Proyecto: La Ruleta de la Fortuna

Enunciado

El objetivo es desarrollar una versión del concurso "La Ruleta de la Fortuna" en **Kotlin**, aplicando principios de **Programación Orientada a Objetos (POO)** moderna y una arquitectura que separe claramente la **lógica del juego** de la **interfaz de usuario** (consola por ahora).

Esta separación es crítica porque en el futuro migraremos la interfaz a **Jetpack Compose**, y una buena arquitectura nos permitirá hacerlo sin tocar la lógica del juego.

Reglas del Juego

- **Jugadores**: Pueden ser humanos o la computadora (IA). Cada uno tiene un monedero que empieza en 0€.
- **La Ruleta**: Contiene gajos con dinero (positivo), y gajos especiales:
  - **Quiebra**: El jugador pierde todo su dinero acumulado en la ronda.
  - **Pierde Turno**: Pasa el turno al siguiente jugador.
- **Mecánica**:
  - Se juegan X rondas (definido en constantes).
  - En cada turno, el jugador tira la ruleta.
    - Si sale dinero: Puede pedir una **consonante**.
      - Si la consonante está en la frase: Gana (Valor x Veces que aparece). **Vuelve a tirar**.
      - Si no está: **Pierde turno**.
    - Si tiene suficiente dinero (coste de vocal), puede **comprar una vocal** antes de tirar o pasar.
    - En cualquier momento de su turno (si tiene el control), puede intentar **resolver el panel**.
- **Ganador**: El jugador con más dinero acumulado al final de todas las rondas.

Especificaciones Técnicas

1. Arquitectura y Paquetes

El proyecto debe estructurarse en paquetes para separar responsabilidades:

- `modelo` : Clases de datos y logica pura (`Jugador` (abstracta), `Panel` , `TiposDeResultado`).
- `motor` : La clase controladora del juego (`MotorJuego`).

- `datos` : Gestión de lectura de ficheros (`RepositorioFrases`).
- `ui` : Interfaz de usuario (Entrada/Salida por consola).

2. Estructura de Carpetas

La solución inicial debe tener la siguiente estructura de carpetas dentro de `src/main/kotlin` :

```text
src/main/kotlin/
├── datos
│   └── RepositorioFrases.kt      // Interfaz y clases de acceso a datos
├── modelo
│   ├── Frase.kt                  // Data class simple
│   ├── Jugador.kt                // Clase abstracta base
│   ├── JugadorIA.kt              // Implementación de IA
│   └── ResultadoRuleta.kt        // Sealed Interfaces para resultados
├── motor
│   └── MotorJuego.kt             // Lógica principal (Game Controller)
└── ui
    └── Main.kt                   // Punto de entrada y gestión de consola
```

IMPORTANTE: Las clases del paquete `modelo` y `motor` **NO PUEDEN TENER `print` ni `scanner`** . Toda la comunicación con el usuario debe ocurrir en el `main` .

(Esto es para que podamos reutilizar el código en futuras versiones con interfaz gráfica)

2. Clases y Conceptos Clave

A. Tipos de Resultado (Sealed Interfaces)

En lugar de usar números mágicos (-1, 0, etc), usa `sealed interfaces` para modelar los resultados de la ruleta y de las acciones.

> Revisa los apuntes de Kotlin y estudia la sección sobre sealed interfaces.

```kotlin
sealed interface ResultadoRuleta {
    data class Premio(val cantidad: Int) : ResultadoRuleta
    object Quiebra : ResultadoRuleta
    object PierdeTurno : ResultadoRuleta
}

sealed interface ResultadoAccion {
    object Acierto : ResultadoAccion
    object Fallo : ResultadoAccion
    object PanelResuelto : ResultadoAccion
    // ... otros estados necesarios
}
```

B. Repositorio de Frases (Memoria vs Fichero)

Para facilitar el inicio, comenzaremos con frases en memoria (una lista en código). La lectura de fichero o de API es una opción de implementación a futuro.

```kotlin
interface RepositorioFrases {
    fun obtenerFraseAleatoria(): Frase
}

// Implementación inicial
class RepositorioFrasesMemoria : RepositorioFrases {
    val frases = listOf(...)
    // ...
}
```

C. Clase `MotorJuego`

Esta clase contiene el estado de la partida:

- Jugadores (lista).
- Turno actual (index del jugador).
- Panel actual (letras descubiertas).
- Monto acumulado en la ronda.

Debe exponer métodos como:

- `fun girarRuleta(): ResultadoRuleta`
- `fun intentarLetra(letra: Char): Boolean`
- `fun comprarVocal(vocal: Char): Boolean`
- `fun resolverPanel(frase: String): Boolean`

D. Jugador Computadora (Herencia y Threads)

La clase `Jugador` debe ser **ABSTRACTA** para practicar herencia y polimorfismo.

