M01 : Programmation avancée

Jean-François Berdjugin

M01 : Programmation avancée

Jean-François Berdjugin

Publication date 2011

Table of Contents

List of Figures

List of Examples

Chapter 1. Présentation

1. Présentation des séances

Nous avons six séances de trois heures, pour découvrir avec *Java* les concepts de programmation orientée objet.

Nous commencerons par découvrir le langage Java dans une approche non objet et l'environnement de développement *eclipse* (Integrated Development Environment).

Puis, nous introduirons la notion de classe et d'objet, nous verrons comment utiliser l'API (Application Programming Interface) Java SE pour créer nos propres objets. Nous créerons nos propres classes et nos propres objets et nous verrons comment les associer.

La partie suivante traite de l'héritage, un moyen de factoriser le code et des concepts associés : le classes abstraites, les interfaces et le polymorphisme.

Les chapitres qui suivent reposent sur la notion d'objets, nous avons les exceptions qui permettent de gérer des situations exceptionnelles comme les erreurs, la généricité qui permet au classe de recevoir de façon dynamique des types et enfin les annotations java qui permettent de fournir au compilateur des informations supplémentaires.

Commencer par créer sur votre disque dur dans votre espace personnel un répertoire nommé Workspace puis lancer eclipse ; lorsque ce dernier vous invitera à choisir un workspace, prenez le répertoire créé.

Chapter 2. Types primitifs, instructions, tableaux et sous-programmes

Pour développer en java, il faut un compilateur java et un ensemble de bibliothèques, nous utiliserons Java SE (Standard environnement) avec le JDK (Java Developpement Kit), celui-ci est déjà présent sur votre ordinateur.

Le JDK contient un ensemble d'outils parmi lesquels nous trouvons :

javac
: le compilateur java,

jar
: l'archiveur, qui met sous forme d'un paquetage les bibliothèques de classes relatives au projet fusionné en un fichier *jar*,

javadoc
: le générateur de documentation, qui génère automatiquement de la documentation à partir des *annotations* du code source,

jdb
: le d'éboueur,

je
: un ensemble d'outils permettant l'exécution de programmes Java sur toutes les plates-formes supportées et qui contient la machine virtuelle java.

Le langage java peut-être utilisé dans le domaine du Web pour exécuter du code sur le client :

appelet
: du code inclus dans une page Web

Java Web Star (JUS)
: un moyen de déploiement d'applications via le Web.

Le langage java peut aussi être utilisé pour s'exécuter sur les serveurs (Web dynamique) :

EJB
: Enterprise Java Bean composant Java, réutilisable, destiné à être déployé sur des serveurs d'applications.

JSP
: Java Server Page, technologie Java de création de pages dynamique en XHTML

Servlet
: Le coeur, le programme java qui s'exécute sur le serveur.

Le langage Java est aussi présent sur les systèmes embarqués, java ME (Micro Edition) et sur les systèmes mobiles. Par contre java n'est pas javascript (JS) ou javaFX.

1. Le premier programme (Hello World)

Nous allons créer dans notre workspace un premier projet nommé, *seance1*. Le projet et l'unité de développement, il peut posséder un compilateur spécifique, une description de la localisation des bibliothèque (le CLASSPATH) spécifique, ...

Dans le projet seance1, créer un paquetage tests. Le paquetage permet de définir un espace de nommage, un paquetage peut contenir d'autre paquetages.

Dans le paquetage test, créer une classe nommée Hello qui possède un main (public static void main(String[] args)). Vous devez avoir ceci :

```
package tests;

public class Hello {

 /**
  * @param args
  */
 public static void main(String[] args) {
  // TODO Auto-generated method stub

 }

}
```

Vous observerez le mots clef *package* qui indique le paquetage de la classe Hello, celle-ci contient le programme principal, celui par qui commence l'exécution : le *main*.

Transformer le code de votre classe comme suit :

```
package tests;

/**
 * Seance1 Exo1
 * @author jub
 * Une classe dont le programme principal affiche Hello World
 */
public class Hello {

 /**
  * Le programme principal
  * @param args
  */
 public static void main(String[] args) {
  //System.out.println nous permet d'afficher sur la sortie standard
  System.out.println("Hello World");

 }

}
```

Compiler et exécuter votre code avec "run as -> java application". Cette opération a produit en .class du bytecode en utilisant le compilateur (javac) et ce bytecode a été exécuté par la machine virtuelle (java).

2. Les commentaires et la javadoc

Les commentaires peuvent être en ligne // ou en bloc /* */. Tout ce qui suit le signe // ou qui est dans le bloc /* */ est considéré comme du commentaire c'est à dire que le compilateur n'en tient pas compte.

Les commentaires seuls sont principalement destinés aux concepteurs de classe, il existe un outil supplémentaire qui permet de la documentation pour les utilisateur : la *javadoc*. La javadoc utilise des annotations (@) pour générer des fichiers HTML. Dans projet choisir "generate javadoc" en indiquant l'emplacement de javadoc.exe (`C:\Program Files (x86)\Java\jdk1.6.0_21\bin\javadoc.exe`).

3. Les types primitifs

Java est fortement typé, toute variable doit avoir reçu un type à la déclaration. Les types primitifs disponibles sont : byte, short, int, long, float, double, char et boolean.

Des variables peuvent être déclarée n'importe où, leur durée de vie est celle des accolades qui les entourent, la syntaxe est :

```
//type nom_variable;
```

```
// int i;
//type nom_variable1, nom_variable_2;
//char c1, c2.
```

4. Les instructions

Comme tout langage, java possède son jeux d'instruction, nous allons parcourir les principales.

4.1. L'affectation

L'affectation utilise le symbole =, la comparaison repose sur ==.

Si une variable est déclarée mais ne reçoit pas de valeur, un Waring est généré par java.

4.2. Les conditionnelles

La comparaison se fait de la manière suivante :

```
if (testBooléen) // testBooléen est un booléen
                 //ou un test dont le résultat est une valeur booléenne
traitement à effectuer si le test est vérifié
[else traitement à effectuer si le test n'est pas vérifié]
```

Ce qui est entre [ ] est optionnel. On l'utilise si on en a besoin. Exemple :

```
if (maxi > 5) // ici testBooléen est un test
              //dont le résultat est true si maxi est supérieur à 5, false sinon.
System.out.println("maxi est plus grand que 5");
else
System.out.println("maxi est plus petit ou égal que 5");
```

Je vous conseil d'utiliser systématiquement les accolades pour définir un bloc: if (cond) {...} else {...}

Lorsque l'on possède plusieurs cas exclusifs sur une valeur entière ou une énumération l'utilisation du switch est possible:

```
witch (expr) {
  case c1:
        //instructions pour expr==c1
        break;
  case c2:
        //instructions pour expr==c2
        break;
  case c2:
  case c3:
  case c4:
        //instructions pour expr==c2 ou expr==c3 ou expr==c4
        break;
  . . .
  default:
        //instructions par défaut
}
```

“Réaliser une classe *Exo1.java* qui contient un programme principal qui permet à partir de deux entiers lus au clavier d'afficher le maximum.”

Le morceau de code suivant permet de lire deux entiers au clavier :

```
Scanner scanner = new Scanner(System.in);
  a = scanner.nextInt();
  b = scanner.nextInt();
```

Tip

CTRL+**ESPACE**permet d'obtenir la complétion.

Tip

SHIFT+**F2**permet d'accéder à la javadoc.

Tip

syso+**CTRL**+**ESPACE** permet d'obtenir System.out.println().

4.3. Les boucles

En java les structures de boucle sont le *while* et le *for*. Le *while* s'utilise de deux manières différentes :

```
while (condition) {
  instructions
}

ou

do{
  instructions
}
while (condition);
```

La boucle est exécutée tant que la condition est vérifiée. Dans la boucle do { } while au moins une itération a lieu.

La boucle *for* et une écriture condensée du *while*.

```
for (init;cond;post-traitement)
{
  traitement
}

équivaut à

init;
while (cond)
{
  traitement
  post-traitement
}
```

L'utilisation la plus courante est la suivante :

```
for (int i = 0; i < limite; i++)
{
    // instructions à exécuter
}
```

Dans cet exemple, `i` prendra les valeurs 0, 1, .., limite-1

Tip

Il n'est pas nécessaire d'incrémenter la variable de boucle (`i`) dans la boucle

Si l'initialisation ou le post-traitement contiennent plusieurs instructions, il faut les séparer par des virgules.

Il existe aussi une autre syntaxe de la boucle for pour parcourir des collections :

```
 for (NomClass element : Collection
{
 // instructions à exécuter
}
```

“Réaliser une classe *Exo2.java* qui contient un programme principal qui permet à partir d'un nombre *n* saisi au clavier, de calculer le maximum de *n* entiers ensuite saisis au clavier. Par exmple si 4, 2, 8, 3, 5 sont saisis, 8 doit être affiché.”

4.4. Les tableaux

Les tableaux ne sont pas encore les objets mais ne sont déjà plus des types primitifs. Ils constituent la première forme de collection, un moyen de regroupé sous un ou plusieurs noms, un ensemble de valeurs.

Les tableaux JAVA ™ correspondent à la représentation intuitive que vous vous faite d'un tableau ou d'un vecteur. Les éléments cases sont accessibles via un indice, dont la numérotation commence à 0. La première case d'un tableau est numérotée 0 est la dernière la longueur du tableau moins 1.

Nous allons apprendre comment déclarer un tableau, puis l'instancier et enfin le manipuler.

4.4.1. Déclaration

Pour déclarer un tableau les crochets*[] doivent être utilisés.*

typeTab[] tab; déclare *tab* comme étant une variable référencent un tableau de type *typeTab*. Tous les éléments de notre tableau sont de même type.

```
typeTab[] tab; //declaration de tab comme étant un tableau
               //de type typeTab
```

Figure 2.1. Déclaration d'un tableau

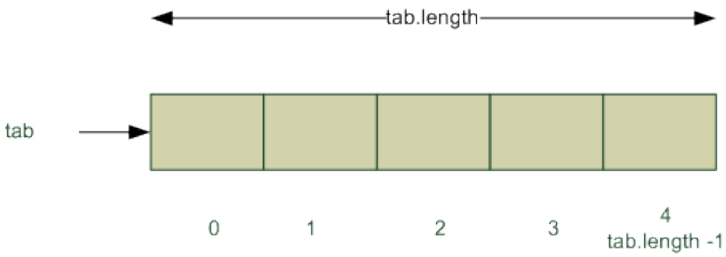
4.4.2. Instanciation (ou construction)

Après la déclaration notre tableau n'existe pas, nous ne disposons que de la référence (le nom), pour qu'il existe, il faut l'instancier (construire). Les tableaux ont une taille fixe (structure de donnée statique), celle taille doit être fixée lors de la construction. Le mot clef permettant la construction est new.

tab = new typeTab[taille]; permet de créé le tableau de type *typeTab*, de taille *taille* est accessible via la référence *tab*.

```
typeTab[] tab;

tab = new typeTab[taille]; //construction d'un tableau de taille : taille
                           // la première case est numérotée 0
                           // la dernière case est numérotée taille - 1
```

Figure 2.2. Construction d'un tableau



4.4.3. Accès aux données

Sous un même nom la référence, nous avons un ensemble de données accessible en utilisant un indice.

4.4.3.1. Lecture/Ecriture

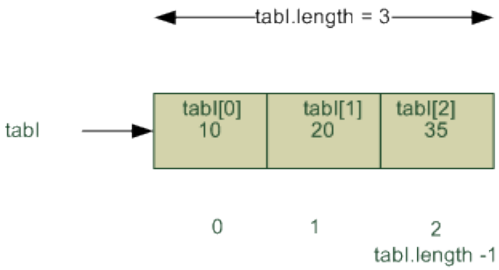
La lecture et l'écriture sont réalisées en spécifiant entre crochet (*[]*) la position de la "case".

tab[i] permet d'accéder à la position *i* du tableau référence par *tab*.

Voici un exemple d'écriture suivi d'un exemple de lecture sur un tableau d'entiers:

```
int[] tabI;
tabI = new int[3];

tabI[0]=10; //ecriture de la valeur 10 à la position 0
tabI[1]=20; //ecriture de la valeur 20 à la position 1
tabI[2]=35; //ecriture de la valeur 35 à la position 2

int i = tabI[0] + tabI[1] + tabI[2]; //i recoit la somme des elements du tableau
```

Figure 2.3. Accès aux éléments d'un tableau



4.4.3.2. Propriétés

Nous allons utiliser comme propriété du tableau sa longueur : *length*.

Nous le verrons plus tard mais l'accès au propriétés d'un objet se fait en utilisant l'opérateur . (point). Ainsi la longueur du tableau *tab* est accessible en utilisant *tab.length*. Le code suivant permet d'afficher la taille du tableau *tabI* ainsi que le dernier et le premier élément.

```
int[] tabI;
tabI = new int[3];
tabI[0]=10;
tabI[1]=20;
tabI[2]=35;

System.out.println(tabI.length);          //affiche la longueur du tableau ici 3
System.out.println(tabI[tabI.length - 1]) //affiche la valeur de la derniere case du tableau ici 35
System.out.println(tabI[0])               //affiche la valeur de la première case du tableau ici 10
```

4.4.3.3. Pièges

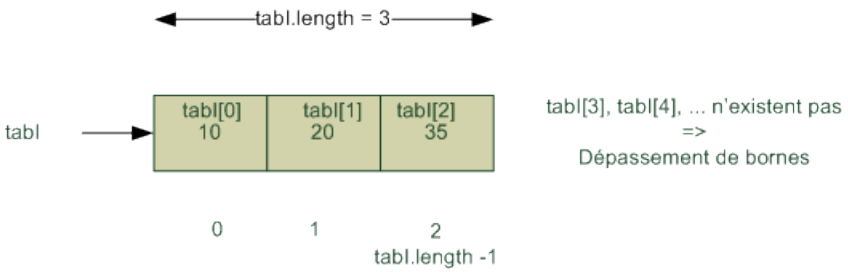
Les tableaux contiennent deux pièges intrinsèques : le dépassement des bornes et les alias involontaires.

4.4.3.3.1. Dépassement des bornes

Les tableaux sont des structures statiques dont la taille est choisie à la création, il est impossible d'accéder à un élément à l'extérieur des bornes. Si notre tableau *tabI* a pour taille 3 : t[-1] et t[3] conduisent par exemple au message *ArrayIndexOutOfBoundsException* (Erreur de dépassement des bornes par l'indice du tableau).

Vous rencontrez probablement se problème un jour ou l'autre aussi n'en oubliez pas la raison.

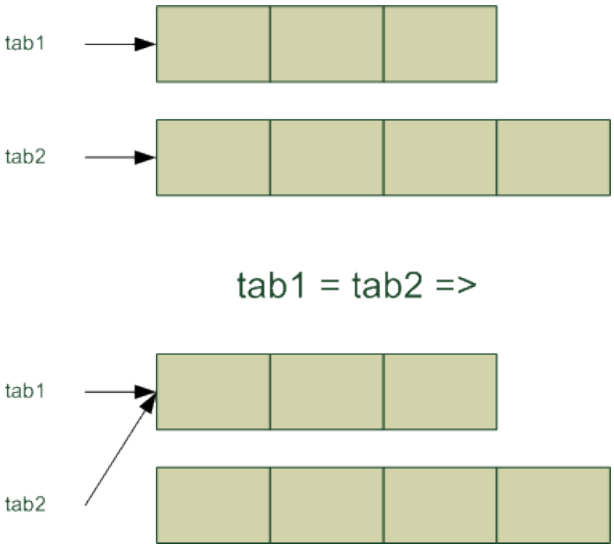
Figure 2.4. Dépassement des bornes



4.4.3.3.2. Alias

Nous accédons aux tableaux via une référence aussi si *tab1* et *tab2* sont deux références de tableaux : *tab1* = *tab2* ne recopie pas le tableau référencé par *tab2* dans celui référencé par *tab1* mais recopie la référence de *tab2* dans *tab1*. *tab1* devient un alias sur *tab2*, ils référencent le même tableau et toute modification par l'un affecte en conséquence l'autre. Nous avons deux noms (*tab1*, *tab2*) pour une même chose (un même tableau).

