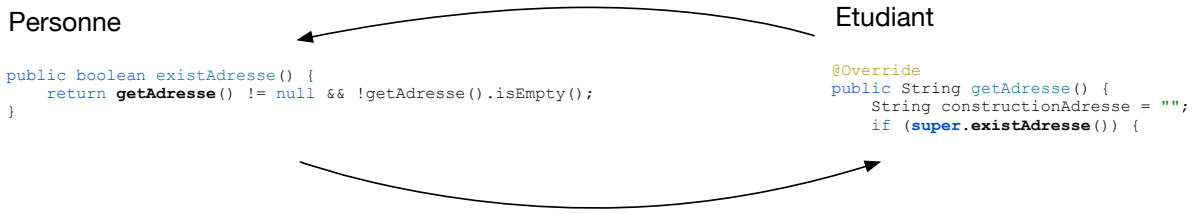
ATTENTION : un problème de boucle peut arriver ! **`StackOverflowError`** qui traduit ici une boucle infinie :

```
Exception in thread "main" java.lang.StackOverflowError
        at tp2.universite.Etudiant.getAdresse(Etudiant.java:62)
        at tp2.universite.Personne.existAdresse(Personne.java:52)
        at tp2.universite.Etudiant.getAdresse(Etudiant.java:62)
        at tp2.universite.Personne.existAdresse(Personne.java:52)
```

POURQUOI ? Vous avez redéfinit les méthodes **`existAdresse()`** et **`getAdresse()`** pour cet exercice. Vous avez bien pensé à utiliser le mot clé **`super`** pour appeler la méthode du parent. Par exemple :

```java
@Override
public String getAdresse() {
    String constructionAdresse = "";
    if (super.existAdresse()) {
```

Mais dans la classe parent **Personne**, la méthode **existAdresse**() appelle la méthode **getAdresse**() **MAIS** de la classe fille (à cause de la redéfinition) qui appelle la méthode **existAdresse**() de **Personne** qui appelle ... et boucle infinie !



Pour remédier à ce problème, modifier **existAdresse**() de **Personne** pour utiliser l'attribut adresse et non la méthode **getAdresse**() :

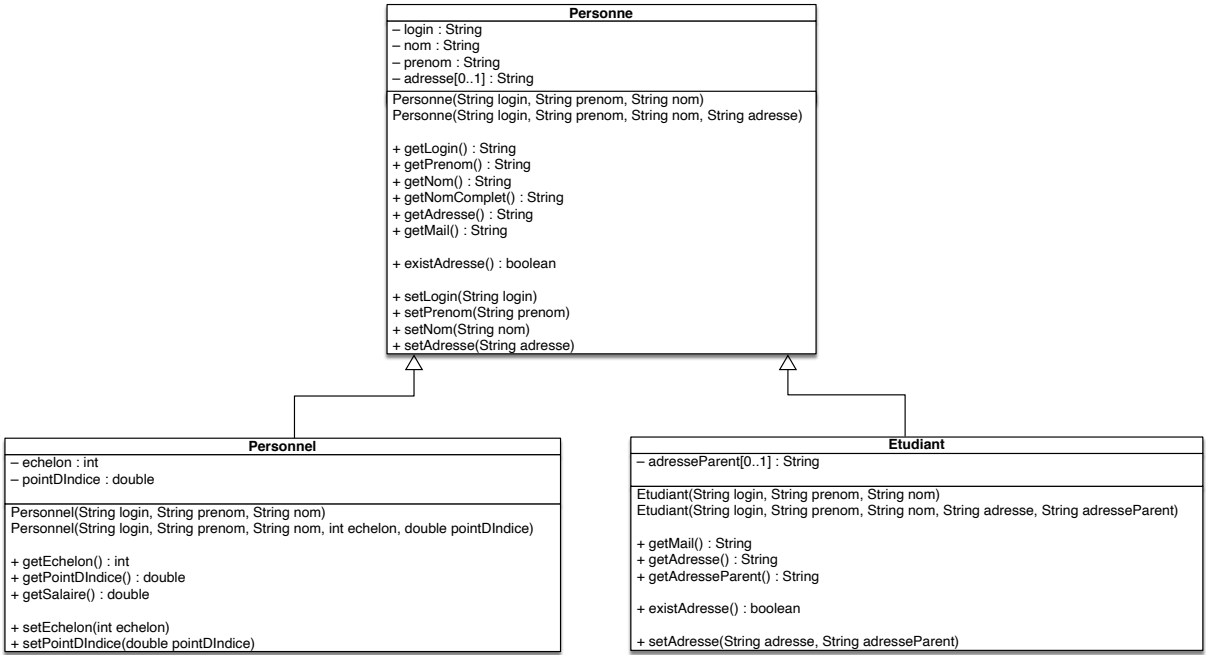
```java
public boolean existAdresse() {
    //return getAdresse() != null && !getAdresse().isEmpty();
    return adresse != null && !adresse.isEmpty();
}
```

