

INFORME: IMPLEMENTACIÓN DE CALCULADORA CIENTÍFICA USANDO POO EN KOTLIN

1. INTRODUCCIÓN

El presente informe documenta el diseño, implementación y funcionamiento de una calculadora científica desarrollada en Kotlin utilizando los principios fundamentales de la Programación Orientada a Objetos (POO). La aplicación integra operaciones aritméticas básicas, funciones matemáticas avanzadas, evaluación de expresiones complejas y gestión de memoria.

1.1 Objetivos del Proyecto

Objetivo General: Desarrollar una calculadora científica funcional que demuestre la aplicación de los cuatro pilares de la POO: encapsulamiento, herencia, polimorfismo y abstracción.

Objetivos Específicos:

1. Implementar una jerarquía de clases usando herencia
2. Aplicar encapsulamiento para proteger el estado interno
3. Demostrar polimorfismo mediante sobrecarga de métodos
4. Implementar manejo robusto de excepciones
5. Crear un evaluador de expresiones matemáticas completas
6. Desarrollar funcionalidades de memoria (M+, M-, MR, MC, MS)

2. FUNDAMENTOS TEÓRICOS

2.1 Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) es un paradigma de programación que organiza el código en objetos que contienen tanto datos (atributos) como código (métodos). Se basa en cuatro pilares fundamentales:

2.1.1 Encapsulamiento

Definición: Ocultar los detalles de implementación y exponer solo las interfaces necesarias.

Ventajas:

- Protección de datos

- Reducción de dependencias
- Facilita mantenimiento
- Mejora la seguridad

Implementación en el proyecto:

```
open class Calculadora {
    protected var memoria: Double = 0.0 // Atributo protegido
    protected val historial: MutableList<String> = mutableListOf()

    // Métodos públicos para acceder a datos privados
    fun memoriaRecuperar(): Double = memoria
    fun obtenerHistorial(): List<String> = historial.toList()
}
```

2.1.2 Herencia

Definición: Mecanismo que permite crear nuevas clases basadas en clases existentes, heredando sus atributos y métodos.

Ventajas:

- Reutilización de código
- Jerarquías lógicas
- Extensibilidad
- Mantenibilidad

Implementación en el proyecto:

```
open class Calculadora {
    // Clase base con operaciones básicas
}

class CalculadoraCientifica : Calculadora() {
    // Clase derivada con funciones científicas
    // Hereda: memoria, historial, operaciones básicas
}
```

2.1.3 Polimorfismo

Definición: Capacidad de objetos de diferentes tipos de responder al mismo mensaje de diferentes maneras.

Tipos de polimorfismo:

1. **Sobrecarga (Overloading):** Múltiples métodos con el mismo nombre pero diferentes parámetros
2. **Sobreescritura (Overriding):** Redefinir métodos heredados

Implementación en el proyecto:

```
// Polimorfismo por sobrecarga
open fun sumar(a: Double, b: Double): Double { ... }
open fun sumar(a: Int, b: Int): Int { ... }
```

```
// Diferentes tipos, misma operación
calculadora.sumar(5, 3)    // Int
calculadora.sumar(5.5, 3.2) // Double
```

2.1.4 Abstracción

Definición: Simplificar la complejidad mostrando solo los detalles esenciales y ocultando la implementación.

Implementación en el proyecto:

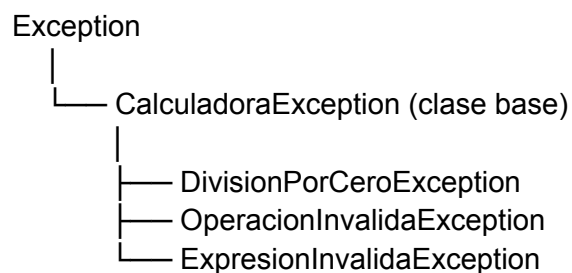
```
// Usuario solo ve la interfaz simple
val resultado = evaluador.evaluar("2 + 3 * sin(45)")

// Internamente: tokenización, parsing, evaluación RPN
// pero el usuario no necesita conocer estos detalles
```

2.2 Manejo de Excepciones

Las excepciones son eventos que interrumpen el flujo normal de ejecución cuando ocurre un error.