Figure 2.5. Alias



4.4.4. Exemples

Example 2.1. Affichage de voyelles

```
public class Exemple1{
  public static void main(String[] args){
    char[] voyelle; //declaration de voyelle comme étant une référence sur un tableau de caractères
    voyelle=new char[6]; //construction d'un tableau de 6 caractères référencé par voyelle
    voyelle[0]='a';
    voyelle[1]='e';
    voyelle[2]='i';
    voyelle[3]='o';
    voyelle[4]='u';
    voyelle[5]='y';
    for (int i=0; i < voyelle.length; i++){
       System.out.println(voyelle[i]);
    }
  }
}
```

Cet exemple permet de créer et d'afficher un tableau de caractères contenant des voyelles.

4.4.5. Exercice

“Ecrire une classe Exo3.java qui contient un programme principal qui permet de demande le nombre de note puis les notes et qui enfin, affiche le nombre de notes supérieures à la moyenne des notes saisies.”

5. Les sous-programmes

La notion de sous-programme en java coorespond à la notion de méthodes de classe. Le passage des paramètres et réalisé par recopie, pour les tableaux et les objets, c'est la référence qui est recopiée.

L'élément de structuration en java, comme dans les autres langages orientés *objet* est la *class*. Les classes sont composée d'*attributs* et de *méthodes*. Java est fourni avec une API (Application Programming Interface) qui contient un ensemble de classes.

5.1. Méthode statiques

Pour continuer à prendre des habitudes de programmation, nous allons séparer notre code en deux parties : les méthodes et leur test.

Vous allez créer un paquetage nommé *mesClasses*, puis dans ce paquetage une classe nommé Tp1, sans main.

La classe Tp1 qui contiend la méthode static static int prixPizza(int a) dont le code est le suivant :

```
public class Tp1
{
//calcul le prix de nb pizza
//sachant que la pizza vaut 10€ et que la dixième est gratuite
//
 public static int prixPizza(int nb)
 {
   final int prixU=10;
   return nb*prixU-((nb/10)*prixU);
 }
}
```

Le programme que nous avons ne peut-être lancé, il ne contient qu'une méthode qui n'est pas un *main*. Nous allons créer une *class TestTp1* dans code/test dont le code est le suivant :

```
public class TestTp1
{
 public static void main(String[] args)
 {
  System.out.println("10 pizzas => " + Tp1.prixPizza(10));
  System.out.println("20 pizzas => " + Tp1.prixPizza(20));
 }
}
```

Par défaut, les classes sont recherchée dans le paquetage courant, ici vous devez avoir recour à `import mesClasses.Tp1;` pour pouvoir avoir accès à la classe Tp1. Sans l'import vous auriez du spécifier le nom complet de la classe : `mesClasses.Tp1`.

Chapter 3. Classes et objets

1. Cours

Si vous maintrisez la notion de classe et la notion d'objet, vous pouvez passer la partie cours et commencer directement les exer

1.1. Définitions

Nous ne prétendons pas ici refaire la théorie des langages à objets. Cependant nous allons préciser certaines notions. Une classe est une description statique d'une famille d'objets ayant même structure et même comportement. Les deux aspects sont donc :

- la donnée d'une composante structurelle (non dynamique) : *variables d'instances* (ou champs, ou attributs, ou propriétés) qui caractérisent l'état d'un objet pendant l'exécution d'un programme.
- la donnée d'une composante dynamique : procédures ou fonctions appelées *méthodes*. Les méthodes manipulent les variables d'instances.

La classe est donc un plan de construction, permettant d'obtenir des objets tous semblables (à l'instar d'un véhicule automobile d'une marque et d'un modèle particulier) mais tous différents (chaque véhicule possède sa couleur, son immatriculation,... le fait de repeindre un véhicule ne modifie pas la couleur des autres).

Un objet est une entité indépendante (dont la structure est connue de lui seul). Pour agir sur un objet, il faut utiliser les méthodes offertes par celui-ci. Cette utilisation passe par l'envoi d'un message (qui peut être vu comme une requête) à l'objet.

Dans la suite, nous allons définir et utiliser des objets .

1.2. Exemple

Nous allons d'abord travailler avec la classe Point, nous allons apprendre à l'utiliser puis à la créer.

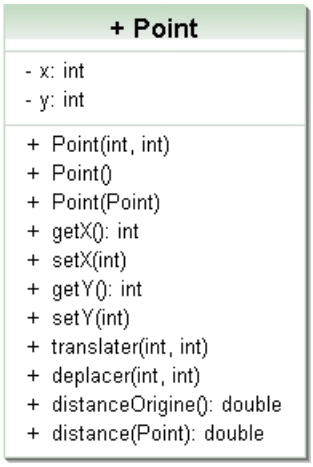
1.2.1. Point

Les points sont pour nous des points en deux dimensions caractérisés par leur abscisse et leur ordonnée. Bien que cela soit en dehors du domaine de ce cours nous illustrerons nos propos avec des diagramme UML. Vous reverrez par la suite de votre scolarité ces diagrammes.

1.2.1.1. Représentation UML

La notation UML (Unified Modeling Language) et un moyen de décrire graphiquement des données et des traitements. Cette notation est composé d'un ensemble de diagramme, nous allons utiliser l'un d'entre eux qui permet de décrire une classe : le diagramme de classe

Figure 3.1. Diagramme de classe de Point



Sur ce diagramme nous voyons que notre classe se compose :

- un nom : Point
- de variables d'instance (ou propriétés, ou attributs) : *x* et *y*. L'état d'un point est défini par *x* et *y* qui sont deux entiers.
- d'un ensemble de méthodes qui vont définir le comportement d'un point :
  - Trois constructeurs qui vont permettre de construire des points. Ce sont les constructeurs qui sont invoqués lors de l'utilisation du mot clef *new*. Les constructeurs portent le noms de la classe et ne renvoient rien. Ici nous avons :
    - Point() qui construit un point en cordonnées (0,0), on parle de constructeur sans paramètre.
    - Point(int, int) qui construit un point a une abscisse et une ordonnée choisie.
    - Point(Point) qui fabrique un point à partit d'un autre point, on parle de constructeur par recopie.
  - Des méthodes qui vont permettre de se renseigner sur l'état et de modifier l'état de l'objet :
    - Les accesseurs (getters) qui permettent de connaître l'état :
      - getX():int qui retourne l'abscisse du point
      - getY():int qui retourne l'ordonnée du point
      - distanceOrigine():double et distance(Point):double qui donne la distance à l'origine et la distance à un autre point.
    - Les manipulateurs (setters) qui permettent de modifier l'état de l'objet
      - setX(int) qui modifie l'abscisse
      - setY(int) qui modifie l'ordonnée
      - deplacer(int,int) et translater(int,int) qui modifient les coordonnées.

Nous avons une description de la classe Point nous en ferons le codage plus tard mais nous allons voir maintenant comment créer des objets points et les utiliser.

1.2.1.2. Utilisation

Les classes sont des types, pour déclarer une référence sur un objet nous utiliserons la syntaxe suivante :

```
NomClasse nomObjet;
```

.

Ce qui nous créé une référence (nomObjet) vers un objet de type NomClasse. Pour créer ou construire ou instancier l'objet, il faut faire appel à un constructeur en utilisant le mot clef *new*. :

```
nomObjet = new NomClasse([parametre])
```

Le constructeur défini l'état initial de l'objet, les variables d'instance sont crées et ont reçu une valeur, pour que l'objet évolue, ait un comportement, il faut utiliser des méthodes. L'accès aux variables d'instance et aux méthodes publiques de l'objet se fait en utilisant la notation . :

```
nomObjet.variablesInsatnce
nomObjet.nomMethode([parametre])
```

Nous pouvons maintenant appliquer ces concepts à l'utilisation de la classe Point.

Figure 3.2. Teste d'un point

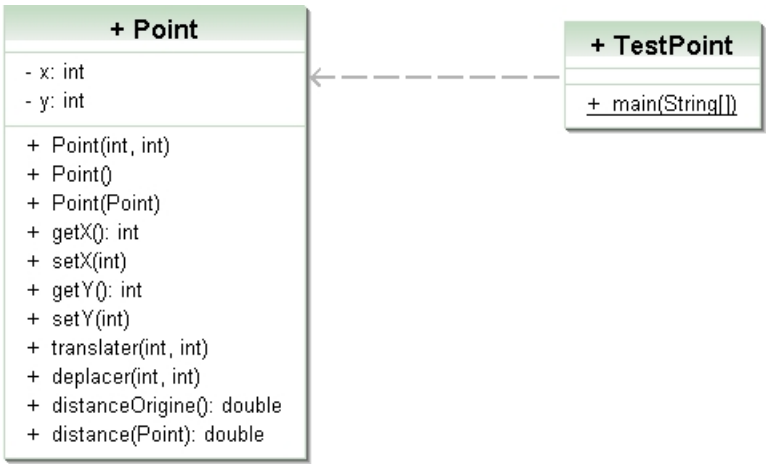
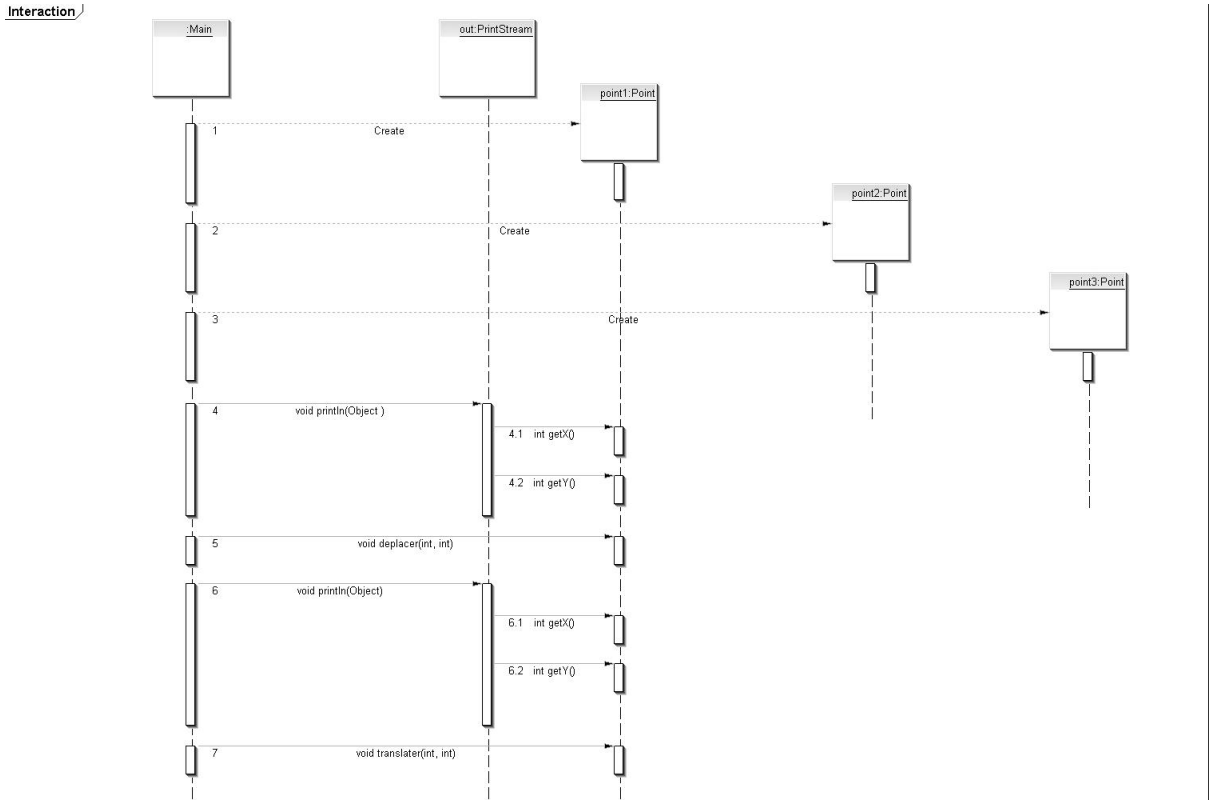


Figure 3.3. Diagramme des séquences



Example 3.1. Utilisation de la classe Point

```
public class TestPoint {

 public static void main(String[] args) {

  Point point1;              //point1 est une reference sur un objet de type Point
  Point point2;              //point2 est une reference sur un objet de type Point
  Point point3;              //point3 est une reference sur un objet de type Point

  point1 = new Point();      //le constructeur sans parametre qui créé un point en (0,0) est appelé.
                             //le point1 reference le Point cree on parle de d'instanciation
  point2 = new Point(1,2);   //le constructeur prenant en parametre deux entiers en utilisé
                             //le point est en (1,2)
  point3 = new Point(point1);//le constructeur par recopie est utilisé, point3 est un clone de point2
                             //le point est en (0,0)

  System.out.println(point1.getX()+" " +point1.getY());
                             //getX() renvoie le x, getY() renvoie le y de Point
                             //0 0
  point1.deplacer(10, -5);   //point1 va en (1O,-5)
  System.out.println(point1.getX()+" " +point1.getY());
                             //10 -5
  point1.translater(2, 5);   // le x de point1 est augemente de 2
                             // le y de point1 est augemente de 5
  System.out.println(point1.getX()+" " +point1.getY());
                             //12 0
  System.out.println(point1.distanceOrigine());
                             //12.0
  System.out.println(point1.distance(point2));
                             //11.180339887498949
  point2.setY(10);           // le y de point2 prend 10
  System.out.println(point2.getY());
                             //10
  System.out.println(point2.getX()+" " +point2.getY());
                             //1 10
  System.out.println(point3.getX()+" " +point3.getY());
                             //0 0
  point3 = point2;           //point3 recoit la reference point2
                             //point3 est un alias sur point2
  point3.deplacer(5, 5);     //point3 est déplacé en (5,5)
  System.out.println(point2.getX()+" " +point2.getY());
                             //5 5
  System.out.println(point3.getX()+" " +point3.getY());
                             //5 5
 }
}
```

Bien évidement avant d'utiliser une classe, il faut l'avoir créée.

1.2.1.3. Codage

Nous allons maintenant étudier le codage de la classe Point, à vous de lire le code suivant et de poser des questions à votre enseignant(e) :

```
/**
 *
 */
package td;

/**
 * @author jub
 *Classe d'un point 2D
 */

public class Point {

/**
 * x: abscisse
```

```
 * y: ordonnée
 */

private int x,y;

/**
 * Permet de créer un point de coordonnées (x,y)
 * @param x l'abscisse
 * @param y l'ordonnée
 */
public Point(int x, int y) {
 this.x = x;  //this.x permet de manipuler la variable d'instance x
 this.y = y;  //this.y permet de manipuler la variable d'instance y
}

/**
 * Permet de créer un point de coordonnées (0,0)
 */
public Point() {
 this(0,0);  //this(0,0) permet d'appeler le constructeur qui prend en parametre deux entiers
}

/**
 * Permet de créer un point de mêmes coordonnées qu'un autre point
 * @param point
 */
public Point(Point point) {
//dans la classe l'appel à getX() et getY() peut-être remplacé par x et y.
//les attributs privé sont visible dans la classe
 this(point.getX(),point.getY());
}

/**
 * Renvoie l'abscisse
 * @return the x
 */
public int getX() {
 return x;
}

/**
 * Définit l'abscisse
 * @param x the x to set
 */
public void setX(int x) {
 this.x = x;
}

/**
 * Renvoie l'ordonnée
 * @return the y
 */
public int getY() {
 return y;
}

/**
 * Définie l'ordonnée
 * @param y the y to set
 */
public void setY(int y) {
 this.y = y;
}



/**
 * translate le point courant de dx et dy (i.e les nouvelles coordonnées de mon Point sont (x+dx,y+dy) )
 * @param dx un int
 * @param dy un int
```

```
 */
public void translater(int dx,
                       int dy)
{
 this.setX(this.getX()+dx);
 this.setY(this.getY()+dy);
}


/**
 * déplace le point courant en (newX,newY)(i.e les nouvelles coordonnées de mon Point sont (newX,newY) )
 * @param x un int
 * @param y un int
 */
public void deplacer(int x,
                     int y)

{
 this.setX(x);
 this.setY(y);
}


/**
 * renvoie la distance à l'origine du Point courant
 * @return distance à l'origine
 */
public double distanceOrigine()
{
 return this.distance(new Point(0,0));
}

/**
 * Renvoie la distance entre deux points
 * @param autrePt
 * @return distance
 */
public double distance(Point autrePt)
{
 double dx,dy;
 dx=this.getX()-autrePt.getX();
 dy=this.getY()-autrePt.getY();

 double res;
 res= Math.sqrt(Math.pow(dx, 2)+ Math.pow(dy,2));
 return res;
}


/* (non-Javadoc)
 * @see java.lang.Object#toString()
 */
public String toString()
{
 String s="("+this.getX()+","+this.getY()+")";
 return s;
}


}
```

2. Exercices

2.1. Mise en place de l'espace de travail

Vous allez lancer eclipse et suivre la démarche suivante :

1. Importer dans le workspace le projet existant (File ->Import-> General -> import existing projets into workspace -> select archive file) : `sceance2` le projet est contenu dans l'archive : `seance2.zip`.