IMPORTANT : attention à l'utilisation des getter dans une classe et à leur effet de bord lors d'une redéfinition.

Exercice 1.2 : ajout d'une classe fille Personnel

Objectifs R2-01 : créer un héritage complexe.

Le modèle précédent ne permet pas de modéliser le personnel de l'université correctement car on a besoin d'attributs particuliers pour le calcul du salaire. On propose le modèle UML suivant.



Explication de certaines méthodes du diagramme UML pour la classe `Personnel`

- Pour déterminer le salaire d'un personnel de l'université, deux informations sont importantes : son échelon entre 1 et 4 (entier), et la valeur du point d'indice entre 1000 et 1200 (réel). Ces contraintes sont à faire respecter si une valeur est donnée en dessous ou en dessus, remplacer par la valeur min ou max.
- Un personnel crée par le constructeur sans échelon et sans point d'indice se verra attribuer les valeurs minimales.

Aide : Pour définir des constantes en java, il faut écrire :

```
public static final int ECHELON_MIN = 1;
```

Les mots clés utilisés :

`static` permet de préciser que c'est un attribut de classe et non d'objet. Donc à une classe et son ensemble d'objets instanciés, cet attribut de classe sera unique, il y aura un seul ECHELON_MIN. Par convention, on écrit une constante en majuscule.

`Final` permet de bloquer l'usage de la valeur primitive int associée. La valeur restera à 1 dans ce cas précis.

Nous reviendrons sur ces mots clés dans les prochaines séances.

- **`getSalaire`**() : retourne le salaire d'un **`Personnel`**, c'est-à-dire le produit de **`echelon`** et **`valeurPointIndice`**. Par exemple, pour un échelon de 2 et un indice de 1000, le salaire sera de 2000.
- L'adresse mail d'un **`Personnel`** est l'adresse mail d'une **`Personne`**.

Dans le package tp2, **créer** la classe **`Personnel`**. **Compléter** cette classe en respectant l'héritage. Aucune redéfinition nécessaire pour cette classe.

Compléter `TestUniversite` et **`UniversiteUtilitaire`** en utilisant le test et la trace ci-après. Vous ajouterez à la classe **`UniversiteUtilitaire`** la méthode d'affichage d'un personnel. Nous ajoutons le personnel dans la liste de personnes pour vérifier qu'un objet de type **`Personnel`** et également de type **`Personne`**. Nous ajoutons ensuite ce personnel dans une nouvelle liste de type **`Personnel`** pour avoir un affichage spécifique du personnel (échelon, point d'indice et salaire). Obligatoire si l'on souhaite afficher les particularités du personnel.

Ajout à TestUniversite

```
// Création de deux personnels
Personnel personnel1 = new Personnel("goulianj", "Jérôme", "Goulian");
Personnel personnel2 = new Personnel("brunetf", "Francis", "Brunet-
Manquat", 0, 1400);
Personnel personnel3 = new Personnel("lejeuna", "Anne", "Lejeune");
personnel3.setEchelon(5);
personnel3.setPointDIndice(900);

// Ajouter les personnels à la liste personnes
personnes.add(personnel1);
personnes.add(personnel2);
personnes.add(personnel3);

// Créer et ajouter les personnels à une liste personnels
ArrayList<Personnel> personnels = new ArrayList<>();
personnels.add(personnel1);
personnels.add(personnel2);
```

```
personnels.add(personnel3);

// Affichage des personnes
// Utilisation d'une simplifacation d'écriture avec le for
for (Personne personne : personnes) {
    UniversiteUtilitaire.affichePersonne(personne);
}

// Affichage du personnel
for (Personnel personnel : personnels) {
    UniversiteUtilitaire.affichePersonnel(personnel);
}
```

