الجمهوريـــــــــــــــة الجزائريـــــــــة الديمقراطيـــــــة الشعبيـــــة
وزارة التعليـــــــم العـــــــالي و البحـــــــث العلـــــمي
جامعة وهران للعلوم والتكنولوجيا « محمد بوضياف. »
République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur Et de la Recherche Scientifique
Université des Sciences et de la Technologie d'Oran- Mohamed BOUDIAF

Programmation Orientée Objet (POO)

Licence Informatique 2ème année

Dr Latifa DEKHICI

Maitre de conférences

Latifa.dekhici@univ-usto.dz

Objectif du module : Le cours de Programmation orienté objet a pour principal objectif d'introduire les étudiants à la programmation dans le paradigme orienté-objet avec le langage Java. Pour cela, ce support de court vient pour présenter les concepts de base de la POO ainsi que leur implémentation et mise en pratique dans Java. Il est constitué de 6 chapitres. Le premier présente le paradigme orienté objet. Le second chapitre aborde les bases du langage java. Dans le troisième chapitre, Le chapitre 4 détaille quelques relations entre les classes telle que la composition et explique l'influence de la dépendance sur le cycle de vie d'un objet. Dans le quatrième chapitre

Sommaire :

Chapitre 1

Paradigme Orienté Objet

1. Introduction

La Programmation orientée objet (POO) est une philosophie de programmation basée sur des concepts issus du monde réel. C'est ce que nous allons expliquer dans ce chapitre.

2. L'orienté objet : Philosophie

La **Programmation orientée objet** (POO) est une manière de résoudre un problème. C'est un paradigme, modèle de référence, une façon d'aborder un problème, une manière de penser et une concrétisation d'une philosophie de programmation.

3. Paradigmes de programmation

a) **Programmation impérative (procédurale) :** Dans la programmation procédurale un programme = algorithme + structure de données. Ses langages dédiés sont Pascal, C,…

b) **Programmation fonctionnelle :** Elle adopte une approche de la programmation beaucoup plus mathématique. Ses langages dédiés sont : Lisp, Matlab, Camel…

c) **Programmation logique :** Dans la programmation logique, la description d'un programme est sous forme de prédicats. Exemple de langage: Prolog.

d) **Programmation Orientée Objet :** Ses langages dédiés sont : java, Small talk…

4. Comparaison entre procédurale et POO

Jusqu'ici comment avons-nous l'habitude de programmer? Et Quelle est la structure de nos codes?

Représentation procédurale :	Représentation Objet :
Séparer les données des traitements. Subdivision basée sur les traitements.	Manipuler un ensemble d'objets qui interagissent entre eux. Subdivision basée sur les données.

5. POO

5.1 . Définition du monde Orienté objet

La programmation orientée objets définit un ensemble d'objets qui interagissent entre eux. En d'autres termes : tout est objet. Elle n'améliore pas subitement la qualité de l'application comme par magie. En revanche, elle aide à mieux organiser le code, à le préparer à des futures évolutions et à rendre certaines portions **réutilisables** pour gagner en temps et en **clarté**. C'est pour cela que les développeurs professionnels l'utilisent dans la plupart de leurs projets.

La POO se concentre dans la subdivision du problème sur les données.

- La **classe** est la définition d'un type **d'objets**.
- **L'encapsulation** est le principe de cacher les détails d'implémentation de l'extérieur par une interface.

5.2 . Notion d'objet et de classe

Dans le monde réel, on peut illustrer un objet par une voiture, un téléphone, un livre, etc.

En Informatique, un objet n'est ni une simple variable ni une simple fonction mais un mélange des deux. Il maintient son état (**paramètres**, caractéristique) dans des variables appelées **attributs**. On implémente son comportement à l'aide de fonctions appelées **méthodes**. Donc :

Objet=Un état +Un comportement

Le type d'un ensemble d'objets qui se ressemblent mais qui n'ont pas les mêmes valeurs est appelé Classe :

Classe= attributs + méthodes.

On peut considérer par exemple les classes suivantes :

a) **Bicyclette :**

Ses caractéristiques ou son **état** dépend de : Marque, type, nombre de vitesses, vitesse courante, couleur,…etc.

Alors que son **comportement** est de : rouler, tourner, accélérer, changer de vitesse, freiner,…

Les **objets** peuvent être maBicyclette , taBicyclette et touteBicyclette avec des valeurs d'états différentes.

b) **Personnage dans un jeu de combat :**

Il a **(état) :** un nom, une force, une localisation, une certaine expérience et enfin des dégâts…etc. Il peut **(comportement) :** frapper un autre personnage ; gagner de l'expérience ; se déplacer..etc. Lors d'une session de jeu, on peut créer plusieurs instances (personnages) différents du même type, chacun est appelé objet.

c) **Un rectangle :**

Un rectangle est caractérisé par ses attributs : longueur, largeur, couleur..etc. Son comportement est déclenché par des méthodes telles que : Dessiner, Calculer son périmètre, Calculer sa surface..etc.

5.3 . Cycle de vie d'un objet

Chaque objet informatique (instance) a un cycle de vie :

- Construction (en mémoire).
- Utilisation: changements d'état - comportements par exécution de méthodes
- Destruction.

6. Visibilité et Encapsulation

6.1 Visibilité d'un attribut ou d'une méthode

La **visibilité** d'un attribut ou d'une méthode indique à partir d'où on peut y avoir accès. On distingue plusieurs niveaux de visibilité, dont les plus importants sont : publique et privé.

a) **Visibilité publique :** La première, **public**, est la plus simple. Si un attribut ou une méthode est publique, alors on pourra y avoir accès depuis n'importe où, depuis l'intérieur de l'objet (dans les méthodes qu'on a créées), comme depuis l'extérieur. Quand on crée un objet, c'est principalement pour pouvoir exploiter ses attributs et méthodes. L'extérieur de l'objet, c'est tout le code qui n'est pas dans sa classe. En effet, quand on crée un objet, cet objet sera représenté par une variable, et c'est à partir d'elle qu'on pourra modifier l'objet, appeler des méthodes, etc.

b) **visibilité privée :** La seconde, **private**, impose quelques restrictions. On n'aura accès aux attributs et méthodes seulement depuis l'intérieur de la classe, c'est-à-dire que seul le code voulant accéder à un attribut privé ou une méthode privée écrit(e) à l'intérieur de la classe fonctionnera.

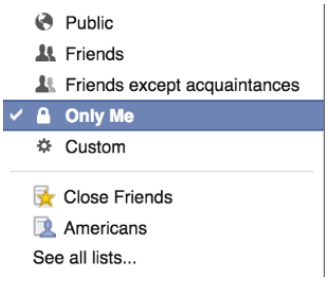
6.2 . Principe d'encapsulation

Le principe **d'encapsulation** est la manière qu'on peut interdire l'accès aux attributs dans une classe c.-à-d. réduire leur visibilité. L'un des avantages de la POO est que l'on peut masquer le code au **ré- utilisateur**, Le concepteur de la classe a englobé dans celle-ci un code détaillé qui peut être assez complexe et il est donc inutile voire dangereux de laisser l'utilisateur manipuler ces objets sans aucune restriction.

Ainsi, il est important d'interdire à l'utilisateur de modifier directement les attributs d'un objet ou les détails des méthodes.

Remarque :

L'encapsulation existe dans d'autre domaine tel que les réseaux sociaux où la visibilité d'une publication ou du profil entier peut être privée, destiné aux amis uniquement, aux amis et leurs amis ou publique. Aussi on peut dire que l'encapsulation est comme une boite noire du code.



7. Motivation et Avantages de la POO

- Permet de concevoir, maintenir et exploiter facilement de gros logiciels.
- Abstraction : objets informatiques proches de ceux du monde réel.
- Modularité et encapsulation permettent une maintenance et une sûreté plus effective et plus simple de projets.
- Accroît la réutilisabilité et la productivité.

Chapitre 2

Langage Java : Les bases

1. Introduction

Comme je programmais en C et bien avant j'utilisais Pascal, je me demandais comment la majorité des développeurs préfèrent java et je croyais que ça faisait partie d'une vague de mode en informatique. Peu à peu et en touchant java, j'ai pu comprendre d'où vient le succès de ce langage relativement récent. Java est un langage facile à apprendre et dont l'utilisation est amusante même quand il s'agit d'un simple TP en Licence. La programmation en orienté objet est implicite puisque tout est classe et objet. Et surtout en java, contrairement au langage c, on ne se soucie pas de pointeurs ou de références. Tout au long de ce support de cours, nous utilisons java pour la programmation orienté objet.

2. Historique de JAVA

Java est un langage de programmation orientée objet créé par James Gosling et Patrick Naughton, employés de Sun Microsystems, avec le soutien de Bill Joy , présenté officiellement en 1995 .La société Sun a été ensuite rachetée en 2009 par la société Oracle qui détient et maintient désormais Java. Java n'est pas un acronyme mais en argot américain signifie *café* qui est la boisson favorite de nombreux programmeurs.

La particularité et l'objectif central de Java est que les logiciels écrits dans ce langage doivent être très facilement portables sur plusieurs systèmes d'exploitation mobiles ou non, avec peu ou pas de modifications.

3. Installer les Outils De Développement

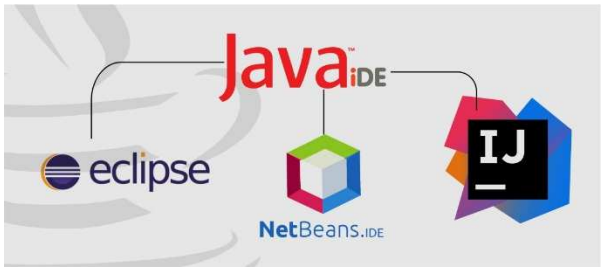
3.1 . Machine Virtuelle et JRE

L'un des principes phares de Java réside dans sa machine virtuelle : celle-ci assure à tous les développeurs Java qu'un programme sera utilisable avec tous les systèmes d'exploitation sur lesquels est installée une machine virtuelle Java.

Lors de la phase de compilation du code source, celui-ci prend une forme intermédiaire appelée **byte code** : interprétable par la machine virtuelle Java. Cette dernière porte un nom : on parle plus de **JRE** (Java **R**untime **E**nvironment).

3.2 . Editeur de Texte vs. Environnent de Développement Intégré

Nous pouvons saisir le code source dans un éditeur de texte puis le compiler sous Ms-DOS à l'aide d'une instruction java Mais il est préférable d'utiliser un outil de développement, ou **IDE** (**I**ntegrated **D**evelopment **E**nvironment) pour la saisie et l'exécution, Exemples : Eclipse, Netbeans et BlueJ.



3.3 . JRE et JDK

Avant de programmer en java le développeur commence par télécharger le kit de développement java (JDK) sur le site **d'Oracle**. Il faut choisir la dernière version stable. Il y a 2 environnements java:

1. **JRE** (**J**ava **R**untime **E**nvironment) contient tout le nécessaire pour que les programmes Java puissent être exécutés sur votre ordinateur ;
2. **JDK** (**J**ava **D**evelopment **K**it), en plus de contenir le JRE, contient tout le nécessaire pour développer, compiler…

Il y a plusieurs catégories de JDK :

a) J2SE (*Java 2 Standard Edition*) : pour développer des applications.
b) J2EE (*Java 2 Enterprise Edition*) : pour des applications web.
c) J2ME (*Java 2 Micro Edition*) : pour appareils portables, comme des téléphones portables.

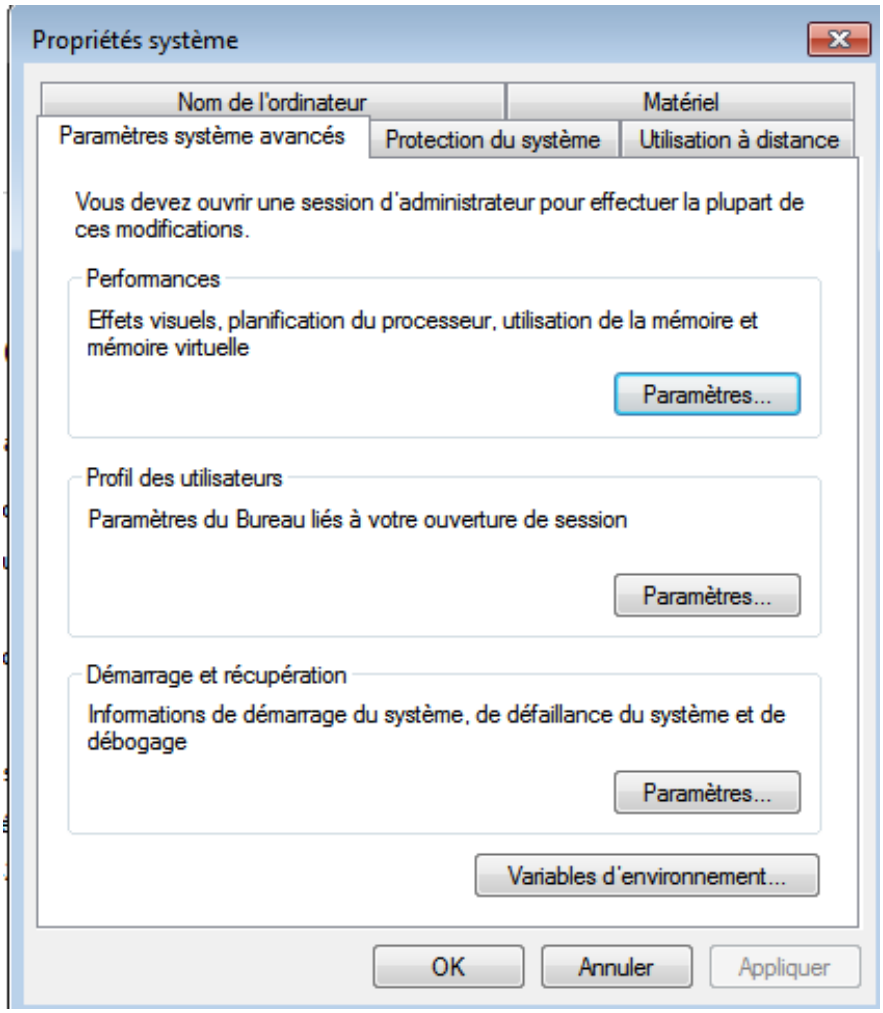
3.4 . Fichiers Java

- Les fichiers contenant le code source de programmes Java ont l'extension **.java.**
- Les fichiers précompilés correspondant aux codes sources Java ont l'extension **.class.**
- Un programme Java, codé sous Windows, peut être précompilé sous Mac et enfin exécuté sous Linux.
- La machine ne peut pas comprendre le byte code, elle a besoin de la machine virtuelle (JRE).