2. Dans le projet `seance2` vous trouverez les paquetages suivants : `tp.etudiant.classes` et `tp.etudiant.tests`, ce sont les paquetages avec lesquels vous travaillerez. Pour le moment, ils contiennent les classes pour la récurrence, vous allez bientôt les faire grandir.

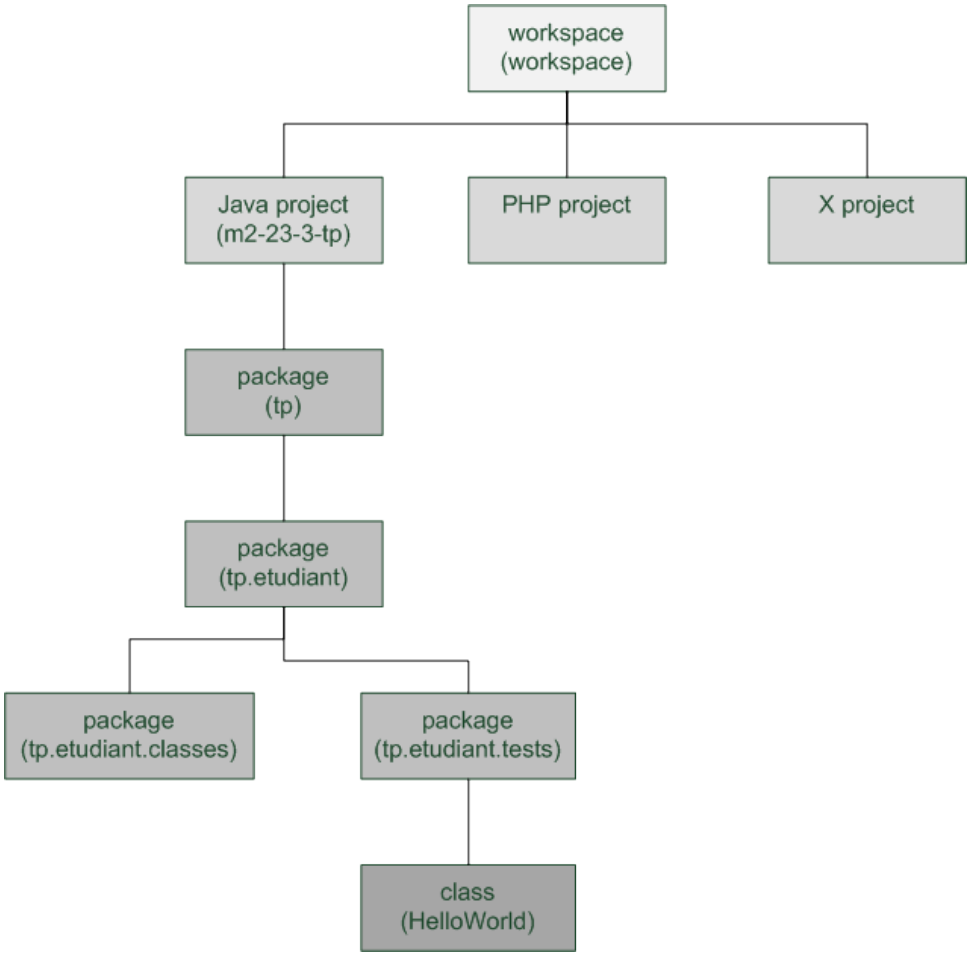
3. Le projet contient aussi une librairie externe (`lib`) contentant le paquetage `tp.prof.classes`, ce paquetage contiennent uniquement les `.class` des classes que vous aurez à tester (Point et Vehicule).

4. Pour finir le projet contient la javadoc des classes du paquetage `tp.prof.classes`. Une javadoc est la documentation d'une API (Application Programming Interface) en HTML obtenue grâce à des commentaires du code. Vous aurez à suivre cette documentation pour utiliser la classePoint, la classe Vehicule et créer les classes de l'énoncé.

Tip

Vous pouvez accéder à la javadoc en positionnant le curseur sur l'élément choisi et en utilisant **SHIFT+F2** sur ce même élément.

Figure 3.4. Workspace



2.2. Utilisation d'une API

Ce TP a pour but de vous faire créer des classes de tests, en manipulant des objets définis par des classes que nous avons élaborées et dont nous vous fournissons les API (*Vehicule*, *Point*) ou qui figurent dans l'API standard Java (*String* et *StringBuffer* et *Date*). Attention les fichiers `Vehicule.class` et *Point.class* devront être importés dans

votre projet (ce qui n'est bien entendu pas le cas des fichiers `String.class`, *StringBuffer.class* et *Date.class* qui sont disponibles par défaut depuis n'importe quel projet).

2.2.1. Application Programming Interface

une API (Application Programming Interface) ou interface de programmation pour les applications comprend toutes les méthodes et les variables utilisables pour les programmeurs pour écrire leurs applications. L'API d'une classe décrit un objet et la manière de le manipuler.

2.2.1.1. Principes d'utilisation

L'Application Programme Interface standard est l'ensemble des bibliothèques mises à votre disposition en Java. Les APIs sont disponibles sur `http://docs.pedago.src/java/api/`ou sur `http://java.sun.com/javase/6/docs/api/`.

Rappels :

- la première étape est de déclarer une instance d'objet,
- avant d'utiliser une méthode sur une instance d'objet, il FAUT construire cette instance (par une clause *new*),
- pour utiliser une méthode d'une classe, il faut l'appliquer à une instance d'un objet de cette classe. Exemple : maString.methodeDeString(paras),
- faire attention aux types des paramètres et au type de retour de la méthode,
- bref, le principe c'est de respecter les signatures des méthodes et de suivre la procédure :

  1. déclaration
  2. construction
  3. utilisation

  des objets.

2.2.1.2. Exemple de l'API d'une classe *Bidon*

Une classe *Bidon.java* a été réalisée, comprenant des constructeurs d'objets et des méthodes s'appliquant sur ces objets :

```
public class Bidon ...
// Constructeurs
public Bidon(int a, int b);
```

Le constructeur, du nom de la classe, prend en paramètre deux entiers.

```
//méthodes
public int bidule(String c);
```

renvoie un entier comme résultat, prend une *String* en paramètre.

2.2.1.3. Exemple d'utilisation de la classe *Bidon*

On peut vouloir utiliser la classe `Bindon.java` en instanciant des objets de cette classe et en appliquant certaines méthodes sur ces objets. On va pour cela, créer une classe `TestBidon.java` comprenant :

déclaration
: de ma variable *monObjet* de type *Bidon*

```
Bidon monObjet ;
```

construction
: en donnant des valeurs aux deux entiers requis par le *constructeur*

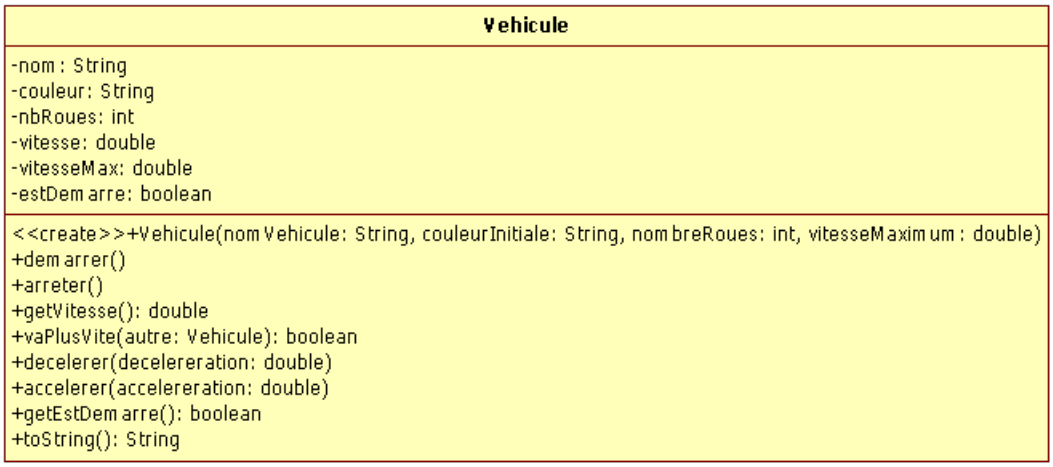
```
monObjet = new Bidon(2,3) ;
```

appel
de la méthode *bidule* ; on stocke le résultat dans une variable de type *int*, on donne une chaîne de caractères en paramètre.

```
int entier ;
entier = monObjet.bidule("bonjour") ;
```

2.2.2. Vehicule

Figure 3.5. Classe véhicule



Les classes *Vehicule* et *Point* sont disponibles dans le paquetage tp.prof.classes, pour les utiliser vous devez les importer. L'*import* spécifie l'espace de nommage, vous devrez rajouter dans vos fichiers :

```
import tp.prof.classes.Point;
import tp.prof.classes.Vehicule;
```

Le bytecode de Vehicule étant disponible, nous allons l'utiliser pour créer dans tp.etudiant.tests une classe TestVehicule. Cette classe ressemblera à :

```
package tp.etudiant.tests;

import tp.prof.classes.Vehicule;

public class TestVehicule {

 /**
  * @param args
  */
 public static void main(String[] args) {
 //au boulot
 }

}
```

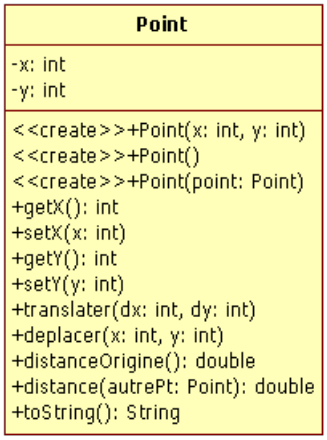
Dans cette classe, vous devrez :

- créer une voiture de couleur rouge à quatre roues ne pouvant dépasser 180km/h, l'afficher
- démarrer la voiture, la faire accélérer de 50km/h, afficher la voiture
- la faire accélérer de 40km/h
- faire afficher sa vitesse
- créer un autre véhicule
- démarrer, accélérer, ...

- comparer les vitesses des deux véhicules.
- ...

2.2.3. Point

Figure 3.6. Classe Point



Vous disposez d'une API documentant la classe *Point*, dans le paquetage *test* créer une classe *TestPoint*, puis vous :

- Créer un Point p1 initialisé à (0,0), l'afficher.
- Créer un Point p2 initialisé à (1,2), l'afficher.
- Afficher la coordonnée x de p1, puis de p2.
- Modifier la coordonnée y de p2 en y affectant la valeur 3 , afficher p2.
- Translater p2 de 1 en x et de 1 en y, afficher p2.
- Déplacer p1 en (2,10), l'afficher.
- Calculer la distance de p2 à l'origine, l'afficher.
- Calculer la distance entre p1 et p2, l'afficher.
- Affecter p1 à p2, afficher les deux.
- Translater p1 de 2,1, afficher p1 et p2 (surpris(e) ?)

2.2.4. String et StringBuffer

2.2.4.1. Principe

L'objet *String* décrit une chaîne de caractères, non modifiable, de taille fixe. En d'autres termes, dès qu'une *String* est créé, elle devient non modifiable. La *String* est le seul objet à pourvoir être créé sans appel explicite du constructeur :

```
String s = "bonjour" ;
```

Ceci est dû au caractère incontournable des objets chaînes de caractère. L'objet *StringBuffer* décrit une chaîne de caractères, modifiable, de taille variable, ce qui permet d'insérer des caractères à une position, de rajouter des caractères en fin, ...

Ces deux classes sont deux classes de la bibliothèque standard de Java. La documentation de ces classes se trouve donc dans l'API standard (en anglais).

2.2.4.2. Réalisation

Vous allez :

- Créer une classe TestString, que vous utiliserez pour la suite.
- Créer une String initialisée à "anticonstitutionnellement".
- Quelle est la longueur de cette String ? (faire afficher le résultat).
- Afficher le troisième caractère.
- Extraire la sous-chaîne allant du deuxième caractère au quatrième, l'afficher.
- Passer la String du départ en majuscule.
- Faire écrire ce résultat et la String originale.
- Afficher la première et la dernière position du caractère 'n' dans la String. Que se passe-t-il, si on demande la position d'un caractère non présent dans la chaîne ?
- Créer un nouvelle String initialisée à bonjour.
- Créer une StringBuffer à partir de la String précédente.
- Concaténer " le monde", faire afficher la StringBuffer résultat.
- Insérer " tout" à la position 7, faire afficher la StringBuffer résultat.
- Créer trois *String s1*, *s2* et *s3* en appelant explicitement le constructeur (

  ```
  String s = new String(...)
  ```

). Les trois valeurs d'initialisation sont *"bonjour"*, *"bonjour"*, et *"Bonjour"*. Comparer *s1* à *s2* et *s1* à *s3* en utilisant successivement l'opérateur == , la méthode *equals(...)* et la méthode *equalsIgnoreCase(...)*. Résultat ?

2.2.5. Date

La classe `java.util.Date` n'est pas d'un usage cohérent, je vous conseil de lire la documentation. Vous observerez aussi qu'elle contient des méthodes dépréciées. Une méthode dépréciée est une méthode dont-on conseil l'utilisation d'une autre méthode plus récente. Ici nous utiliserons les méthodes dépréciées.

2.2.5.1. Introduction

La classe *Date* permet de manipuler des dates. Cette classe fait partie du paquetage java.util. Il faut donc mettre

```
import java.util.Date;
```

en première ligne d'un fichier qui veut utiliser cette classe. Certaines méthodes de cette classe sont dépréciés, mais nous allons tout de même l'utiliser. Sa remplaçante présente (pédagogiquement parlé) beaucoup moins d'intérêts. Il ne faut donc pas s'inquiéter de messages comme :

```
Note : TestDate.java uses or overrides a deprecated API.
Note : Recompile with -Xlint :deprecation for details.
```

2.2.5.2. Réalisation

Vous aller :

- Créer une date initialisée à la date du jour
- L'afficher
- Afficher le mois,
- Modifier l'année en 1999, afficher la date
- Créer une date initialisée à votre date de naissance, l'afficher

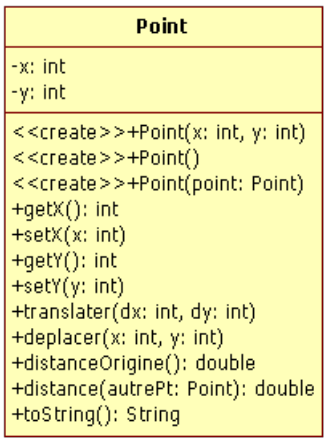
- Comparer ces deux dates.

2.3. Création d'une classe

Le but de ce TP est de vous faire écrire vos premières classes décrivant un objet (Point et Personne). Une fois ces classes décrivant le fonctionnement d'un objet écrites, les instances de ces classes vont s'utiliser de la même façon que n'importe quel objet de l'API standard.