Para simular que la IA "piensa", en esta versión de consola usaremos `Thread.sleep(1500)`. *Nota: En futuras versiones con interfaz gráfica, cambiaremos esto para no bloquear la pantalla.*

```kotlin
abstract class Jugador(val nombre: String) {
    var dinero: Int = 0
    abstract fun realizarMovimiento(motor: MotorJuego)
}

class JugadorIA : Jugador("Computadora") {
    override fun realizarMovimiento(motor: MotorJuego) {
        Thread.sleep(1500) // "Pensando..."
        // lógica de la IA
    }
}
```

3. Pruebas Unitarias

Se incluye un fichero de pruebas en `src/test/kotlin/motor/MotorTest.kt` y `src/test/kotlin/modelo/JugadorTest.kt`.

Para ejecutarlo en Android Studio:

1. Abre el archivo `MotorTest.kt` y `JugadorTest.kt`.
2. Haz clic en el icono verde de "Play" al lado de la clase `MotorTest` y `JugadorTest` o de cada método `@Test`.
3. Tu código debe pasar estas pruebas para considerarse correcto.

Nota: Algunos tests están comentados. Descoméntalos a medida que implementes la lógica.

> Los test seguramente muchos no los habéis visto en clase, ni trabajado con este concepto, pero es una excelente forma de probar el código, además de ser algo fundamental para el desarrollo de software moderno. Para conocer más, busca en internet "pruebas unitarias en Kotlin" o "pruebas unitarias en Java".

Anexos (información adicional)

Anexo I: Arquitectura y Conceptos

1. El Modelo (Model)

En arquitectura de software, el **Modelo** representa los datos y la lógica de negocio. Son clases que "saben" cómo jugar o qué datos existen, pero "no saben" cómo se muestran.

- Ejemplo: `Jugador` tiene dinero, pero no sabe si ese dinero se pinta en verde en una pantalla o se imprime en texto en la consola. Esto permite reutilizar `Jugador` en una app móvil, web o de consola sin cambiar código.

2. Sealed Interfaces

Son como enumerados (`enum`) supervitaminados. Permiten definir una jerarquía cerrada de clases.

- Mientras que un `enum` solo tiene valores constantes, una subclase de una `sealed interface` puede contener datos (como `Premio(500)`).
- Obligan a usar `when` exhaustivos, reduciendo bugs por casos no controlados.

  Ejemplo de implementación (Motor):

  ```kotlin
  fun girarRuleta(): ResultadoRuleta {
      val aleatorio = (1..24).random()
      return when (aleatorio) {
          1 -> ResultadoRuleta.Quiebra
          2 -> ResultadoRuleta.PierdeTurno
          else -> ResultadoRuleta.Premio(50) // Simulado
      }
  }
  ```

  Ejemplo de uso (UI):

```kotlin
val resultado = motor.girarRuleta()
when (resultado) {
    is ResultadoRuleta.Premio -> {
        println("¡Has ganado ${resultado.cantidad}€!")
        jugador.sumarDinero(resultado.cantidad)
    }
    ResultadoRuleta.Quiebra -> {
        println("¡Oh no! Lo has perdido todo.")
        jugador.quiebra()
    }
    ResultadoRuleta.PierdeTurno -> {
        println("Pierdes el turno.")
    }
}
```

3. Asincronía

- `Thread.sleep(ms)` : Duerme al hilo actual. Si lo usas en el hilo principal de una app móvil, la pantalla se congela y el usuario cree que la app se ha roto.
- **Corrutinas** (`suspend fun`): Permiten "pausar" la ejecución de una función sin bloquear el hilo, permitiendo que la interfaz siga respondiendo. (Lo veremos más adelante).

Anexo IV: Ejemplo de Ejecución Simulada

Así debería verse una partida típica en la consola:

text

```
 Bienvenido a La Ruleta de la Fortuna

> Introduce el nombre del Jugador 1: Ana
> Introduce el nombre del Jugador 2 (IA): Bot

--- RONDA 1 ---
Categoría: PELÍCULAS
Pista: Clásico del cine
Panel: _ _   _ _ _   _ _ _ _ _ _   _ _   _ _ _ _ _

>> Turno de Ana (0€)
[1] Girar Ruleta  [2] Comprar Vocal (250€)  [3] Resolver
> 1

¡La ruleta gira...!
 Resultado: 100€
> Introduce una consonante: L
¡Correcto! La letra 'L' aparece 4 veces.
Ganancia turno: 400€ (100€ x 4)

Panel: L _   _ _ _   _ _ _ _ _ _   _ _   L L _ _ _

>> Turno de Ana (400€)
[1] Girar Ruleta  [2] Comprar Vocal (250€)  [3] Resolver
> 2
> Introduce una vocal: E
¡Correcto! La letra 'E' aparece 2 veces.
Coste vocal: -250€

Panel: L _   _ _ E   _ I E _ _ _   _ E   L L E _ _

>> Turno de Ana (150€)
[1] Girar Ruleta  [2] Comprar Vocal (250€)  [3] Resolver
> 1
¡La ruleta gira...!
 Resultado: Quiebra
¡Oh no! Ana ha perdido todo su dinero de la ronda.

>> Turno de Bot (0€)
 Bot está pensando...
 Bot decide: Girar Ruleta
¡La ruleta gira...!
 Resultado: 50€
 Bot elige consonante: S
¡Correcto! La letra 'S' aparece 2 veces.

...
```