Jerarquía de excepciones en el proyecto:

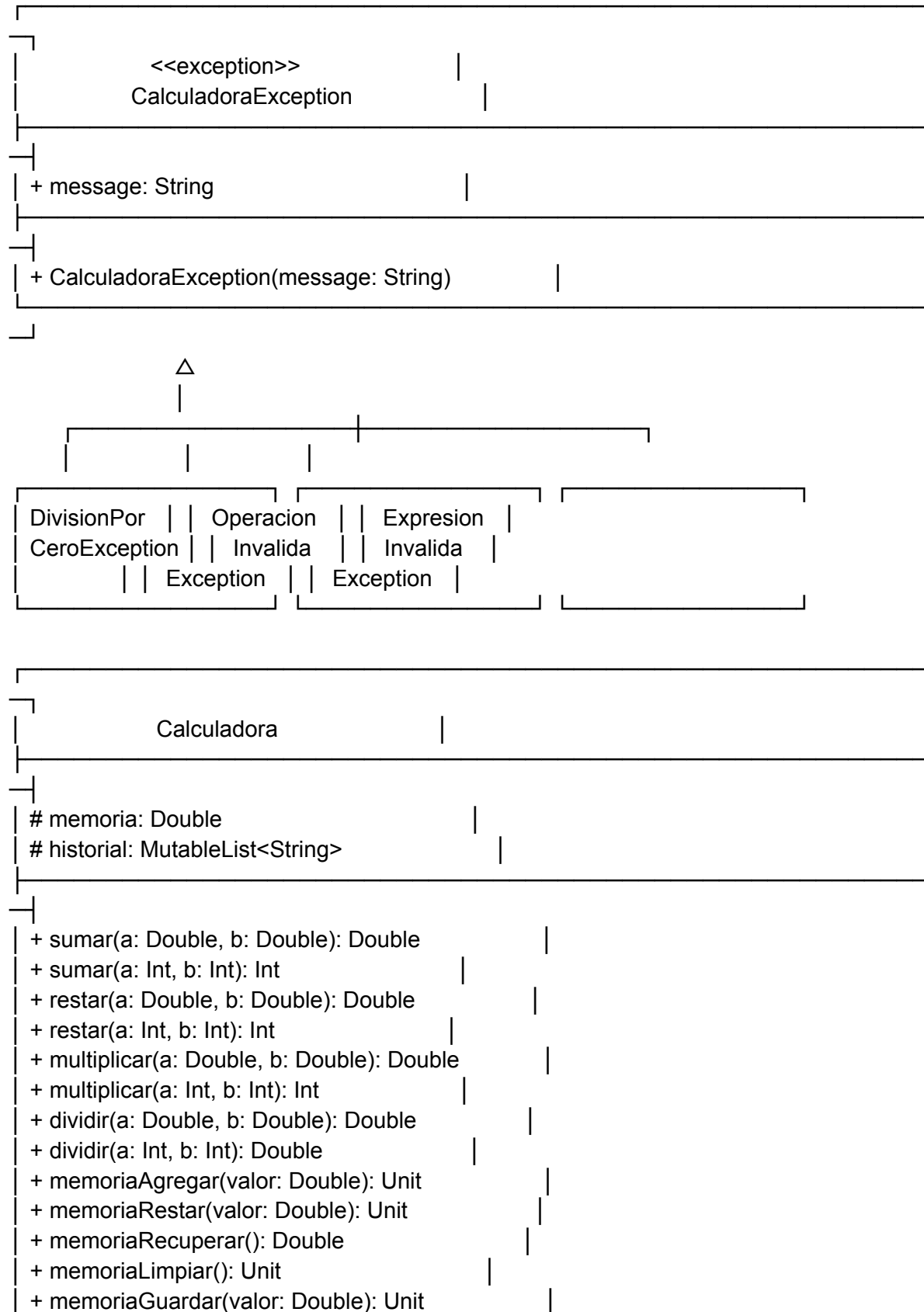


