

Reporte de la Práctica 1: Repaso de clases y herencia.

Julio César Lozano Garnica. No de cuenta UNAM: 420095390

25 de agosto del 2025

Este es el reporte de la primera práctica de laboratorio de la materia *Estructuras de Datos* impartida por el profesor Erick Quintero Villeda y su ayudante de laboratorio Sandra Valeria Rivera Lara.

Complicaciones al realizar la práctica

No fue muy difícil desarrollar la práctica, aunque recordar la sintaxis de Java y la lógica de programación orientada a objetos, si me requirió regresar a mis apuntes del semestre pasado de la clase de Introducción a la Ciencias de la Computación I.

¿Qué es la herencia en Java y cómo se utilizó en esta práctica?

Es la propiedad que tienen las clases “hijas” de adquirir características de las clases “padre”. En esta práctica se utilizó para definir una *super clase* *PoligonoRegular* en la que se definen varios atributos y métodos que nos sirvieron más adelante en sus clases hijas para poder heredar esas características y poder definir nuestros métodos en sus clases correspondientes de manera adecuada.

Definición formal del algoritmo calcularPerimetro (PoligonoRegular)

Entrada: Un objeto de tipo *PoligonoRegular*, junto con un atributo tipo *int* de nombre *numeroLados* que definirá el número de lados del poligono regular, y un atributo tipo *double* de nombre *longitudLado* que definirá la medida de la longitud de uno de los lados del poligono regular.

Salida: Un número que resulta el producto de *numeroLados*, por *longitudLado*. Es decir, el perímetro del poligono regular.

1. return *numeroLados* * *longitudLado*

Definición formal del algoritmo calcularPerimetro (Circulo)

Entrada: Un objeto de tipo *Circulo*, junto con un atributo tipo *double* de nombre *PI* que definirá el número redondeado de pi, y un atributo tipo *double* de nombre *radio* que definirá la medida de la longitud del radio del círculo.

Salida: Un número que resulta el producto de *PI*, por *radio*, por 2. Es decir, el perímetro del círculo.

1. return 2 * *PI* * *radio*

Definición formal del algoritmo calcularArea (TrianguloEquilatero)

Entrada: Un objeto de tipo *TrianguloEquilatero*, junto con un atributo tipo double de nombre *longitudLado* que definirá la medida de la longitud del lado del triángulo equilátero.

Salida: Un número que resulta la división del producto *longitudLado* al cuadrado, por la raíz de 3, entre 4. Es decir, el área del triángulo equilátero.

1. return $(\text{Math.pow}(\text{this.longitudLado}, 2) * \text{Math.sqrt}(3)) / 4$

Definición formal del algoritmo calcularArea (Pentagono)

Entrada: Un objeto de tipo *Pentagono*, junto con un atributo tipo double de nombre *longitudLado* que definirá la medida de la longitud del lado del pentagono, .

Salida: Un número que resulta la división del producto del perímetro del pentagono, por su apotema, entre dos. Es decir, el área del cuadrado.

1. return $(\text{calcularPerimetro}() * \text{calcularApotema}()) / 2$

Definición formal del algoritmo calcularArea (Cuadrado)

Entrada: Un objeto de tipo *Cuadrado*, junto con un atributo tipo double de nombre *longitudLado* que definirá la medida de la longitud del lado del cuadrado.

Salida: Un número que resulta la *longitudLado* al cuadrado. Es decir, el área del cuadrado.

1. return $\text{Math.pow}(\text{this.longitudLado}, 2)$

Definición formal del algoritmo calcularArea (Circulo)

Entrada: Un objeto de tipo *Circulo*, junto con un atributo tipo double de nombre *PI* que definirá el número redondeado de pi, y un atributo tipo double de nombre *radio* que definirá la medida de la longitud del radio del círculo.

Salida: Un número que resulta el producto de *PI*, por *radio* al cuadrado Es decir, el área del círculo.

1. return $PI * \text{Math.pow}(\text{this.radio}, 2)$

Responde a las siguientes preguntas

a. ¿Por qué sería una mala decisión de diseño que la clase *Circulo* extienda a la clase abstracta *PoligonoRegular*?

Porque la clase *Circulo* necesite atributos y métodos que la clase *PoligonoRegular* no tiene.

b. ¿Cuáles son las ventajas de que exista la interfaz **Figura**?

Nos sirve como una checklist de los métodos que tenemos que implementar luego en la clase abstracta *PoligonoRegular* y en la clase *Circulo*.

Respuestas de la IA

- a. ¿Por qué sería una mala decisión de diseño que la clase `Circulo` extienda a la clase abstracta `PoligonoRegular`?

