

Examen de septembre écrit

Durée : **2h00** - Tous les documents *personnels*, manuscrits ou imprimés, sont autorisés.
Tous les appareils électroniques sont interdits et doivent être rangés.

NE RIEN ÉCRIRE AU CRAYON NI À L'ENCRE ROUGE S.V.P.

Ce sujet comporte quatre pages.

Question de cours

On considère les trois classes publiques suivantes :

```
public class QuestCours {
    public static void main(String[] a) {
        Mere o1= new Mere();
        System.out.println(o1.toStr());
        System.out.println(o1.toString());

        Fille o2= new Fille();
        System.out.println(o2.toStr());
        System.out.println(o2.toString());
        System.out.println(o2.toString(true));
        o1= new Fille();
        System.out.println(o1.toStr());
        System.out.println(o1.toString());
    }
}

public class Mere {
    public String toStr()    { return "dans Mere"; }
}

public class Fille extends Mere {
    public String toStr()    { return "dans Fille"; }
    public String toString() { return super.toString(); }
    public String toString(boolean b) { return super.toStr(); }
}
```

q1 Pourquoi les lignes

```
    System.out.println(o2.toString(true));
    o1= new Fille();
```

sont-elles autorisées ? Que signifient-elles, pour la compilation et pour l'exécution ?

q2 Pourquoi la ligne

```
System.out.println(o1.toString(true));
```

est-elle interdite comme instruction de la méthode `main` (erreur de compilation), y compris lorsqu'elle est placée après `o1= new Fille()` ? Quel type de message indique le compilateur ?

q3 L'affichage à l'écran résultant de l'exécution est le suivant :

```
 dans Mere
Mere@1cc88d
 dans Fille
Fille@1cc8c2
 dans Mere
 dans Fille
Fille@1cc8b0
```

Expliquez ses différentes lignes, sachant que les valeurs 1cc88d, 1cc8c2 et 1cc8b0 sont dépendante du contexte matériel et logiciel d'exécution des instructions.

Présentation du thème abordé

On s'intéresse à la représentation de figures de \mathbb{R}^2 , convexes et régulières. On se limite au cercle, au carré et au polygone régulier à n cotés. Dans un premier temps, ces figures sont considérées comme non positionnées dans \mathbb{R}^2 . Seules leurs grandeurs caractéristiques nous intéressent : le rayon pour le cercle, le côté pour le carré et le nombre de côtés et le rayon pour le polygone régulier. Il s'agit de calculer le périmètre de figures différentes, opération générale dont les spécificités de calcul dépendent de la figure particulière considérée.

Les classes `Cercle`, `Carre` et `Polygone` dépendent toutes trois d'une même classe `Figure`. La méthode `perimetre` de cette dernière classe étant nécessairement abstraite (puisque l'on ne sait pas calculer en toute généralité le périmètre d'une figure), il faudra également définir une méthode `perimetre` pour chacun des trois cas de figure.

Dans un premier temps, hors de tout souci de représentation des figures à l'écran, on définira cinq classes d'objets, les quatre ci-dessus plus une classe `Test1` pour tester le fonctionnement de cette structure. Une exécution, parmi d'autres, de la classe `Test1` conduit à l'affichage à l'écran suivant :

```
figure cercle : 45.63770653159439
figure carre : 20.04629295309989
figure cercle : 32.83541225056175
figure polygone : 18.879348219986454
figure cercle : 43.34234227419014
```

Le terme figure apparaît à chaque fois, tandis que le nom de la figure et la valeur de son périmètre sont particuliers à chaque ligne. Cet affichage est obtenu en utilisant les méthodes publiques `toString` renvoyant une chaîne de caractères, définies pour chacune des quatre premières classes.

Très important Toute indentation mal faite ou incorrecte sera notée négativement.

