# Principios de POO aplicados en la Calculadora Científica

## Introducción

El presente trabajo desarrolla una aplicación en Kotlin que simula el funcionamiento de una calculadora científica, implementando los principios fundamentales de la Programación Orientada a Objetos (POO): encapsulamiento, herencia y polimorfismo.

El proyecto se estructura en tres clases principales:

- Calculadora: clase base que define las operaciones aritméticas básicas y la gestión de memoria.
- CalculadoraCientifica: subclase que amplía las funcionalidades agregando operaciones trigonométricas, logarítmicas, exponenciales, factoriales y matriciales.
- Matriz2x2: clase interna utilizada para representar matrices de dos dimensiones.

## Principios de la Programación Orientada a Objetos

### Encapsulamiento

El encapsulamiento se aplica mediante el uso de modificadores de acceso (protected, private) que restringen la visibilidad de los atributos internos:

protected var memoria: Double = 0.0

protected var resultadoActual: Double = 0.0

private var modoGrados: Boolean = true

Esto asegura que las variables no puedan ser modificadas directamente desde fuera de la clase, protegiendo la integridad de los datos.
En su lugar, se proporcionan métodos controlados (getters y setters) como memoriaObtener(), memoriaGuardar(), establecerResultado() para interactuar con esos atributos

### Herencia

La clase CalculadoraCientifica **hereda** de la clase Calculadora mediante la palabra clave ::

class CalculadoraCientifica : Calculadora()

De esta forma, la calculadora científica **reutiliza** todas las funciones básicas (sumar, restar, multiplicar, dividir) y **agrega** nuevas funcionalidades específicas como:

- Funciones trigonométricas (seno, coseno, tangente),

- Potencias y raíces (potencia, raizCuadrada, raizNesima),
- Logaritmos y exponenciales,
- Operaciones binarias y matriciales.

Esta relación de herencia refleja una jerarquía natural:

Una Calculadora Científica **es una** Calculadora, pero con más capacidades.

**Polimorfismo**

El polimorfismo se evidencia mediante **sobrecarga de métodos** en la clase Calculadora. Por ejemplo, el método sumar() se define tres veces con distintos tipos de parámetros:

open fun sumar(a: Double, b: Double): Double = a + b

open fun sumar(a: Int, b: Int): Int = a + b

open fun sumar(a: Float, b: Float): Float = a + b

Gracias a esto, el programa **decide automáticamente** cuál versión del método usar según los tipos de datos pasados como argumento.

Además, los métodos están marcados como open, permitiendo que en clases derivadas (como CalculadoraCientifica) puedan **ser sobreescritos** en caso de necesitar comportamientos diferentes.

## Diseño de la Solución

### Descripción del diseño

El sistema está estructurado de manera jerárquica y modular:

- La clase **Calculadora** define la estructura general y las operaciones aritméticas.
- La clase **CalculadoraCientifica** amplía el comportamiento añadiendo nuevas funciones científicas.
- La clase **Matriz2x2** se define dentro de la calculadora científica para representar matrices pequeñas y operar con ellas.

Esto promueve la **reutilización del código** y la **extensibilidad**, ya que se pueden crear nuevas clases derivadas (por ejemplo, CalculadoraProgramador) sin modificar la clase base.

## Resultados y Ejemplos Esperados

DEMOSTRACIÓN CALCULADORA CIENTÍFICA - PARADIGMA ORIENTADO A OBJETOS

1. DEMOSTRACIÓN DE HERENCIA - Operaciones Básicas

15.5 + 7.3 = 22.8

20.0 - 8.5 = 11.5

6.0 * 7.0 = 42.0

45.0 / 9.0 = 5.0

2. DEMOSTRACIÓN DE POLIMORFISMO

Suma de enteros: 12 + 8 = 20

Suma de doubles: 12.5 + 8.3 = 20.8

3. FUNCIONES CIENTÍFICAS - Trigonometría

Modo cambiado a GRADOS

sin(30.0°) = 0.49999999999999994

cos(60.0°) = 0.5000000000000001

tan(45.0°) = 0.9999999999999999

Modo cambiado a RADIANES

sin(0.5235987755982988 rad) = 0.49999999999999994

4. POTENCIAS Y RAÍCES

2.0 ^ 8.0 = 256.0

√64.0 = 8.0

√[3.0]27.0 = 3.0

5. LOGARITMOS Y EXPONENCIALES

log(100.0) = 2.0

ln(2.718281828459045) = 1.0

6. OPERACIONES CON BINARIOS

1010 (binario) = 10 (decimal)

15 (decimal) = 1111 (binario)

1010 (binario) = 10 (decimal)

1100 (binario) = 12 (decimal)

10 + 12 = 22

22 (decimal) = 10110 (binario)

1010 + 1100 = 10110 (en binario)

7. FUNCIONES DE MEMORIA (ENCAPSULAMIENTO)

MS | Memoria guardada: 100.0

M+ 25.0 | Memoria actual: 125.0

M- 15.0 | Memoria actual: 110.0

MR | Memoria recuperada: 110.0

8. EXPRESIONES COMPUESTAS

Evaluando expresión: 25+15

25.0 + 15.0 = 40.0

Evaluando expresión: 6*7

6.0 * 7.0 = 42.0

Evaluando expresión: sin(30)

sin(30.0°) = 0.49999999999999994

Evaluando expresión: log(100)

log(100.0) = 2.0

9. MANEJO DE EXCEPCIONES

EXCEPCIÓN CAPTURADA: Error: División por cero

EXCEPCIÓN CAPTURADA: Error: Raíz cuadrada de número negativo

10. FUNCIONES AVANZADAS

5! = 120

DEMOSTRACIÓN COMPLETADA - TODOS LOS CONCEPTOS DE POO VERIFICADOS

- ✓ Herencia: CalculadoraCientifica hereda de Calculadora

- ✓ Polimorfismo: Sobrecarga de métodos sumar()

- ✓ Encapsulamiento: Funciones de memoria protegidas

- ✓ Manejo de excepciones: Divisiones y raíces inválidas

## Conclusión

El código presentado demuestra la correcta aplicación de los principios de la POO en Kotlin.
A través de encapsulamiento, se asegura la protección de los datos internos; con herencia, se facilita la reutilización y extensión del código; y mediante polimorfismo, se logra flexibilidad en las operaciones.

El diseño modular permite agregar nuevas funcionalidades científicas o programables sin alterar la estructura base, cumpliendo así con los principios de un software mantenible y escalable.