2.3.1. Point

Figure 3.7. Classe Point



Vous avez déjà utilisé la classe *Point* (celle de tp.prof.classes). En vous inspirant de l'exemple du cours, implantez cette classe. La classe `TestPoint.java` du TP précédent doit continuer de fonctionner. Avant de créer votre propre classe *Point*, vous devez modifier TestPoint pour l'import soit

```
import tp.etudiant.classes.Point;
```

et non plus `tp.prof.classes.Point;`.

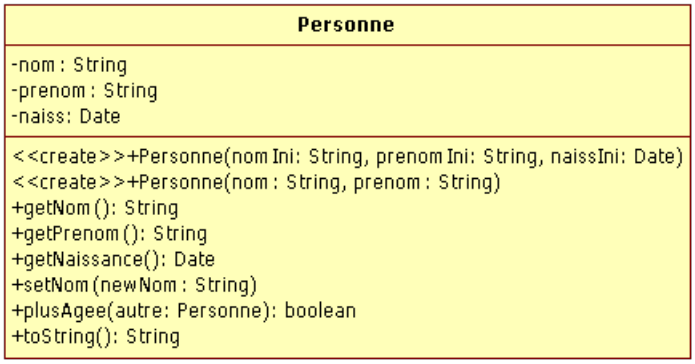
Vous placerez donc la classe *Point* dans le paquetage *tp.etudiant.classes*.

Important

Vous devez suivre l'API et non pas seulement le diagramme de classe UML.

2.3.2. Personne

Figure 3.8. Classe Personne



La classe *Personne* décrit (sommairement) une personne . On mémorise trois informations, le nom de la personne, le prénom de la personne et la date de naissance de la personne. On peut comparer l'âge de personnes.

2.3.2.1. Implantation de la classe Personne

Outre de diagramme de classe UML précédent, vous disposez de la documentation de la classe Personne, elle contient l'API de *Personne*. Vous devrez suivre cette API pour implanter la classe *Personne*, dans le paquetage *tp.etudiant.classes*.

2.3.2.2. Réalisation d'une classe de test

Écrire une classe *TestPersonne*, dans le paquetage *test*, dotée d'une méthode *main* qui fait appel aux différentes méthodes que vous aurez écrites dans la classe *Personne*.

2.4. Variables de classe

Le but de ce *TP* est de consolider l'écriture de classes décrivant un objet et d'y rajouter les variables et méthodes de classes.

2.4.1. Rappels

Les objets ont des comportements et des états séparés, pour qu'ils puissent partager des données, une solution est l'utilisation de variables de classes. Les variables de classes sont des variables partagées par tous les objets de cette classe. Les méthodes de classe permettent d'accéder à des méthodes sans avoir à créer d'objets. La classe utilitaire *java.lang.Math* en est un exemple, elle contient des constantes (

```
Math.PI
```

,

```
Math.E
```

) et des méthodes comme

```
double Math.sqrt(double x)
```

qui calcul une racine carré.

2.4.1.1. Variables de classes, constantes

Example 3.2. Variable de classe

```
public class MaClasse
{
static int maVariableDeClasse ;
...
```

Une variable de classe se déclare comme une variable d'instance, mais en rajoutant le modifieur *static*.

Une constante se déclare comme une variable de classe mais en rajoutant le modifieur final. La signification de *final* est : non modifiable, donc constant.

Example 3.3. Variable de classe constante

```
public class MaClasse
{
final static int MA_CONSTANTE ;
...
```

il est de tradition de noter les constantes en majuscules en séparant les (éventuels) mots composant l'identificateur par le caractère _.

Les variable de classe ou constantes peuvent s'initialiser de deux manière :

directement
: au moment de la déclaration.

Example 3.4. Initialisation et déclaration d'une variable de classe constante

```
public class MaClasse
{
static int maVariableDeClasse = 0 ;
...
```

dans un bloc d'initialisation static.
: C'est simplement un bloc délimité par static{ et } dans lequel on fait ces initialisations (traditionnellement, on utilise un tel bloc pour des initialisations plus complexes que des simples affectation).

Example 3.5. Bloc d'initialisation static

```
public class MaClasse
{
static String maVariableDeClasse ;
static
{
uneMethodeQuiInitialiseMaVariable(maVariableDeClasse) ;
}
...
```

Si on ne fait aucune initialisation, une initialisation par défaut (0 pour les types *byte*, *short*, *int*, *long*, *float*, *double*, false pour *boolean*, null pour un type *objet*) est faite.

Les variable de classe ou constantes peuvent se modifier n'importe quand (dans un constructeur, dans une méthode, ...).

2.4.1.2. Méthodes de classes

Une méthode de classe se définit comme une méthode d'instance, mais en rajoutant le modifieur static.

Une méthode de classe est une méthode qui n'est pas associé à une instance particulière. L'utilisation des méthodes de classes se fait pour une des raisons suivante :

- aucun objet n'est impliqué ; c'est la cas des méthodes de la classe Math ou des fonctions ou procédures que nous avons écrite en algorithme.
- on veut symétriser l'écriture d'une méthode opérant sur deux instances d'une classe (typiquement une méthode de comparaison) :

```
public static boolean soldeSuperieur(CompteBancaire compte1,
CompteBancaire compte2)
```

- on a une méthode qui est spécialisée dans la manipulation des variables de classes.

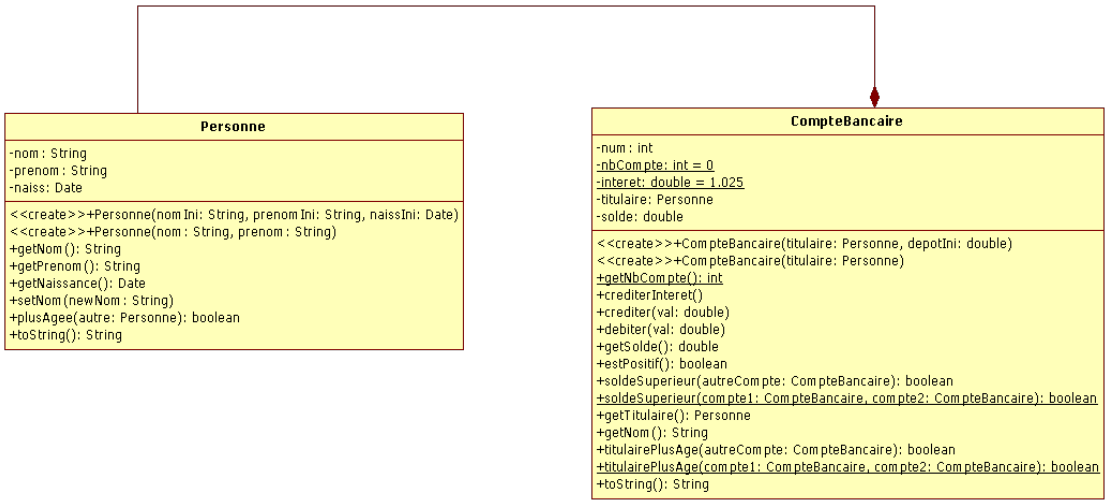
2.4.2. Classe CompteBancaire

Avant de commencer cette partie vous devez avoir la classe Personne opérationnelle, nous allons la réutiliser.

2.4.2.1. Introduction

La classe CompteBancaire décrit (sommairement) un compte bancaire. On mémorise deux informations, le titulaire (une Personne) du compte, le solde du compte. Ce compte peut être crédité et débité. On veut pouvoir gérer le nombre de comptes créés. On utilise une constante pour le taux d'intérêts, commun à tous les comptes et de (2,5% par exemple). Enfin, deux méthodes de classes permettent d'obtenir le nombre de comptes créés et de comparer le solde de deux comptes. Une modélisation UML de cette classe est donnée.

Figure 3.9. Comptebancaire



2.4.2.2. Implantation de la classe CompteBancaire

En utilisant l'API créer une classe *CompteBancaire* dans le paquetage *tp.etudiant.classes*.

2.4.2.3. Réalisation d'une classe test

Écrire une classe *TestCompte*, dans le paquetage *test*, dotée d'une méthode *main* qui fait appel aux différentes méthodes que vous aurez écrites dans la classe *CompteBancaire*.

Chapter 4. Collaborations entre classes

Nous avons utilisé conjointement plusieurs classes, nous allons formaliser ce que nous avons fait.

1. Classes/Association Les bases

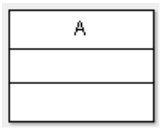
Nous allons étudier comment associer simplement des classes, nous utiliserons la notation UML avec à chaque fois le code Java correspondant.

1.1. Cours et exemples

1.1.1. La classe

Commençons par le plus simple : la classe vide.

Figure 4.1. Classe vide



Sa traduction java est :

```
public class A {}
```

Une classe vide ne sert à rien. Une classe pour être utile doit être peuplée d'attributs (variables d'instance ou variables de classe) et de méthodes (constructeurs, méthodes d'instance, méthodes de classe). Le méthodes et les variables ont une accessibilité :

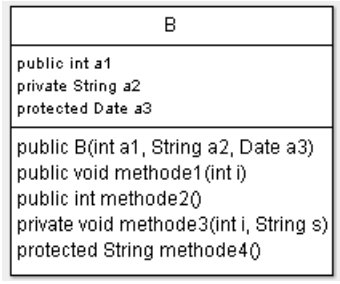
public
: accessible depuis n'importe où,

private
: accessible seulement dans la classe,

protected
: accessible depuis les classes filles et/ou dans le même package.

C'est vision est simplifiée par rapport à java qui est complexifié par la notion de paquetage (package).

Figure 4.2. Classe avec trois compartiments

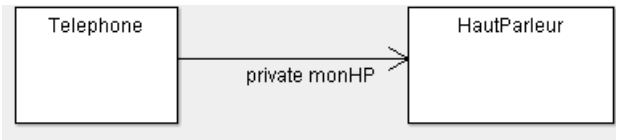



```
package td.exemples;

import java.util.Date;

public class B {
 public int a1;
 private String a2;
 protected Date a3;

 public B(int a1, String a2, Date a3) {

 }

 public void methode1(int i)
 {

 }

 public int methode2()
 {
  return 0;
 }

 private void methode3(int i, String s)
 {

 }

 protected String methode4()
 {
  return "";
 }
}
```

1.1.2. L'association

L'association permet d'exprimer une relation entre deux classes. Elle exprime une connexion sémantique entre deux classes (relation logique). Elle est représentée par un trait plein qui peut-être complété par des flèches. Les flèches définissent la visibilité.

Figure 4.3. Association simple




```
public class Telephone {
 private HautParleur monHP;
}
```

Sur le diagramme précédent la classe Telephone "voit" la classe HautParleur. Nous avons associé à un téléphone, un haut parleur. Mais comment faire pour associer à un téléphone 15 ou n touches ? En utilisant des multiplicités.

1.1.3. Les multiplicités

Les multiplicités sont comparables aux cardinalités du système Merise (mais sont placées de l'autre coté), elles servent à indiquer le nombre minimum et maximum d'instances de chaque classe dans la relation liant 2 ou plusieurs classes :

Digit
: Le nombre exacte qui peut être implémenté sous forme de tableau.

* ou 0..*
: de zéro a plusieurs qui peut être implémenté sous la forme d'une collection souvent un vecteur.

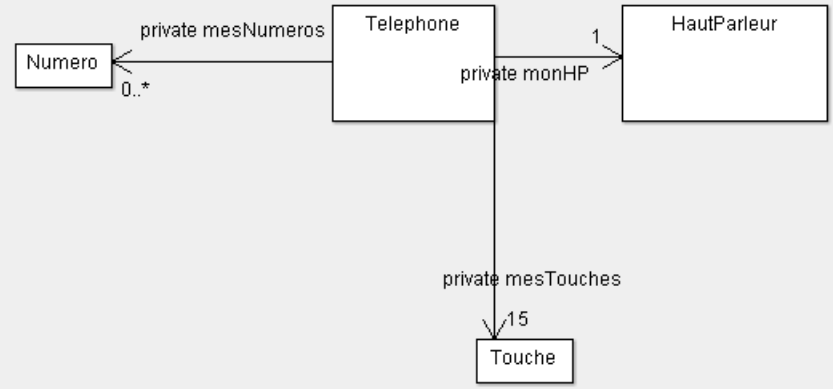
0..1
: zéro ou un, le zéro est implémenté en utilisant une référence nulle.

1..*
: un à plusieurs.

n..m
: de n à m.

1

Figure 4.4. Association avec multiplicité

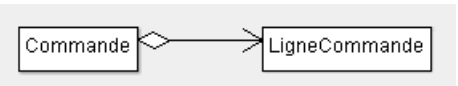



```
public class Telephone {
 private HautParleur monHP;
 private Touche[] mesTouches;
 private Vector<Numero> mesNumeros;
}
```

1.1.4. L'agrégation

Il existe des cas particuliers d'association : les agrégations et les compositions. L'agréation traduit la notion de tout-partie ou est une partie de. Elle est soumise à des restrictions une instance de classe ne peut se composer elle même et il ne peut-y avoir de cycle dans les dépendances.

Figure 4.5. Agrégation



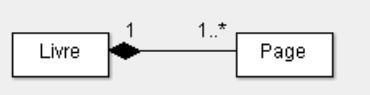
[1] Il aussi possible d'exprimer des plages plus complettes comme 1, 2..10, 100..*.

La traduction java reste la même que pour l'association.

1.1.5. La composition

La composition est une forme de particulière de l'agréation ou le tout est responsable du cycle de vie de la partie. La partie doit-être créée (*new*) et détruite dans le tout. De plus une partie ne peut faire partie de plusieurs tous.

Figure 4.6. Composition




```
package td.exemples;

import java.util.Vector;

public class Livre {
 private Vector<Page> pages;
 public Livre()
 {
  pages = new Vector<Page>();
  Page page = new Page();
  pages.add(page);
 }
}
```

Dans l'exemple précédent, la page est crée dans le Livre, ainsi sa vie est liée à celle du livre.

1.2. Exercices

Nous allons dans un premier temps réfléchir sur un diagramme de classe puis dans un deuxième temps introduire l'exemple qui sera vu en TP.

1.2.1. Diagramme de classe d'une maison

Une maison est composée de pièces, elles mêmes composées de murs. Les pièces contiennent ou ne contiennent pas des objets. La disparition de la maison entraîne celle des pièces.

1.2.2. La voiture

Une voiture est caractérisée par sa marque et son modèle, elle est possédée par une personne qui a un nom. La voiture possède un moteur qui ne peut-être réutilisé après la vie de la voiture. Le moteur est caractérisé par sa puissance et sa marque. Proposer un diagramme de classe et le code java associé.

2. Classes/Association Le détail

UML dispose d'un mécanisme d'extension : les stéréotypes. Ils se présentent sous forme de guillemets ou d'accolade, ils permettent d'étendre la sémantique des éléments de modélisation. Nous verrons des stéréotypes d'association qui augmente le sens de l'association pour faciliter le codage .

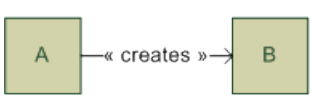
Nous verrons aussi la possibilité d'imbriquer une classe dans une classe interne. L'imbrication nous rendra de grands services lors de la réalisation d'IHM (Interface Homme-machine).

2.1. Cours et exemples

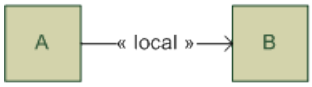
2.1.1. Les stéréotypes d'association

Nous allons au travers de quatre stéréotypes d'association étudier du code Java.

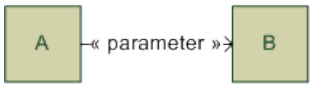
Figure 4.7. Stéréotypes d'association




```
public class A
{
        public B makeB()
        {
                return new  B();
        }
}
```




```
public class A
{
        public void f()
        {
                B b = new B() ;
                //utilisation de b
        }
}
```




```
public class A
{
        public void f(B b)
        {
                //utilisation de b
        }
}
```

2.1.1.1. create

Le stéréotype "create" indique que la source va créer la cible est la rendre disponible pour le reste de l'application.