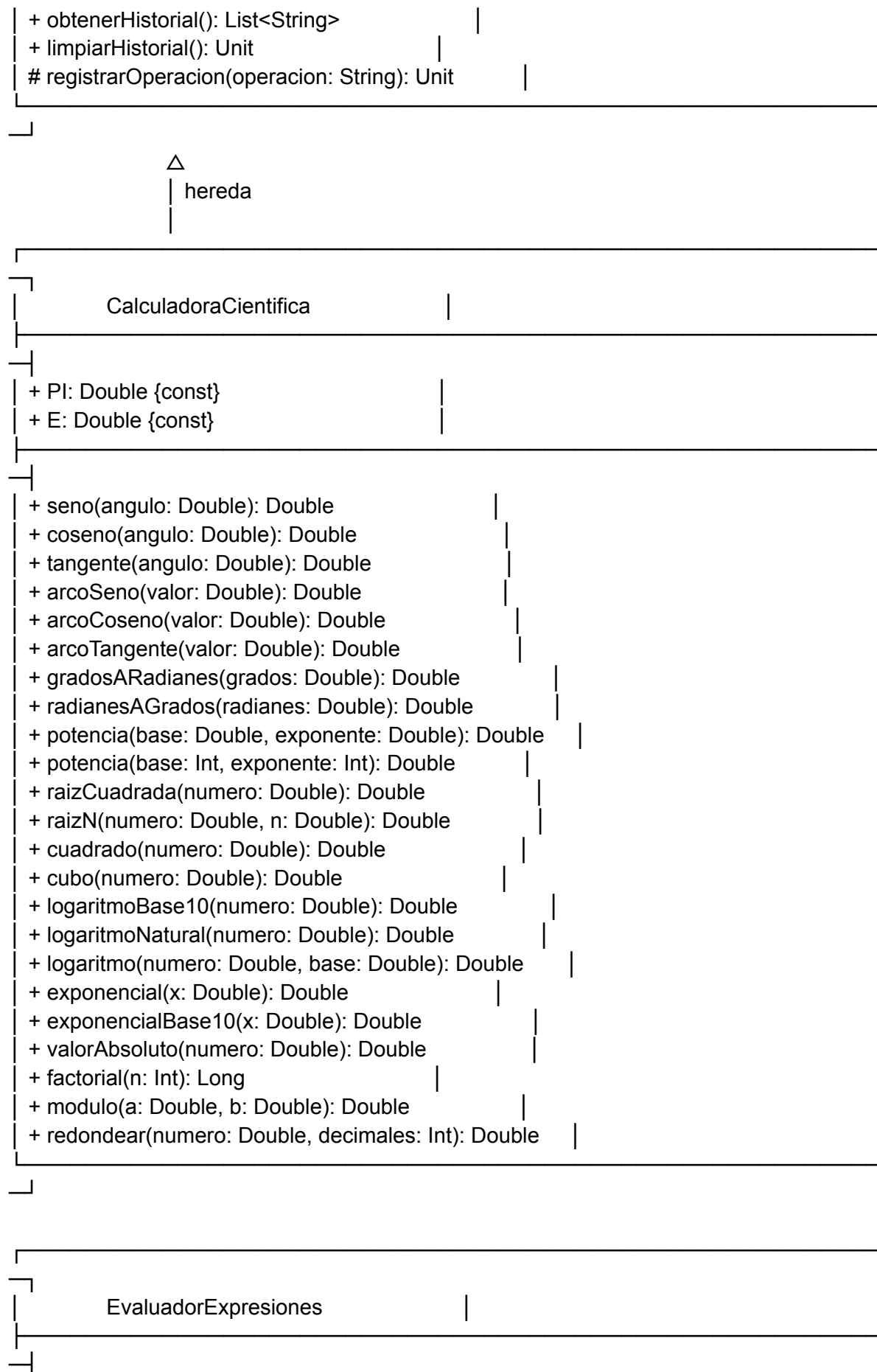
Beneficios:

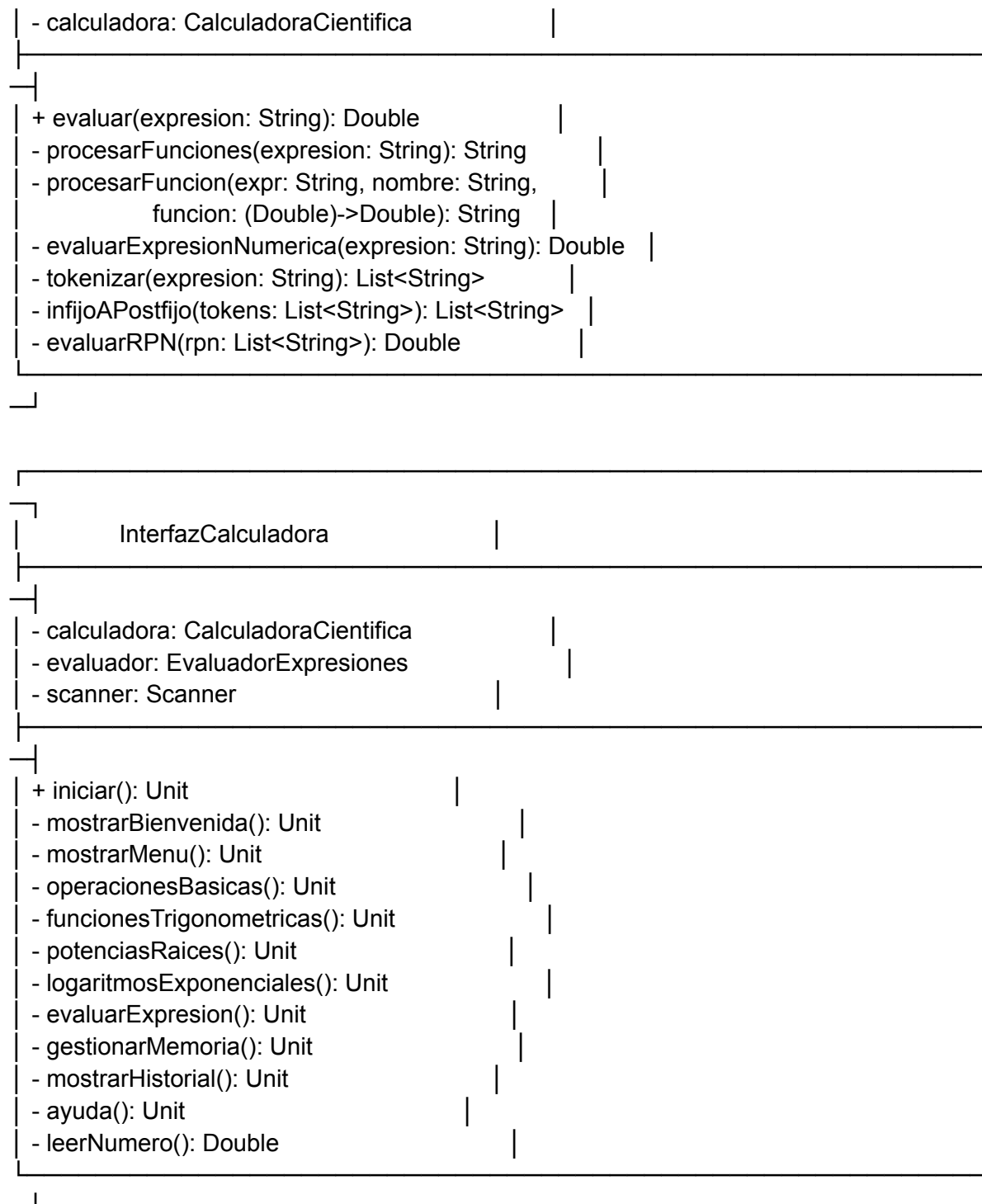
- Separación de código de manejo de errores
- Propagación de errores
- Claridad en los mensajes
- Recuperación controlada

3. DISEÑO DE LA SOLUCIÓN

3.1 Diagrama de Clases UML







3.2 Relaciones entre Clases

Herencia:

