2I002 - Introduction à la programmation Objet
Groupe 2 – Test n°3 – Correction

Thomas Robert Remi Cadene

Partie 1 – Questions de cours (7 pts)

1. **(1pt) De quelle classe héritent toutes les classes ? Donnez la signature de deux méthodes de cette classe.**

`Object`. `public string toString()`, `public boolean equals(Object o)`, etc. (cf javadoc)

2. **(1pt) Expliquez brièvement les intérêts de l'héritage.**

Permet de mutualiser du code qui sera commun a plusieurs classes filles, de déclarer des signatures communes dont le fonctionnement pourra etre spécialisé, permet de modéliser une logique de modélisation des objets pour aller de classes générales vers des classes de plus en plus particulières.

3. **(1pt) Expliquez comment choisir entre utiliser l'héritage et la composition d'objets.**

L'heritage modélise une relation "est" : un Rectangle "est" une Figure, la classe fille est une spécialisation de la classe mère. La composition modélise l'utilisation, un lien, une relation "a" : un Cercle "a" un Point qui représente son centre.

4. **(1pt) Citez deux cas d'utilisation du mot clé `super`.**

Appel au constructeur parent, appel à une méthode parente (utile seulement quand elle a été redéfinie)

5. **(0.5pt) Quel modificateur d'accès permet à une variable d'être visible dans les classes filles de celle où elle est déclarée ?**

protected

6. **(0.5pt) Si la classe `B` hérite de `A`, peut-on écrire `A a = new B()` ? `B b = new A()` ?**

oui. non.

7. **(1pt) Quel est l'intéret du concept d'abstraction ?**

Permet de déclarer des comportements attendus mais dont on ne connait pas le comportement exact car la classe modélise un objet qui est "abstrait" a ce niveau de modélisation, et qui devra etre défini concretement lors de l'heritage.

8. **(1pt) Quelle est la difference entre une classe abstraite et une interface ?**

Une interface ne declare que des methodes abstraites, aucun constructeur. Une classe abstraite modélise réellement une classe, qui possède la plupart du temps du code concret, avec quelques méthodes abstraites.

Partie 2 – Code (3 pts UML, 12.5 pts code)

On veut développer des classes qui permettent de calculer la surface totale d'un ensemble de figures. Pour cela, on aura les classes `Figure`, `Cercle`, `Rectangle` et `EnsembleFigure`.

Figure. La classe `Figure` possède un **constructeur à un paramètre** : le nom de la figure, ainsi qu'un **getter** pour récupérer ce nom et **une méthode** `double surface()` qui retournera la surface de la figure en question. Cette surface est inconnue au niveau de la figure puisqu'on ne sait pas de quelle figure il s'agit.

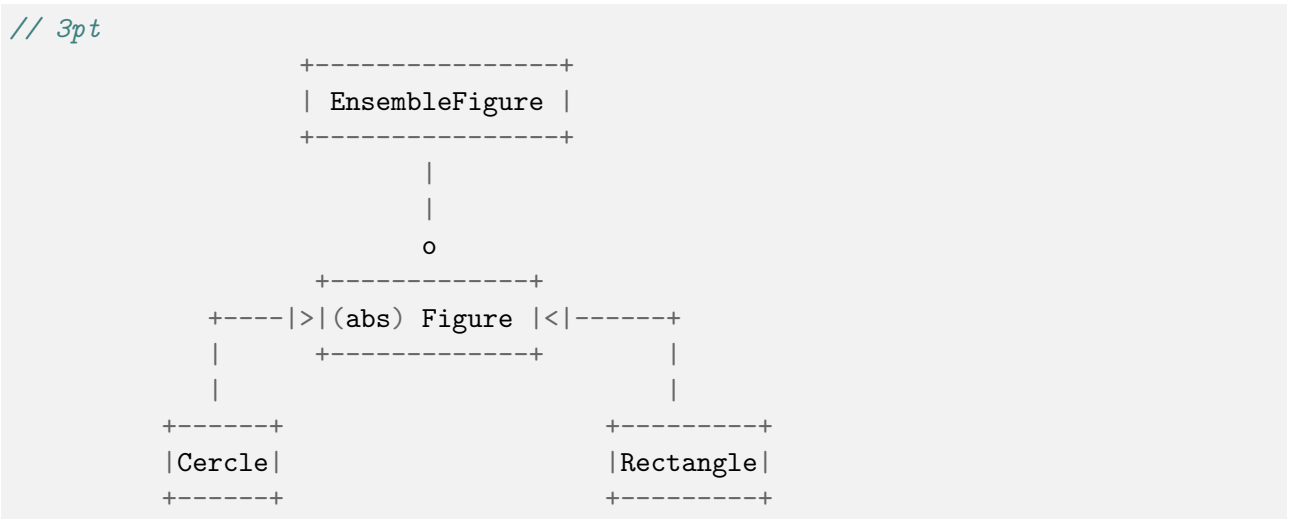
Cercle et Rectangle. La classe `Cercle` possède un **constructeur à deux paramètres** : le nom et le rayon du cercle, et une **méthode** `double surface()`. **De même pour la classe** `Rectangle` avec la hauteur et la largeur du rectangle à la place du rayon du cercle.

