TD JAVA n°2: Héritage et Polymorphisme

Exercice 1 :

Quelles sont les erreurs contenues dans ces définitions de classes ?

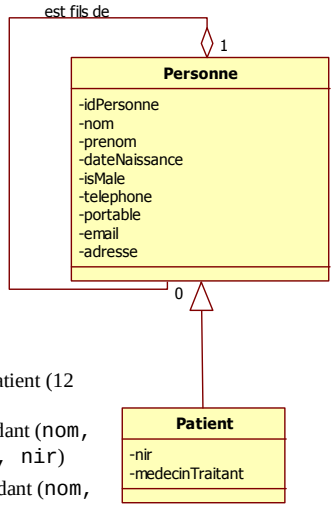
```
public class A
{
      private double x ;

      public A()
      {
          x = 0.0 ;
      }
      public void setX(double nX)
      {
          if(nX > 0.0)
                  x = nX;
      }
      public getX()
      {
          return x ;
      }
}
public class B extends A
{
      public B()
      {
          this.x = 1.0 ;
      }
      public double getY()
      {
          return x * 2.0;
      }
}
```

Exercice 2 : Cabinet Médical

2.1 Héritage : classe `Patient`

↳ 2.1.a Définir une classe **`Patient`**.
Utiliser l'héritage en prenant comme classe de base la classe **`Personne`**.

- Définir les nouveaux **attributs** :
  - → Le numéro de sécurité sociale (**`nir`**) sera mémorisé dans une variable de type **`String`**.
  - → Le nom et prénom du médecin traitant seront pour l'instant mémorisés directement dans une variable de type **`String`** appelée **`medecinTraitant`**
- Définir les constructeurs suivants :
  - → **constructeur par défaut** (sans code)
  - → **constructeur** avec tous les paramètres caractérisant un patient (12 paramètres)
  - → **constructeur avec 6 arguments significatifs** sans ascendant (`nom, prenom, dateNaissance, isMale, adresse, nir`)
  - → **constructeur avec 7 arguments significatifs** avec ascendant (`nom, prenom, dateNaissance, isMale, adresse, ascendant, nir`)



- Définir les **getteurs / setteurs** correspondant aux nouveaux attributs
- Redéfinir la **méthode `toString`**

↳ 2.1.b Redéfinir la méthode `equals` qui est déjà une méthode de la classe `Object`.
Notre choix de conception en ce qui concerne l'égalité de la classe `Patient` est le suivant :
Deux objets de la classe `Patient` seront considérés comme « égaux » si et seulement si leur `nom, prenom, nir` et `dateNaissance` sont identiques.

java.lang
Class Object

`java.lang.Object`

`public class Object`

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

`boolean`	**`equals`**`(Object obj)` Indicates whether some other object is "equal to" this one.

Remarque : penser à utiliser l'opérateur booléen **`instanceof`** (voir cours)

2.2 Complément pour le TP ⇒ Héritage : classe `Professionnel`

Cette classe n'est pas à écrire en TD (sauf si le temps le permet),
mais elle devra être implémentée en TP !!!!
Pour la suite on suppose cette classe écrite !!!

De la même manière, on pourrait définir une classe **`Professionnel`** en utilisant l'héritage et en prenant comme classe de base la classe ***`Personne`*** (voir annexe 2).

2.3 Classe abstraite

↳ Qu'est-ce qu'une classe abstraite ?

↳ Transformer la classe `Personne` en classe abstraite

2.4 Polymorphisme : Gestion d'une liste de `Personne` *avec un tableau statique*

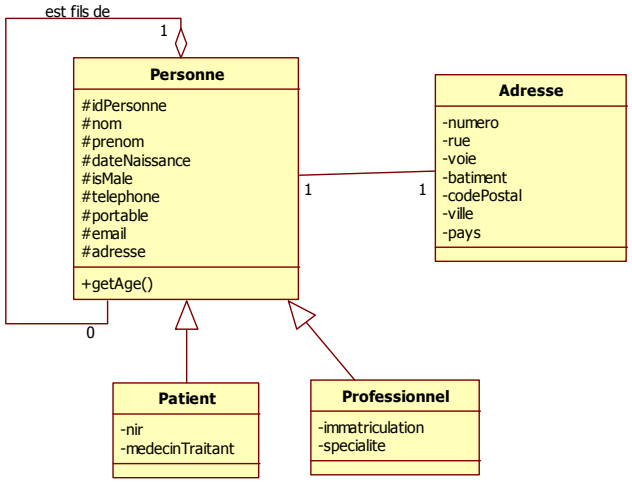
Le but de cet question est d'écrire une classe `ListePersonneCabinetTab` (voir annexe 2) qui permet de gérer une liste de personnes à l'aide d'un tableau statique.