3.5 . PATH et CLASSPATH

A l'étape suivante de l'installation, on doit définir 2 *variables système* ***PATH*** *et* ***CLASSPATH***. Dans Windows, dans le *Panneau de configuration* ou dans le menu *Paramètres*, sur l'icône *Système|Avancé|Variables d'environnement*, on modifie la variable PATH pour ajouter « ; »+<le chemin de JDK>.


Panneau de configuration ▸ Système et sécurité ▸ Système
Fichier Edition Affichage Outils ?
Page d'accueil du panneau de configuration
Gestionnaire de périphériques
Paramètres d'utilisation à distance
Protection du système
Paramètres système avancés
Informations système générales
Édition Windows
Windows 7 Édition Intégrale
Copyright © 2009 Microsoft Corporation. Tous droits réservés.


Propriétés système
Nom de l'ordinateur
Matériel
Paramètres système avancés
Protection du système
Utilisation à distance
Vous devez ouvrir une session d'administrateur pour effectuer la plupart de ces modifications.
Performances
Effets visuels, planification du processeur, utilisation de la mémoire et mémoire virtuelle
Paramètres...
Profil des utilisateurs
Paramètres du Bureau liés à votre ouverture de session
Paramètres...
Démarrage et récupération
Informations de démarrage du système, de défaillance du système et de débogage
Paramètres...
Variables d'environnement...
OK
Annuler
Appliquer

4. Programmation Java

4.1 . L'essentiel

Tous les programmes Java sont composés d'au moins une classe. Exemple :

```
public class Essentiel { }
```

Le point de départ de tout programme Java est la méthode suivante :

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

On peut afficher des messages dans la console grâce à : `System.out.println("Ma Faculté de MI");` qui affiche un message avec un saut de ligne à la fin.

4.2 . Variables et Opérateurs

En Java, nous avons 2 types de variables :

1. variables de type simple ou « **primitif** » ;(A voir dans les sous sections suivantes)
2. variables de type **complexe** ou des « **objets** » ; (A voir dans le chapitre suivant)

4.2.1. Variables de type numérique

Les noms de types numériques simples commencent par une minuscule.

Nous avons les entiers :

byte (1 octet)	*byte temperature= 19 ;// Ceci est un exemple de déclaration*
short (2 octets)	*short vitesseMax= 32000 ;*
int (4 octets)	*int tempSoleil= 15600000 ; //La température est exprimée en kelvins*
long (8 octets)	*long anneeLumiere=9460700000000000L;*

Afin d'informer la machine virtuelle que le type utilisé est **long**, on doit ajouter un "**L**" à la fin du nombre, sinon le compilateur essaiera d'allouer ce dernier dans une taille d'espace mémoire de type entier.

Pour les réels on trouve float (4 octets) qui est utilisé pour les nombres avec une virgule flottante.

```
float pi = 3.141592653f;
```

Ou encore : `float nombre= 2.0f;`

On remarquera qu'on ne mets pas une virgule, mais un point ! Et on remarquera aussi que même si le nombre en question est rond, on écrit « .0 » derrière celui-ci, le tout suivi de « f ».

Le type double (8 octets) est identique à float, si ce n'est qu'il contient plus de chiffres derrière la virgule et qu'il n'a pas de suffixe.

```
double division= 0.33333333333333333333333333333333333333333334d;
```

Ici encore, on peut utiliser une lettre - le « d » juste pour parfaire la déclaration de la variable et allouer l'espace requis.

4.2.2. Variables stockant un caractère

La déclaration d'un caractère prend la forme suivante :

```
char caractere='A' ;
```

4.2.3. Variables de type booléen

La déclaration d'une variable logique prend la forme suivante :

```
boolean question=true;
```

4.2.4. Type String

L'utilisation n'est pas comme en C.En java, String est un type complexe , c'est-à-dire une classe mais particulière. Nous trouvons 4 manières de déclaration de chaine de caractères :

```
String phrase; phrase = "Mohamed Boudiaf"; //1ère méthode
String str = new String(); str = " Oran  ";//2ème  méthode
String chaine1 = " ouest ";           //3ème  méthode
String chaine = new String("Algérie !");//4ème  méthode.
```

Pour faciliter la réutilisation, il y a une convention de nommage. En fait, c'est une façon d'appeler les classes, les variables, etc. Il faut essayer de respecter au maximum cette convention. La voici :

- tous les noms de **classes** doivent commencer par une **majuscule** ;
- tous les noms de **variables** doivent commencer par une **minuscule** ;
- si le nom d'une variable est composé de plusieurs mots, le premier commence par une minuscule, le ou les autres par une majuscule, et ce, sans séparation ;

4.3 . Opérateurs arithmétiques

Les opérateurs numériques sont : +,-,*, / ,%, ++, += ,--,-= comme en C. L'opérateur + permet d'additionner deux variables numériques mais aussi de concaténer des chaînes de caractères.

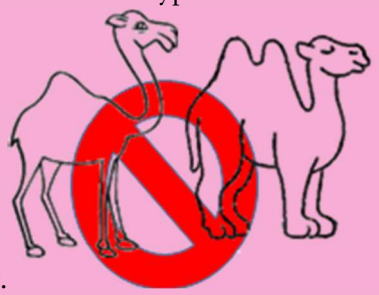
Voici quelques exemples de calcul sur des entiers :

```
int nbre1, nbre2, nbre3;  //Déclaration des variables
nbre1 = 1 + 3;            //nbre1 vaut 4
nbre2 = 2 * 6;            //nbre2 vaut 12
nbre3 = nbre2 / nbre1;    //nbre3 vaut 3 car / est entière ici
nbre1 = 5 % 2;            //nbre1 vaut 1, car 5 = 2 * 2 + 1
nbre1 = nbre1 + 1;// incrémente
nbre1 += 1; // incrémente
nbre1 --; // décrémente par 1
```

```
nbr1 /=2 ; // divise nbr1 par 2
```

4.4 Conversions, ou « cast »

Java est fortement typé , on ne peut utiliser un opérateur sur deux types différents sans appliquer



le cast.

La conversion est d'écrire le nouveau type entre parenthèse avant la variable à caster. Il est parfois impossible de faire la conversion entre certains types complexe (à voir en chapitre héritage).

a) De double à int

```
double nbre1 = 10, nbre2 = 3;
int resultat = nbre1 / nbre2;//impossible
int resultat = (int)(nbre1 / nbre2);//affiche 3
System.out.println("Le résultat est = " + résultat);//pour afficher
```

Sachez aussi qu'on peut tout à fait mettre des opérations dans un affichage :

```
System.out.print("Résultat = " + nbre1/nbre2); //affiche 3
```

b) D'un type int en type float :

```
int i = 2;
float t= (float)i; // =2.0f , important pour le considérer float
```

c) D'un type int en double :

```
int i = 123;
double z = (double)i; //  vaut 123.0
```

d) Et inversement :

Le cast d'un réel n'arrondit pas le nombre.

```
double x = 1.23; double y = 2.9999999;
int k = (int)x;          //k vaut 1
k = (int)y;              //k vaut 2
```

e) Priorité d'opération

Si on déclare deux entiers et qu'on mette le résultat dans un double, l'opérateur s'applique sur ces entiers avant d'être affecté

```
int nbre1 = 3, nbre2 = 2; double resultat1 = nbre1 / nbre2;
```

résultat1 est 1 au lieu de 1,5 car la division est entière selon les opérandes.

Le cast après le résultat ne servira à rien :

```
double resultat2 = (double)(nbre1 / nbre2); // résultat2= 1
```

Le cast sur l'un des opérandes entiers servira à avoir un résultat réel exacte.

```
double resultat3 = (double)nbre1 / nbre2; // résultat= 1.5
System.out.println("Le résultat est = " + resultat3);
```

Pour avoir un résultat correct, il faut caster chaque type au type approprié avant de faire l'opération.

f) Du numérique vers String

La méthode statique valueOf apparait dans beaucoup de classe comme la classe String et sert à convertir.

Il suffit de mettre la variable numérique à convertir comme paramètre de la méthode.(On reparlera de cette méthode pour la conversion des classes Color, Date….)

```
int i = 12;
String js = new String();
js = String.valueOf(i);
```

g) Du String vers numérique

Les méthodes clés pour les conversions entre types simple et complexe sont valueOf et intValue() , floatValue(), doubleValue().Voici un exemple :

```
int i = 12;
String js = String.valueOf(i);
int k = Integer.valueOf(j).intValue(); // car Integer.valueOf(j)
retourne un Integer(objet) et non pas un int
```

Aussi on peut écrire `int k=Integer.parseInt(j) ;`

Pour les autres types simples (respectivement float et double), nous utilisons respectivement **Float**, **Double** qui sont des types complexes (classes) comme Integer et qui contiennent les méthodes appropriées (valueOf() pour la conversion et floatValue(), doubleValue() pour extraire la valeur simple)

4.5 . Lire Entrées Clavier

a) Classe Scanner

Afin de lire les entrées en mode console et non pas sur des interfaces graphiques (à voir jtextArea par exemple en mode design sur IDE), une des manières les plus faciles à retenir par l'étudiant est l'importation de la classe Scanner en haut du fichier avant de définir le prototype de la classe , la création d'une instance du Scanner appelée par exemple *lecteur* puis la lecture de ses entrées dans le corps de la classe.

```
Import java.util.Scanner ; //importer la classe scanner
Public class Test{
Public static void main(String []arg){
// créer un objet lecteur :
Scanner lecteur = new Scanner(System.in);
System.out.println("Veuillez saisir un mot :");
// attend et lit la réponse de l'utilisateur sur la console :
String str = lecteur.nextLine();
System.out.println("Vous avez saisi : " + str);
}}
```

b) Lire les types numériques

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
double d = sc.nextDouble();
long l = sc.nextLong();
byte b = sc.nextByte();
```

Pareil pour float. Pour les caractères, il n'existe pas une méthode directe il faut lire une chaine de caractère avec nextLine et prendre le premier caractère avec **str.charAt(0) ;**

Remarque : Il faut utiliser deux nextLine() après une lecture des nombres pour vider le tampon.

4.6 . Conditions et boucles

Les conditions et les boucles sont comme en C sauf qu'en java, il existe une autre écriture intéressante de boucle appelée for each , à voir dans les paragraphes qui suivent.

4.7 . Tableaux

4.7.1. Tableau à une dimension

La syntaxe de déclaration est la suivante : *<type du tableau><nom du tableau> [] = {<contenu du tableau>};*

Exemple :

```
int tableauEntier[] = {0,1,2,3,4,5,6,7,8,9};
double tableauDouble[] = {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0};
char tableauCaractere[] = {'a','b','c','d','e','f','g'};
```

```
String tableauChaine[] = {"chaine1", "chaine2", "chaine3" , "chaine4"};
```

Un tableau vide de six entiers :

```
int tableauEntier[] = new int[6];
```

Ou encore

```
int[] tableauEntier2 = new int[6];
```

4.7.2. Tableaux multidimensionnels

```
int premiersNombres[][] = { {0,2,4,6,8},{1,3,5,7,9} };
```

4.7.3. Utiliser et rechercher dans un tableau

L'indice est mis entre crochet :

```
System.out.println(tableauCaractere[0]);
```

Le parcours jusqu'à sa longueur **length** :

```
for(int i = 0; i < tableauCaractere.length; i++)
      {  //traitement}
```

La syntaxe intéressante de boucle en java sur les tableaux, voire les listes (voir chapitre 4) est la suivante :

```
String universités[] = {"ustoMB", "ustHB", "univ-oran1"};
for(String mot : universités)
      System.out.println(mot);
```

4.8 . Méthodes de Classe

4.8.1. Pour Les Chaines de Caractères

La longueur d'une chaine comme la longueur d'un tableau est à la l'aide de la méthode **length.**

```
String chaine = new String("USTOMB!");
int longueur = chaine.length();   //Renvoie 7
```

La méthode **equals**() permet de vérifier (donc de tester) si deux chaînes de caractères sont identiques. C'est avec cette fonction que nous effectuerons les tests de condition sur les String. Exemple:

String str1 = new String("oran"), str2 = new String("wahran"); str3 = "ORAN";

```
if (str1.equals(str2))
System.out.println("Les deux chaînes sont identiques !");
else
System.out.println("Les deux chaînes sont différentes !");
```

Le programme affichera *Les deux chaînes sont différentes.*

```
if (str1.equals(str3))
System.out.println("villes identiques!");
else
```

```
System.out.println("villes différentes !");
```

Le programme affichera *villes différentes* car la méthode est sensible aux majuscules.

Le caractère à une position est par charAt et la sous chaine et avec subString.La première position est 0

```
String nbre = "1234567";
char carac = nbre.charAt(4); //Renverra ici '5'
nbr.subString(0,3) ; // sous chaine
```

4.8.2. Pour Math

A titre d'exemple, on peut citer la fonction aléatoire random qui diffère légèrement de celle du C++, la fonction absolue et exponentielle. **Math** est une classe à importer.

