Examen de Programación Orientada a Objetos (Plan 2014)

29 de junio de 2021

Nº matrícula:_____ **Grupo:**_____ **Nombre:** _____

Apellidos: _____

NOTA: La duración del examen es de 80 minutos.

Problema. (10 puntos)

Teniendo en cuenta la codificación de la interfaz `Figura`:

```
1     public interface Figura {
2           double getArea();
3           double getPerimetro();
4     }
```

Y de la clase `Circulo`:

```
1     public class Circulo extends FiguraCurva {
2           private double radio;
3
4           public Circulo(double radio) {
5                 super(new double[]{radio, radio});
6                 this.radio = radio;
7           }
8
9           public double getPerimetro() {
10                return 2 * Math.PI * this.radio;
11          }
12    }
```

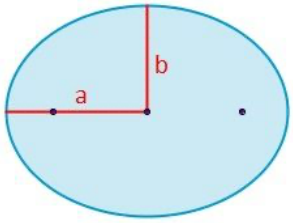
1.1 (2 puntos). Codificar la clase abstracta `FiguraCurva` que implementa la interfaz `Figura` y de la cual hereda la clase `Circulo`.

```
public abstract class FiguraCurva implements Figura {
    private double[] radios;

    public FiguraCurva(double[] radios) {
        this.radios = radios;
    }

    @Override
    public double getArea() {
        return Math.PI * radios[0] * radios[1];
    }

    @Override
    public abstract double getPerimetro();
 }
```

1.1 (Continuación).

1.2 (1 punto). Teniendo en cuenta la jerarquía de interfaces/clases anteriores, codificar la clase `Elipse`.

Nota: Para el cálculo del perímetro de una elipse se puede utilizar la siguiente aproximación:



$$Perímetro \approx 2\pi \cdot \sqrt{\frac{a^2 + b^2}{2}}$$

siendo a y b los semiejes mayor y menor de la elipse

```
public class Elipse extends FiguraCurva {
       private double radioMayor, radioMenor;

       public Elipse(double radioMayor, double radioMenor) {
          super(new double[]{radioMayor, radioMenor});
          this.radioMayor = radioMayor;
          this.radioMenor = radioMenor;
       }

       public double getPerimetro() {
          // fórmula aproximada
          double radio = Math.sqrt((Math.pow(radioMayor,2) +
                         Math.pow(radioMenor,2))/2);
          return 2 * Math.PI * radio;
       }
}
```

1.3 (1 punto). Teniendo en cuenta la codificación de la clase `Elipse`. ¿Sería posible hacer que la clase `Circulo` heredara de la clase `Elipse` y no de la clase `FiguraCurva`? En caso afirmativo codificar cómo quedaría la nueva clase `Circulo`. En caso negativo justificar porqué no se podría dar esa jerarquía de herencia.

```
public class Circulo extends Ellipse {

	public Circulo(double radio) {
		super(radio, radio);
	}
}
```

1.4 (1 punto). Se quiere reescribir el método `toString()` de la clase `Object`. Su salida debe indicar la siguiente información: *nombre de la clase*, *listado de los radios*, *perímetro* y *área* de la figura correspondiente. A modo de ejemplo, un círculo y una elipse tendrían las siguientes salidas:

```
Circulo [radios:(2.5, 2.5), perímetro:15.71, área:19.63]
Elipse [radios:(3.0, 2.0), perímetro:16.02, área:18.85]
```

Indicar en qué clase de la jerarquía definida anteriormente debería incluirse la reescritura del método y codificarlo.

```
Se incluiría en la clase FiguraCurva

@Override
public String toString() {
    return this.getClass().getName() +
            " [radios:" + Arrays.toString(this.radios) +
            ", perímetro:" + this.getPerimetro() +
            ", área:" + this.getArea() + "]";
}
```