2.1.1.2. local

Le stéréotype "local" indique que la source va créer une instance de la cible, conservée dans une variable locale et non un attribut.

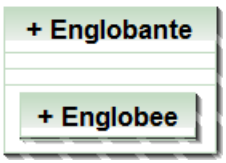
2.1.1.3. parameter

Le stéréotype "parameter" indique que la source accède à la cible via une référence reçue en paramètre

2.1.2. Les classes internes

Il est possible de déclarer une classe dans une autre classe (ou une interface)[2]. Une classe interne a un accès complet sur tous les attributs des ses classes englobantes. Les classes internes nous seront utiles pour traiter les événements. En UML les classes internes sont représentées par un cercle avec une croix mais on peut aussi les trouver simplement imbriquées.

Figure 4.8. Classes imbriquées




```
public class Englobante {
```

[2]Java est très riche, il permet des classes internes statique de classe, des classes internes de méthodes, des Innerclass de classe celle que nous étudierons et des classes anonyme de méthode.

```
 public class Englobee {

 }
}
```

Il existe d'autre possibilités pour qualifier les associations comme les classes d'association et le qualifieurs d'association.

3. Exercice

Vous aller créer un projet seance3 dans lequel, vous mettrez votre code.

3.1. Communication entre objets

Nous allons illustrer la communication entre objets au sein d'un même processus. Les objets communiquent par messages, lorsqu'un objet utilise une méthode d'un autre objet, il envoie, à ce dernier, un message lui demandant de rendre un service. La notation UML(Unified Modeling Language) reconnaît au sein du diagramme de classe plusieurs associations qui traduisent la communication :

Association
: une association simple représente une relation sémantique durable entre deux classes

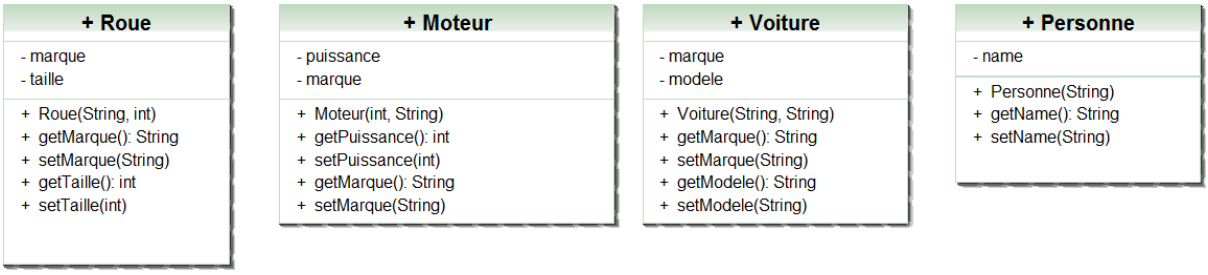
L'agrégation partagée
: Les agrégations sont des assocaitions non symétriques particulières qui signifient "contient" ou "est composé de" Pour l'agrégation partagée, les cycles de vie sont indépendants, les objets sont créés et détruit séparément. Un élément ne peut appartenir qu'a un seul agrégat composite

L'agrégation de composition ou composition
: Les cycles de vie sont liés, la destruction de l'agrégat composite entraine la destruction de tous ses ensembles. Le composite est responsable du cycle de vie de tous ses éléments.

3.1.1. Les classes d'origine

Coder les classes *Voiture*, Personne, *Moteur* et Roue du diagramme de classe UML suivant.

Figure 4.9. Classe isolées



A ce stade nos classes ne peuvent traduire des associations, comment traduire qu'un véhicule appartient à une personne ?

3.1.2. Association simple

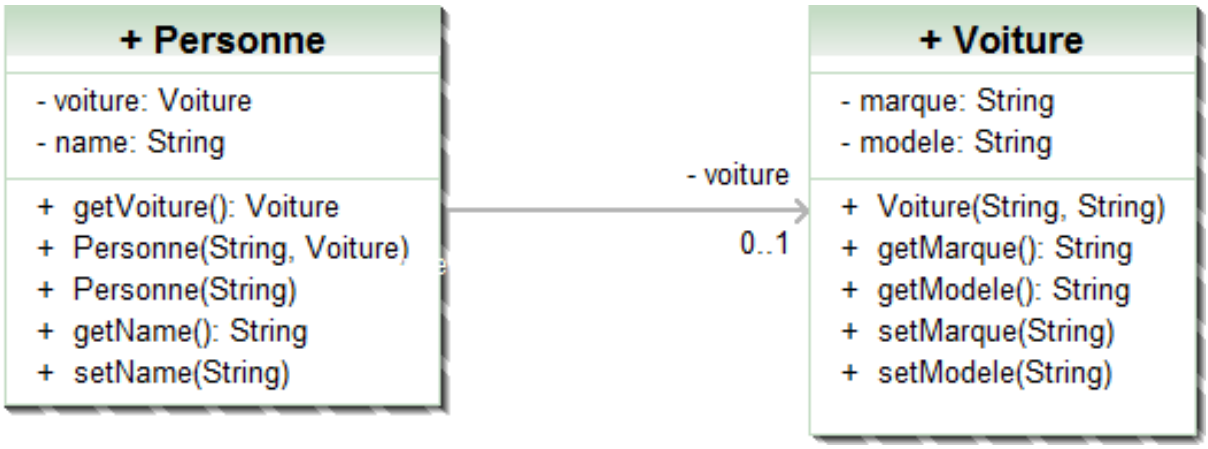
La communication entre classe se fait en rajoutant des variables d'instance. Si une voiture veut voir une personne alors la classe Voiture doit contenir une variable d'instance de type Personne, l'inverse et bien entendu vrai et les deux sens de visibilité peuvent cohabiter.

Le sens de la visibilité n'est pas le seul paramètre, tout comme pour les MCD (Modèle Conceptuel de Données) des cardinalités peuvent être exprimées, les cardinalités multiples seront impléméntées sous forme de collections ou de tableaux.

Faite en sorte qu'une personne possède au plus une voiture et que la voiture ne connaisse pas sont possesseur. Le digramme UML suivant doit pouvoir vous aider.

Une fois les classes modifiées, et en utilisant une classe munie d'un main faîte en sorte que la personne "Brard" puisse posséder une ferrari de modèle "pas cher". La personne "brard" peut-elle posséder deux voitures et peut-elle changer le modéle de sa voiture ? Si la personne "brard" ou la voiture disparait, l'autre disparait-il aussi ?

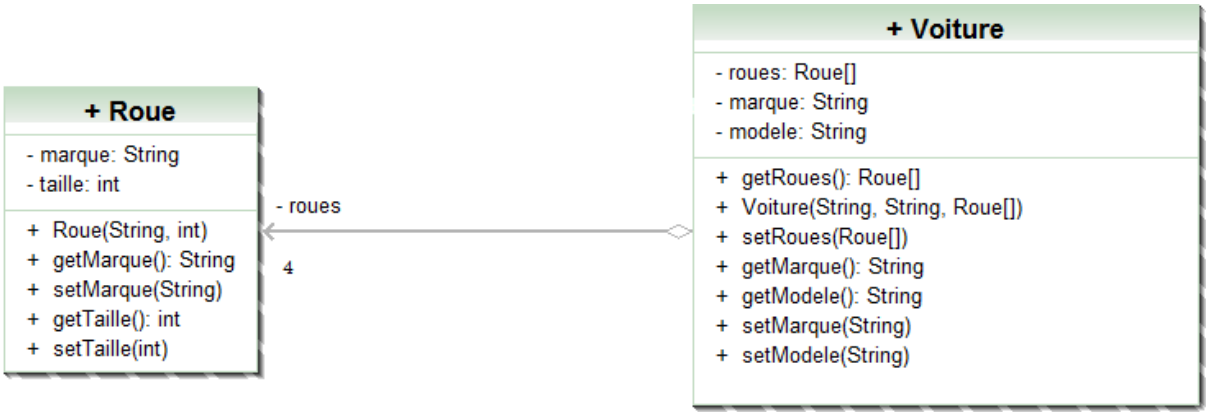
Figure 4.10. Association simple



3.1.3. Agrégation (partagée)

L'association partagée correspond à une association "groupe-élément", "tout-partie" où la disparition du groupe n'entraîne pas la disparition de l'élément. Modifier vos classes avec le diagramme suivant puis tester en faisant en sorte que la voiture Ferrari précédente ait quatre roues.

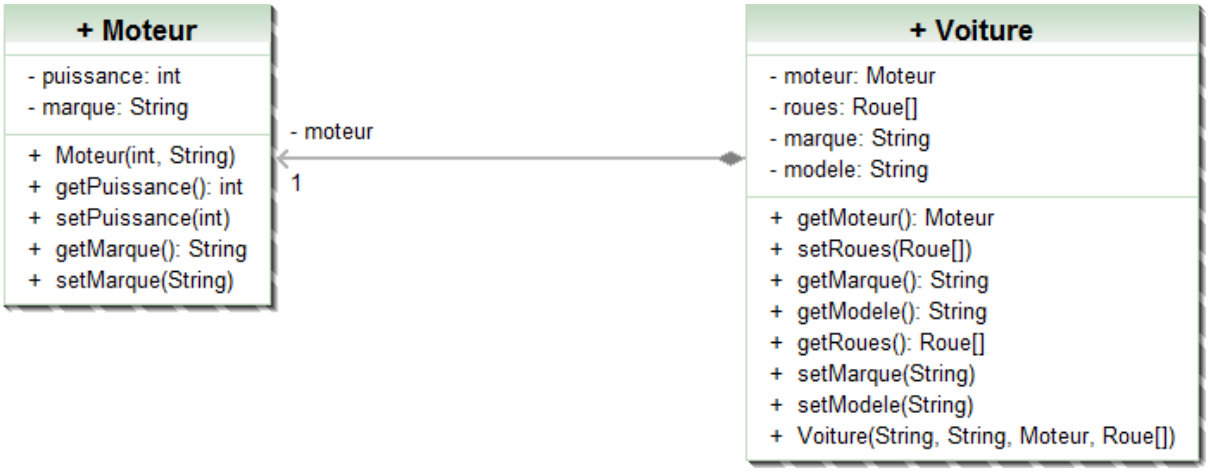
Figure 4.11. Agrégation partagée



3.1.4. La composition

Nous allons maintenant traduire le fait que le moteur fait partie intégrante de la voiture, si la voiture est détruite alors le moteur l'est aussi. Implémenter le diagramme suivant.

Figure 4.12. Agrégation de composition

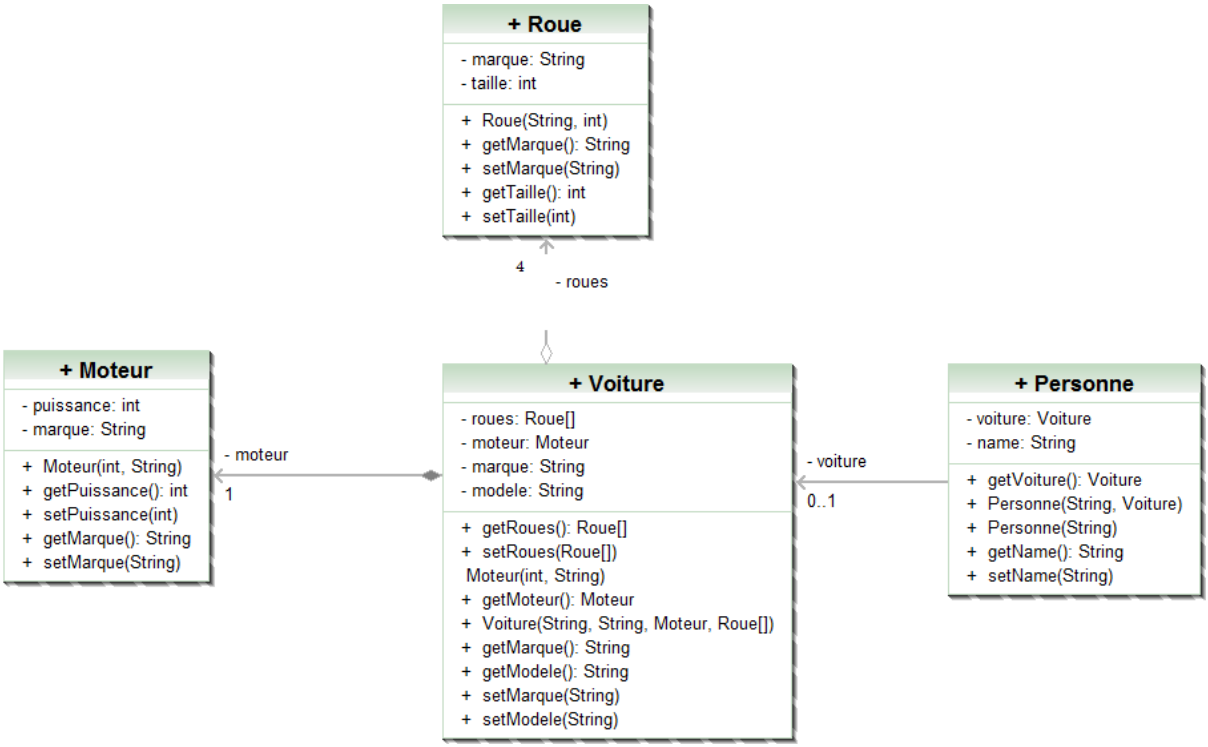


Nous avons modifier le constructeur de voiture pour que la voiture soit munie à la construction d'un moteur. Dans ce constructeur un nouveau moteur doit être construit (*new*) à l'image de celui passé en paramétre.

Créer un moteur (initial), puis une voiture, qui posséde une copie de ce moteur enfin modifier le moteur initiale, celui de la voiture est-il modifié.

Au final nous avons réalisé tout ceci :

Figure 4.13. Diagramme de classe complet



Nous venons de voir comment faire communiquer des classes, nous allons maintenant apprendre une nouvelle façon de factoriser le code en réutilisant des classes.

Chapter 5. Héritage, classes abstraites, interfaces et polymorphisme

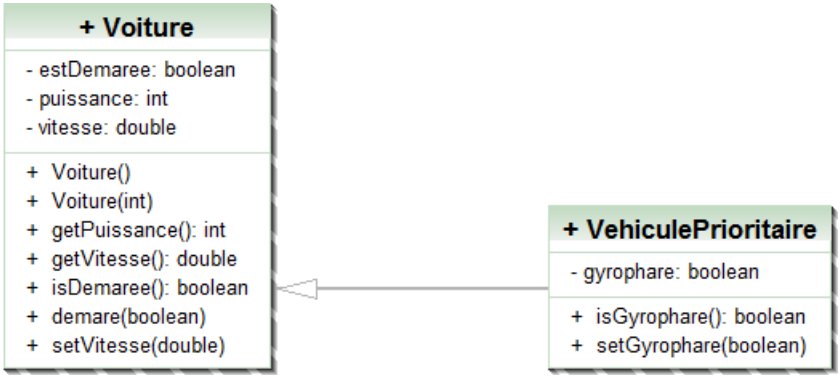
1. Cours et exemples

L'héritage est un moyen de réutiliser du code, à l'image d'un arbre généalogique, une classe fille hérite des caractéristiques de sa classe mère. L'héritage va nous permettre d'utiliser des classes abstraites : des classes dont une partie du code n'est pas encore connu mais dont la signature des méthodes est connue. Le classes abstraites ont une forme extrême, des classes sans aucun code et avec uniquement des méthodes statiques : les interfaces. L'héritage va aussi nous ouvrir la voie du polymorphisme, la capacité que possède un objet à changer de type à l'exécution.

1.1. Héritage

L'héritage est représenté en UML avec une flèche triangulaire. La classe cible est la classe mère, l'autre est la classe fille. On parle encore de superclasse pour la classe mère, et de sous-classe pour la classe fille. L'héritage peut-être traduit par "est une sorte de". La classe fille "est une sorte de" classe mère. Sur l'exemple qui suit la classe mère est Voiture et la classe fille est VehiculePrioritaire. En java, il n'y a pas d'héritage multiple une classe fille n'a qu'une classe mère. Une classe mère par contre peut-avoir plusieurs classes filles.