Extrait de l'affichage des personnels

```
… [affichage des personnes]

-----------------
Login : goulianj
Nom complet : Jérôme Goulian
Mail : Jérôme.Goulian@univ-grenoble-alpes.fr
Adresse : aucune
Echelon : 1
Point d'indice : 1000.0
Salaire : 1000.0

-----------------
Login : brunetf
Nom complet : Francis Brunet-manquat
Mail : Francis.Brunet-manquat@univ-grenoble-alpes.fr
Adresse : aucune
Echelon : 1
Point d'indice : 1200.0
Salaire : 1200.0

-----------------
Login : lejeuna
Nom complet : Anne Lejeune
Mail : Anne.Lejeune@univ-grenoble-alpes.fr
Adresse : aucune
Echelon : 4
Point d'indice : 1000.0
Salaire : 4000.0
```

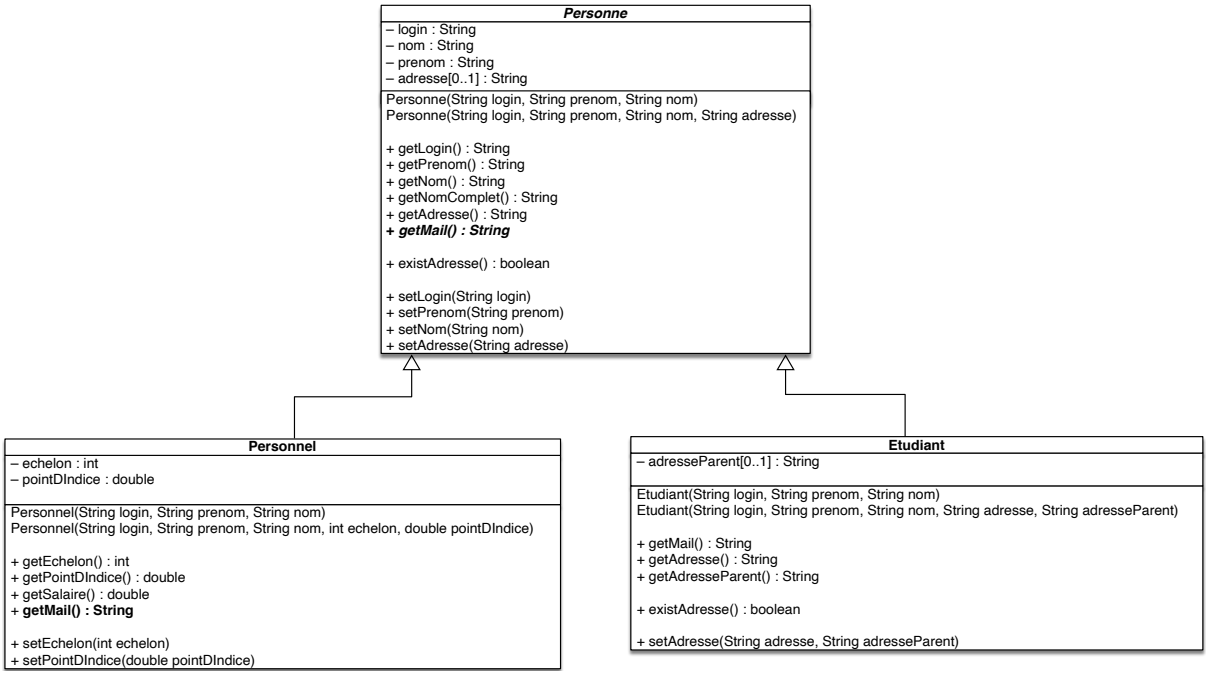
Exercice 1.3 : abstraction de la classe Personne

Objectifs R2-01 : rendre une classe abstraite

On décide maintenant que l'on ne créera pas d'instance (d'objet) de la classe **`Personne`** qui devient dont une classe abstraite (***`Personne`*** – en italique notation UML) comme illustré dans le diagramme UML ci-dessous. Pour que le modèle soit cohérent, la méthode **`getMail()`** de la classe ***`Personne`*** devient aussi abstraite (***`getMail()`*** – en italique notation UML) et la classe **`Personnel`** est dotée d'une méthode **`getMail()`**.