```
double X =  Math.random();// un nombre aléatoire entre 0 et 1
double abs = Math.abs(-120.25);//La fonction valeur absolue
double d = 2; double exp= Math.pow(d, 3);//La fonction puissance d3
```

Ces fonctions sont dites statiques c'est-à-dire ne dépendent pas d'une instance de la classe Math mais appartiennent à la classe. (à voir la notion de statique dans un prochain chapitre).

Exercices du chapitre 2

Exercice 1 :

1. Donner le nom complet de ces abréviations
JRE
JDK
2. Quelle est la différence entre JRE et JDK ?

Exercice2 : Ecrire les programmes qui :

1) affiche la concaténation de deux chaines
2) affiche le produit de deux entiers.
3) Calcul le reste de la division
4) Affiche admis ou ajourné après calcul de la moyenne.
5) affiche pour x allant de 1 à 10 la valeur réelle de 1/(x-5)

Corrections :

Exercice 1

3. Donner le Nom complet de ces abréviations	
JRE	JAVA RUNTIME ENVIRONEMENT
JDK	*JAVA DEVELOPEMENT KIT*
4. Quelle est la différence entre JRE et JDK ?	
L'environnement JRE contient le nécessaire pour l'exécution, Le kit développement JDK=JRE+ le nécessaire pour la compilation.	

Exercice2 :

```
//1)
import java.util.Scanner ;
public class Exercice2{
public static void main (String[] args)
{Scanner   lu=new Scanner(System.in) ;
String ch1=lu.nextLine( ) ;
String ch2=lu.nexLine( ) ;
System.out.println(ch1+ch2) ;
//2)
int x=lu.nexInt( ) ;
int y=lu.nextInt( ) ;
System.out.println(x*y) ;
//3)
System.out.println(x%y) ;
//4)
double note1=lu.nextDouble( ) ;
double note2=lu.nextDouble( ) ;
double moyenne=(note1+note2)/2 ;
if (moyenne>=10) System.out.println(« admis ») ;
else System.out.println(« ajourné ») ;
for(int i=1 ;i<10 ;++i)
if (i!=5)   System.out.println( 1/(double)(i-5)) ;//cast
obligatoire
}}
```

Class
Attributes
Methods

Chapitre 3
Classes et Objets

1. Introduction

Maintenant que nous avons compris comment écrire un programme simple de calcul, de lecture et d'affichage en java avec une seule classe, nous passerons dans ce chapitre à l'utilisation de plusieurs classes dont une est principale pour l'interaction avec l'utilisateur.

2. Notions de classes

2.1 . Définiton

Une classe est un type, modèle. Elle existe en fichier *.Java (code source) en mode de développement et en fichier *.Class lors de l'utilisation. **Le nom de la classe est le même que le nom du fichier.**

Exemple : si le nom du fichier est Velo.java , la classe serait : `public class Velo{}`

Sur un éditeur de texte, si on modifie le nom de la classe, on doit renommer le fichier .java. Si on est sur un IDE, le renommage est semi-automatique grâce à l'action ***refactor*** ou ***rename*** *selon l'IDE*

Exemple : `public class Moto{}` alors Moto.java

Dans les identificateurs, Les accents sont permis en java

2.2 . Contenu

Une classe contient :

a **Nom** : par convention commence en majuscule en java et porte le même nom que le fichier.

b **Attributs** : peuvent être de type simple (exemple : short numéro ;) ou un objet d'une autre classe, c'est-à-dire type complexe (exemple : Personne propriétaire ;// Personne est une classe). Les attributs peuvent aussi être publiques ou privés.

c **Méthodes** : peuvent être des fonctions retournant une valeur ou des procédures , on met alors void au début. Elles peuvent dépendre de l'objet ou non, on dit alors qu'elles sont statiques (voir paragraphes suivants)

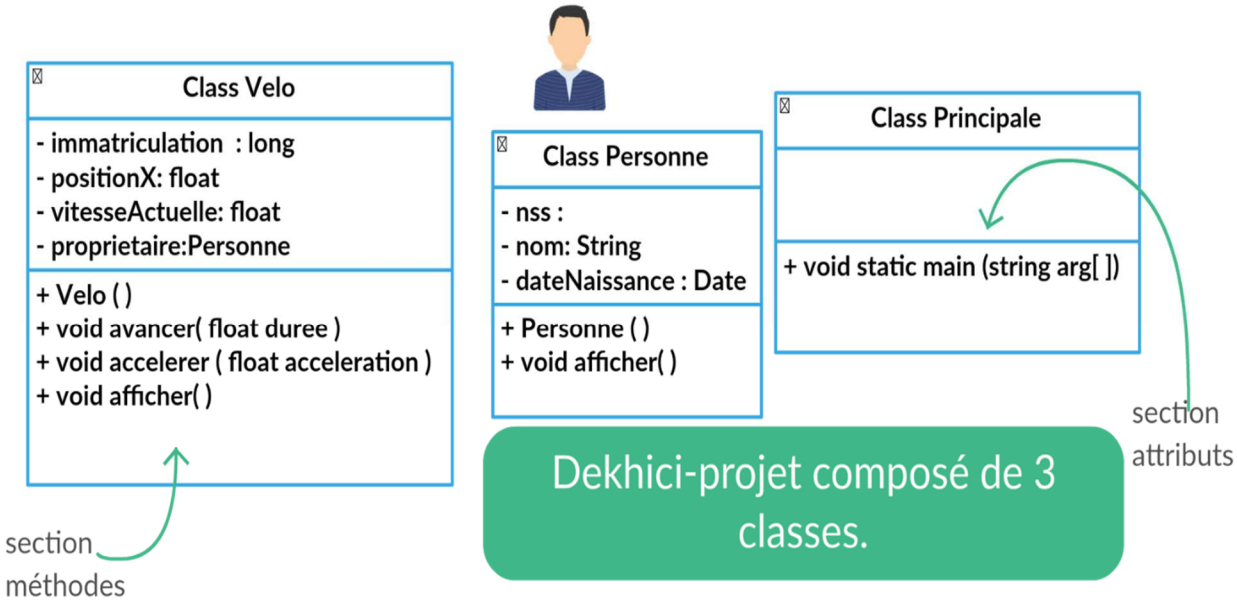
On aura alors dans le code :

```
public class Velo {

long immatriculation ;

Personne proprietaire ;// type complexe

float positionX,vitesseActuelle ;

}
```

on peut regrouper dans la même déclaration, les attributs de même type et modificateur (mot réservé de la visibilité, complétude…).

2.3 . Schématisation de la classe

On peut schématiser une classe en UML(Unified Modelling Language) comme dans le diagramme de classe ci-dessous. Beaucoup d'étudiants négligent la forme rectangulaire alors qu'elle est importante.



3. Visibilité

Le principe d'encapsulation (vu en chapitre 1) en java engendre trois niveaux de visibilités ***publiques, privées*** ou ***protégées*** qui sont respectivement définies dans la classe au moyen des mots-clés ***public, private*** et ***protected***.

3.1 .public

Le mot-clé ***public*** indique que les propriétés et méthodes d'un objet seront accessibles depuis n'importe où dans le programme principal. Signe en UML peut être :+. Voir figure précédente.

3.2 . private

Le mot-clé ***private*** (Signe en UML peut être : - , Voir figure précédente.) permet de déclarer des attributs et des méthodes qui ne seront visibles et accessibles directement que depuis l'intérieur même de la classe. C'est à dire qu'il devient impossible de lire ou d'écrire la valeur d'un attribut privé directement.

```
Velo v1 ; // v1 est un vélo ;
```

et vitesseActuelle est privée dans Velo

Dans une classe différente de la classe Velo on ne peut pas écrire :

~~`v1.vitesseActuelle=60 ;`~~

On ne peut pas modifier l'attribut privé. On ne peut pas connaitre la valeur de l'attribut privé.

~~`System.out.println(v1.vitesseActuelle) ;`~~

On peut par contre accéder aux propriétés privées via des méthodes (fonctions) dites ACCESSESSEURS (voir section suivante sur les méthodes)

3.3 protected

Le mot-clé **protected** ou protégé (Signe en UML peut être : #) (à voir en chapitre 5 qui concerne l'héritage).

Package : pour simplifier n'est que le dossier contenant une ou plusieurs classes

Le tableau suivant résume la couverture (visibilité) de chaque **modificateur**.

Modificateur	Class	package	Sous classe	monde
Publique	O	O	O	O
protégé	O	O	O	N
Sans	O	O	N	N
Privé	O	N	N	N

3.4 Implémentation de la classe avec les visibilités

Voici ci-dessous à quoi ressemble une classe avec modificateurs

```
public class Velo {

private long immatriculation;

private float positionX, vitesseActuelle  ;

private Personne proprietaire ;}
```

Un programme en java peut contenir plusieurs classes *principales c-à-d contenant la fonction main. Au moment de l'exécution, on peut choisir une de ces classes pour le lancer.*

4. Notion d'objet

Il reste beaucoup de notions à détailler concernant les classes. Mais ces détails ne peuvent pas être expliqués sans connaitre les objets.

Un objet est une instance d'une classe. Pour chaque instance d'une classe, le code est le même, seules les données des attributs (propriétés) changent.

Il est préférable de déclarer (plus utile) les objets dans une autre classe telle qu'une classe principale ou dans une classe générant des instances (par exemple veloFactory pour Velo). Ici, on déclare des vélos V1,V2 dans la classe principale

Exemple :

```
public class Principale{

public static void main(String [] args) {

Velo v1, v2 ; }}
```

5. Méthodes de classes

5.1 Définition

Une méthode est un regroupement d'instructions semblable aux fonctions et procédures sauf qu'elle s'exécute toujours sur un objet précis. Seules les méthodes de la classe ont droit à l'accès aux attributs (privés) de cette classe.

Elles sont aussi appelées fonctions par abus de langage (on utilise x.f au lieu de f(x) si x est l'objet dont dépend la méthode f). Une méthode peut être :

publique: accessible en dehors de la classe (les autres objets) c.à.d. visible dans l'interface.

privée: nécessaire à l'implémentation interne de la classe mais n'est pas visible dans l'interface.

Une méthode peut retourner quelque chose (fonction) ou ne rien retourner (void procédure).

Elle peut recevoir un ou plusieurs arguments entre parenthèses, qu'elle utilisera en cours de son exécution.

Exemple : dans notre classe Vélo , on peut avoir une méthode *avancer* dont voici le code :

```
public void avancer(float duree)
{positionX= positionX +vitesseActuelle*duree ;}
```

5.2 Méthodes statiques :

Une méthode statique est une méthode qui peut être appelée même sans avoir instancié la classe. Une méthode statique ne peut accéder qu'à des attributs et méthodes statiques.

Peut-on comprendre le résultat de l'instruction suivante dans la fonction main ?

```
public class Principale{
public static void main(String [] args) {
Velo v1,v2 ;
 avancer(30) ;// 30 est en mn.}}
```

Non car on ne sait pas quel objet est à avancer .On doit alors plutôt écrire, si v1 est un vélo :

```
v1.avancer(30);
```

par contre on peut utiliser ces fonctions sans se soucier de l'instanciation:

```
Math.random() ; String.valueOf(12); //static
```

Voici un autre exemple d'une fonction prédéfinie mais non statique :

```
monNom.length(); // non static car elle dépend de mon nom.
```

5.3 Signature de méthode

En java, on peut donner le même nom à deux méthodes différentes mais pas la même signature .

La signature d'une méthode est :

- Le nom de la méthode et
- La liste des types de ses arguments.

Pour invoquer une méthode, il faut toujours respecter sa signature.

Exemple les trois méthodes suivantes sont différentes même si elles ont le même nom:

```
produit(Matrice m1, int x) ;

produit (int x, int y) ;

produit (double x, Vector v) ;
```

Aussi :

```
int aleatoire(int x, int y) ; //est différente de

int aleatoire(int x)
```

5.4 Accesseurs ou mutateurs :Getter et setter

Comme on a vu dans une sous-section précédente concernant les attributs privés, on ne peut pas à partir d'une autre classe, accéder **directement** en lecture ou écriture à une propriété déclarée privée dans sa classe.

Exemple :

~~v1.immatriculation= 123;~~

~~System.out.print(v1.immatriculation);~~ `//ne fonctionne pas ailleurs.`

On **peut** alors déclarer dans Velo, 2 méthodes un setter setImmatriculation pour pouvoir modifier le numero et un getter getImatriculation pour avoir sa valeur :

```
public void setImmatriculation(int nouvelleImmatriculation)

{immatriculation= nouvelleImmatriculation ;}

public int getImmatriculation()

{return immatriculation ;}
```

On procède pareil pour les autres attributs privés si on veut mais pas forcément.

Dans la classe principale, on peut alors utiliser ces méthodes :

```
v1.setImmatriculation(123);

System.out.print(v1.getImmatriculation());
```

On peut ne pas coder de setPositionX pour obliger le passage via la méthode avancer(float durée) selon la vitesse et la durée. On peut ne pas mettre de setter pour un attribut dont la modification est interdite exemple *prénom* dans la classe *Personne*.

En IDE comme NetBeans , on peut insérer via un menu tous les accesseurs.

5.5 Constructeur

Un constructeur est une méthode particulière dont le rôle est de créer un nouvel objet.
Les constructeurs ont principalement trois tâches :

- réserver un emplacement en mémoire pour le nouvel objet,
- initialiser les attributs de cet objet, c'est-à-dire leur donner une valeur initiale,
- rendre cet objet.

a) Constructeur par paramètres