Figure 5.1. Héritage



La notion d'héritage est traduite en java par le mot clef extends.

```
public class VehiculePrioritaire extends Voiture {
 private boolean gyrophare;

 public boolean isGyrophare() {
  return gyrophare;
 }

 public void setGyrophare(boolean gyrophare) {
  this.gyrophare = gyrophare;
 }
}
```

L'héritage permet d'enrichir, la classe VehiculePrioritaire possède les méthodes et les attributs (public ou protected) de la classe Vehicule plus ses propres attributs.

L'héritage permet aussi de spécialiser, de *redéfinir* le comportement de la classe mère en *redéfinissant* ses méthodes. En java, la classe Object est la mère de toute classe, elle possède la méthode

```
public String toString()
```

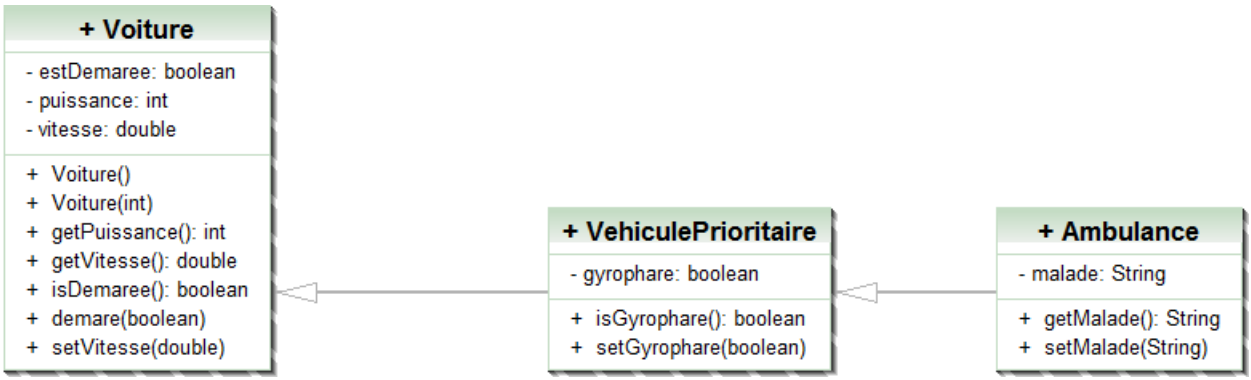
qui affiche le nom de la classe et la référence mémoire, vous avez déjà redéfini cette méthode pour obtenir un affichage plus convivial.

Enfin l'héritage permet de réutiliser du code sans en avoir la source (si la classe n'est pas final), pour hériter de Voiture, seul son *.class* est utile, le *.java* ne l'est pas.

Lors de la construction d'un objet de type d'une classe fille un constructeur de la classe mère est appelé en java le mot clef *super* à la même sens que *this* dans la classe fille. super(...) permet de faire un appel explicite à un constructeur de la classe mère, super.attribut ou super.methode() permet d'utiliser un attribut ou une méthode de la classe mère.

Par défaut, *super()* est utilisé, c'est donc le constructeur par défaut si il existe ou alors le constructeur sans paramètre de la classe mère qui est appelé.

Figure 5.2. Chaînage des constructeurs



Si le code de Voiture() est

```
public Voiture() {
  this(5);
 }
```

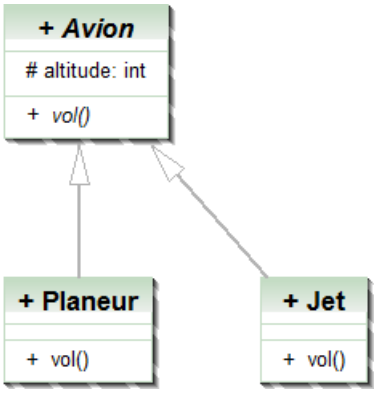
new Ambulance() conduira à la création d'une ambulance de 5 chevaux. Ambulance() est le constructeur par défaut, il existe car il n'y a pas d'autre constructeur dans Ambulance. Ambulance() fait appel implicitement (super()) à VehiculePrioritaire() qui lui même fait appel au constructeur sans paramètre Voiture(). Voiture() n'est pas un constructeur par défaut et il fait appel (this(5)) à une surcharge Voiture(int).

1.2. Classe abstraite

Une classe est dite abstraite si l'on ne connaît pas le code d'au moins une de ses méthodes. Attention une méthode sans code n'est pas une méthode dont le code est vide. maMethode(); n'a pas de code, maMethode(){ } est une méthode dont le code est vide. Une classe abstraite ne peut-être instanciée, alors à quoi sert-elle ? Elle permet de définir un comportement pour toutes ses classes filles sans en connaître le code. Toutes les classes filles sous peine de rester abstraite doivent implémenter la méthode abstraite.

En UML une classe abstraite et ses méthodes abstraites sont signalées par la propriété {abstract} est ou par un nom en italique.

Figure 5.3. Classe Abstraite



Le code java associé est le suivant :

```
public abstract class Avion {
 protected int altitude;

 public abstract void vol() ;

}
```

```
ublic class Planeur extends Avion {

 @Override
 public void vol() {
  // TODO Auto-generated method stub
  altitude +=100;
  if (altitude > 1000) altitude=1000;
 }

}
```

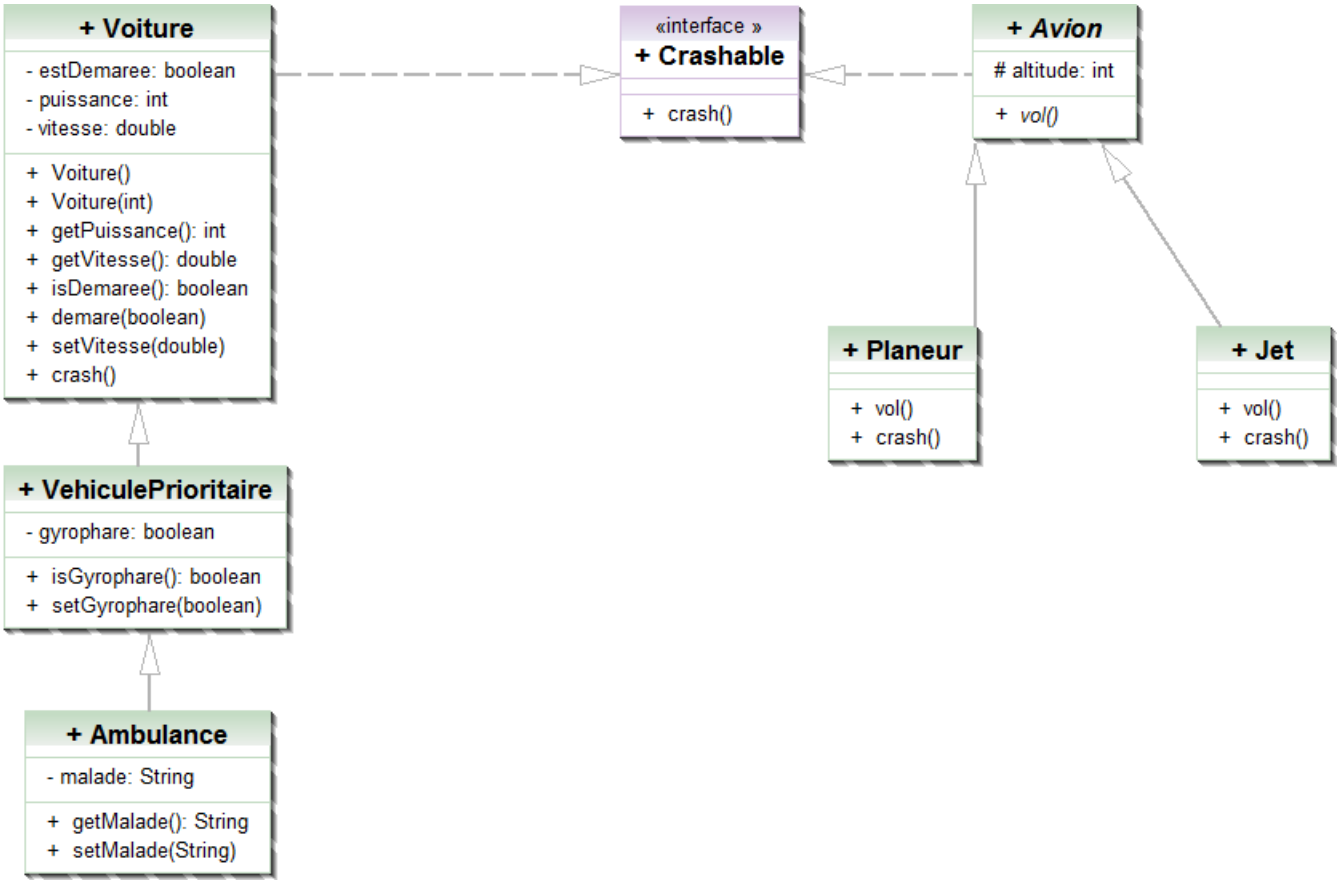
```
public class Jet extends Avion {

 @Override
 public void vol() {
  // TODO Auto-generated method stub
  if (altitude > 1000) altitude=10000;
 }

}
```

Tous les avions vols mais on ne sait pas comment. Par contre, on sait comment un Jet ou un Planeur volent.

1.3. Interface

Une interface est une classe abstraite pure. Elles sont très utiles en java qui ne possède pas d'héritage multiple mais qui permet d'implémenter plusieurs interfaces. Vous avez déjà rencontré l'interface comparable<T> qui contient la méthode int compareTo(T o). Pour rendre des objets comparables, il suffit d'implémenter l'interface comparable à savoir donner un code à la méthode compareTo(T o). En UML une interface est un des deux stéréotypes de classe, l'autre est "utility" : une classe utilitaire est une classe qui ne contient que des attributs et des méthodes statiques. L'interface est représentée soit avec l'annotation "interface" soit avec une "lolipop". Le fait de réaliser (implements) une interface est noté avec un flèche en pointillés à tête triangulaire.

Figure 5.4. Interface



Le code java associé est

```
public interface Crashable {
 public void crash();
}
```

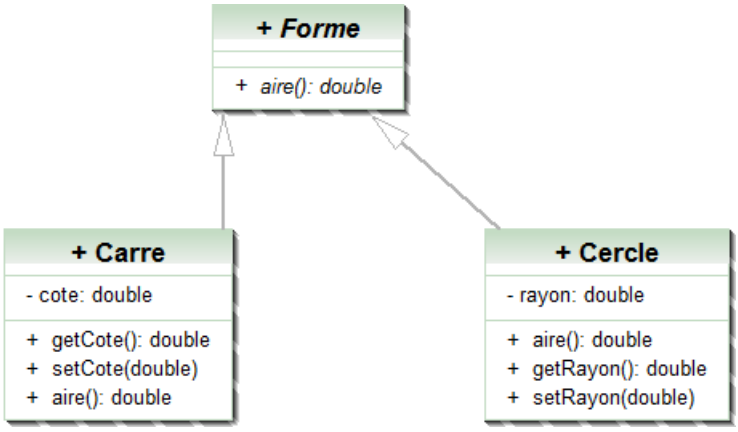
Voiture a du être modifier pour réaliser l'interface Crashable comme suit :

```
public class Voiture implements Crashable{
 ...

 @Override
 public void crash() {
  // TODO Auto-generated method stub
  ...
 }
 ...

}
```

1.4. Polymorphisme

L'héritage est un moyen de réaliser le polymorphisme en java. Le polymorphisme est le moyen pour un objet de changer de type à l'exécution. L'idée est d'autoriser le même code à être utilisé avec différents types. Vous l'avait fait en utilisant Collections.sort(). Soit l'exemple suivant :

Figure 5.5. Polymorphisme



la méthode de Forme aire() est abstraite, elle est implémentée dans Carre et dans Cercle de deux manière différente. Nous pouvons calculer l'aire totale de plusieurs formes comme suit :

```
float aireTotal(Forme[] tabl) {
   float s=0;
   for(int i = 0; i < tab1.length; i++) {
      s += tabl[i].aire(); // le programme sait automatiquement quelle fonction appeler
   }
   return s;
}

// ...
Carre c1 = new Carre(); c1.setCote(10);
Carre c2 = new Carre(); c2.setCote(20);
Cercle cer1 = new Cercle(); cer1.setRayon(1);
Forme[] tableau = { c1, c2, cer1};
int a = aireTotal(tableau);
```

La méthode aireTotal() attend un tableau de Forme et exécute le même algorithme quelque soit la Forme. Nous avons donc un algorithme qui peut s'appliquer à n'importe qu'elle sorte de Forme. Il nous fout donc pouvoir transformer tout objet d'un classe fille de Forme en un objet de type Forme (upcating ou surclassement). A une référence d'un type donné, soit Forme, il est possible d'affecter une valeur qui correspond à une référence vers un objet dont le type effectif est n'importe quelle sous classe directe ou indirecte de Forme. Forme f1 = new Carre() est possible, de même que Forme f2 = new Cercle(). f1 et f2 ne dispose que des méthode de Forme mais peuvent retrouver leur type d'origine. f1= (Carre) f1 et f2=(Cercle) f2 on parle de downcasting. Le downcasting permet de liberer les fonctionnalités cachées.

2. Exercices

Créer dans votre workspace un projet nommé `seance3`.

2.1. Héritage, classe abstraite et interface

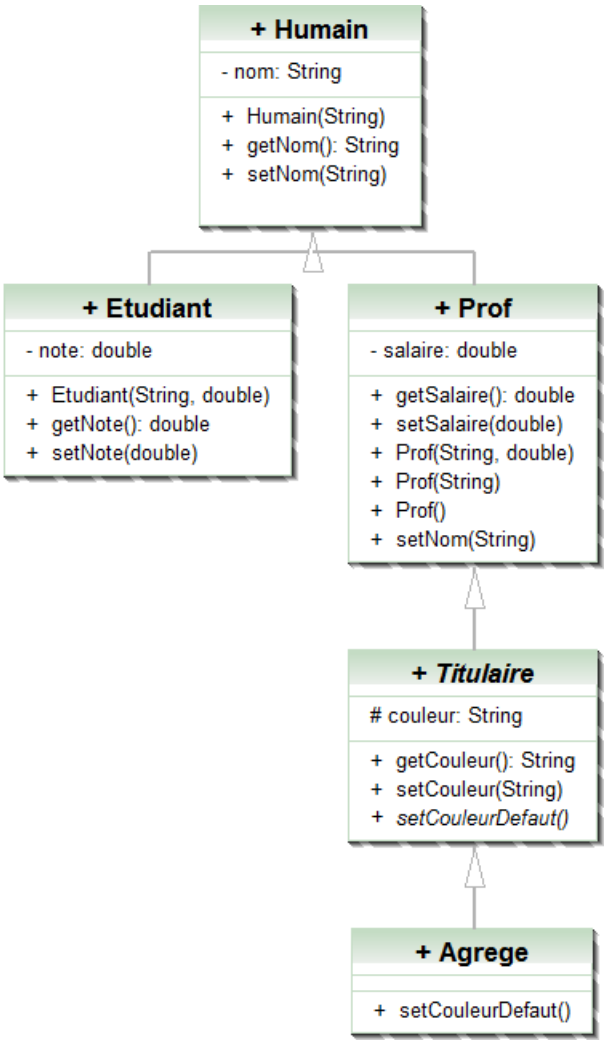
Nous allons étudier comment réutiliser des classes.

2.1.1. Héritage

L'héritage est un principe de la programmation orientée objet, permettant entre autre la réutilisabilité et l'adaptabilité des objets. Elle se nomme ainsi car le principe est en quelque sorte le même que celui d'un arbre généalogique. Ce principe est basé sur des classes dont les "*filles*" héritent des caractéristiques de leur(s) "*mère*(s)". Les classes possèdent des attributs et/ou des méthodes qui leurs sont propres, et qui pourront être transmis aux classes filles découlant de la classe mères. Chacune des classes filles peut donc posséder les mêmes caractéristiques que sa classe mères et bénéficier de caractéristiques supplémentaires à celles de ces classes mères. Chaque classe fille peut, si le programmeur n'a pas défini de limitation, devenir à son tour classe mère.