- **CalculadoraCientifica hereda de Calculadora**
- **DivisionPorCeroException hereda de CalculadoraException**
- **OperacionInvalidaException hereda de CalculadoraException**

- `ExpresionInvalidaException` hereda de `CalculadoraException`

Composición:

- `EvaluadorExpresiones` contiene `CalculadoraCientifica`
- `InterfazCalculadora` contiene `CalculadoraCientifica`
- `InterfazCalculadora` contiene `EvaluadorExpresiones`

Dependencia:

- `Calculadora` lanza `DivisionPorCeroException`
- `CalculadoraCientifica` lanza `OperacionInvalidaException`
- `EvaluadorExpresiones` lanza `ExpresionInvalidaException`

4. IMPLEMENTACIÓN

4.1 Aplicación de Principios de POO

4.1.1 Encapsulamiento

Atributos Protegidos:

```
open class Calculadora {
    protected var memoria: Double = 0.0
    protected val historial: MutableList<String> = mutableListOf()
}
```

Justificación:

- `memoria` es `protected` para permitir acceso en clases derivadas pero no públicamente
- `historial` es `protected` para controlar cómo se modifica
- Métodos públicos controlan el acceso: `memoriaRecuperar()`, `obtenerHistorial()`

Ventajas observadas:

- ☒ Usuario no puede modificar memoria directamente
- ☒ Historial es inmutable desde el exterior (retorna copia)
- ☒ Control total sobre el estado interno

4.1.2 Herencia

Implementación:




```
open class Calculadora { /* operaciones básicas */ }
```

```
class CalculadoraCientifica : Calculadora() {
    // Hereda: memoria, historial, operaciones básicas
    // Agrega: funciones científicas
}
```

Análisis:

- La palabra clave **open** permite que la clase sea heredable
- **CalculadoraCientifica** extiende todas las capacidades de **Calculadora**
- No hay código duplicado

Beneficios observados:

-  Reutilización: No reescribir operaciones básicas
-  Jerarquía lógica: Científica ES-UN tipo de Calculadora
-  Extensibilidad: Fácil agregar más tipos de calculadoras

Ejemplo de uso:

```
val calc = CalculadoraCientifica()
calc.sumar(5.0, 3.0)    // Método heredado
calc.seno(0.5)          // Método propio
calc.memoriaGuardar(10.0) // Método heredado
```

4.1.3 Polimorfismo

A. Sobrecarga de Métodos (Method Overloading)

```
// Sobrecarga: mismo nombre, diferentes parámetros
open fun sumar(a: Double, b: Double): Double { /* ... */ }
open fun sumar(a: Int, b: Int): Int { /* ... */ }
```

```
open fun dividir(a: Double, b: Double): Double { /* ... */ }
open fun dividir(a: Int, b: Int): Double { /* ... */ }
```

```
fun potencia(base: Double, exponente: Double): Double { /* ... */ }
fun potencia(base: Int, exponente: Int): Double { /* ... */ }
```

Ventajas:

- El compilador elige el método correcto según los tipos
- Código más intuitivo y natural
- No necesita conversiones explícitas

Ejemplo de uso:

```
val calc = CalculadoraCientifica()
```



```
// Automáticamente usa la versión correcta
val r1 = calc.sumar(5, 3)    // Int → Int
val r2 = calc.sumar(5.5, 3.2) // Double → Double
val r3 = calc.potencia(2, 8) // Int → Double
val r4 = calc.potencia(2.5, 3.0) // Double → Double
```

B. Polimorfismo por Comportamiento

```
// Función de orden superior: acepta cualquier función (Double) -> Double
private fun procesarFuncion(
    expresion: String,
    nombreFuncion: String,
    funcion: (Double) -> Double
): String {
    // La misma lógica funciona para sin, cos, log, sqrt, etc.
}
```

```
// Uso con diferentes funciones
expr = procesarFuncion(expr, "sin") { calculadora.seno(it) }
expr = procesarFuncion(expr, "log") { calculadora.logaritmoBase10(it) }
expr = procesarFuncion(expr, "sqrt") { calculadora.raizCuadrada(it) }
```