Comparaison. Par ailleurs, on voudrait pouvoir **comparer les figures** en fonction de leur surface. Pour cela, **utilisez à bon escient l'interface standard** `Comparable<Figure>` qui déclare la signature `public int compareTo(Figure f)` tel que :

- `x.compareTo(y) > 0` ⇔ `x.surface() > y.surface()`
- `x.compareTo(y) = 0` ⇔ `x.surface() = y.surface()`
- `x.compareTo(y) < 0` ⇔ `x.surface() < y.surface()`

EnsembleFigure. La classe `EnsembleFigure` permet de stocker un **ensemble de figures** et d'en calculer la surface totale. Elle a un **constructeur sans paramètre**, une **méthode** `add(...)` (paramètre d'entrée à déterminer) qui permet d'y ajouter une figure, une **méthode** `double surface()` qui renvoie la surface totale, c'est à dire la somme des surfaces des figures, et une méthode `getLargest()` qui retourne la plus grande figure de l'ensemble.
**

Question. En utilisant au mieux les concepts d'héritage, d'abstraction, et les `ArrayList`, représentez le **diagramme UML** puis donnez le **code** pour le problème ci-dessus.**




```java
// 1pt abstract
// 1pt implements
// 1pt constr get
// 1pt compareTo()
// 1pt surface()
public abstract class Figure implements Comparable<Figure> {
    private nom;
    public Figure(String nom) { this.nom = nom; }
    public String getNom() { return nom; }
    public abstract double surface();
    public int compareTo(Figure f) { return this.surface() - f.surface(); } // 1pt
}
```

```java
// Cercle + Rectangle
// 1pt extends
// 1pt constr + attrs
// 1pt surface()
// 1pt pour la 2e classe
public class Cercle extends Figure {
    private double r;
    public Cercle(String n, double r) { super(n); this.r = r; }
    public double surface() { return Math.pi * r * r; }
}

public class Rectangle extends Figure {
    private double h,w;
    public Rectangle(String n, double h, double w) { super(n); this.h = h; this.w =
    ↪  w; }
    public double surface() { return h * w; }
}

// 1pt attr + constr + add()
// 1pt surface
// 1.5pt getLargest()
public class EnsembleFigure {
    public List<Figure> l;
    public EnsembleFigure() { l = new ArrayList<Figure>(); }
    public void add(Figure f) { l.add(f); }
    public double surface() { double s; for (Figure f: l) s += f.surface(); return
    ↪  s; }
    public Figure getLargest() { return Collections.max(l); }
    // ou
    public Figure getLargest() {
        if (l.size() == 0) return null;
        Figure maxF = l.get(0);
        for (Figure f: l)
            if (f.compareTo(maxF) > 0) maxF = f;
        return maxF;
    }
}
```