POURQUOI rendre une classe abstraite ? on considère ici que la classe **`Personne`** n'a d'intérêt que pour la **mise en commun** d'un état et de comportements (attributs et méthodes) pour les classes filles. On ne souhaite plus créer des personnes mais seulement

soit du personnel soit des étudiants. La classe abstraite **Personne** reste cependant un type que l'on pourra utiliser pour regrouper du personnel et des étudiants si nécessaire.



Mettre à jour la classe **Personne** pour respecter le modèle UML : rendre la classe ***Personne*** abstraite et rendre la méthode ***getMail()*** abstraite également.

Mettre à jour la classe **Personnel** pour respecter le modèle UML : ajouter la méthode **getMail()** qui retourne une chaîne de caractères correspondant à « prenom.nom@univ-grenoble-alpes.fr »..

Vérifier que la procédure principale **main()** de la classe **TestUniversite** continue de se comporter comme dans l'exercice précédent. Pour se faire, il faudra **commenter** les instructions créant des objets de type **Personne** et utilisant ces objets. Ces instructions provoquent maintenant une erreur : 'Personne' is abstract; cannot be instantiated. Pour rappel, une classe abstraite ne peut pas créer d'objet, mais vous pouvez l'utiliser comme type comme par exemple avec la liste de **Personne** dans laquelle on ajoute des étudiants et du personnel qui ne sont rien d'autres que des personnes.

Début de la trace

```
-----------------
Login : sanzf
Nom complet : Floriant Sanz
Mail : Floriant.Sanz@etu.univ-grenoble-alpes.fr
Adresse : 13 rue à côté (Adresse des parents 23 rue beaucoup plus loin)

-----------------
Login : portepa
Nom complet : Pierre-antoine Porte
Mail : Pierre-antoine.Porte@etu.univ-grenoble-alpes.fr
Adresse : 30 avenue Berlioz

-----------------
Login : burlatn
Nom complet : Nils Burlat
```

```
Mail : Nils.Burlat@etu.univ-grenoble-alpes.fr
Adresse : 1 rue pas loin (Adresse des parents 10 rue des parents)

Etc…
```

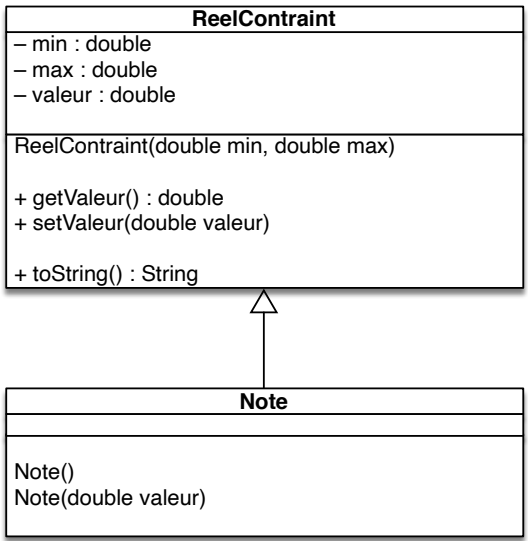
Exercice 1.4 : contraindre des réels

Objectifs R2-01 : créer une classe mère et sa classe fille, puis les utiliser.

On propose une classe **`ReelContraint`** qui permet de définir des réels pour lesquels on assure que leur valeur est comprise dans un intervalle fixé par une valeur minimale (**`min`**) et une valeur maximale (**`max`**).

On propose une classe **`Note`** qui hérite de la classe **`ReelContraint`** qui va permettre de représenter des notes obtenues par les **`Etudiant`**s. Une note est bien sûr comprise entre **`0`** et **`20`**.

Le diagramme UML correspondant :



Dans le paquetage **`tp2`**, créer un package **`contrainte`** puis créer les classes **`ReelContraint`**, **`Note`**, **`ContrainteUtilitaire`** et **`TestContrainte`**.