```
public Velo(long immatriculationP ,floatpositionXP,float
vitesseActuelleP)

  {immatriculation=immatriculationP  ;

  vitesseActuelle=vitesseActuelleP;

  positionX=positionXP;

}
```

On peut augmenter le nombre d'arguments dans un autre constructeur dans la même classe en ajoutant le propriétaire :

```
public Velo(long immatriculationP ,float positionXP, float
vitesseActuelleP, Personne pro)

{immatriculation=immatriculationP;

vitesseActuelle=vitesseActuelleP;

positionX=positionXP;

proprietaire=pro ;

}
```

b) Notion de this

Pour pouvoir utiliser le même identificateur de paramètre que l'attribut, on utilise le mot réservé **this** qui signifie ceci ou l'objet qui l'utilise :

```
public Velo(long immatriculation ,float positionX, float
vitesseActuelleP, Personne pro)

{this.immatriculation= immatriculation  ;

// Le premier est l'attribut de ceci (objet) et le deuxième est le
paramètre
```

```
vitesseActuelle= vitesseActuelleP;

this.positionX= positionX;

proprietaire=pro ;

}
```

Le mot this peut être utilisé pour appeler un premier constructeur (par exemple this()) dans un autre constructeur d'une différente signature.

c) Constructeur sans paramètres (par défaut du développeur)

```
public Velo()

{ immatriculation=1 ; // valeur par défaut

vitesseActuelle=1.0f ;

positionX=0.0f;

}
```

d) Constructeur par défaut de java

La **définition** d'un constructeur n'est pas obligatoire (si on ne souhaite pas initialiser les données membres par exemple) dans la mesure où un **constructeur** par défaut (appelé parfois **constructeur** sans argument) est défini par le compilateur **Java** si la classe n'en possède pas.

5.6 Destructeur

Le destructeur est aussi une méthode spéciale de la classe mais qui est **exécutée à la « mort » de l'objet**.

Le destructeur a pour rôle de libérer l'espace occupé par l'objet.

Si aucun destructeur n'est défini, c'est le destructeur par défaut qui est exécuté.

En Java la récupération de l'espace mémoire est implicite.

6. Instanciation des objets et utilisation

```
public class Principale {

public static void main (Sring[] args)

{

Velo v1=new Velo() ; //utilisation de constructeur par défaut.
```

```
    Personne p1= new Personne(1,"Isam");// supposant que Personne
    possède un tel  constructeur

    Velo v2= new Velo(1115,0.5,60); // on ignore le propriétaire

    System.out.println(v1.getNumero( )); //utilisation de getter

    v2.SetProprietaire(p1); //utilisation de setter

    v1.avancer(150.0);

    System.out.print(v1.getPositonX( ));

}}
```

7. Modificateurs de classe (ClassModifiers)

Une classe peut aussi avoir un modificateur. Une classe ne peut être privée que si sa déclaration est imbriquée dans une autre.

Modificateur	Rôle
abstract	la classe contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires pour ne plus être abstraite.
final	la classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées finales ne peuvent donc pas avoir de classes filles.
private	la classe n'est accessible qu'à partir du fichier où elle est définie
public	La classe est accessible partout

Les modificateurs **abstract** et **final** ainsi que **public** et **private** sont mutuellement exclusifs.

Marquer une classe comme final peut permettre au compilateur et à la machine virtuelle de réaliser quelques petites optimisations.

Exercices du chapitre 3

Exercice 1 : Qu'affiche cette fonction main ?

```
class Voiture  { public static int vitesse = 100;}

public class TestConduite {

        public static void accélérer (Voiture c)

                { c.vitesse += 30;  }

        public static void main(String[] args)

        { Voiture v1 = new Voiture();  accélérer (v1);
Voiture v2 = new Voiture();  accélérer (v2);  System.out.println("speed="+v2..vitesse); } }
```

Exercice 2

1. Schématiser en diagramme de classe un livre connu par sa côte, son titre, l'année d'edition, le nom de son auteur, nbrCopiesDisponibles ;
2. coder la classe
3. Definir le constructeur et les getters de tous les attributs et la méthode afficher()
4. Définir des setters du titre et de l'année d'édition.
5. Définir une méthode qui vérifie si deux livre sont du même auteur.
6. Définir deux méthodes emprunter() et rendre().
7. Instancier dans une classe principale deux livres en utilisant Scanner et tester les méthodes.

Exercice3

1. Schématisez en diagramme une classe permettant de représenter un point sur un plan. Un point est défini par son nom, sa **position** sur l'axe des X et des Y, ainsi que sa **couleur**.
2. Codez le schéma.
3. Codez le constructeur par paramètres et le constructeur par défaut
4. Ecrivez ensuite les différentes méthodes permettant de déplacer un point, de modifier sa couleur et d'afficher ses coordonnées.
5. Ecrivez dans une classe principale l'instanciation de deux points x et y et de calculer la distance entre eux.
6. Testez les autres méthodes.

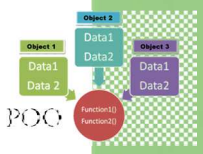
Correction :

Exo 1 :

Speed=160

Exo 3 :

Voir exemple chapitre suivant., page 30.



Chapitre 4

Relation entre Classes

1. Introduction

Jusqu'ici nous avons appris à implémenter une classe qui représente un concept et une classe principale qui sert à l'exploiter et à l'interaction avec l'utilisateur. Mais peut-on créer plusieurs classes et les utiliser dans une ou plusieurs classes principales et quel serait l'intérêt ?

2. Relations entre les différents objets

Une association (relation) exprime une connexion sémantique entre deux classes. Elle décrit une sémantique commune ayant un effet structurel.

Nous abordons dans ce chapitre les relations suivantes :

- L'association
- L'agrégation
- La composition simple ou multiple
- L'héritage (non multiple en java)

3. Dépendance ou non dépendance

3.1. Dépendance de classes

Quand une méthode de la classe A admet comme paramètre un objet de la classe B, on construit un objet local. La relation entre les deux objets est temporaire (le temps de l'exécution de la méthode).

Exemple

Soit une classe permettant de représenter un Cercle sur un plan. Il est défini par son centre (chapitre3.exercice2.Point) et son rayon qui est réel. La classe possède les méthodes pour calculer sa surface et son périmètre.

On peut importer la classe du package chapitre3.exercice2. au début du fichier Cercle en utilisant l'instruction :

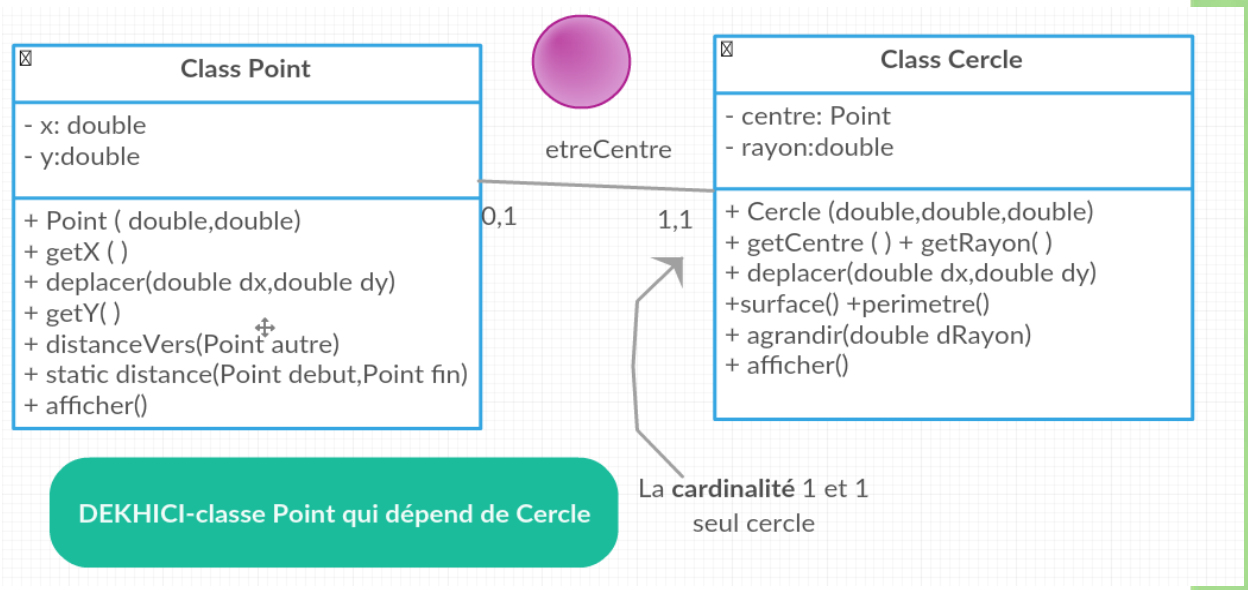
```
import
chapitre3.exercice2.Point ;
```

On peut réutiliser une classe existant dans un autre package à l'aide de **import**

Dans la modélisation, on évite de mettre des cardinalités 1,1 et 1,1 des deux côtés.

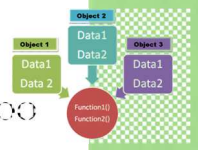
On peut définir le centre de 2 manières différentes. Pour la manière avec dépendance ci-dessous le diagramme de classe avec les cardinalités :

On suppose qu'un point créé pour être le centre d'un cercle, ne peut appartenir qu'à ce cercle. Une fois le cercle est détruit, son centre n'existera plus.




```
import chapitre3.exercice2.Point ;

import java.lang.Math ;

public class Cercle{

private Point centre ;

private double rayon ;

public double surface()

{return Math.pow(rayon,2)*Math.PI ;}

public double perimetre()

{return 2*rayon*Math.PI ;}

}
```

import java.lang.Math ;
pour utiliser la constante PI



Essayons

Pour les getter et setter, modifier un objet interne est inutile

```
public double getRayon() {return rayon ;}

public Point getCentre() {return centre ;}

public void setRayon(double nouveauRayon) {rayon= nouveauRayon ;}
```

Inutile de ramener un nouveau point de l'extérieur avec setter et écraser le centre actuel

~~`Public void setCentre(Point nouveauCentre) {centre=nouveauCentre ;}`~~

Comme *centre* dépend de *cercle* , il a été déclaré dans class Cercle:

```
    private Point centre;
```

et dans le constructeur de Cercle on ajoute l'instanciation du centre comme un nouveau point en ramenant seulement les caractéristiques en argument:

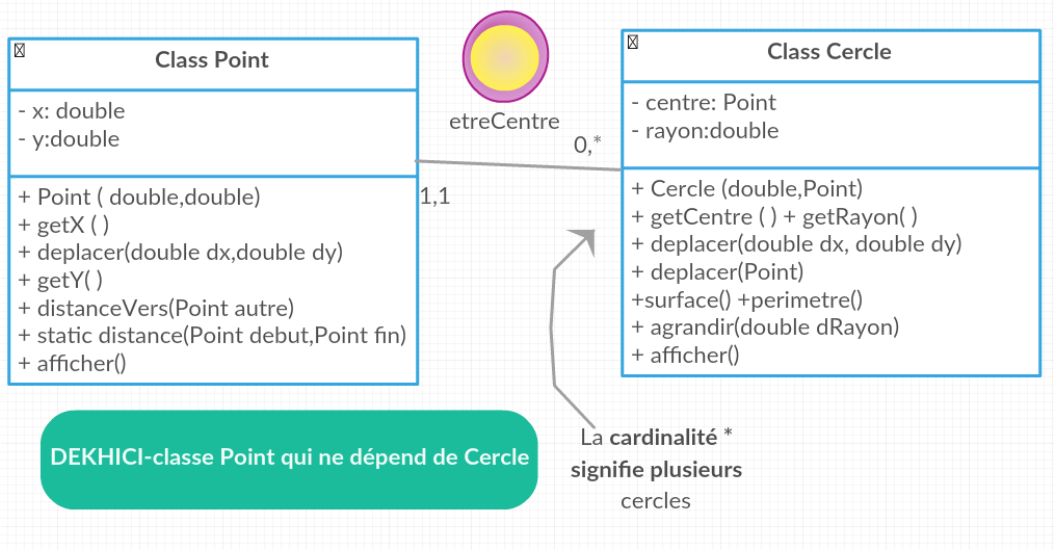
```
  public Cercle(double x, double y,double rayon)
  {
   centre=new Point(x,y,.....);
  This.rayon=rayon ;
}
```

Dans une classe principale l'instanciation d'un cercle c1 et le test des méthodes :

```
Public static void main(String arg[ ]){
Cercle c1=new Cercle( 3.12 , 43 ,13.66) ;   }
```

3.2. Association sans dépendance

Si on considère comme hypothèse que le cercle utilise comme centre un point déjà instancié, on aura une indépendance. Pour la deuxième manière sans dépendance ci-après le diagramme :



En ramenant en paramètre un point qui existe déjà, le constructeur deviendra :

```
public Cercle(Point p, double rayon)

{ centre=p;

This.rayon=rayon ; }
```

Il sera possible de le modifier avec un setter:

```
public void setCentre(Point nouveauCentre)
{centre=nouveauCentre ;}
```

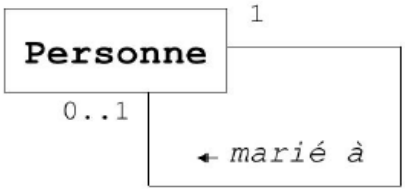
Il sera obligatoire d'avoir une création préalable d'un point p1 dans le programme principal avant de le considérer comme centre :

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

Point p1=new Point( 3.12 , 43 …..) ;

Cercle c1=new Centre( p1 ,13.66) ;

}
```

3.3. Association d'une classe sur elle-même

Il est également possible de définir une association entre une classe et elle-même.

