POLITÉCNICA

Escuela Técnica Superior de Ingeniería de Sistemas Informáticos
Universidad Politécnica de Madrid

Examen de Programación Orientada a Objetos (Plan 2014) — **20 de enero de 2021**

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

Apellidos: _____

NOTA: La duración del examen es de 80 minutos.

Problema. (10 puntos)

Se quiere definir una jerarquía de interfaces/clases para trabajar con figuras planas geométricas, específicamente con polígonos.

1.1 (0.5 puntos). Codificar la interfaz `Figura` que contiene los métodos `getArea()` y `getPerimetro()` sin parámetros. Ambos devuelven un valor de tipo `double` que se corresponde con el área y el perímetro de una figura genérica.

```
public interface Figura {
        double getArea();
        double getPerimetro();
}
```

1.2 (0.5 puntos). Indica cuál de las siguientes afirmaciones sobre una interfaz es **falsa**:

a) Una interfaz es una colección de métodos abstractos.
b) Una interfaz puede contener miembros privados.
c) Una interfaz puede contener atributos públicos estáticos finales (*public static final*).
d) Si una clase implementa una interfaz, debe reescribir todos sus métodos o declararlos como abstractos.

1.3 (2 puntos). Codificar la clase abstracta `Poligono` que implementa la interfaz `Figura`. Para ello se deben tener en cuenta las siguientes reglas:

- Un `Poligono` queda definido por sus lados. Los lados de un `Poligono` son de tipo `double`.
- Se debe incluir la reescritura del método `toString()` de la clase `Object`. Su salida debe indicar la siguiente información: *nombre de la clase*, *listado de los lados*, *perímetro* y *área* del polígono. A modo de ejemplo, un cuadrado y un rectángulo tendrían las siguientes salidas:

  `Cuadrado (lados:[2.0, 2.0, 2.0, 2.0], perímetro:8.0, área:4.0)`
  `Rectangulo (lados:[4.0, 2.0, 4.0, 2.0], perímetro:12.0, área:8.0)`
- Se deben implementar todos los métodos que son comunes a cualquier polígono.

1.3 (Continuación).

```
import java.util.Arrays;

public abstract class Poligono implements Figura{
    private double[] lados;

    public Poligono(double[] lados) {
        this.lados = lados;
    }

    @Override
    public abstract double getArea();

    @Override
    public double getPerimetro() {
        int perimetro = 0;
        for(double lado: lados){
            perimetro += lado;
        }
        return perimetro;
    }

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

1.4 (0.5 puntos). Indica cuál de las siguientes afirmaciones sobre una clase abstracta es **cierta**:

a) Se pueden instanciar objetos de clases abstractas siempre que la clase abstracta implemente todos sus métodos.
b) Una clase abstracta no puede tener constructores.
c) Una clase abstracta debe contener al menos un método abstracto.
d) Nunca se pueden instanciar objetos de una clase abstracta.

1.5 (1 punto). Codificar la clase `Rectangulo` que hereda de la clase `Poligono`. Tener en cuenta que un `Rectangulo` se define indicando su ancho y su alto.

```
public class Rectangulo extends Poligono {
	private double alto, ancho;

	protected Rectangulo(double ancho, double alto) {
		super(new double[]{ancho, alto, ancho, alto});
		this.alto = alto;
		this.ancho = ancho;
	}

	public double getArea() {
		return this.alto * this.ancho;
	}

}
```

1.6 (0.5 puntos). Codificar la clase `Cuadrado`. Decidir cuál es la mejor jerarquía de herencia para esta clase. Tener en cuenta que un `Cuadrado` se define indicando únicamente su lado.

```
public class Cuadrado extends Rectangulo{

    protected Cuadrado(double lado) {
        super(lado, lado);
    }
}
```

1.7 (0.5 puntos). Codificar la versión *estática* del método `getArea`:

```
public static double getArea(double alto, double ancho) {
	return alto * ancho;
}
```

1.8 (2 puntos). Codificar la clase `ListaFiguras` que permite gestionar listas genéricas de figuras. La clase debe contener los siguientes métodos:

- `sacarFigura` que tiene un único parámetro de tipo entero que indica la posición, dentro de la lista, de la figura que se quiere recuperar. Este método puede 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` es más prioritaria que la excepción `ElementoNoExistente`.
- `meterFigura` que tiene como único parámetro la figura que se desea incluir en la lista.
- Se debe codificar la reescritura del método `toString()` de la clase `Object`. Su salida debe indicar la siguiente información: *nombre de la clase* y *polígonos incluidos*. A modo de ejemplo, una lista con un cuadrado y un rectángulo tendría como salida:

```
ListaFiguras [Cuadrado (lados:[2.0, 2.0, 2.0, 2.0], perímetro:8.0, área:4.0), Rectangulo (lados:[4.0, 2.0, 4.0, 2.0], perímetro:12.0, área:8.0)]
```

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

public class ListaFiguras <T extends Figura> {
    private List<T> lista;

    public ListaFiguras() {
        lista = new ArrayList<>();
    }

    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);
    }

    public void meterFigura(T figura){
        lista.add(figura);
    }

    @Override
    public String toString() {
        return "ListaFiguras " + lista;
    }
}
```

1.8 (Continuación).

1.9 (1 punto). Codificar la clase `ListaVacia`.

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

1.10 (0.5 puntos). Codificar la declaración de un objeto `listacuadrados` que instancia una `ListaFiguras` que únicamente contiene elementos de tipo `Cuadrado`**:**

```
ListaFiguras<Cuadrado> listacuadrados = new ListaFiguras<>();
```

1.11 (1 punto). Teniendo en cuenta la codificación de la clase `Test`, indicar cuál sería el resultado de la ejecución de la misma, identificando las líneas en las que se generan las correspondientes salidas o los errores.

```
1  public class Test {
2    public Test(){
3          Cuadrado c1 = new Cuadrado(2);
4          Rectangulo r1 = new Rectangulo(4, 2);
5          ListaFiguras lista = new ListaFiguras();
6          try {
7                Figura f = lista.sacarFigura(5);
8                System.out.println(f.toString());
9          } catch (ListaVacia|ElementoNoExistente e) {
10               System.out.println(e.getMessage());
11         }
12         lista.meterFigura(c1);
13         lista.meterFigura(r1);
14         try {
15               System.out.println(lista.sacarFigura(5));
16         } catch (ListaVacia|ElementoNoExistente e) {
17               System.out.println(e.getMessage());
18         } finally {
19               System.out.println(lista);
20         }
21   }
22   public static void main(String[] args) {
23         new Test();
24   }
25 }
```

Línea 7 -> Lista Vacía...
Línea 15 -> Elemento no existente en la pila...
Línea 19 -> ListaFiguras [Cuadrado (lados:[2.0, 2.0, 2.0, 2.0], perímetro:8.0, área:4.0), Rectangulo (lados:[4.0, 2.0, 4.0, 2.0], perímetro:12.0, área:8.0)]