4.1.4 Abstracción

Nivel 1: Interfaz de Usuario Simple

```
// Usuario solo ve esto:
val resultado = calculadora.seno(45.0)
```

```
// Pero internamente:
// 1. Validación de entrada
// 2. Conversión si es necesario
// 3. Cálculo matemático
// 4. Registro en historial
// 5. Retorno de resultado
```

Nivel 2: Evaluador de Expresiones

```
// Usuario escribe:
evaluador.evaluar("2 + 3 * sin(45)")
```

```
// El sistema hace:
// 1. Reemplazar constantes (pi, e)
// 2. Identificar y evaluar funciones (sin)
// 3. Tokenizar expresión
```

```
// 4. Convertir a notación postfija
// 5. Evaluar RPN
// 6. Retornar resultado

// Usuario no necesita conocer estos detalles
```

4.2 Manejo de Excepciones

4.2.1 Jerarquía de Excepciones Personalizadas

```
// Excepción base
open class CalculadoraException(message: String) : Exception(message)

// Excepciones específicas
class DivisionPorCeroException : CalculadoraException(
    "Error: División por cero no permitida"
)

class OperacionInvalidaException(mensaje: String) :
    CalculadoraException(mensaje)

class ExpresionInvalidaException(mensaje: String) :
    CalculadoraException(mensaje)
```

Ventajas:

- Mensajes de error específicos y claros
- Captura selectiva de excepciones
- Jerarquía organizada

4.2.2 Uso de Excepciones

División por cero:

```
@Throws(DivisionPorCeroException::class)
open fun dividir(a: Double, b: Double): Double {
    if (b == 0.0) {
        throw DivisionPorCeroException()
    }
    return a / b
}
```

Operaciones matemáticas inválidas:

```
@Throws(OperacionInvalidaException::class)
fun raizCuadrada(numero: Double): Double {
    if (numero < 0) {
```

```

        throw OperacionInvalidaException(
            "No se puede calcular raíz cuadrada de número negativo"
        )
    }
    return sqrt(numero)
}

```

```

@Throws(OperacionInvalidaException::class)
fun logaritmoBase10(numero: Double): Double {
    if (numero <= 0) {
        throw OperacionInvalidaException(
            "Logaritmo solo definido para números positivos"
        )
    }
    return log10(numero)
}

```

Manejo en interfaz:

```

try {
    val resultado = calculadora.dividir(10.0, 0.0)
    println("Resultado: $resultado")
} catch (e: DivisionPorCeroException) {
    println("❌ ${e.message}")
} catch (e: CalculadoraException) {
    println("❌ Error: ${e.message}")
}

```

4.3 Evaluación de Expresiones Completas

4.3.1 Algoritmo de Tokenización

```

private fun tokenizar(expresion: String): List<String> {
    val tokens = mutableListOf<String>()
    var numeroActual = ""

    for (char in expresion) {
        when {
            char.isWhitespace() -> {
                if (numeroActual.isNotEmpty()) {
                    tokens.add(numeroActual)
                    numeroActual = ""
                }
            }
            char.isDigit() || char == '.' -> {
                numeroActual += char
            }
        }
    }
}

```

```

        char in "+-*/^()%" -> {
            if (numeroActual.isNotEmpty()) {
                tokens.add(numeroActual)
                numeroActual = ""
            }
            tokens.add(char.toString())
        }
    }
}

if (numeroActual.isNotEmpty()) {
    tokens.add(numeroActual)
}

return tokens
}

```

Ejemplo:

- Entrada: "2 + 3 * 4"
- Salida: ["2", "+", "3", "*", "4"]