Exemple :



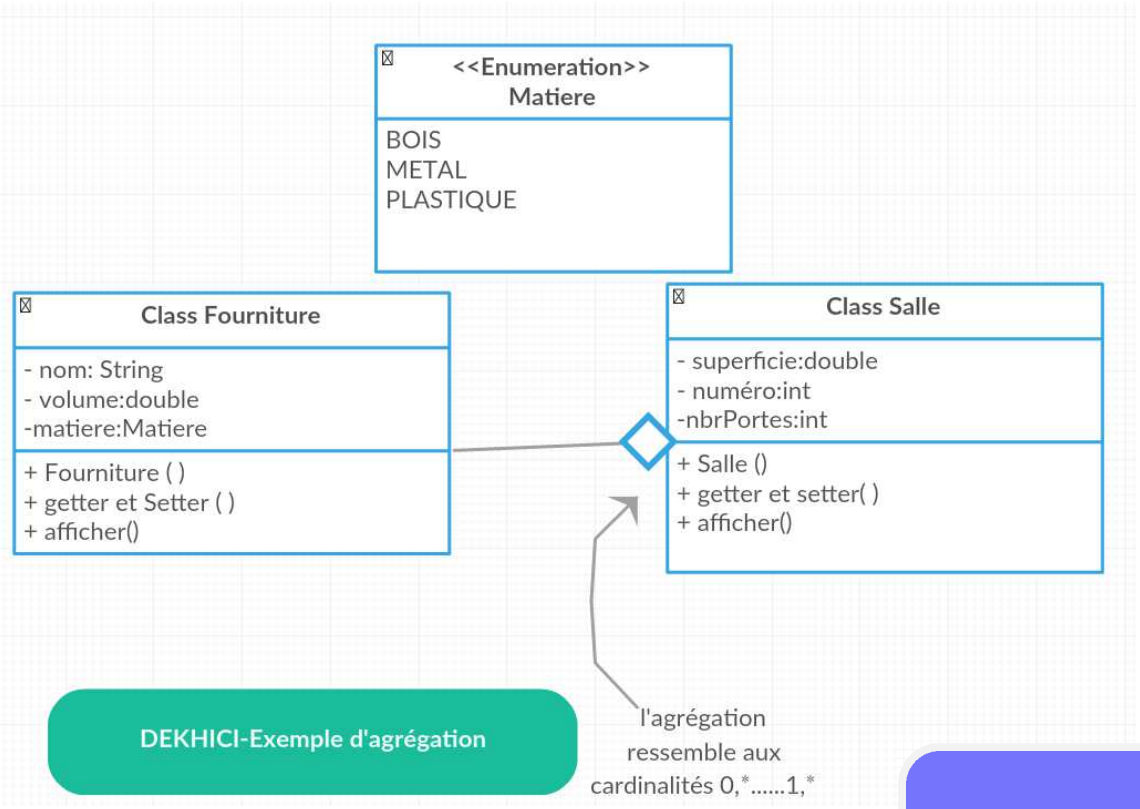

```
public class Personne{

private String nom;

private Personne epoux = null; // l'association est représentée par
la Personne epoux.

private boolean marie = false;

public Personne( String nom ){

this.nom = nom;

}

public void seMarier( Personne p ){

epoux = p; marie = true;

}

public boolean getMarie(){

return marie;

}

...

}
```

4. Agrégation et composition

4.1. Agrégation

L'agrégation exprime une association avec relation de subordination. Elle est représentée par un trait reliant les deux classes et dont l'origine (classe contenante) se distingue par un losange de l'autre extrémité (classe subordonnée). Une des classes "regroupe" d'autres classes. On peut dire que l'objet T utilise des instances de la classe T'.

Exemple :

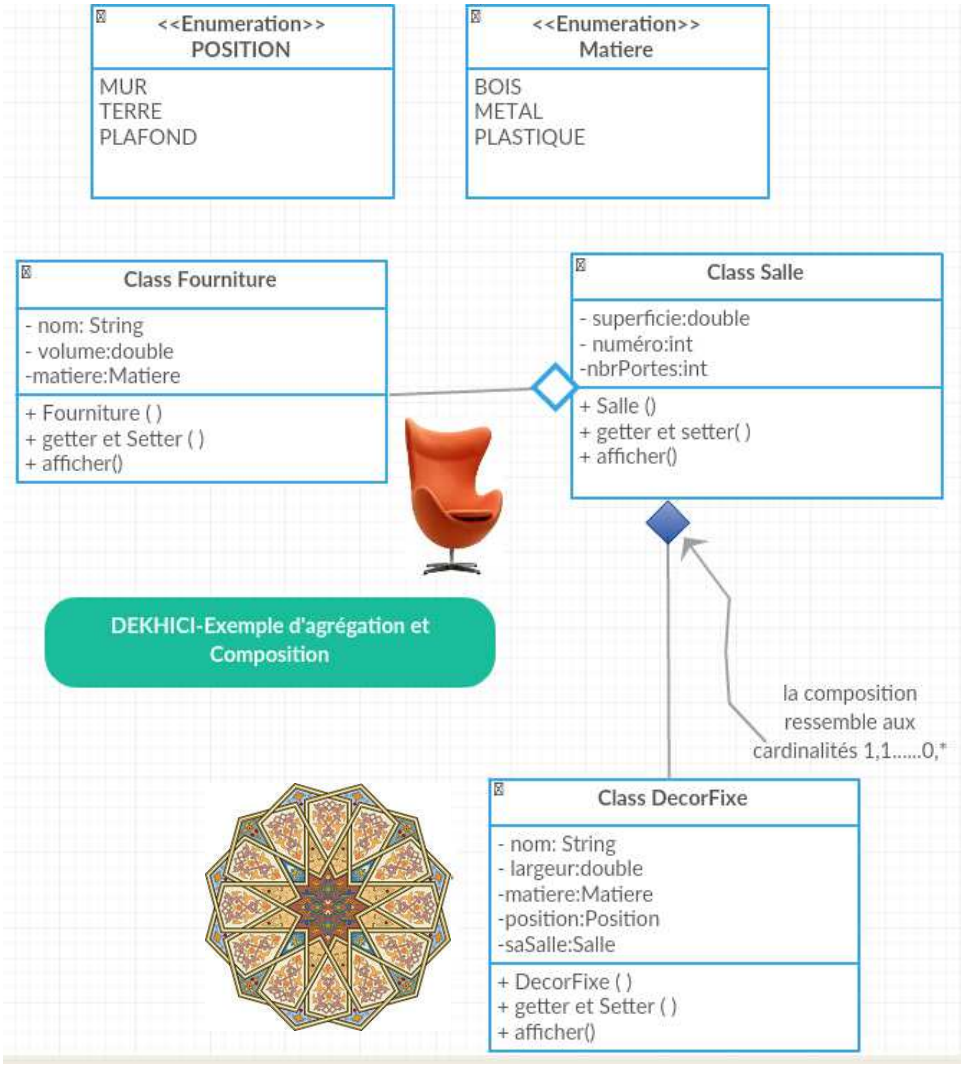


L'énumération en java est un fichier regroupant des constantes.

4.2. Composition

La composition est une variante plus forte de l'agrégation. En général, le cycle de vie des deux classes en relation est dépendant. Si la classe contenante est détruite, la classe subordonnée est détruite. Elle est représentée par un trait reliant les deux classes et dont l'origine (classe contenante) se distingue par un losange noir de l'autre extrémité (classe subordonnée).

Exemple : Soit le diagramme suivant. Une salle est composée de plusieurs décors Fixes et chaque objet de décoration appartient à une salle. Si la salle est détruite , les décors fixés seront détruit.



4.3. Codage de la composition vs. de l'agrégation

a) Déclaration d'une liste de contenants (agrégats et composants) :

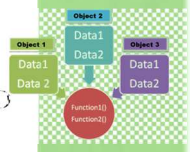
```
public class Salle{
```

L'agrégation est représentée par le vector ou List :

```
    private Vector<Fourniture>listeFournitures;
```

La composition est aussi représentée par un Vector :

```
    private Vector< DecorFixe >sesDecorFixes;
    private double superficie;
    private int numero,nbrPortes;
```



b) Initialisation de la liste de contenants :

Dans le constructeur chaque liste de type Vector doit être initialisée à un vector vide **new Vector<TYPE>(nbrElement)** d'une longueur connue(la mettre comme argument de Vector) ou pas. Par exemple ci-dessous un constructeur avec un numéro comme paramètre :

```
public Salle(int numero){

this.numero=numero ;

listeFournitures = new Vector<Fourniture>();

sesDecorFixes= new Vector<DecorFixe>();

superficie=9.0 ;nbrPortes=1 ;

}
```

c) Ajout de contenant

Pour les agrégats :

Pour remplir une salle par des fournitures, il suffit de **ramener de l'extérieur un agrégat existant** et l'ajouter à sa liste en utilisant la méthode add.

```
public void ajouterFourniture(Fourniture f ){

this.listeFournitures.add( f );// add est une méthode dans
Vector pour l'ajout

}
```

Pour les composants :

Pour remplir une salle par un decorFixe, il faut **ramener les paramètres nécessaires** de l'extérieur, **créer localement le composant** decorFixe dans salle et l'ajouter à la liste.

```
public void ajouterDecorFixes(String nom,double largeur,Matiere
matiere, Position position){

DecorFixe d=new DecorFixe(nom,largeur,matiere,position, this) ;

// le dernier this va initaliser saSalle de ce nouveau decor d
par la salle actuelle.

 this.sesDecorFixes.add(d);

}
```

d) Affichage et Traitement sur un contenant

Le getter de la liste entière ne diffère pas des autres getters :

```
public Vector<DecorFixe>getSesDecorFixes(){

return sesDecorFixes;

}
```

En cas d'agrégation, il est facile aussi de modifier la liste entière par un setter.

Pour afficher le i^{ème} élément de la liste , il suffit d'écrire :

```
 public void afficherDecor(int i)

{system.out.println(this.sesDecorsFixes.get(i).afficher() ;}
```

e) Parcours de la liste des contenants

Enfin, pour afficher tous les attributs de Salle :

```
  public void afficher()

{System.out.print(« numéro= »+numéro+ «superficie= » +superfici
e+ «nbr. De Portes= »   +nbrPortes+ « liste de ses
Fournitures :») ;
```

voici la première manière pour parcourir une liste de composants ou d'agrégats:

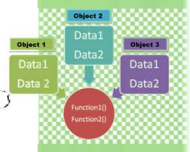
```
for(int i=0 ;i<listeFournitures.size() ;++i)  // size =taille
vector

{ listeFournitures.get(i).afficher() ;}

System.out.print(« ses décors fixes :») ;
```

voici la deuxième manière pour parcourir une liste de composants ou d'agrégats:

```
for(DecorFixe d :this.sesDecorFixes)  // une boucle for each
sans index

{ d.afficher() ;}

}
```

Dans la classe Principale, on veut créer une salle :

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

Fourniture chaiseCONFOJAVA=new Fourniture
("chaise",125000,Matiere.BOIS);// constante BOIS est utilisée
ici.

Salle salleConference=new Salle(154);