Cette classe aura comme **attributs** :
- un **tableau** de 30 personnes que vous pouvez appeler **`liste`**
- un entier **`nbPersonnes`** permettant de mémoriser le nombre de personnes dans le tableau.

↳Compléter l'annexe 2. ***(aidez-vous éventuellement du cours transparent n° 26 `GroupeTD` et n°30 `ListeDeFormes`)***

↳Quelle remarque vous inspire cette classe ?

Annexe1 : Classe `Professionnel` « simplifiée » héritée de `Personne`



Sur ce diagramme, nous avons ajouté la classe **`Professionnel`** qui hérite de la classe **`Personne`** et contient 2 nouveaux **attributs** :

- le numéro d'**`immatriculation`** du professionnel mémorisé sous forme de **`String`**.
- la dénomination de la **`specialite`** du professionnel mémorisé sous forme de **`String`** (Exemple : généraliste, pédiatrie, gynécologie, gastro-entérologie…)

Cette classe devra respecter les **Règles d'écriture d'une classe** donnée dans le TD/TP précédent.

Cette classe `Professionnel` *n'est pas à écrire en TD (sauf si le temps le permet), mais elle devra être implémentée en TP … Pour la suite on supposera cette classe écrite !!!*

Attention pour faciliter l'implémentation de ce TD/TP, la classe `Professionnel` proposée ici a été simplifiée. Dans une « vraie » analyse, il faudrait considérer que les professionnels sont des médecins ayant une ou plusieurs spécialités médicales (généraliste, pédiatrie, gynécologie, gastro-entérologie…), avec zéro, un ou plusieurs domaines de compétences (médecine naturelle, homéopathie, ostéopathie …)

Annexe 2 : Classe `ListePersonneCabinetTab` à compléter …

```
public class ListePersonneCabinetTab
{
  // attributs



  // méthode permettant d'ajouter une personne maPers
  // dans le tableau
  public void ajouterPersonne (Personne maPers)
     {




     }

  // méthode permettant d'afficher toutes les personnes du tableau
  public void afficheListe()
     {




     }

  // méthode permettant d'afficher les patients présents ds le
tableau
  public void affichePatient()
     {




     }

  // méthode permettant d'afficher les professionnels présents
  // ds le tableau
  public void afficheProfessionnel()
     {




     }
}
```

Correction TD JAVA n°2: Héritage et Polymorphisme

Exercice 1 :

Rappel :

- **classe A :** classe mère
- **classe B :** classe fille ou sous-classe hérite de la classe A (mot cle extend)

si les variables de A sont **private**, il faut les déclarer en **protected** pour être accessibles depuis les sous-classes

```
public class A
      public getX() :  public double getX() // déclarer un type retourné , ou void
```

⇒ **Remarque : seul le constructeur n'a ni type retourné, ni void**

```
public class B extends A
{
      public B()
      {
        this.x = 1.0 ; // ⇒   x est declaré privé dans la classe A, on ne peut pas y accéder depuis B .
      }                       // Solution : déclarer x protected
                              // ou remplacer    this.x = 1.0     par     this.setX(1.0)
      public double getY()
      {
     return x * 2.0;  // Solution : déclarer x protected ou this.getX()*2.0
      }
}
```

Exercice 2 : Cabinet Médical

2.1 Héritage : classe `Patient`

↳ 2.1.a Définir une classe **`Patient`** en prenant comme classe de base la classe **`Personne`.**

Modifications à apporter à la classe ***`Personne`*** *:*
Passer tous les attributs ***`(liste des attributs)`*** *en* ***`protected`***
ou utiliser les getteurs comme ***`this.getIdPersonne()`***

