SIO 2 SLAM – Bloc 2 - Ch. 15 – Interfaces

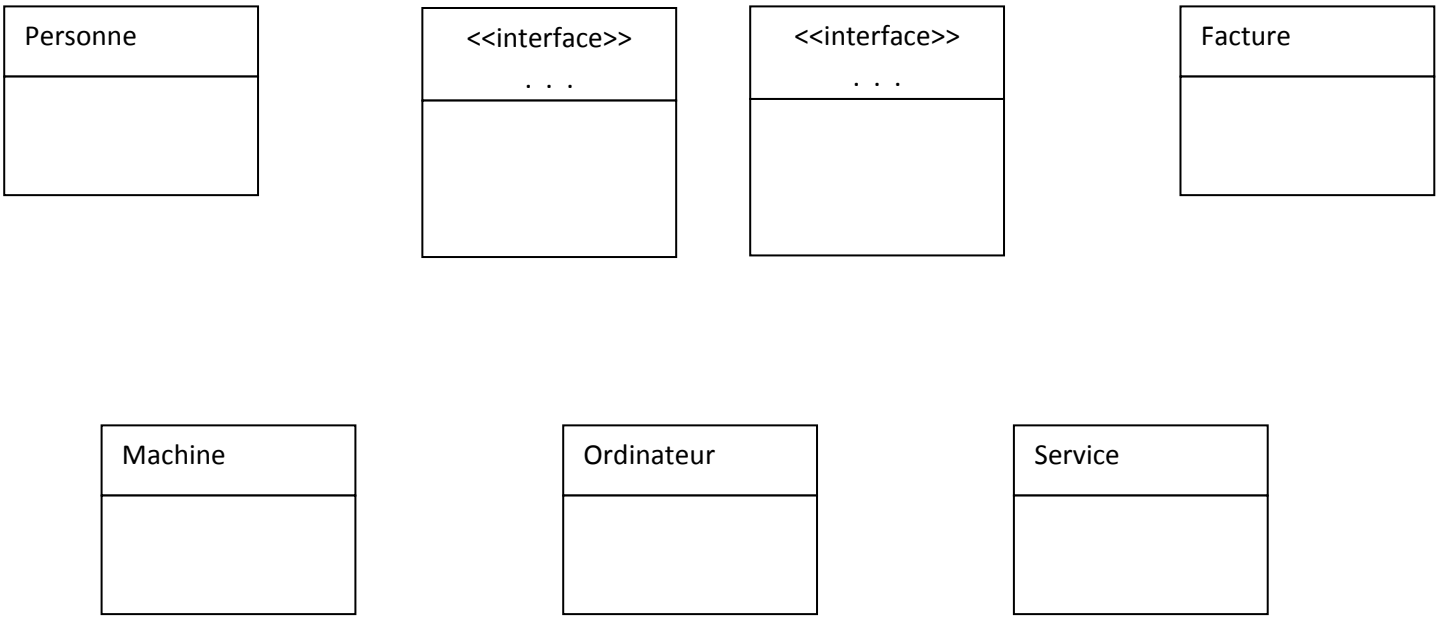
1 Les interfaces

1.1 Exercice préliminaire

Nous devons concevoir une application qui gère des machines et des ordinateurs dans une entreprise industrielle. De plus cette entreprise a aussi une activité commerciale : elle vend des ordinateurs et des services, qui sont facturés aux clients.

Travail à faire : compléter le diagramme de classes en tenant compte des règles suivantes.

- Créez l'interface **Appareil** avec les méthodes **allumer()** et **eteindre()** (dans le carré vide de gauche).
- Les ordinateurs et les machines sont des appareils.
- Une personne doit pouvoir allumer et éteindre un appareil.
- Un ordinateur ou un service sont payants et peuvent être facturés. Créez l'interface **Payant** avec la méthode **getPrix()**.
- Une facture est constituée d'objets payants.



1.2 Définition

Une **interface** définit un ensemble de comportements – qui ne sont pas implémentés – qui vont être utilisés par des classes clientes. Elle ne comporte que des méthodes abstraites (c'est-à-dire sans corps).

On peut voir une interface comme une classe abstraite particulière qui ne comporte que des méthodes abstraites (et des constantes), bien qu'une interface ne soit pas à proprement parler une classe.

Les « sous classes » qui héritent de l'interface - on dit qui **implémentent** ou qui **réalisent l'interface** – auront les comportements communs de l'interface, mais elles devront **implémenter**, c'est-à-dire redéfinir concrètement, toutes les méthodes déclarées dans l'interface.

En Java, une classe pourra avoir (implémenter) plusieurs interfaces alors qu'une classe ne peut dériver que d'une seule classe.