salleConference.setNbrPortes(2);
```




```
        // ajout d'un agrégat.
        salleConference.ajouterFourniture(chaiseCONFOJAVA);
salleConference. ajouterDecorFixes(cadre,30,matiere.METAL,
Position.MUR ) ;
salleConference.afficher();
}
```

4.4. La composition multiple

La composition peut être multiple : la classe contenante a alors plusieurs classes subordonnées.

Exemple :

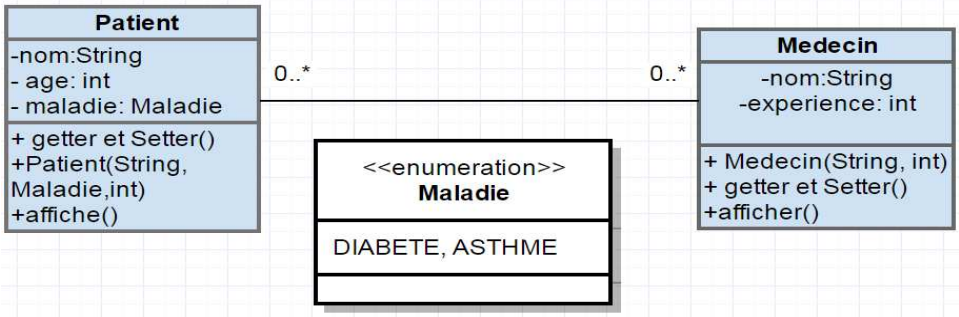
```
public class Vehicule
{
private Moteur moteur; // la composition multiple est représentée par moteur
private Roue[] roues; // le tableau roues
private Caisse caisse; // caisse
private Habitacle habitacle; // et habitacle
public Vehicule(){ //composition
moteur = new Moteur();
roues = new Roue[4];
caisse = new Caisse();
habitacle = new Habitacle();
}
public Vehicule( Moteur moteur,Roue[] roues, Caisse caisse, Habitacle habitacle){//agrégation
this.moteur = moteur;
this.roues = roues;
this.caisse = caisse;
this.habitacle = habitacle;
```

}

...

Exercices du chapitre 4

Exercice1 : Soit le diagramme suivant. Un Médecins soigne plusieurs Patient et chaque patient peut être chez plusieurs medecins. On suppose que Medecin et Maladie sont déjà programmées. Donc on s'intéresse seulement à Patient ainsi que ses médecins et à une classe principale.

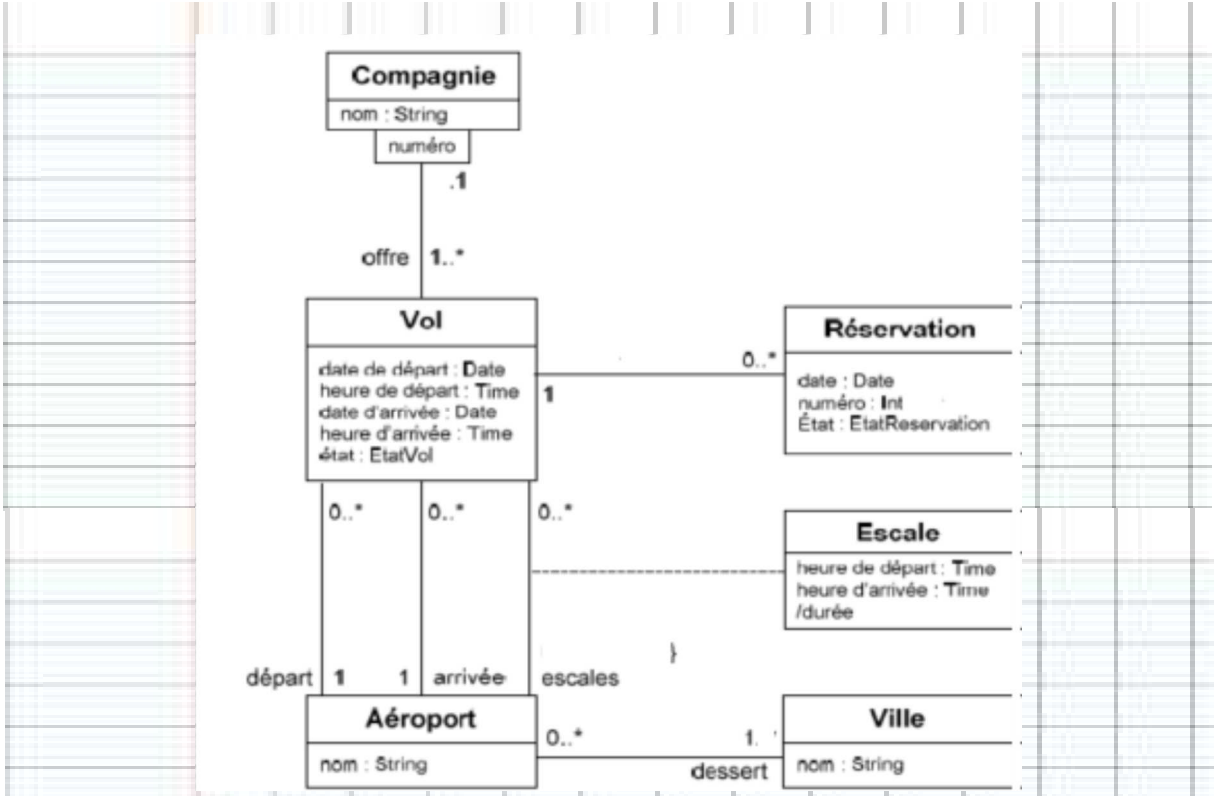


1. Ecrire les instructions qui permettent de déclarer la classe Patient et tous ses Attributs
2. Ecrire le constructeur avec paramètres de Patient
3. Ecrire le setter de age
4. Ecrire la méthode afficher() de Patient
5. Ecrire la méthode qui ajoute un nouveau médecin au Patient.
6. Ecrire les getter et setter de tous les attributs de Patient et le constructeur de Médecin
7. Dans une procédure main de la classe principale , lire 2 variables ceNom, cetAge.
8. Créer un nouveau Patient p1 avec les 2 variables précédentes et la maladie DIABETE.
9. Ecrire les instructions qui ajoutent à p1 un médecin avec les propriétés (« dr Doctor », 35).
10. Afficher p1.

Exercice 2. Une compagnie aérienne (ex : air Algérie) offre plusieurs vols. Le vol part d'un aéroport(exemple : Ahmed Ben Bella) d'une ville(ex :Oran) vers un autre aéroport (exemple: Mohamed Boudiaf) d'une ville (ex :Constantine) et peut faire escale dans un ou plusieurs aéroports (ex :Houari Boumediene d'Alger)pour une certaine durée. Plusieurs réservations (confirmées ou pas) peuvent se faire sur un vol. Le diagramme de classes ci-dessous montre les relations entre les classes.

1. Ajoutez au diagramme les méthodes nécessaires (réserver(), confirmer()…) et de base.

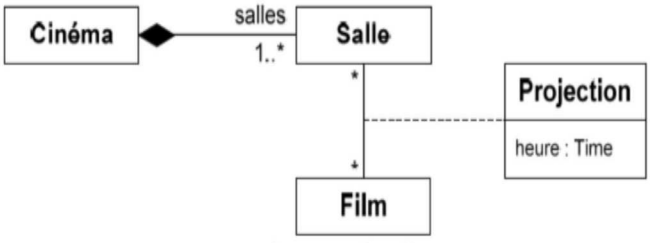
2. Codez en java ces classes et Testez en une classe principale des exemples d'instances.

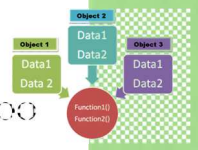


Exercice 3:

Une salle de cinéma ayant un nom (ex : cinémathèque Larbi Ben Mhidi), se situant dans une ville est composée de plusieurs salles ayant des codes(ex :MIRA 03). Une projection dans le calendrier est programmée dans une salle de cinéma et concerne un film (titre, durée, genre) (ex :Essaha, 1h ,Comédie musicale).

1. Ajoutez au diagramme les attributs, les méthodes nécessaires (au cinéma projeter(….)…) et de base (afficher….).





Correction :

Exercice 1 :

1. Ecrire les instructions qui permettent de déclarer la classe Patient et **tous** ses Attributs y compris le(s) medecin(s).

```
public class Patient{

private String nom ;

private int age ;

private Maladie maladie;

private Vector<Medecin>sesMedecins; //ou une des implémentations de
List  tel que ArrayList

// respect de convention (majuscule, minuscule)
```

2. Ecrire le constructeur avec paramètres de Patient

```
public Patient( String nom ,int a ,Maladie m)

{this.nom=nom ;age=a;maladie=m; // montrez l'utilité de this

sesMedecins= new Vector<Medecin>(); // ou ArrayList ou LinkedList
```

3. Ecrire le setter de age

```
public void setAge(int nouvelAge)

{age= nouvelAge ;}
```

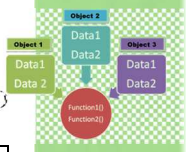
4. Ecrire la méthode afficher() de Patient

```
public void afficher()

{System.out.print(«  « +nom+ » »+age+ » »+maladie) ; // message » »
pas important

for (Medecin m :sesMedecins)

{m.afficher();}
```

5. Ecrire la méthode qui ajoute un nouveau médecin au Patient.

```
public void ajouterMedecin(Medecin m)  // médecin existant déja

{sesMedecins.add(m) ;}
```

6. Ecrire les getter et setter de tous les attributs de Patient et le constructeur de Médecin




```
// voir getter et setter
```

7. Ecrire dans une procédure main de la classe principale , les instructions qui permettent de définir un scanner et de lire 2 variables ceNom, cetAge.

```
Public static void main(String[] args)

{Scanner clavier=new Scanner(System.in);

String ceNom= clavier.nextLine();

String cetAge = clavier.nextInt();
```

8. Ecrire l'instruction qui crée une nouveau Patient p1 avec les 2 variables précédentes. et la maladie DIABETE.

```
Patient p1=new Patient(ceNom,cetAge, Maladie.DIABETE) ;
```

9. Ecrire les instructions qui ajoute à p1 un médecin avec les propriétés (« dr Doctor », 35).

```
Medecin m1=new Medecin(« dr Doctor », 35);

p1.ajouterMedecin(m1);
```

10. Afficher p1.

```
p1.afficher() ;
```

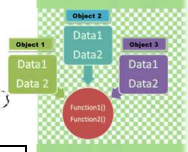
Exercice 2 :

```
import java.sql.Time;
import java.util.Date;
import java.util.Vector;
```

```
/**
 *
 * @author Dekhici
 */
public class Vol {
private Date dateDepart,dateArrivee;
private Time heureDepart,heureArrivee;
private Aeroport aeroportDepart,AeroportArrivee;
  private EtatVol etatVol;
  private Vector<Escale>listeEscales;
  private Vector<Reservation>listeReservations;
  private Compagnie compagnie;
   public Vol(Compagnie c,Date dateDepart,Date dateArrivee,Time heureDepart,Time
heureArrivee,Aeroport aeroportDepart,Aeroport AeroportArrivee)
   {this.compagnie=c; // this est facultative ici
this.dateArrivee=dateArrivee;
   this.dateDepart=dateDepart;
   this.heureArrivee=heureArrivee;
   this.heureDepart=heureDepart;
   this.aeroportDepart=aeroportDepart;
   this.AeroportArrivee=AeroportArrivee;
listeEscales =new Vector<Escale>();
listeReservations=new Vector<Reservation>();
etatVol=EtatVol.AHEURE;//constante AHEURE de l'ensemble <enum>
```

```
    }
   public void ajouterEscale(Aeroport aeroport,int duree ,Time heureDepart,Time
heureArrivee)
{ Escale e=new Escale(aeroport,duree,heureDepart,heureArrivee);
listeEscales.add(e);
    }

            //getter et Setter
   public void afficher()
{ System.out.println(" "+this.aeroportDepart.getNom()+
" "+this.dateDepart+
" "+this.heureDepart+
" "+this.AeroportArrivee.getNom()+
// ajouter les autres champs
                        " "+this.etatVol.toString()//print peut imprimer l'objet
sans toString
);
for (Escale e:this.listeEscales)
{ e.afficher();
      }
for (Reservation r:this.listeReservations)
{ r.afficher();
      }

   }
void reserver(int i) {
   {Reservation r=new Reservation(i,this);// son vol est this
   this.listeReservations.add(r);

   }   }}
package GestionVol;
import java.util.Calendar;
import java.util.Date;
/**
 *
 * @author Dekhici
 */
public class Reservation {
private Date dateReservation;
   private int num;
   private Vol vol;
   private boolean etatRes;//ou avec enum
   public Reservation(int n,Vol v)
   {        dateReservation=Calendar.getInstance().getTime();
   num=n;
   vol=v;
   etatRes=false;
}
   public void confirmer()
   {etatRes=true;}

   //getter et setter

    void afficher() {
// à remplir
    }
}
```

```
public class Aeroport {
private String nom;
    Ville ville;
     public Aeroport(Ville ville,String nom)
    {this.nom=nom;
    this.ville=ville;}
public String getNom() {return nom;}
public void setNom(String nouveauNom){this.nom=nouveauNom;}
```



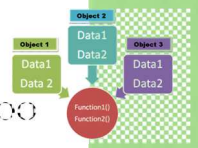

```
    public Ville getVille() {return ville;}
     public void setVille(Ville nouvelleVille){this.ville=ville;}

void afficher() {
    // à remplir
    }


}
```

```
public class Compagnie {
private String nom;
private int num;
public Compagnie (String nom,int num)
   {this.nom=nom;
   this.num=num;
   }
   //getter et setter
   //afficher
}
```

```
public class Escale {
private Aeroport aeroport;
private int duree ;
private Time heureDepart,heureArrivee;
public Escale(Aeroport aeroport,int duree ,Time heureDepart,Time heureArrivee)
{ this.aeroport=aeroport;
    this.duree=duree;
    this.heureArrivee=heureArrivee;
    this.heureDepart=heureDepart;
}

    void afficher() {
        // à remplir
    }
}
```

```
public enum EtatVol {
ANNULE,
   RETARDE,
   ARCHIVE,
   AHEURE;
}
```

```
import java.sql.Date;
import java.sql.Time;
public class Gestion {
public static void main(String[] args) {
        Compagnie c1=new Compagnie("airAlgérie",1);
        Ville v1,v2,v3;
        v1=new Ville("Oran");
        v2=new Ville("Constantine");
        v3=new Ville("Alger");
v1.ajouterAeroport("ahmed BenBella");
v2.ajouterAeroport("Mohamed Boudiaf");
v3.ajouterAeroport("Boulmediane");
        Aeroport a1,a2;
        a1 =v1.getListAeroport().get(0);// le premier aeroport de la ville Oran
        a2=v2.getListAeroport().get(0);// le premier aeroport de constantine
        Date d1;
        d1=Date.valueOf("2016-02-02");
        Time t1,t2,t3,t4;
t1=Time.valueOf("08:00:00");t2=Time.valueOf("11:00:00");
        Vol vol1 = new Vol(c1,d1,d1,t1,t2,a1,a2);
```



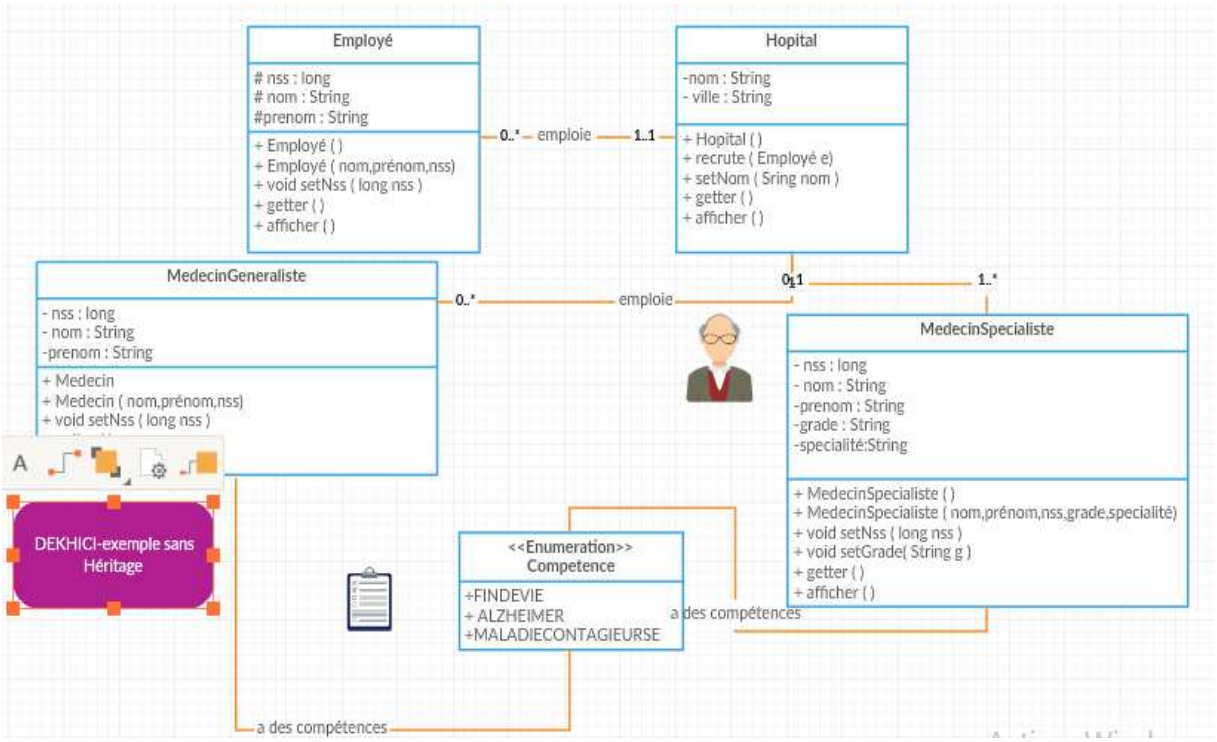