Vous remarquerez qu'on a décidé volontairement de ne pas mettre `idPatient`.
On rajoutera ce champs quand on travaillera avec les Bases de Données : c'est à ce moment là qu'on en aura besoin…
Un patient a déjà un numéro de sécurité sociale (NIR) appelé dans le programme `numSecu`.
Est-il unique ? … et bien non…
Par exemple, pour les personnes de même sexe, nées à 100 ans d'intervalle, dans la même commune, dans le même numéro d'ordre

Le numéro de sécurité sociale n'est donc pas suffisant pour identifier de façon unique et certaine un individu, du fait de l'existence de doublons :
Et c'est bien parce que ce numéro n'est pas unique qu'il nous faudra rajouter un `idPatient`…
et même peut être pas besoin de cet idPatient si on ne crée qu'une seule table personne…on aura déjà ***`idPersonne`****…? ? ? ?*

A propos du type de `nir` : Le numéro de sécurité sociale sera mémorisée dans une variable de type `String`. En effet il peut y avoir des lettres pour la corse (A ou B) voir annexe.

A propos des constructeurs pourquoi ne pas avoir choisi un constructeur du type :
`public Patient (String nir, String medecinTraitant)` c-a-d avec seulement les 2 attributs de Patient ? car au TD/TP n°3 on souhaitera lever une exception si le patient n'a pas d'adresse d'où adresse est un argument significatif pour les constructeurs..


```
package com.iut.cabinet.metier;

import java.sql.Date;

public class Patient extends Personne{
      ////////////////////////////
      /// Nouveaux attributs
      private String nir;
      private String medecinTraitant;

      ////////////////////
      // Constructeurs
       ////////////////////
       public Patient()
        {     }

       // Constructeurs avec tous les arguments
       // arguments de la Classe Personne
       // et les 2 attributs de Patient (numSecu et medecinTraitant)
       public Patient (Integer idPersonne,String nom,String prenom,Date
dateNaissance,boolean isMale,String telephone, String portable, String email, Adresse
adresse,Personne unAscendant, String nir, String medecinTraitant)
       {
             super(idPersonne,nom,prenom,dateNaissance,isMale,telephone,
                   portable,email,adresse,unAscendant);
             this.nir=nir;
             this.medecinTraitant=medecinTraitant;
       }


       // constructeur avec 7 arguments significatifs AVEC ascendant
       // 6 arguments venant de la classe Personne :nom, prenom, date de Naissance,
       // isMale, adresse, ascendant et 1 argument de la classe Patient : nir
       public Patient (String nom,String prenom,Date dateNaissance,boolean isMale,
Adresse adresse,Personne unAscendant,String nir) {
             super (nom,prenom,dateNaissance,isMale,adresse,unAscendant);
             this.nir=nir;
// pour éviterla duplication du code de this.nir=nir
// on peut appeler avec this le constructeur écrit précédemment
// this(null,nom,prenom,dateNaissance,isMale,null,null,null,adresse,unAscendant,
nir,null);
          }


       // constructeur avec 6 arguments significatifs sans ascendant
      // 5 arguments venant de la classe Personne : nom, prenom, date de Naissance,
      // isMale, adresse et 1 argument de la classe Patient : nir
       public Patient (String nom,String prenom,Date dateNaissance,boolean isMale,
Adresse adresse, String nir) {
             super (nom,prenom,dateNaissance,isMale,adresse);
             this.nir=nir;
// pour éviter la duplication de code : appel du constructeur précédemment
// this(nom,prenom,dateNaissance,isMale,adresse,null,nir);
       }


       ////////////////////////////
       // Getteurs et Setteurs
       ////////////////////////////
       // numSecu
       public void setNir(String nir)
       {    this.nir=nir;      }

       public String getNir()
       {       return nir;     }
```

```
        ////////////
        // medecinTraitant
        public void setMedecinTraitant(String medecinTraitant)
        {       this.medecinTraitant = medecinTraitant;      }

        public String getMedecinTraitant()
        {       return medecinTraitant;     }

        ////////////////////////
        // Autre(s) méthode(s)
        ////////////////////////
        public String toString()  {
                 return super.toString() + "\n" +
                 "NIR: "+nir +"\n"+
                 "Medecin Traitant: "+medecinTraitant;
                 }
    }
```

↳ 2.1.b Redéfinir la méthode `equals` qui est déjà une méthode de la classe `Object`.
Notre choix de conception en ce qui concerne l'égalité de la classe `Patient` est le suivant :
Deux objets de la classe `Patient` seront considérés comme « égaux » si et seulement si leur `nom, prenom, nir` et `dateNaissance` sont identiques.