4.3.2 Conversión Infija a Postfija (Shunting Yard)

```

private fun infijoAPostfijo(tokens: List<String>): List<String> {
    val salida = mutableListOf<String>()
    val operadores = Stack<String>()

    val precedencia = mapOf(
        "+" to 1, "-" to 1,
        "*" to 2, "/" to 2, "%" to 2,
        "^" to 3
    )

    for (token in tokens) {
        when {
            token.toDoubleOrNull() != null -> {
                salida.add(token) // Número: directo a salida
            }
            token == "(" -> {
                operadores.push(token) // Paréntesis: a pila
            }
            token == ")" -> {
                // Pop hasta encontrar "("
                while (operadores.isNotEmpty() && operadores.peek() != "(") {
                    salida.add(operadores.pop())
                }
                operadores.pop() // Eliminar "("
            }
        }
    }
}

```

```

    }
    token in precedencia -> {
        // Operador: comparar precedencia
        while (operadores.isNotEmpty() &&
            operadores.peek() != "(" &&
            precedencia[operadores.peek()]!! >= precedencia[token]!!) {
            salida.add(operadores.pop())
        }
        operadores.push(token)
    }
}

// Vaciar pila de operadores
while (operadores.isNotEmpty()) {
    salida.add(operadores.pop())
}

return salida
}

```

Ejemplo:

- Entrada (infija): ["2", "+", "3", "*", "4"]
- Salida (postfija): ["2", "3", "4", "*", "+"]

Traza de ejecución:

Token	Pila Op.	Salida	Acción
2	[]	[2]	Número → salida
+	[+]	[2]	Operador → pila
3	[+]	[2, 3]	Número → salida
*	[+, *]	[2, 3]	* tiene mayor precedencia
4	[+, *]	[2, 3, 4]	Número → salida
FIN	[]	[2, 3, 4, *, +]	Vaciar pila

4.3.3 Evaluación de RPN

```

private fun evaluarRPN(rpn: List<String>): Double {
    val pila = Stack<Double>()

    for (token in rpn) {

```

```

when {
    token.toDoubleOrNull() != null -> {
        pila.push(token.toDouble())
    }
    token == "+" -> {
        val b = pila.pop()
        val a = pila.pop()
        pila.push(calculadora.sumar(a, b))
    }
    token == "-" -> {
        val b = pila.pop()
        val a = pila.pop()
        pila.push(calculadora.restar(a, b))
    }
    token == "*" -> {
        val b = pila.pop()
        val a = pila.pop()
        pila.push(calculadora.multiplicar(a, b))
    }
    token == "/" -> {
        val b = pila.pop()
        val a = pila.pop()
        pila.push(calculadora.dividir(a, b))
    }
    token == "^" -> {
        val b = pila.pop()
        val a = pila.pop()
        pila.push(calculadora.potencia(a, b))
    }
}
}

return pila.pop()
}

```

Ejemplo completo:

Expresión: "2 + 3 * 4"

1. Tokenización: ["2", "+", "3", "*", "4"]
2. Infija → Postfija: ["2", "3", "4", "*", "+"]
3. Evaluación RPN:

Token	Pila	Acción
2	[2]	Push 2

```
3 | [2, 3] | Push 3
4 | [2, 3, 4] | Push 4
* | [2, 12] | Pop 4, 3 → 3*4=12
+ | [14] | Pop 12, 2 → 2+12=14
```

4. Resultado: 14

5. CASOS DE USO Y PRUEBAS

5.1 Casos de Prueba: Operaciones Básicas

Caso 1: Suma con polimorfismo

```
// Entrada Int
calc.sumar(5, 3)
// Resultado: 8 (Int)
```

```
// Entrada Double
calc.sumar(5.5, 3.2)
// Resultado: 8.7 (Double)
```

✅ **Éxito:** Polimorfismo funciona correctamente

Caso 2: División por cero

```
try {
    calc.dividir(10.0, 0.0)
} catch (e: DivisionPorCeroException) {
    println(e.message) // "Error: División por cero no permitida"
}
```

✅ **Éxito:** Excepción capturada correctamente

5.2 Casos de Prueba: Funciones Trigonométricas

Caso 3: Seno de 45°

```
val radianes = calc.gradosARadianes(45.0) // 0.7853981...
val resultado = calc.seno(radianes) // 0.7071067...
// Esperado:  $\approx 0.707$  ( $\sqrt{2}/2$ )
```