```
vol1.ajouterEscale(v3.getListAeroport().get(0)
,1,Time.valueOf("09:00:00"),Time.valueOf("10:00:00")); //les declarer avant,
possible
vol1.afficher();
        vol1.reserver(12);
        vol1.reserver(13);
vol1.afficher();
        //vol1.getListReservation().get(1).confirmer();

        }  }
```

Chapitre 5 Héritage et Polymorphisme

1. Héritage

1. But de l'héritage

Soit le diagramme de classe qui décrit les employés d'un établissement de santé par exemple un hôpital :

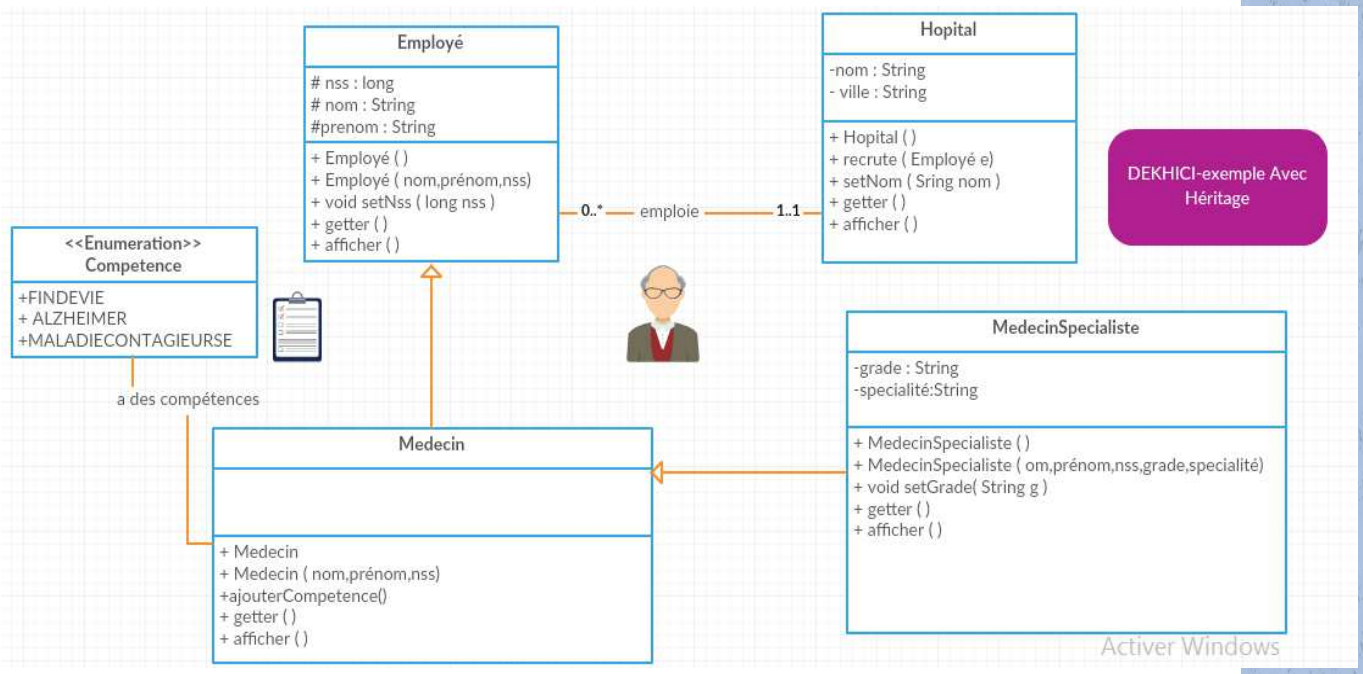


On remarquera que pour chaque concept (médecin généraliste, médecin spécialiste et employé ordinaire), une classe a été créée avec redondance de définition d'attributs. Aussi la relation entre la classe Hôpital et les autres classes engendra 3 listes selon le type d'employé. Si on veut ajouter la date de naissance ou un comportement sous forme de méthode, nous serons obligés de le coder partout. En terme de modélisation de réalité, le médecin n'est-il pas un employé ? Pour ces raisons, l'héritage en POO a été défini.

2. Définition

La relation d'héritage indique que la "sous-classe" (classe fille) est une spécification de la "super-classe" (classe mère). La classe fille hérite de tous les attributs et méthodes de la classe mère, on va du général au particulier.

En UML, Elle est représentée par un trait continu reliant les deux classes et dont l'origine (classe mère) se distingue de l'autre extrémité (classe fille) par un triangle vide. Ci-dessous le nouveau diagramme de classe avec héritage :



Traduction Java :

> En java, l'héritage est représenté par <Fille>**extends**<Mere>

L'héritage est représenté par le mot clé extends

```
public class Medecin extends Employé{

}
```

3. Protection des attributs

Il est impossible d'utiliser dans la classe fille des attributs privés de Mere . Pour cela, il faut rendre les attributs de la classe Mere protected au lieu de private pour les voir dans les classes filles et ne pas les voir ailleurs.

Dans le diagramme UML, cette protection est représentée par #

4. Héritage Multiple

L'héritage mulltiple n'est pas permis en java

Exemple :

Il n'est pas possible d'écrire:

```
public class MedecinResident extends
Medecin,Etudiant{

} ou

public class Chat extends Carnivore,Domestique{

}
```

L'héritage des méthodes et attributs publiques et protégés est transitif Grand-Mere-> Mere ->fille

Réflexion : Pourquoi l'heritage multiple n'est pas permis en Java alors qu'il l'est en C++ ? Quelle solution adopter pour résoudre ce problème de modélisation? Voir un peu plus loin en polymorphisme et chapitre suivant.

5. Mot réservé Super

Le mot clé **super** permet de récupérer les éléments (Méthodes) de la classe Mere. Il remplace le nom de la classe Mere. C'est comme si l'on dit Maman au lieu de prononcer son prénom.

Exemple :

Dans le Constructeur sans paramètre de Médecin, on utilise super() au lieu de dire Employé() , le constructeur de la classe mère :

```
public Medecin(){
super();
//complément de la construction
sesCompetences=new Vector<Competence>();}
}
```

Pour utiliser Employé.afficher(), on écrit super.afficher() qui signifie afficher comme la classe Mère :

```
public void afficher(){
super.afficher();
for(Competence c:sesCompetences)
System.out.println(String.valueOf(c));}
```

6. Utilisation de l'héritage

L'un des principes de la programmation orientée objet est l'utilisation parfois d'objets généralisés (ayant des attributs et des comportements qui se ressemblent) sans se soucier des détails. Par exemple ci-dessous, on cherche à recruter des employés et les afficher. Ces employés peuvent être des médecins ou des médecins spécialistes.

```
public static void main(String arg[])
{ Hopital h=new Hopital();
Medecin m=new Medecin();
Employe e=new Employe();
…
h.recruter(m); h.recruter(e);
…
h.getSesEmployes().get(9).afficher();
}}
```

afficher() est une méthode polymorphe qui existe partout et qui a le même objectif mais ayant un contenu spécifique à chaque type d'employé. Il est très recommandé de garder le même nom de méthode.

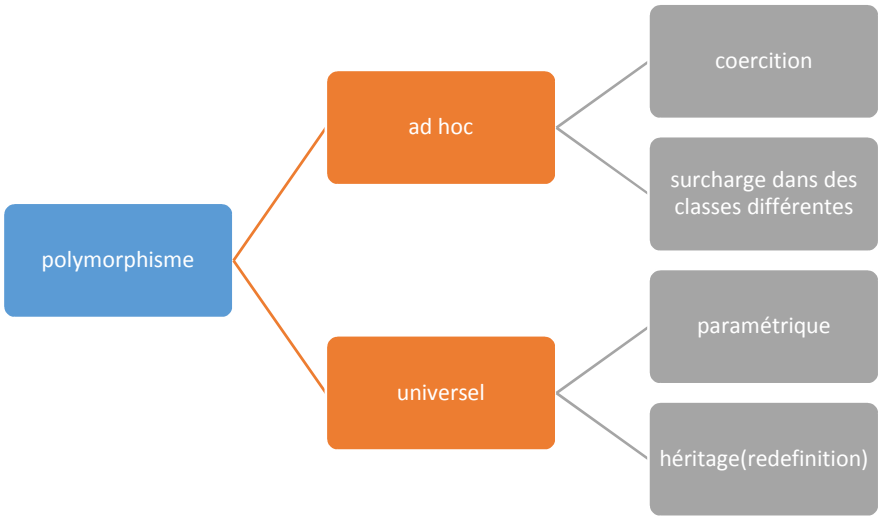
2. Polymorphisme

1. Définition du polymorphisme

Le mot polymorphisme du grec signifie qui peut prendre plusieurs formes. Cette caractéristique est un des concepts essentiels de la POO. Alors que l'héritage concerne les classes (et leur hiérarchie), le polymorphisme est relatif aux méthodes des **objets** (instances).

2. Types de polymorphisme

Il y a plusieurs types de polymorphisme selon la redéfinition de signature, de paramètre ou l'utilisation dans la même classe ou dans des classes indépendantes (ad hoc) ou pour un héritage.



3.8.1. polymorphisme ad hoc

Il permet d'avoir des fonctions de même nom, avec des fonctionnalités similaires, dans des classes différentes.(exemple : toString(), ValueOf())

a **Polymorphisme de coercition (conversion implicite).**

int valInt = 2;

double valDouble = 2.2;

double resultat = valInt + valDouble; // Conversion implicite de valInt en double.

b Polymorphisme ad hocs de surcharge (surcharge des méthodes).

Il est donc possible par exemple de **surcharger** une méthode avec un nombre de paramètres.

L'exemple de constructeur, en est un cas de surcharge. Ainsi, on peut par exemple définir plusieurs méthodes homonymes addition() effectuant une somme de valeurs. Elle pourra retourner la somme de deux entiers, la somme de deux flottants, la concaténation de deux caractères, de plusieurs tableaux….etc :

```
int addition(int, int)
float addition(float, float)
String addition(char, char)
```

On appelle **signature** le nombre et le type (statique) des arguments d'une fonction. C'est donc la signature d'une méthode qui détermine laquelle sera appelée.

Remarques : Il faut faire attention lorsque pour un même nom de méthode dans un même fichier, la signification des arguments n'est pas la même mais les paramètres en nombre et en types sont identiques, l'implémentation d'un tel cas n'est pas possible.

3.8.2. Polymorphisme Paramétrique

Le polymorphisme paramétrique, appelé généricité, représente la possibilité de définir plusieurs fonctions de même nom mais possédant des paramètres différents (en nombre et/ou en type). Le polymorphisme paramétrique rend ainsi possible le choix automatique de la bonne méthode à adopter en fonction du type de donnée passée en paramètre. Par exemple Vector et ArrayList sont génériques ArrayList <Object> :

```
ArrayList <Personne>listP ; …..listP.add(unePersonne) ;
ArrayList <String>listS ; …..listS.add(uneChaineCaractere) ;
listS.get() //retourne un String.
```

Même chose pour LinkedList<Type> et Vector<Type>.

3.8.3. Polymorphisme d'héritage

La possibilité de redéfinir une méthode dans des classes héritant d'une classe de base s'appelle la spécialisation. Il est alors possible d'appeler la méthode d'un objet sans se soucier de son type intrinsèque : il s'agit du polymorphisme d'héritage.

Exemple :

Dans un jeu d'échec comportant des Pieces : Roi, Reine, Fou, Cavalier, Tour et Pion, héritant chacun de Piece. On a la méthode mouvement polymorphe par héritage.