Justificatif du choix de conception :

- `nom` et `prenom` => cela paraît évident
- `dateNaissance` => car si on a un DUPONT Jean et un DUPONT Jean (junior), le `nir` serait dentique, de même que le `nom` et le `prénom`, il faut bien faire la différence sur l'âge (`dateNaissance`)…
- `nir` pour deux homonymes DUPONT Jean et DUPONT Jean né le même jour… (commune ou numéro de registres seront différents donc `nir` différents)
- pas de `id` dans la comparaison => car `id` est un *identifiant technique* : si on saisit une personne et si on veut vérifier si elle existe déjà dans le registre des `Patient`, il ne faut pas comparer son `id`, mais son `nom`, `prenom`, âge (`dateNaissance`) et `nir`…De même, il n'y a pas de `id` dans les constructeurs *à x arguments significatifs*…Le id n'est jamais dans le constructeur, mais se créé au moment où on enregistre dans le base de données.

Pour comparer structurellement 2 objets, il faut changer (on dit redéfinir la méthode equals())

```
@Override public boolean equals (Object o)
//@Override pour le TP
// => annotation qui demande au compilateur de vérifier que l'on redéfinie bien une
méthode.
        {
             if (!(o instanceof Patient))
// si on compare un Patient et un objet autre qu'un Patient
// il faut à tout prix faire ce test car on n'a pas encore vu les exceptions...
                  {return false;}
             else
                  {
                   Patient pat= (Patient) o;
// si o n'était pas un Patient cela déclencherait une ClassCastException...
// c'est pour cela qu'on a fait le test avant...
                   if ( this.nom.equals(pat.nom) &&
// pour pouvoir utiliser this.nom il faut passer ce champ en protected dans la classe
Personne !!!… si le champ reste private, il faut utiliser le getteur : pat.getNom()…
                  this.prenom.equals(pat.prenom) &&
                        this.dateNaissance.equals(pat.dateNaissance)
                              &&
                        this.nir.equals(pat.nir) )
                         return true;
                   else return false;
                  }
        }
```


Remarque : ATTENTION à bien noter l'utilisation d'*`equals`*et non de* == *!!!
Est-ce que les étudiants sont un peu malins ?
J'ai déjà écrit ce code pour la classe `Adresse` *(TP précédent), donc ils « doivent » l'avoir déjà vu… mais s'en souviendront-ils ?*

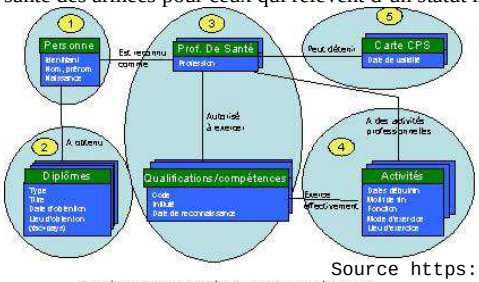
Modifications à apporter à la classe `Personne` :

Passer tous les attributs `(liste des attributs)` en `protected`

ou utiliser les *getteurs* comme

`this.getIdPersonne()` (cf le premier exercice)