Nous allons travailler avec le diagramme suivant :

Figure 5.6. Arbre d'héritage



Commencer par coder la classe *Humain*.

Créer la classe *Etudiant* qui hérite de *Humain* (*public class Etudiant extends Humain*). La classe *Etudiant* disposera de toutes les méthodes de la classe *Humain*. *Humain* ne possédant pas de constructeur par défaut [1] vous devrait faire appel explicitement au constructeur public Humain(String nom) avec super(param). super possède la même syntaxe que *this* mais référence la classe mère.

De même coder la classe *Prof*, cette classe doit *redéfinir* la méthode setNom(String nom). Le constructeur Prof()[2] créé un *prof* avec un *salaire* de 0 et sans *nom*(*null* ou *""*), le constructeur *Prof(String nom)* créé un *prof* de nom *nom* avec un *salaire* de 0. *La surcharge n'est pas la redéfinition, la surcharge consiste à avoir plusieurs méthodes de même nom et la redéfinition consiste à réécrire le code d'une méthode héritée.* La méthode setNom de Prof redéfinie celle d'Humain et impose que le nom soit mis en majuscule (toUpperCase).

Tester votre code.

2.1.2. Classe abstraite et interface

En programmation orientée objet (POO), une classe *abstraite* est une classe dont l'implémentation n'est pas complète et qui n'est pas instanciable. Elle sert de base à d'autres classes *dérivées* (*héritées*).

[1] Si une classe n'a pas de constructeur, elle a un constructeur par défaut parcontre dès qu'une classe à un constructeur, elle n'a plus de constructeur par défaut. Le constructeur sans parmètre n'est pas le constructeur par défaut.
[2] Prof() est ici le constructeur sans paramètre de la classe Prof.

Créer la classe abstraite *Titulaire* (*public abstract class Titulaire extends Prof*) qui possède la méthode abstraite *setCouleurDefaut* (*public abstract void setCouleurDefaut();*), l'attribut *couleur* peut être déclaré *protected* (# en UML) pour pouvoir être modifié dans les classes héritées, il peut aussi être accesible via un setter public.

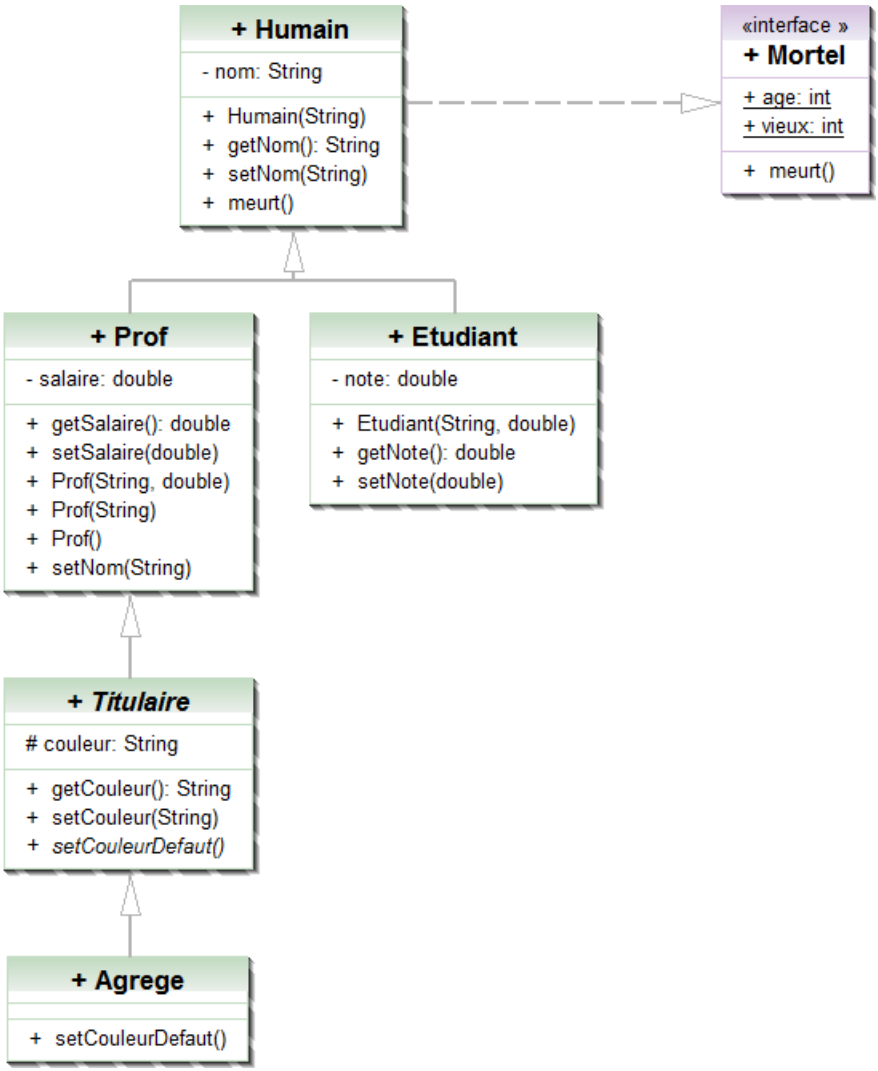
Créer la classe *Agrege*, les agrégés sont "roses". Tester votre code.

Une *interface* est une classe abstraite sans implémentation. Toutes les méthodes sont abstraites. Seule l'interface de la classe apparaît. Nous allons créer une interface *Mortel* dont le code est le suivant :

```
public interface Mortel {
 public final static int vieux = 40;
 public int age=18;
 public void meurt();

}
```

Comment pouvons nous rendre les humains mortels en implémentant l'interface (implements Mortel). Réaliser cette opération et faite mourir un agrégé (écran un message à l'écran).

Figure 5.7. Interface



2.1.3. Polymorphisme

Le polymorphisme veut dire que le même service, aussi appelé opération ou méthode, peut avoir un comportement différent suivant la Classe dans laquelle il est utilisé. Nous l'avons vu avec la surcharge mais il peut être introduit en utilisant l'héritage. Utiliser les lignes suivantes :

```
Object a = new Agrege(); //surclassement
Agrege o = new Object();
```

Laquelle fonctionne ?

Essayons maintenant

```
Object a = new Agrege(); //surclassement
Humain h = (Humain) a; //transtipage
h.setNom("toto");
```

Comme nous le voyons, la méthode à exécuter est déterminé à l'exécution et non pas à la compilation, par contre le surclassement est réalisé à la compilation.

3. Mis en oeuvre avec SWING

Nous allons illustrer le chapitre précédent avec l'utilisation de la bibliothèque graphique SWING (et AWT). Le code est fourni, pour chaque commentaire, il faut expliquer le concept objet mis en oeuvre (héritage, utilisation d'une méthode redéfinie, utilisation d'une méthode de la classe mère, polymorphisme, ...). La lecture de la javadoc peut vous aider.

```
import javax.swing.BoxLayout;
import javax.swing.JColorChooser;
import javax.swing.JFrame;
import javax.swing.JTextArea;

public class MaFenetre extends JFrame {  //1
 private JTextArea textArea;
 private JColorChooser colorChooser;
 public MaFenetre(String titre)
 {
  super(titre); //2
  this.setDefaultCloseOperation(EXIT_ON_CLOSE);
  textArea = new JTextArea(20,40);
  colorChooser = new JColorChooser();
  this.setLayout(new BoxLayout(this.getContentPane(), BoxLayout.Y_AXIS)); //2
  this.add(textArea); //3
  this.add(this.colorChooser); //3

  this.colorChooser.getSelectionModel().addChangeListener(new MonChangeListener(this));
  this.pack(); //4
  this.setVisible(true); //4
 }
 public JColorChooser getColorChooser() {
  return colorChooser;
 }
 public JTextArea getTextArea() {
  return textArea;
 }
 public void setTextArea(JTextArea textArea) {
  this.textArea = textArea;
 }
}
```

```
package tests;

import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class MonChangeListener implements ChangeListener { //4
 private MaFenetre maFenetre;
 public MonChangeListener(MaFenetre maFenetre)
 {
  this.maFenetre = maFenetre;
 }
 public void stateChanged(ChangeEvent e) { //5
  this.maFenetre.getTextArea().setForeground(this.maFenetre.getColorChooser().getColor());
```

```
 }

}
```

```
package tests;

public class TestMafenetre {

 /**
  * @param args
  */
 public static void main(String[] args) {
  MaFenetre fenetre = new MaFenetre("Ma premiere fenetre");

 }

}
```

Proposer un diagramme de classe correspondant aux classes précédentes.

Chapter 6. Exceptions

Une exception représente un événement exceptionnel, le plus souvent une erreur. L'utilisation du mécanisme d'exception java repose sur : la définition de l'exception, le lancement de l'exécution et le rattrapage de l'exception. Nous allons appréhender les exceptions au travers de la gestion des erreurs

1. Cours

1.1. La définition des exceptions

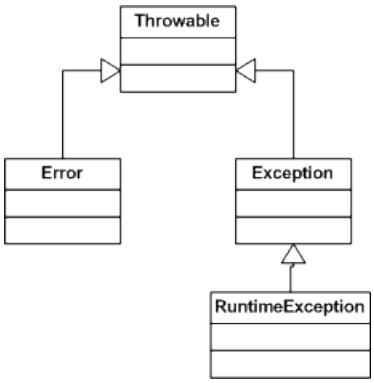
En java les exceptions sont représentées par des classes filles de `Throwable`, on distingues trois types d'erreurs ou de gravité :

java.lang.Error
: Elle représente les erreurs critiques qui ne sont pas censées être rattrapées en temps normal. Par exemple `OutOfMemoryError` est une classe fille de `Error`, elle est lancée lorsque la mémoire système est insuffisante.

java.lang.Exception
: Ce sont les erreurs qui doivent généralement être gérées comme par exemple une erreur d'entrée/sortie (`IOException`).

java.lang.RuntimeException
: Elle représente des erreurs pouvant éventuellement être gérée par le programme, comme par exemple `ArrayIndexOutOfBoundsException` que vous pouvez rencontrer lorsque l'indice est hors du tableau.

Figure 6.1. Exceptions



Pour définir nos classes d'exception, c'est simple, il nous faut hériter de `Exception` ou `RuntimeException`.

```
class MonException extends Exception {} ou class MonException extends RuntimeException {}
```

1.2. Le lancement d'une exception

Le lancement d'une exception se fait à l'aide du mot clef *throw* : `throw new MonException`(). Quand une exception est lancée, toutes les instructions suivantes sont ignorées et on remonte la pile des appels des méthodes : on dit que l'exception se propage. Une méthode qui est susceptible de lancer une exception sans la rattraper, doit le déclarer en utilisant le mot clef *throws* : `public void maMethode()throws MonException1, MonException2{ ... throw new MonException1 ...}`. Cette déclaration n'est pas obligatoire pour les exceptions de la catégorie `Error` ni pour celles de la catégorie `RuntimeException`.

1.3. La capture ou rattrapage de l'exception

La capture d'un exception est réalisée avec un bloc try/catch :

```
try {
//bloc succeptible de lever un exception
}
catch (Exception1 e) {
//Si l'exception Exception1 est levée elle est capturée ici
}
catch (Exception2 e) {
//Si l'exception Exception2 est levée elle est capturée ici
}
finally {
//dans tous les cas
}
```

La ou les clauses *catch* doivent-être utilisées dans le sens inverse de l'héritage, la classe fille (la plus "fine") avant sa classe mère.

Si une exception se produit dans le *try* les instructions qui suivent ne sont pas exécutées et le l'exécution se continue au *catch* correspondant.

La clause *finally* est optionnelle, elle est toujours exécutée que l'exception soit levée ou non.

2. Exercice

Le but de cette partie est de reprendre la classe CompteBancaire et d'associer une exception à la méthode debiter() si le débit demandé fait passer le solde en négatif.

2.1. Création de l'exception

Créer une `Exception` nommée `SoldeInsuffisantException`.

2.2. Le lancement de l'exception

Reprendre la méthode `debiter()` de la classe `CompteBancaire` et lancer l'exception `SoldeInsuffisantException` si le débit demandé fait passer le solde en négatif (Utilisation de *throw* et *throws*).

2.3. Capture de l'exception

Dans le test du compte bancaire capturer l'exception `SoldeInsuffisantException` lors qu'un retrait trop important et faire afficher le message retrait impossible.

Chapter 7. Généricité ou polymorphisme paramétrique

La généricité c'est la possibilité d'écrire des classes et des méthodes dans lesquelles les types sont passés en paramètres (type abstrait de données).

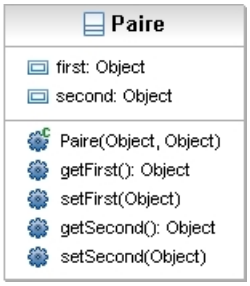
1. Cours

La généricité est apparue avec java 1.5, elle permet de répondre aux limites du polymorphisme, commençons donc par identifier les avantages et les limites du polymorphisme.

1.1. Les limites du polymorphisme

Sans la généricité le polymorphisme est réalisé conjointement avec l'héritage, les méthode peuvent prendre en paramètre des objets d'une classe mère (généralement Object) ce qui permet de passer tout type à l'appel de la méthode. Il est aussi possible de définir des méthodes abstraites dans une classe mère et d'appeler ces méthodes devenues non abstraites dans les classes filles.

Figure 7.1. Paire



Nous avons dans l'exemple précédent une Paire constitué de deux éléments, le code suivant fonctionne :

```
Paire paire = new Paire(10, "toto"); //10 est un objet Integer //"toto" est un objet de type

String String s1 = (String) paire.getSecond(); //correct
```

Par contre le code suivant compile mais lève une erreur à l'exécution :

```
String s2 = (String) paire.getFirst(); //erreur à l'execution java.lang.ClassCastException
```

Le compilateur ne peut vérifier la cohérence des types de même on ne peut garantir , par exemple que sans gestion des exceptions que first et second soient de même type.

1.2. La généricité en java

Voici une nouvelle version de la classe Paire, qui utilise la généricité et deux types X et Y reçu en paramètre.

Figure 7.2. Paire2

Le code associé est :

```
public class Paire2<X,Y> {
 private X first;
 private Y second;
 public Paire2(X first, Y second) {
  super();
  this.first = first;
  this.second = second;
 }
 public X getFirst() {
  return first;
 }
 public Y getSecond() {
  return second;
 }
 public void setFirst(X first) {
  this.first = first;
 }
 public void setSecond(Y second) {
  this.second = second;
 }

}
```

Les <> permettent le passage des types abstraits, ici X et Y.*En java les méthodes peuvent recevoir des types différents de la classe.*

Lors de l'utilisation, il faut préciser les types :

```
Paire2<Integer, String> paire = new Paire2<Integer,String>(10, "toto");
  //10 est un objet Integer
  //"toto" est un objet de type String
  String s1 = paire.getSecond();
  //correct
```

A la compilation les erreurs de transtypage seront levées :

```
//String s2 = (String) paire.getFirst();
//erreur à la compilation Cannot cast from Intger to String
```

La généricité en java contredit le sens intuitif, la généricité repose sur la traduction homogène, il n'existe à l'exécution qu'une classe qui est paramétrée par toute les instanciations.

1.3. Généricité et sous-typage

Soit le code suivant ne compile pas, Vector<Object> n'est pas une classe mère de Vector<String> :

```
Vector<String> strings = new Vector<String>();
Vector<Object> objects = strings;
```

Par contre, avec `public class Vector2 extends Vector<String> {}` le code suivant compile :

```
Vector2 vector2 = new Vector2();
Vector<String> strings = vector2;
```

1.4. Généricité contrainte

Il est possible de contraindre les types paramétriques (borne supérieure) pour indiquer les services qu'ils doivent rendre.

Par exemple, public class Paire3<X extends Comparable<X> & Serializable,Y> {}, spécifie que le premier type doit implémenter l'interface Comparable et l'interface Serializable. Le mot clef extends désigne la contrainte et le mot clef & la composition de contraintes. *En java, la définition des contraintes peut-être récursive.*

1.5. Les jokers

Un dernier point pour vous permettre la lecture de l'API java : le super-type ou joker (?).