A dessin de la hiérarchie des classes

Dessiner la hiérarchie des classes `Carre`, `Cercle`, `Figure` et `Polygone`, sous la forme d'un arbre, dans lequel vous indiquerez, bien sûr le nom de la classe, mais aussi en dessous de lui les noms des champs et celui des méthodes, ces dernières portant éventuellement juste à la suite de leur nom une indication supplémentaire, * si la méthode est abstraite, + si elle implémente une méthode abstraite et ++ si elle redéfinit une méthode.

B classe *Figure*

Elle comporte une méthode abstraite `perimetre` ; la méthode `toString` renvoie la chaîne de caractères `"figure "` .

Donner la déclaration complète de cette classe.

C classe *Cercle*

Elle comporte un champ, au moins un constructeur et bien sûr les deux méthodes.

Donner la déclaration complète de cette classe.

D classe *Carre*

Elle comporte un champ, au moins un constructeur et bien sûr les deux méthodes.

Donner la déclaration complète de cette classe.

E classe *Polygone*

Pour calculer le périmètre d'un polygone, il suffit de connaître la longueur d'un des côtés du polygone : le périmètre total s'en déduit alors naturellement.

On considère le polygone de rayon 1, donc avec ses points sur le cercle trigonométrique; on considère un côté du polygone, dont le milieu est l'intersection du côté avec l'axe des x positifs : la hauteur du triangle isocèle formé depuis l'origine et ayant pour base ce côté est sur l'axe des x. Il est alors aisé de calculer la demi longueur du côté en tenant compte du sinus de la moitié de l'angle à l'origine du triangle isocèle. On connaît ainsi la longueur d'un côté du polygone.

Donner la formule mathématique donnant la longueur d'un côté d'un polygone régulier à n côtés.

Donner la déclaration complète de la classe `Polygone` . Si vous n'avez pas réussi à trouver la formule permettant de calculer la longueur du côté, indiquez-le au bon endroit à l'aide d'un commentaire; débrouillez vous pour que votre programme reste compilable.

F classe *Test1*

On va maintenant utiliser ces classes pour réaliser l'affichage en écran texte indiqué plus haut.

La classe ne comporte aucun champ et une seule méthode. Cette dernière comporte trois parties : la déclaration de deux variables locales, l'une pour le nombre de figures (ici initialisée à 5) et l'autre pour un tableau de `Figure` ; viennent alors les instructions pour initialiser aléatoirement les figures, tant pour leur type, que pour les caractéristiques de chacun des types; enfin la dernière instruction est dédiée à l'affichage des valeurs des périmètres, qui est ainsi réalisé :

```
// affichage de tous les perimetres
for (int f= 0; f<nbrFig; f++)
    System.out.println(figures[f].toString());
```

Le choix entre les trois types de figures est aléatoire. Un cercle aura un rayon compris entre 0 et 10, la longueur du côté d'un carré sera comprise entre 0 et 7, et le polygone aura entre 5 et 15 côtés, son rayon étant compris entre 0 et 12.

Donner la déclaration complète de la classe `Test1` .

G polygone approchant un cercle

On désire approcher un cercle par un polygone. Le rayon du polygone sera évidemment celui du cercle, et le nombre de côtés est un paramètre de l'approximation. Il y a plusieurs solutions envisageables.

La plus simple serait de dire qu'un cercle approché est à la fois un cercle et un polygone. Pourquoi, en Java, cette solution n'est nullement envisageable ?

On peut alors envisager entre autres deux solutions.

Si l'on considère qu'une fois approché le cercle devient un polygone à part entière, approcher le cercle revient tout simplement à construire un polygone (mais on ne sait plus que ce polygone provient d'un cercle). Dans ce cas, il suffit de surcharger les constructeurs de `Polygone`. Donner la définition de la classe `Polygone` modifiée.

Si l'on considère qu'un cercle approché est un cercle auquel on adjoint le polygone qui l'approche, on peut envisager d'étendre la classe `Cercle` par la classe `CercleApproche`. Donner la définition de cette classe. On peut avoir besoin d'accéder aux valeurs des points du polygone : que faut-il modifier d'autre ?

Existe-t-il une troisième solution ?