```
roi.mouvement() ; cavalier.mouvement() ;
```

3. Utilisation de polymorphisme

Le polymorphisme comme l'héritage permet de réduire des redondances et de prévoir l'ajout d'autres variantes de classes et de méthodes dans le projet dans le futur.

Exemple de compte Bancaire :

Sans le polymorphisme

```
if ( monCompteBancaire

instance of (PEA))
{moncomptebancaire.calculeInteretPEA();}

else  if ( monCompteBancaire

instance of (PEL)){
moncomptebancaire.calculeInteretPEL() ;}else

moncomptebancaire.calculeInteretLivretA() ;
```

On ne peut pas hériter d'une classe déclarée final.

avec le polymorphisme :

Le code sera simplement :

```
moncomptebancaire.calculeInteret() ;
```

avec possibilité de réutilisation si on crée un autre type de compte.

Une méthode déclarée final n'est pas redéfinissable.

4. Règles de Polymorphisme de méthode

- Si la première instance de fonction est statique, alors toutes les méthodes polymorphes doivent l'être.
- Si la première instance de fonction est publique, protégée ou privée, alors toutes les méthodes polymorphes doivent avoir la même classe d'accès
- Si la première instance renvoie des exceptions, alors toutes les méthodes polymorphes doivent renvoyer les mêmes exceptions.
- Si une méthode d'une classe mère n'est pas redéfinie ou « polymorphée », à l'appel de cette méthode par le biais d'un objet enfant, c'est la méthode de la classe mère qui sera utilisée.

Exercices du chapitre 5

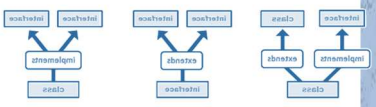
Exercice 1 :

Soient les classes Point et PointNom ainsi définis :

```
Class Point
{ public Point (int x, int y) { this.x = x ; this.y = y ; }
  public static boolean identiques (Point a, Point b)
  { return ( (a.x==b.x) && (a.y==b.y) ) ; }
  public boolean identique (Point a)
  { return ( (a.x==x) && (a.y==y) ) ; }
  private int x, y ;
}
Class PointNom extends Point
{ PointNom (int x, int y, char nom)
  { super (x, y) ; this.nom = nom ; }
  private char nom ;
}
```

Quels résultats fournit ce programme ? Expliquer les conversions mises en jeu et les règles utilisées pour traiter les différents appels de méthodes :

```
public class Poly
{ public static void main (String args[])
  { Point p = new Point (2, 4) ;
    PointNom pn1 = new PointNom (2, 4, 'A') ;
    PointNom pn2 = new PointNom (2, 4, 'B') ;
    System.out.println (pn1.identique(pn2)) ;
    System.out.println (p.identique(pn1)) ;
    System.out.println (pn1.identique(p)) ;
```



```
System.out.println (Point.identiques(pn1, pn2)) ; }}
```

2. Doter la classe PointNom d'une méthode statique *identiques* et d'une méthode *identique* fournissant toutes les deux la valeur true lorsque les deux points concernés ont à la fois les mêmes coordonnées et le même nom. Quels résultats fournira alors le programme précédent ? Quelles seront les conversions mises en jeu et les règles utilisées ?

Chapitre 6

Notions Complémentaires en POO

1. Abstraction

1.1. Définition

abstract signifie non complète. Une méthode ou classe abstraite doit être obligatoirement redéfinie. Une classe est abstraite si elle contient au moins une méthode abstraite.

1.2. Exemple

On souhaite représenter une classe forme ayant un point de départ (centre, coin….) et prévoir le calcul de son aire.

```
 public abstract class Forme {
Point sonPoint ;
abstract float aire() ;}
public class Carre extends Forme {
float cote;
float aire() {  return cote * cote;  }
}
 public class Cercle extends Forme {
   float rayon;
   float aire() { return Math.PI*rayon*rayon; }
```

On peut par la suite déclarer dans un programme principal une Liste.

```
public class Geometrie{
 public static void main(String [] args)
{List<Formes> desFormes ;
```

On l'initialise vide de 40 formes maximum. Comme List de java.util contient elle aussi des

méthodes abstraites, il est impossible de créer une liste sans préciser exactement quel type de List, ici ArrayList :

```
sesFormes=new ArrayList<Formes>(40) ;
```

On la remplit de carré et de cercles en utilisant des constructeurs sans et avec paramètres) :

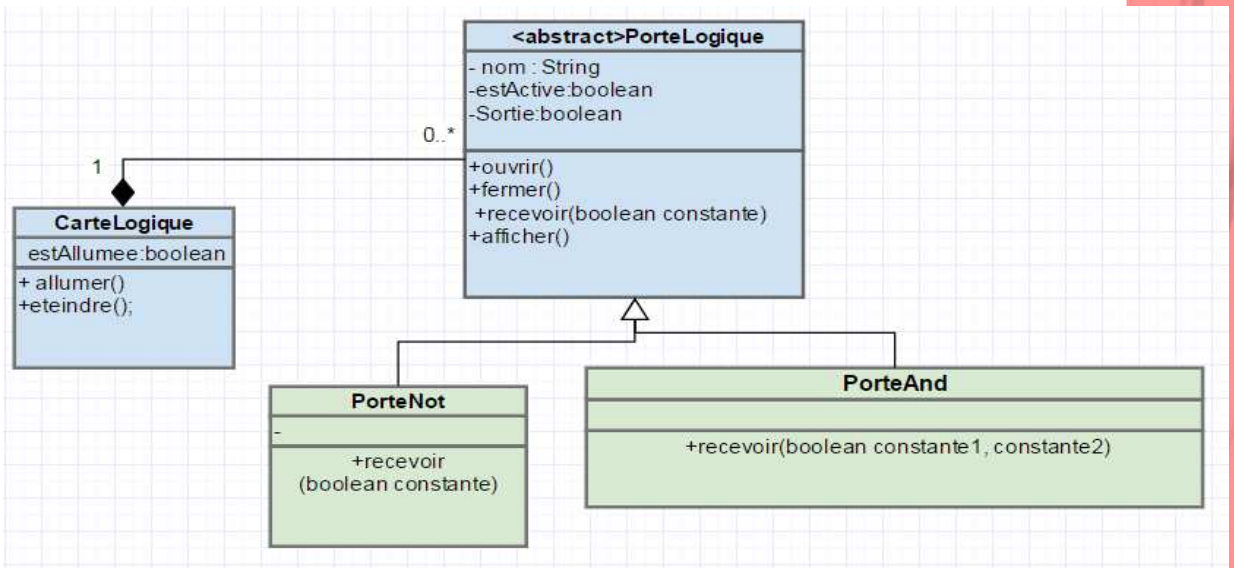
```
desFormes.add(new Cercle(new Point(0,0), 3.5)) ;

desFormes.add(new Carre(new Point(0,6), 4)) ;

desFormes.add(new Carre()) ;

desFormes.add(new Carre(new Point(), 7.4)) ;

desFormes.add(new Cercle()) ;
```

On affiche aisément les aires des formes sans se soucier de l'instance qui appelle la méthode aire() ;

```
 for(Forme f :desFormes)

System.out.println(f.aire()) ;

}}
```

1.3. Exercice

Une carte logique est composée de plusieurs portes logiques. Si porteLogique(And , XOR, Not…) est active et sa carteLogique est allumée, la porte peut *recevoir*( ) une ou des entrées qui modifient son attribut *sortie*. Une porte peut s'activer et se désactiver par son ouverture et fermeture.



1. Ecrire les attributs de la classe CarteLogique

```
Private boolean estAllumee ;
```

```
Private List<PorteLogique> listePorteLogiques ; // ou autre collection
```

2. Ecrire le constructeur de la classe CarteLogique

```
Public CarteLogique()

{estAllumee=true ;// ou false

listePorteLogiques=new ArrayList< PorteLogique >() ;
```

3. Ecrire la méthode allumer() de la classe CarteLogique

```
Public void allumer() {estAllumee=true ;}
```

4. Ecrire la méthode fermer() de la classe abstraite PorteLogique

```
Public void fermer() {estActive=false ;}
```

5. Comment déclarer la méthode recevoir dans la classe PorteLogique

```
abstract public void recevoir(boolean constante) ;
```

6. Redéfinir la méthode recevoir(…) dans la classe PorteAnd

```
public boolean recevoir(boolean var1,boolean var2)   //
peut ajouter throws ExceptionEteinte
{                  if                  (this.estActive&&
this.saCarteLogique.getEstAllumee()) {
 if  (var1  &&  var2)  {sortie=true;return  sortie}  else
{sortie=false;return sortie;}
} else {    return sortie;
//peut  ajouter  throw  exception(new  ExceptionEteinte())
;}
```

7. Dans la classe CarteLogique, écrire une méthode qui vérifie si les deux éléments à la $i^{ème}$ et $j^{ème}$ positions sont de même type de portes.

```
Public boolean sontIdentique(int i, int j)

{if
(this.listePorteLogique.get(i).instanceOf(this.listePorteLog
ique.get(i).getClass()) return true; else return false;}

// pouvant combiner getClass et equals  ou combiner if et
instanceOf
```

8. Ecrire 1 seule instruction qui calcule l'application de deux portes logiques p1 et p2 à entrée binaire sur trois constantes initiales x,y,z et l'affichage du résultat :(x p1 y) p2 z

```
System.out.println(p2.recevoir(p1.recevoir(x,y),z) ; //
accepter même plusieurs instructions au lieu d'une
seule.
```

2. Interfaces

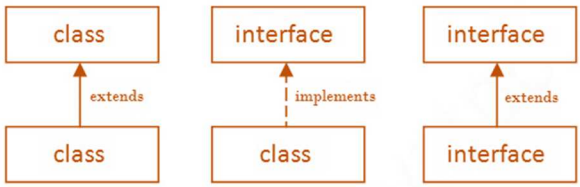
2.1. Définition

Une interface permets de regrouper des comportements (méthodes) et non pas des attributs. Elle peut contenir des constantes (static final). Et par la suite les imiter (implémenter).

```
public interface SportDeBall{

public void envoyerBalle() ;

public void recevoirBalle() ;

public void toucherBall() ;

}
```

Le mot abstract n'est pas forcément devant les méthodes car il y a le terme interface en prototype .

Elle résout l'héritage multiple. Aussi une interface peut hériter de plusieurs interfaces.



- Une interface est une classe 100 % abstraite mais sans le mot abstract.
- Aucune méthode d'une interface n'a de corps.
- Une interface sert à définir un supertype et à utiliser le polymorphisme.
- Une interface s'implémente dans une classe en utilisant le mot clé **implements**.
- On peut implémenter autant d'interfaces qu'on veut dans une classe.
- Il faut redéfinir toutes les méthodes de l'interface (ou des interfaces) dans la classe.

2.2. Exemple

Client et fournisseur herite de personne.Client en ligne , fournisseur en ligne envoie des message par internet , nous pouvons donc créer interface comportment en ligne pour ces deux.

List est une interface qui contient les méthodes abstraites.

3. Généricité

3.1. Définition

La généricité est un concept très utile pour développer des objets travaillant avec plusieurs types de données. On passe donc moins de temps à développer des classes traitant de façon identique des données différentes. List est une interface générique.

3.2. Exemples

Chose est une classe générique comme ArrayList et LinkedList et Vector.

```
public class Chose<T> {

  //Variable d'instance
  private T valeur;

  //Constructeur par défaut
  public Chose(){
    this.valeur = null;
  }

  //Constructeur avec paramètre inconnu pour l'instant
  public Chose (T val){
    this.valeur = val;
  }

  //Définit la valeur avec le paramètre
  public void setValeur(T val){
    this.valeur = val;
  }

  //Retourne la valeur déjà « castée » par la signature de
la méthode !
public T getValeur(){
return this.valeur;
  }

public static void main(String[] args) {
Chose <Integer> val = new Chose <Integer>(12);
  int nbre = val.getValeur();

System.out.print(nbre);
    }}
```

Nous pouvons définir des classes génériques avec deux types

```
public class ChoseAvec<T, S> {
//Variable d'instance de type T
  private T valeur1;

  //Variable d'instance de type S
  private S valeur2;
```

4. Exercices sur les interfaces et la généricité

Soit l'interface qui itère dans un tableau d'entier:

```
public interface IterateurTabInt {

public abstract int suivant();

public abstract int indiceDuSuivant(); }
```

soit une classe **ItérateurDesPairsTab** qui implémente de l'interface **IterateurTabInt** . Elle a comme propriétés un tableau d'entier appelé ***vecteur*** et une propriété **positionActuelle**. La classe parcourt uniquement les cases contenant un nombre pair dans ***vecteur***.

1. Ecrire l'entête d'une classe ItérateurDesPairsTab

```
public class ItérateurDesPairsTab implements
IterateurTabInt{
```

2. Coder la méthode **suivant**() de **ItérateurDesPairsTab** qui renvoie le prochain élément pair depuis la **positionActuelle** et mis à jour **positionActuelle**

```
public int suivant ()
{ for(int i=positionActuelle; i<vecteur.length ;++i)
{if (vecteur[i]%2==0) {positionActuelle=i ;
return vecteur[i] ;}// sortir de la fonction

 positionActuelle=0;

}
```

3. Peut-on ajouter un attribut dynamique à **IterateurTabInt** ? pourquoi ?

4.

Non, les interfaces n'ont pas des attributs dynamiques mais statiques et finaux

5. On veut rendre l'interface **IterateurTabInt** générique. Coder **IterateurTab,** une interface générique pour tout type T et pas forcément pour les entiers ?

```
public interface IterateurTab<T> {
```

```
public abstract T suivant();

public abstract int indiceDuSuivant(); }
```

Références

1. Yakov Fain, « Programmation Java pour les enfants, les parents et les grands-parents », Smart Data Processing, Inc., New Jersey ; Juin 2005, ISBN: 0-9718439-4-5.
2. HOUNGUE, Pelagie. « Programmation Orientée Objet » . Ressources UVA Fr - Sciences Informatiques ;2018.
3. https://openclassrooms.com/fr/courses/26832-apprenez-a-programmer-en-java. dernière consultation en 2018