Les jokers permettent de relâcher les contraintes sur les types paramétriques pour rendre les méthodes plus réutilisables :

<?>
: Désigne un type inconnu.

<? extends A>
: Désigne un type inconnu qui est A ou un sous-type de A.

<? super A>
: Désigne un type inconnu qui est A ou un sur-type de A.

Le joker est utilisé à la déclaration et non à l'instantiation.

Maintenant, vous pouvez comprendre la déclaration de la méthode `public static <T extends Comparable<? super T>> void sort(List<T> list)` sort est une méthode de classe qui peut trier des List générique à la condition que le type générique implement Comparable avec un de ses super-types.

2. Exercices

Vous aller essayer de créer un classe générique Promo qui représente une promotion, puis l'utiliser avec une sous classe de Personne : PersonnePlus.

2.1. La classe générique Promo

Coder la classe générique Promo :

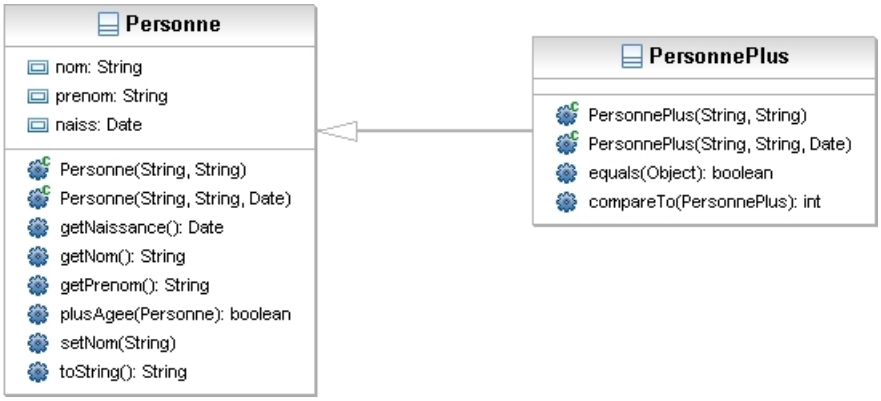
Figure 7.3. Promo

La méthode getPromo() renvoie une promo trié, vous pouvez utiliser la méthode sort de la classe Collections sur une copie de l'attribut promo.

2.2. La classe PersonnePlus

Créer une classe fille de la classe Personne qui implement Comparable et qui redéfinie la méthode equals de Object :

Figure 7.4. PersonnePlus



Vous pouvez utiliser pour compareTo et equals le code suivant :

```
public boolean equals(Object obj) {
  if (this == obj)
   return true;
  if (! (obj instanceof PersonnePlus))
   return false;
  PersonnePlus personne = (PersonnePlus) obj;
  return this.getNaissance().equals(personne.getNaissance()) &&
  this.getNom().equals(personne.getNom()) &&
  this.getPrenom().equals(personne.getPrenom());
 }


 @Override
 public int compareTo(PersonnePlus o) {
  int nomOk, prenomOk, naissOk;
  nomOk = this.getNom().compareTo(o.getNom());
  prenomOk = this.getPrenom().compareTo(o.getPrenom());
  naissOk = this.getNaissance().compareTo(o.getNaissance());
  if (nomOk == 0)
   if (prenomOk == 0)
    return naissOk;
   else
    return prenomOk;
  else
   return nomOk;

 }
```

2.3. Tester la promo

Tester la classe Promo avec des PersonnePlus.

Chapter 8. Annotations et Java Persistance API

Les annotations permettent de marquer certains éléments du code pour les munir de fonctionnalité à la compilation ou à l'exécution, elles représente une alternative au fichiers de configuration XML.

Les commentaire de la javadoc sont une forme d'annotation. Par exemple @Deprecated permet au compilateur et à la javadoc d'émettre un warning.

Nous allons utiliser les annotations pour mettre en oeuvre la persistance.

1. Présentation de Java Persistence API

La persistance est la sauvegarde et la restauration des données, dans le cas qui nous intéresse dans un SGBD. La JPA est une alternative au JDBC (Java DataBase Connectivity) qui met en oeuvre un mapping entre une classe et une table. Elle peut-être mise en oeuvre avec des fichiers XML ou des annotations. La JPA est principalement utilisée sur des serveurs d'application mais peut aussi être utilisé dans des applications autonomes.

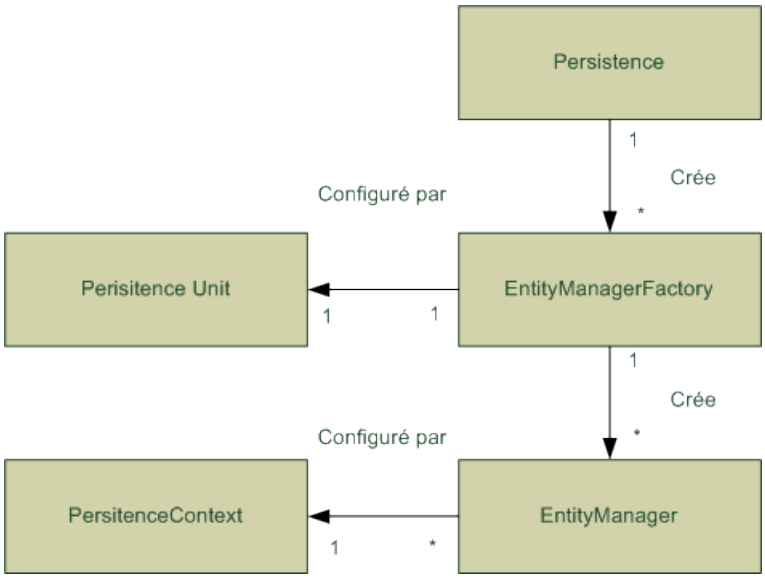
La JPA propose une nouvelle approche, l'idée est de réaliser une association entre une classe et une table ou requête (Object-Relational Mapping).

Les objets de la classes sont en correspondance avec la table, ce qui a pour avantage :

- d'avoir des POJO (Plain Old Java Object) donc pas d'inferfaces à implémenter,
- de disposer des mécanismes d'héritage et de serialisation.

La contrepartie est l'utilisation de nouvelles annotations et de nouveaux objets : EntityManagerFactory, EntityManager, Persistence unit, ...

Figure 8.1. Relations entre les différents concepts de la JPA



Les annotations sont un moyen d'ajouter des propriétés à certains éléments du langage Java(`http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`). Elles sont utilisées à la compilation ou à l'exécution pour automatiser certaines opérations. Vous connaissez déjà les annotations utilisées pour la génération de la javadoc (@pram, @see, @return, ...).

Voici quelques annotations utilisées pour la JPA:

@Entity
: Rend la classe utilisable par un EntityManager

@Table
: Nom de la table associée dans la base de données

@Column
: Colonne associée à un attribut

@Id
: qui définie un attribut comme constituant de la clef primaire

Il existe d'autres annotations qui permettent de maintenir les contraintes d'intégrité référentielles : @ManyToOne, @JoinColumn ou encore @NamedQuery qui permet de définir des requête JPA-SQL[1].

Nous allons illustrer ces concepts sur un exemple.

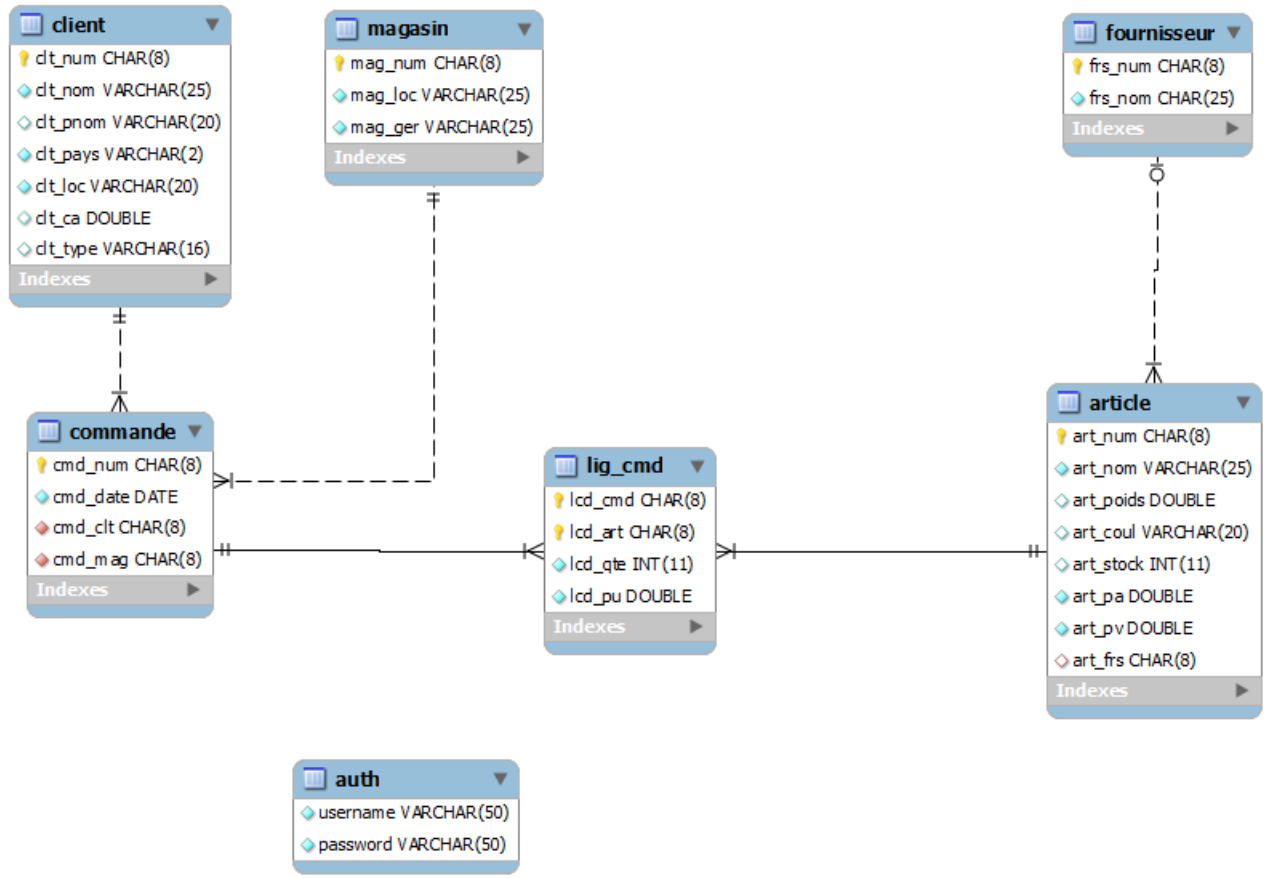
2. Mise en oeuvre

Nous allons utiliser netBeans pour mettre en oeuvre la JPA, la même opération est possible avec eclispe muni du plugin Dali.

2.1. Création de la base donnée

Utiliser dans wamp ou Uwamp le script script_tp_inf340.sql qui va créer la base tpdb avec comme propriétaire tpuser de mot de passe tppassword. Cette base contient les tables suivantes :

Figure 8.2. MLD de tpdb



[1] Encore un nouveau langage :-)

2.2. Configuration de netbeans

Dans l'onglet service créer une nouvelle connexion de base de donnée vers jdbc:mysql://localhost:3306/tpdb et comme utilisateur tpuser et mot de passe tppassword.

Vous pourrez maintenant gérer votre base de donnée depuis netBeans. Consulter par exemple le contenu de la table fournisseur.

2.3. Création des entités

Commencer par créer un projet de type application java, rajouter sur ce projet les librairies mysql et persitence.

Pour ce même projet choisir le JDK6.

Dans le paquetage par défaut créer une entité depuis une base de données et choisir votre connexion et les tables disponibles.

Vous devez avoir vos entités de créés, vous pouvez observer la présence des annotations (@id : clef primaire, @Clumn : la colonne, @OneToMany pour les clefs étrangères, ..).

```
@Entity
@Table(name = "fournisseur")
@NamedQueries({
    @NamedQuery(name = "Fournisseur.findAll", query = "SELECT f FROM Fournisseur f"),
    @NamedQuery(name = "Fournisseur.findByFrsNum", query = "SELECT f FROM Fournisseur f WHERE f.frsNum =
    @NamedQuery(name = "Fournisseur.findByFrsNom", query = "SELECT f FROM Fournisseur f WHERE f.frsNom =
public class Fournisseur implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "frs_num")
    private String frsNum;
    @Basic(optional = false)
    @Column(name = "frs_nom")
    private String frsNom;
    @OneToMany(mappedBy = "fournisseur")
    private Collection<Article> articleCollection;

    public Fournisseur() {
    }

    public Fournisseur(String frsNum) {
        this.frsNum = frsNum;
    }

    public Fournisseur(String frsNum, String frsNom) {
        this.frsNum = frsNum;
        this.frsNom = frsNom;
    }

    public String getFrsNum() {
        return frsNum;
    }

    public void setFrsNum(String frsNum) {
        this.frsNum = frsNum;
    }

    public String getFrsNom() {
        return frsNom;
    }

    public void setFrsNom(String frsNom) {
        this.frsNom = frsNom;
    }

    public Collection<Article> getArticleCollection() {
```

```
        return articleCollection;
    }

    public void setArticleCollection(Collection<Article> articleCollection) {
        this.articleCollection = articleCollection;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (frsNum != null ? frsNum.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not set
        if (!(object instanceof Fournisseur)) {
            return false;
        }
        Fournisseur other = (Fournisseur) object;
        if ((this.frsNum == null && other.frsNum != null) || (this.frsNum != null && !this.frsNum.equals(
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "javaapplication1.Fournisseur[frsNum=" + frsNum + "]";
    }

}
```

Vous trouverez aussi un fichier XML, C'est le fichier qui permet de configurer l'unité persistante, il contient :

name
: nom de l'entity manager

transaction-type
: JTA ou RESSOURCE_LOCAL

jta-data-source
: définit le nom JNDI (Java Naming and Directory Interface) global d'une source de données JTA (Java Transaction API) ou non, ici il n'est pas utilisé car notre ressource est locale.

mapping-file/jar-file/class
: L'ensemble des classes gérées par l'unité de persistance

properties
: Liste de propriété pour la configuration du fournisseur

```
  <?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/20
  <persistence-unit name="JavaApplication1PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>javaapplication1.Commande</class>
    <class>javaapplication1.Article</class>
    <class>javaapplication1.Client</class>
    <class>javaapplication1.Fournisseur</class>
    <class>javaapplication1.LigCmd</class>
    <class>javaapplication1.Magasin</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/tpdb"/>
      <property name="javax.persistence.jdbc.password" value="tppassword"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.user" value="tpuser"/>
```

```
    </properties>
  </persistence-unit>
</persistence>
```

Ce qu'il nous faut retenir c'est le nom de la persitence-unit, ici JavaApplication1PU car c'est par ce nom que nous allons l'utiliser.

2.4. Programme te test

Nous allons maintenant, utiliser la persistance, dans le main utiliser le code suivant :

```
EntityManagerFactory emf =Persistence.createEntityManagerFactory("JavaApplication1PU");
       //la créeation de la fabrique de gestionnaire d'entité
       EntityManager em = emf.createEntityManager();
       //la création du gestionnaire d'entité"
       em.getTransaction().begin();
       //Le debut d'une transaction
       Fournisseur f = new Fournisseur("T01", "toto");
       //la fabrication d'un nouvel objet qui sera mappé dans la base
       em.persist(f);
       //le mapping
       em.getTransaction().commit();
       //la fin de la transaction
       em.close();
       emf.close();
```

Vous pouvez observer depuis l'onglet services, les modifications.

Le code suivant repose sur l'utilisation d'une requête nommée sous forme d'annotation :

```
Query query = em.createNamedQuery("Fournisseur.findAll");
       List<Fournisseur> l =  query.getResultList();
       System.out.println(l);
```