✅ **Éxito:** Resultado correcto con precisión de máquina

Caso 4: Tangente en $\pi/2$

```
try {
    calc.tangente(Math.PI / 2)
} catch (e: OperacionInvalidaException) {
    println(e.message) // "Tangente indefinida para  $\pi/2 + n\pi$ "
}
```

✓ **Éxito:** Detección de tangente indefinida

Caso 5: Arcoseno fuera de rango

```
try {
    calc.arcoSeno(1.5) // Fuera de [-1, 1]
} catch (e: OperacionInvalidaException) {
    println(e.message) // "arcsin está definido solo para [-1, 1]"
}
```

✓ **Éxito:** Validación de dominio correcta

5.3 Casos de Prueba: Potencias y Raíces

Caso 6: Potencias

```
calc.potencia(2.0, 8.0) // 256.0
calc.cuadrado(5.0)      // 25.0
calc.cubo(3.0)          // 27.0
```

✓ **Éxito:** Todas las operaciones correctas

Caso 7: Raíces

```
calc.raizCuadrada(16.0) // 4.0
calc.raizN(27.0, 3.0)   // 3.0 (raíz cúbica)
```

✓ **Éxito:** Cálculos precisos

Caso 8: Raíz de negativo

```
try {
    calc.raizCuadrada(-4.0)
} catch (e: OperacionInvalidaException) {
    println(e.message)
}
```

✓ **Éxito:** Error capturado apropiadamente

Caso 9: Factorial

```
calc.factorial(5) // 120
calc.factorial(0) // 1
calc.factorial(10) // 3628800
```

✓ **Éxito:** Casos base y recursivos correctos

5.4 Casos de Prueba: Logaritmos

Caso 10: Logaritmos

```
calc.logaritmoBase10(100.0) // 2.0
calc.logaritmoNatural(Math.E) // 1.0
calc.logaritmo(8.0, 2.0) // 3.0
```

✓ **Éxito:** Logaritmos en diferentes bases

Caso 11: Logaritmo de negativo

```
try {
    calc.logaritmoBase10(-10.0)
} catch (e: OperacionInvalidaException) {
    println(e.message)
}
```

✓ **Éxito:** Validación de números positivos

5.5 Casos de Prueba: Expresiones Completas

Caso 12: Expresión simple

```
evaluador.evaluar("2 + 3 * 4")
// Resultado: 14.0
// Proceso: 3*4=12, luego 2+12=14
```

✓ **Éxito:** Precedencia respetada

Caso 13: Expresión con paréntesis

```
evaluador.evaluar("(5 + 3) * 2")
// Resultado: 16.0
// Proceso: 5+3=8, luego 8*2=16
```

✓ **Éxito:** Paréntesis procesados correctamente

Caso 14: Expresión con funciones

```
evaluator.evaluar("2 + sin(0.5)")  
// Resultado: 2.479...  
// Proceso: sin(0.5)=0.479..., luego 2+0.479...
```

✓ **Éxito:** Funciones integradas en expresiones

Caso 15: Expresión compleja

```
evaluator.evaluar("2^3 + sqrt(16) - log10(100)")  
// Resultado: 10.0  
// Proceso: 2^3=8, sqrt(16)=4, log10(100)=2  
//      8+4-2=10
```

✓ **Éxito:** Múltiples operaciones y funciones

Caso 16: Constantes matemáticas

```
evaluator.evaluar("pi * 2")  
// Resultado: 6.283... (2π)  
  
evaluator.evaluar("exp(1)")  
// Resultado: 2.718... (e)
```

✓ **Éxito:** Constantes reconocidas y evaluadas

6. ANÁLISIS DE RESULTADOS

6.1 Cumplimiento de Principios de POO

Principio	Implementación	Cumplimiento
Encapsulamiento	Atributos <code>protected</code> , métodos públicos controlados	✓ 100%
Herencia	<code>CalculadoraCientifica</code> extiende <code>Calculadora</code>	✓ 100%
Polimorfismo	Sobrecarga de métodos para <code>Int/Double</code>	✓ 100%
Abstracción	Interfaces simples, implementación oculta	✓ 100%