Le changement d'identification des professionnels de santé
Jusqu'à la mise en fonctionnement du Répertoire Partagé des Professionnels de Santé (RPPS), les professionnels de santé (PS) continueront à être identifiés par un n° ADELI. A partir de la mise en fonctionnement du RPPS, tout nouvel inscrit à l'Ordre sera identifié par un n° RPPS.
Les professionnels en exercice se verront attribuer un numéro RPPS qui leur sera communiqué par l'Ordre.
Concrètement, c'est quoi? :
Le RPPS, c'est un répertoire contenant pour chaque professionnel de santé un identifiant unique et pérenne, le numéro RPPS. Ce numéro sera « non significatif » : on ne pourra en déduire aucune information d'âge, de localisation, etc. ;
un ensemble de données d'intérêt général, fiables et qualifiées, basées sur une nomenclature commune partagée par l'ensemble des partenaires du projet;
un système informatique permettant aux acteurs habilités d'échanger et de partager ces données.
Quels bénéfices en tireront les médecins?
Comme tous les professionnels de santé, les médecins recevront chacun un n° RPPS qui sera leur seul et unique identifiant tout au long de leur parcours professionnel. Ce numéro se substituera alors au n° Adeli qui disparaîtra. L'avantage, c'est qu'un professionnel n'aura plus à changer de numéro s'il s'installe dans une autre région, ce qui était le cas avec le n° Adeli. La bascule avec le nouveau système RPPS sera complètement transparente. Le RPPS permettra aussi de réduire et de simplifier un certain nombre de démarches administratives. Chargé d'enregistrer les informations sur les médecins, de les mettre à jour et de les communiquer aux autres administrations concernées dans le respect de la réglementation en vigueur, l'Ordre jouera le rôle de guichet unique pour les médecins.
À savoir : c'est l'État qui jouera le rôle de guichet unique pour les médecins fonctionnaires, et le Service de santé des armées pour ceux qui relèvent d'un statut militaire.



Représentation simplifiée du contenu du répertoire

Source https://editeurs.gip-cps.fr/index.php?page=RPPS_FAQ

2.3 Classe abstraite : voir cours !!!!

↳ Qu'est-ce qu'une classe abstraite ?

Une classe **abstraite** est une classe ***non instanciable***.
Une classe abstraite n'a pas forcément une méthode abstraite…

Alors que méthode abstraite => classe abstraite non instanciable
classa abstraite => pas forcément méthode abstraite…..

↳Quelle classe peut être transformée en classe abstraite. Effectuer la transformation…

La classe **`Personne`** peut être modifiée en **classe abstraite** car on instanciera directement des `Patient` ou des `Professionnel`, mais jamais directement des personnes….
La classe `Personne` devient la classe de référence. Elle ne sert « jamais » puisqu'on instanciera des `Patient` ou des `Professionnels`.
La classe `Patient` (`Professionnel`) sont les seules classes ***fonctionnelles***(***concrètes***)
La classe `Personne` est alors la classe de ***référence*** (***abstraite***)

Ce qui change par rapport au TD/TP 1 :

```
public abstract class Personne{   // classe abstraite sans méthode abstraite
                                  // mais vu comme une classe de référence
…
      protected Integer idPersonne;      // pour pouvoir être utilisé dans
                                         // les classes filles
…
}
```

2.4 Gestion d'une liste de personnes avec un tableau statique : classe `ListePersonneCabinetTab`

⇒ **IIlustration du polymorphisme :** ***Manipulation uniforme des objets de plusieurs classes sans en connaître le type par l 'intermédiaire d'une classe de base commune***

Une `Personne` est polymorphe : elle peut se décliner en deux sous-classes : `Patient` et `Professionnel` qui possèdent leur propre méthode d'**affichage (`toString`)**. Lors de l'affichage d'un véhicule, la JVM se « débrouille » pour retrouver la bonne méthode à appeler.

⇒ **class** `ListePersonneCabinetTab` **:** ***un exemple similaire a été donné en cours (Transparent n°26 : GroupeTD) Le rappeler ! ! ! ! ! !***

```
package com.iut.cabinet.essai;

import com.iut.cabinet.metier.Personne;
import com.iut.cabinet.metier.Patient;
import com.iut.cabinet.metier.Professionnel;

public class ListePersonneCabinetTab {
      //attributs
      Personne[] liste = new Personne[30];
      int nbPersonnes = 0;

      // méthode permettant d'ajouter une personne maPers dans le tableau
      public void ajouterPersonne (Personne maPers)
            {
            if (nbPersonnes < liste.length)
                  liste[nbPersonnes++] = maPers;
            }
```



```
      // méthode permettant d'afficher toutes les personnes du tableau
      public void afficheListe()
      {
            for (int i=0; i<nbPersonnes;i++)
              {
              System.out.println(liste[i]);
              System.out.println("----------------------------------------");
              }
      }

      // méthode permettant d'afficher les patients présents ds le tableau
      public void affichePatient()
       {
          for (int i=0; i<nbPersonnes;i++)
            {
             if (liste[i] instanceof Patient)
             {
             System.out.println(liste[i]);
              System.out.println("----------------------------------------");
             }
            }
        }

//     méthode permettant d'afficher les professionnels présents ds le tableau
      public void afficheProfessionnel()
        {
        for (int i=0; i<nbPersonnes;i++)
            {
            if (liste[i] instanceof Professionnel)
             {
            System.out.println(liste[i]);
             System.out.println("----------------------------------------");
             }
            }
        }

}//fin classe
```