1.3 Ecriture d'une interface en Java

Voici le code correspondant au cas précédent :

```
public interface Payant
{
    double getPrix();     // on déclare juste le type de retour et le nom de la méthode
                          // le mot clé public n'est pas nécessaire
}
```

Classe **Service** qui implémente l'interface **Payant** :

```
public class Service implements Payant
{
    private String libelle;
    private int nombreJours;
    private double prixJournee;

    public Service(String lib, int nb, double prix)
    {
        libelle = lib;
        nombreJours = nb;
        prixJournee = prix;
    }

    // Implémentation de getPrix
    public double getPrix()
    {
        return nombreJours * prixJournee;
    }
}
```

Méthode main :

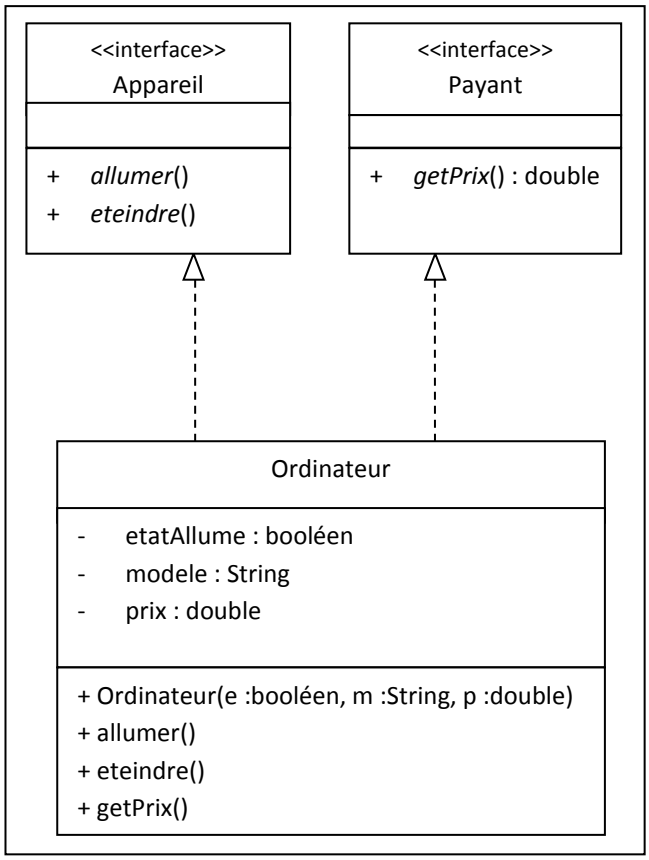
```
    public static void main(String[] args)
    {
        Payant p1 = new Service("dev site web", 5, 500);

        System.out.println( p1.getPrix() );
    }
```

Qu'est-ce qui s'affiche ? . . . . . .

Justifier . . . . . .

Exercice : Ecrivez le code de l'interface **Appareil** et de la classe **Ordinateur** conformément au diagramme suivant :



On vous donne le code de la classe Facture :

```
public class Facture
{
    private ArrayList<Payant> lesPayants;

    public Facture()
    {
        lesPayants = new ArrayList<Payant>();
    }

    public void ajouterPayant(Payant unPayant)
    {
        lesPayants.add(unPayant);
    }

    public double getMontantFacture()
    {
        double montant = 0;
        for (Payant unPayant : lesPayants)
        {
            montant += unPayant.getPrix();
        }
        return montant;
    }
}
```

Le code de la méthode **main** est :

```
public static void main(String[] args)
{
    Payant p1 = new Service("dev site web", 5, 500);
    Payant p2 = new Service("admin db", 10, 600);
    Payant p3 = new Ordinateur(false, "HP ProDesk", 1000);
    Payant p4 = new Ordinateur(false, "Lenovo ThinkCentre", 500);

    Facture facture = new Facture();

    facture.ajouterPayant(p1);
    facture.ajouterPayant(p2);
    facture.ajouterPayant(p3);
    facture.ajouterPayant(p4);

    System.out.println( facture.getMontantFacture() );

}
```

Travail à faire : dire ce qui s'affiche après l'exécution de **main**, en justifiant.

. . . . . .

2 Pourquoi utiliser des interfaces ?

Pour définir un couplage faible entre les classes :
Ceci est un souci permanent dans la conception objet, cela permet de masquer l'implémentation d'une classe en proposant des collaborations au niveau des interfaces.

Utiliser des interfaces proposées par l'API Java afin de profiter de classes fonctionnelles :
Par exemple l'interface **Comparable** déclare la méthode **compareTo.** Cette méthode permet de comparer deux instances d'une classe qui l'implémente. De nombreuses classes utilisent cette interface pour réaliser des traitements.