H polygone décrit par ses points

On peut décrire un polygone par l'ensemble de ses sommets, dont les coordonnées seront stockées dans un tableau de `Point`

On considérera qu'un des points du polygone se trouve sur l'axe des y positifs. On rappelle que la matrice de rotation d'un point d'un angle α autour de l'origine est la suivante :

$\cos \alpha$	$-\sin \alpha$
$\sin \alpha$	$\cos \alpha$

Soit un point (x,y) . Donner les coordonnées du point résultant de la rotation d'angle α de ce point autour de l'origine et

Donner la déclaration de la classe `Point` qui permet de décrire et d'utiliser un point de \mathbb{R}^2 . Donner la déclaration de la classe `PolyPoints`, s'inscrivant dans la structure d'héritage, qui ajoute à un polynôme sa description en points.

```

abstract class Figure {
    abstract double perimetre();
    public String toString(){return "figure "};
}

class Cercle extends Figure{
    private double rayon;

    Cercle(double r) {rayon= r;}
    double perimetre() {return 2*Math.PI*rayon;}
    public String toString() {
        return super.toString()+"cercle : "+perimetre();
    }
}

class Carre extends Figure{
    private double cote;

    Carre(double c) {cote= c;}
    double perimetre(){ return 4*cote; }
    public String toString() {
        return super.toString()+"carre : "+perimetre();
    }
}

class Polygone extends Figure{
    private int nbCotes; private double rayon;

    Polygone(int n, double r) { nbCotes= n; rayon= r; }
    double perimetre() { return 2*Math.sin(Math.PI/nbCotes)*nbCotes*rayon; }
    public String toString() {
        return super.toString()+"polygone : "+perimetre();
    }
}

class Test1 {
    public static void main(String[] args) {
        int nbrFig= 5;
        Figure figures[]= new Figure[nbrFig];

        // creation de nbrFig figures
        for (int f= 0; f<nbrFig; f++) {
            int choix= (int)(Math.random()*10000)%3;
            if (choix==0)
                figures[f]= new Cercle(Math.random()*10);
            else if (choix==1)
                figures[f]= new Carre(Math.random()*7);
            else figures[f]= new PolygoneRegulier(
                (int)(Math.random()*15),Math.random()*12);
        }
        // affichage de tous les perimetres
        for (int f= 0; f<nbrFig; f++)
            System.out.println(figures[f].toString());
    }
}

```

```

abstract class Figure {
    abstract double perimetre();
    public String toString(){return "figure "};
}

class Cercle extends Figure{
    private double rayon;

    Cercle(double r) {rayon= r;}
    double perimetre() {return 2*Math.PI*rayon;}
    public String toString() {
        return super.toString()+"cercle : "+perimetre();
    }
}

class Carre extends Figure{
    private double cote;

    Carre(double c) {cote= c;}
    double perimetre(){ return 4*cote; }
    public String toString() {
        return super.toString()+"carre : "+perimetre();
    }
}

class PolygoneRegulier extends Figure{
    private int nbCotes; private double rayon;

    PolygoneRegulier(int n, double r) { nbCotes= n; rayon= r; }
    double perimetre() { return 2*Math.sin(Math.PI/nbCotes)*nbCotes*rayon; }
    public String toString() {
        return super.toString()+"polygone : "+perimetre();
    }
}

class Test1 {
    public static void main(String[] args) {
        int nbrFig= 5;

        // creation de nbrFig figures
        Figure figures[]= new Figure[nbrFig];
        for (int f= 0; f<nbrFig; f++) {
            int choix= (int)(Math.random()*10000)%3;
            if (choix==0)
                figures[f]= new Cercle(Math.random()*10);
            else if (choix==1)
                figures[f]= new Carre(Math.random()*7);
            else figures[f]= new PolygoneRegulier(
                (int)(Math.random()*15),Math.random()*12);
        }
        // affichage de tous les perimetres
        for (int f= 0; f<nbrFig; f++)
            System.out.println(figures[f].toString());
    }
}

```