Compléter la classe **`ReelContraint`** en ajoutant les attributs, le constructeur et les méthodes proposés par le modèle UML de la classe **`ReelContraint`** en tenant compte des spécifications suivantes :

- **Spécification 1 :** la valeur initiale (à la contruction) d'un **`ReelContraint`** est la valeur minimale (**`min`**).
- **Spécification 2 :** la méthode **`setValeur(double valeur)`** est une méthode **non interactive** (aucune demande à l'utilisateur par le terminal) qui ne modifie la valeur du **`ReelContraint`** que si **`valeur`** est dans l'intervalle [**`min`**, **`max`**].
- **Spécification 3** : la méthode **`toString()`** retourne une chaine de caractère représentant l'entier contraint : « valeur : X, min : Y, max : Z ». Un objet de type ReelContraint pourra être utilisé directement dans une instruction `System.out.prinln(unObjetReelContraint)`qui affichera le résultat de la métode **`toString()`**. Pour aller plus loin, cette méthode est redéfinie de la méthode **`Object`** dont toutes les classes héritent.

- **Spécification 4 :** créer une méthode `saisir(double min, double max)` qui retourne un objet de type `ReelContraint` dans la classe `ContrainteUtilitaire`. Cette méthode est une méthode **interactive** qui permet de veiller à ce que la valeur saisie par l'utilisateur soit bien comprise dans l'intervalle [`min`, `max`] ; en d'autre terme, avant de créer l'objet de type `ReelContraint` cette méthode demande une nouvelle valeur tant que la valeur saisie à la console n'est pas comprise dans l'intervalle [`min`, `max`].

Compléter la méthode principale `main` de `TestContrainte` pour tester la classe `ReelContraint`.

CONSEIL : essayez d'être exhaustif dans vos tests. Dans cet exercice, essayez tous les constructeurs, toutes les méthodes ET les valeurs limites.

Compléter la classe `Note` en ajoutant uniquement les deux constructeurs proposés dans le diagramme :

- Un constructeur par défaut qui initialise le constructeur parent `ReelContraint` avec `min 0`, `max 20` et une `valeur` initiale égale à `min`
- Un constructeur à un paramètre qui initialise le constructeur parent `ReelContraint` avec `min 0`, `max 20`, mais dont on connaît la `valeur` initiale (comprise entre 0 et 20, si la valeur donnée est inférieure ou supérieure, la valeur prend min ou max en fonction)

Compléter la méthode principale `main` de `TestContrainte` pour tester la classe `Note` en :
- Déclarant des objets de type `Note` construits avec les deux constructeurs.
- Testant la méthode `toString()` ainsi que l'**accesseur** (getter) et les **mutateurs** (setter et `saisir()`)
- Créer une méthode `saisir()` qui retourne un objet de type Note dans la classe `ContrainteUtilitaire` et la tester.

Modifier la classes `Personnel`, `Etudiant` et `UniversiteUtilitaire` pour utiliser les classes `ReelContrait` et `Note` :

- **Modifier** la classe Personnel pour utiliser `ReelContraint` à la place du type double pour le point d'indice.
- **Ajouter** les notes et le calcul de la moyenne comme dans le TP1 pour la classe `Etudiant` en utilisant le type `Note` et une liste dynamique `ArrayList` (vu en R1.01) pour gérer la liste des notes d'un étudiant (plus besoin d'un indice pour gérer le nombre de notes ajoutées comme pour un tableau)

Exercice 1.5 : les prénoms et noms composés (bonus)