Muy buena pregunta. Te explico el problema de diseño:

En geometría, un círculo no es un polígono regular.

Un polígono regular se define como una figura de lados rectos, todos iguales y con ángulos internos iguales (triángulo equilátero, cuadrado, pentágono regular, etc.). Un círculo, en cambio, no tiene lados ni ángulos, sino una curva continua.

¿Por qué sería un mal diseño que `Circulo` extienda a `PoligonoRegular`?

- a) Violación de la jerarquía lógica (herencia incorrecta) La herencia en programación modela una relación “es un”.
Un cuadrado es un polígono regular
Un círculo NO es un polígono regular Entonces, la relación no es válida.
- b) Comportamientos incoherentes Si `PoligonoRegular` define métodos como: `abstract double getNumeroDeLados();` `abstract double getLongitudDeLado();` Un círculo no puede implementar estos métodos de manera coherente (¿cuántos lados tiene? ¿infinito? ¿0?). Forzar a `Circulo` a implementarlos sería antinatural.
- c) Mala extensibilidad Si mañana quieres añadir más figuras (hexágonos, triángulos, etc.), la jerarquía se vuelve inconsistente porque `Circulo` estaría en una rama que no le corresponde.
- d) Violación del Principio de Sustitución de Liskov (LSP) Según este principio, cualquier subclase debe poder usarse donde se espera la superclase. Pero si en tu programa esperas un `PoligonoRegular` y pasas un `Circulo`, muchos métodos no funcionarían como se espera → romperías el LSP.

En resumen: sería mala decisión de diseño porque un círculo no cumple las propiedades de un polígono regular. Lo correcto es que ambos hereden de una clase común más general (`Figura`, `Forma`, etc.), pero no directamente uno del otro.

¿Quieres que te haga un diagrama UML rápido para que se vea más claro?

- b. ¿Cuáles son las ventajas de que exista la interfaz **Figura**?

¡Excelente! Vamos paso a paso con las ventajas de tener una interfaz `Figura` en un diseño orientado a objetos como el de tu práctica en Java:

- a) Unificación y generalización La interfaz `Figura` permite que todas las formas (círculos, cuadrados, triángulos, polígonos regulares, etc.) compartan un contrato común. Ejemplo: todas deben saber calcular área y perímetro.

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

Así puedes tratar a todas las figuras por igual sin importar su tipo específico.
- b) Polimorfismo Con `Figura`, puedes escribir código genérico que funcione con cualquier figura:

```
List<Figura> figuras = new ArrayList<>();
figuras.add(new Circulo(5));
figuras.add(new Cuadrado(4));
figuras.add(new TrianguloEquilatero(3));
for (Figura f : figuras) {
    System.out.println("Área: " + f.getArea());
    System.out.println("Perímetro: " + f.getPerimetro());
}
```

Aquí no importa si es círculo o polígono: el programa sabe cómo calcular gracias a la interfaz.
- c) Flexibilidad y extensibilidad Si en el futuro quieres agregar más figuras (Hexágono, Elipse, Trapecio), solo implementas la interfaz y no necesitas cambiar el código que ya usa `Figura`. Se sigue el principio abierto/cerrado: abierto para extensión, cerrado para modificación.
- d) Independencia de implementación Una interfaz no impone cómo se calculan área o perímetro, solo exige que se implementen. Esto permite que cada clase (`Circulo`, `PoligonoRegular`, etc.) resuelva los métodos de la forma que le corresponde.

e) Cumplimiento de buenas prácticas (SOLID)

Liskov Substitution: cualquier Figura puede sustituir otra en el mismo contexto.

Interface Segregation: defines solo el contrato esencial (getArea, getPerimetro) sin obligar a métodos inútiles.

Facilita inyección de dependencias si más adelante quieres desacoplar aún más.

En resumen: la interfaz Figura da abstracción, polimorfismo, flexibilidad y extensibilidad, permitiendo diseñar sistemas más ordenados y fáciles de mantener..

¿Quieres que te arme un ejemplo comparativo con y sin interfaz para que se vea más claro en código real de Java?

Análisis sobre las respuestas de la IA

En mi caso utilice a ChatGPT y en general creo que son buenas respuestas, pero muy cargadas de información, tal vez si fuera la primera vez que se habla del tema sobre herencia, agradecería tanta explicación, pero en esta ocasión creo que me sobre saturó con sus respuestas, además de que habla de cosas que creo están fuera de un contexto general de conocimiento sobre el tema, como es en el caso de las buenas prácticas SOLID o el cumplimiento del Principio de Sustitución de Liskov.