1.5 (0.5 puntos). Codificar la versión *estática* del método `getPerimetro` de la clase `Circulo`.

```
public static double getPerimetro (double radio) {
      return 2 * Math.PI * radio;
}
```

Teniendo en cuenta la codificación de la clase `ListaFiguras`:

```
1     public class ListaFiguras <T extends Figura> {
2         private List<T> lista;
3
4         public ListaFiguras() {
5             lista = new ArrayList<>();
6         }
7
8         public T sacarFigura(int posicion) {
9             return lista.get(posicion);
10        }
11
12        public void meterFigura(T figura){
13            lista.add(figura);
14        }
15
16        @Override
17        public String toString() {
18            return "ListaFiguras " + lista;
19        }
20    }
```

1.6 (0.5 puntos). Codifica las correspondientes sentencias de importación de librerías/paquetes necesarios para que el código de la clase `ListaFiguras` no genere errores:

```
import java.util.List;
import java.util.ArrayList;

# también es válido, pero menos correcto -> import java.util.*
```

1.7 (1 punto). Teniendo en cuenta la codificación de la clase `Test`:

```
1  public class Test {
2    public Test(){
3          ListaFiguras lista = new ListaFiguras();
4          // Circulo y Elipse utilizados en el ejercicio 1.4
5          lista.meterFigura(new Circulo (2.5));
6          lista.meterFigura(new Elipse (3.0, 2.0));
7          System.out.println(lista);
8          Figura f = lista.sacarFigura(2);
9          System.out.println(f.toString());
10   }
11   public static void main(String[] args) {
12         new Test();
13   }
14 }
```

Indicar cuál sería el resultado de la ejecución de la misma, identificando la línea o líneas en las que se generan salidas o errores y los correspondientes valores de salida y naturaleza de los errores.

```
7 -> ListaFiguras [Circulo [radios:(2.5, 2.5), perímetro:15.71, área:19.63]
                   Elipse [radios:(3.0, 2.0), perímetro:16.02, área:18.85]]


8 -> Exception java.lang.IndexOutOfBoundsException
```

1.8 (1 punto). Recodificar el método `sacarFigura` de la clase `ListaFiguras` para que pueda generar la excepción `ListaVacia` cuando se intenta sacar un elemento de una lista vacía o la excepción `ElementoNoExistente` cuando se intenta sacar un elemento de una posición no existente. La excepción `ListaVacia` tiene más prioridad que la excepción `ElementoNoExistente`.

```
   public T sacarFigura(int posicion) throws ListaVacia, ElementoNoExistente {
       if(lista.size() == 0) throw new ListaVacia();
       if(lista.size() < posicion) throw new ElementoNoExistente();
       return lista.get(posicion);
   }
```

1.9 (0.5 puntos). Codificar la clase `ListaVacia`:

```
public class ListaVacia extends Exception {
    public ListaVacia(){
        super("Lista Vacía...");
    }
}
```

1.10 (1 punto). Recodificar el constructor de la clase `Test` del ejercicio 1.7 para adecuarlo a la recodificación realizada en el punto 1.8 del método `sacarFigura`.

```
public Test(){
	ListaFiguras lista = new ListaFiguras();
	// Circulo y Elipse utilizados en el ejercicio 1.4
	lista.meterFigura(new Circulo (2.5));
	lista.meterFigura(new Elipse (3.0, 2.0));
	System.out.println(lista);
	try {
		Figura f = lista.sacarFigura(1);
		System.out.println(f.toString());
	} catch (ListaVacia|ElementoNoExistente e) {
		System.out.println(e.getMessage());
	}
}
```

1.11 (0.5 puntos). Teniendo en cuenta la siguiente sentencia:

```
ListaFiguras<Circulo> listaFiguras = new ListaFiguras<>();
```

Indicar si genera un error y de que tipo o, en caso de ser correcta, que restricciones de uso tiene la variable `listaFiguras`.

```
La variable listaFiguras sólo puede contener elementos de tipo Circulo.
```