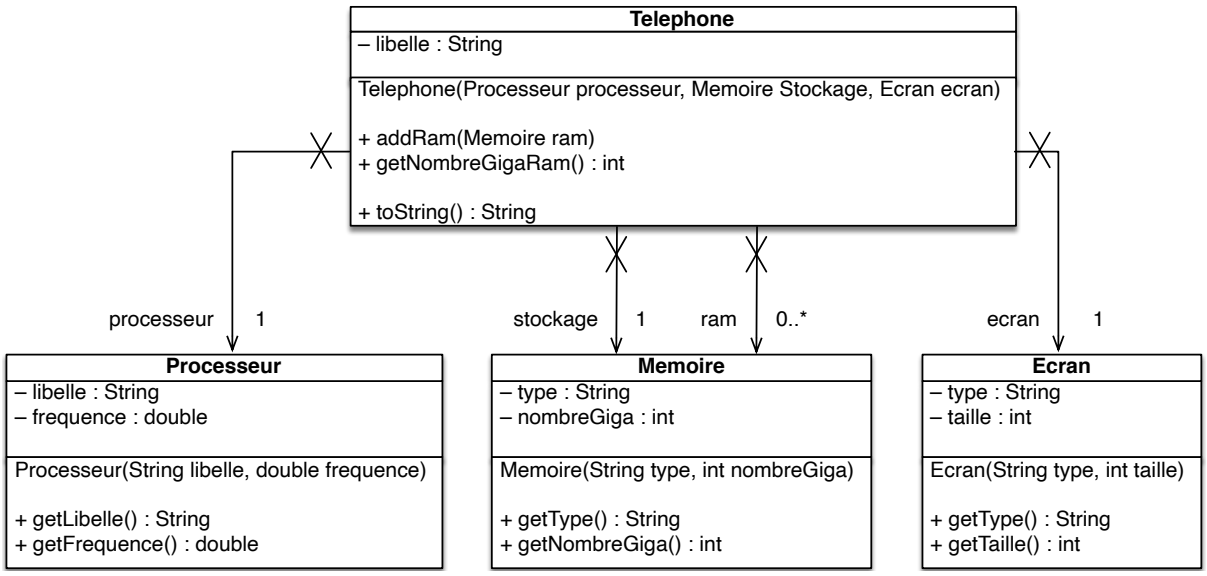
Les prénoms et noms composés ne sont pas bien pris en compte pour le moment. **Modifier** la bonne classe pour les prénoms et noms composés aient une majuscule après le tiret - , exemples : Pierre-Antoine, Brunet-Manquat.

Exercice 2 : Téléphone

Objectifs R2-01 : création d'objets, utilisation de méthodes.
Objectifs R2-03 : gestion de version (GIT). **GIT En cours d'élaboration.**

Cet exercice reprend les notions vues dans le TP1 : création de classes et d'objets, et association de classes. Il n'y a pas de piège dans cet exercice.

Nous souhaitons créer un catalogue de pièces de téléphones pour confectionner des téléphones en fonction des besoins des clients (bas de gamme, milieu de gamme, etc.). Ci-dessous le diagramme UML correspondant :



Explication de certaines méthodes du diagramme UML

- Un téléphone est composé d'un processeur, d'un écran, d'un stockage et de mémoires ram (représentés par une liste dynamique de mémoires 0..* sur le diagramme)
- **`addRam(…)`** dans la classe **`Telephone`** permet d'ajouter de la mémoire au téléphone
- **`getNombreGigaRam()`** dans la classe **`Telephone`** retourne la somme de la mémoire ram du téléphone
- **`toString()`** dans la classe **`Telephone`** retourne une chaine de caractères (voir la trace attendue). Pour rappel, cette méthode est redéfinie de la méthode **`Object`** dont toutes les classes héritent. Elle sera utilisée dans une instruction `System.out.prinln(Telephone)` qui affichera le résultat de la métode **`toString()`**.

Dans le paquetage **`tp2`**, créer un package **`telephone`** puis créer les classes **`Telephone`**, **`Processeur`**, **`Memoire`**, **`Ecran`** et **`TestTelephone`**. Vous trouverez ci-après le début du test à compléter et la trace attendue.

TestTelephone

```
public static void main(String[] args) {

    // Liste de processeurs disponibles : Octa-Core 2.84Ghz et Octa-Core 3Ghz
    // A COMPLETER

    // Liste de ram disponibles : LPDDR5 4G et LPDDR5 8G
    // A COMPLETER

    // Liste de stockage disponibles : 3 mémoires UFS Storage 3.1 de
grandeur 64/128/256
    // A COMPLETER

    // Liste d'écran disponibles : 3 amoled de tailles 5/6/7
```

```
    // A COMPLETER

    ///// Construction des téléphones
// Bas de gamme
Telephone telephone1 = new Telephone("Bas de gamme", processeur1,
stockage1, ecran1);
telephone1.addRam(ram1);

    // A COMPLETER
```