⇒ **Remarque : Pas de constructeur ….**

↳**Question :** ***Comment instancier dans une classe Test un objet*** **`maListe`** ***de classe*** **`ListePersonneCabinetTab`** ***stockant des*** **`Personne`** ***?***

```
        // Instanciation de la Liste de Personne
      ListePersonneCabinetTab maListe = new ListePersonneCabinetTab();
```

⇒ **Remarque : Application utilisant la classe** `ListePersonneCabinetTab` **(demandée en TP)**

```
package com.iut.cabinet.essai;

import java.sql.Date;

import com.iut.cabinet.metier.Adresse;
import com.iut.cabinet.metier.Patient;
import com.iut.cabinet.metier.Professionnel;
import com.iut.cabinet.util.DateUtil;

public class EssaiCabMed_v22 {
```

```
      public static void main(String args[])
      {
    // ici on se contente d'appeler le constructeur.
    new EssaiCabMed_v22 ();
      }

      EssaiCabMed_v22 ()
      {
        // Instanciation de la Liste de Personne
            ListePersonneCabinetTab maListe = new ListePersonneCabinetTab();
        // Rq : pas de constructeur dans la classe ListePersonneCabinetTab
        // Le constructeur appelé est le constructeur par défaut

      System.out.println(" \n --------------------------------");
      System.out.println(" \n ------ TEST des PATIENTS ---------");
      System.out.println(" \n --------------------------------");

      // Instanciation de 2 patients
      Patient patient1=new Patient(1,"DUPONT","Julie",
                  DateUtil.toDate("21/05/1960",DateUtil.FRENCH_DEFAUT),
                  false,"0555434355","0606060606","julie.dupont@tralala.fr",
       new Adresse("15","avenue Jean Jaurès",null,null,"87000","Limoges","France"),
                    null,
                    "260058700112367","MARTIN Paul"); // les 2 attributs de patient

      Patient patient2=new Patient(2,"DUPONT","Toto",
      DateUtil.toDate("25/12/1991",DateUtil.FRENCH_DEFAUT),
                  true,"0555434355","0605040302","toto.dupont@etu.unilim.fr",
new Adresse("185","avenue Albert Thomas",null,"Résidence La Borie","87065","Limoges","France"),
                  patient1,
                  "260058700112367","MARTIN Paul"); // les 2 attributs de patient ....

      // Ajout des patients dans la liste...
      maListe.ajouterPersonne(patient1);
      maListe.ajouterPersonne(patient2);

      // Affichage des caractéristiques de TOUTES les personnes
      System.out.println(" \n ---- TOUTE LA LISTE----");
      maListe.afficheListe();

      System.out.println(" \n ---- LES PATIENTS----");
      maListe.affichePatient();
      System.out.println();

      System.out.println(" \n --------------------------------");
      System.out.println(" \n ------ TEST des PROFESSIONNELS -------");
      System.out.println(" \n --------------------------------");

       // Instanciation de ??? professionnels
      Professionnel prof1= new Professionnel();
      prof1.setNom("MARTIN");

      // Ajout des patients dans la liste...
      maListe.ajouterPersonne(prof1);
      // Affichage des caractéristiques de TOUTES LES PERSONNES
      System.out.println(" \n ---- TOUTE LA LISTE----");
      maListe.afficheListe();
      System.out.println(" \n ---- LES PATIENTS----");
      maListe.affichePatient();

      System.out.println(" \n ---- LES PROFESSIONNELS----");
      maListe.afficheProfessionnel();

      System.out.println();
      }
}
```