Trace attendue

```
Bas de gamme, processeur = Octa-Core (2.84GHz), ram = 4Giga [LPDDR5 4G] ,
stockage = [UFS Storage 3.1, 64Giga] , ecran = [Amoled, 5 pouces]

Bas de gamme+, processeur = Octa-Core (2.84GHz), ram = 8Giga [LPDDR5 4G + LPDDR5 4G] ,
stockage = [UFS Storage 3.1, 128Giga] , ecran = [Amoled, 5 pouces]

Milieu de gamme, processeur = Octa-Core (2.84GHz), ram = 8Giga [LPDDR5 4G + LPDDR5 4G] ,
stockage = [UFS Storage 3.1, 128Giga] , ecran = [Amoled, 6 pouces]

Haut de gamme, processeur = Octa-Core (3.0GHz), ram = 16Giga [LPDDR5 8G + LPDDR5 8G] ,
stockage = [UFS Storage 3.1, 256Giga] , ecran = [Amoled, 7 pouces]
```

IMPORTANT : avant de passer à la suite du TP (exercice bonus), demandez à votre enseignant de valider votre travail !

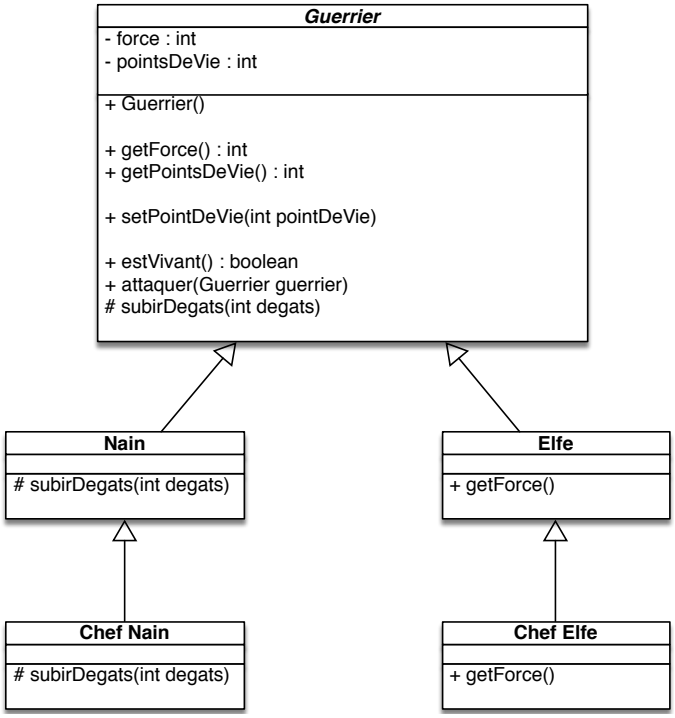
Exercice 3 : Exercice « fil rouge » : La bataille de Faërun (étape 2)

Cet exercice sera présent dans chaque TP et permettra de revenir sur les notions abordées dans les TDs et les TPs sous la forme d'un petit jeu sur le terminal. Cette séquence n'est pas forcément à commencer dès les premières semaines, elle pourra être commencée à tout moment quand vous serez plus à l'aise avec la notion d'objet. **Si vous n'avez pas encore réalisé l'étape 1, rendez-vous à la fin du TP1 !**

IMPORTANT : cet exercice sera évalué dans le cadre de la ressource R02.03 : bonne pratique, réalisation, documentation, gestion de version, test, etc.

Dans la suite, vous compléterez et modifierez les classes du package ***`jeu`****.*

Dans cette deuxième étape du jeu, vous devez créer les classes **`Nain`**, **`Elf`**, **`ChefNain`** et **`ChefElfe`** en respectant le modèle UML proposé à droite du document (la classe ***`Guerrier`*** devient une classe abstraite et les autres classes sont des classes filles). Les spécifications de ces nouvelles classes sont disponibles dans les documents présentant le jeu. Créer une classe **`TestEtape2`** contenant des tests pour ces nouvelles classes et validant leurs spécifications : combats entre différentes classes et affichage des dégâts donnés par l'attaquant et des dégâts réellement subits par le défenseur.



ATTENTION, plusieurs solutions sont possibles pour réaliser la demande. MAIS nous souhaitons une solution avec un héritage et orientée objet ! Donc n'ajoutez pas d'attributs qui ne sont pas dans le diagramme. L'important est de comprendre que les spécificités des guerriers nain, elfe, etc doivent se retrouvent dans leurs classes !

Si vous hésitez sur une solution, demandez à votre enseignant.