CICLO: DAM/DAW
PROGRAMACIÓN

INFORME AHORCADO

Alumno:
DANIEL GARCIA ORTEGA
76667976Y

Contenido

(Una vez realizado el informe, no olvidar actualizar esta tabla del índice **(F9 + Actualizar toda la tabla)**, con el fin de que se actualicen todos los epígrafes y números de página)

Informe del proyecto Juego del Ahorcado RA02

1. Introducción y Descripción del Proyecto

Este proyecto consiste en una implementación robusta del clásico juego del Ahorcado en Java, diseñada para ejecutarse en consola. El objetivo principal es demostrar el dominio de los fundamentos de la Programación Orientada a Objetos (POO) y el control de flujo estructurado, respetando la restricción de **no utilizar arrays ni colecciones** (Listas, Mapas, etc.), basando la lógica puramente en la manipulación de cadenas (`String`) y tipos primitivos.

Arquitectura (MVC Simplificado)

El código se organiza siguiendo una separación de responsabilidades clara:

Modelo (`JuegoAhorcado`): Encapsula el estado de la partida (palabra, intentos, letras acertadas/falladas) y la lógica de negocio.

Vista (`AsciiArt`, `Main`): Se encarga de la presentación visual (dibujos ASCII) y la interaccion con el usuario (entrada/salida).

Controlador (`Main`): Gestiona el flujo del programa, el menú principal y la coordinación entre el usuario y el juego.

Utilidades (`DiccionarioSencillo`): Provee servicios auxiliares (generación de palabras) sin mantener estado.

```
classDiagram
    class Main {
        +main(args)
        -jugarModoAleatorio()
        -iniciarJuego()
    }
    class JuegoAhorcado {
        -String palabraSecreta
        -String letrasAcertadas
        -int intentosRestantes
        -EstadoPartida estado
        +intentarLetra(char)
        +resolver(String)
        +getDuracion()
    }
    class AsciiArt {
        +stage(int intentos) String
    }
    class DiccionarioSencillo {
        +palabraAleatoria(int nivel) String
    }
    class EstadoPartida {
        <<enumeration>>
```

```
    EN_CURSO
    VICTORIA
    DERROTA
}

Main --> JuegoAhorcado : Instancia y Controla
Main ..> AsciiArt : Usa
Main ..> DiccionarioSencillo : Usa
JuegoAhorcado --> EstadoPartida : Tiene
```

2. Justificación Técnica - RA02: Utilización de Objetos

a) Fundamentos OOP (Clases, Objetos, Encapsulación)

El proyecto no es un simple script lineal, sino un sistema orientado a objetos.

Clases: Se han definido clases para modelar entidades (`JuegoAhorcado`) y excepciones (`LetraRepetidaException`).

Encapsulación: Todos los atributos de `JuegoAhorcado` son `private`. El acceso desde el exterior se realiza exclusivamente a través de métodos públicos (`getters` y métodos de acción como `intentarLetra`), protegiendo la integridad del estado interno.

b) Juego Ejecutable

El proyecto compila sin errores y se ejecuta en cualquier consola Java estándar. La clase `Main` contiene el punto de entrada `public static void main`.

c) Instanciación de Clases Predefinidas

Se hace uso de la API estándar de Java:

`java.util.Scanner`: Para la lectura robusta de la entrada del usuario.

`java.util.Random`: Para la selección aleatoria de palabras y niveles.

`java.time.LocalDateTime` y `Duration`: Para registrar el inicio de la partida y calcular la duración final.

d) Métodos Coherentes

La clase `JuegoAhorcado` expone una interfaz pública lógica:

intentarLetra(char)`: Procesa un intento de carácter.

resolver(String)`: Permite arriesgarse a adivinar la palabra completa.

pedirPista()`: Revela una letra a cambio de intentos.

e) Métodos Estáticos

Se utilizan métodos estáticos (`static`) para funcionalidades que no dependen del estado de un objeto específico:

AsciiArt.stage(int)`: Devuelve el dibujo correspondiente a un número de vidas.

DiccionarioSencillo.palabraAleatoria(int)`: Genera una palabra sin necesidad de instanciar la clase diccionario.

f) Parámetros y Validaciones

Todos los métodos validan sus entradas. Por ejemplo, `intentarLetra` verifica si el carácter es una letra válida (`Character.isLetter`) antes de procesarlo, lanzando `IntentoInvalidoException` si no lo es.

g) Uso de Librerías Estándar (Sin Arrays)

La restricción de no usar arrays se ha resuelto creativamente usando la clase `String`.

Almacenamiento: Las letras acertadas se concatenan en un `String letrasAcertadas`.

Búsqueda: Se usa `indexOf()` para verificar si una letra ya ha sido jugada o si está en la palabra secreta.

h) Constructores y Aserciones

El constructor de `JuegoAhorcado` inicializa el estado de manera consistente. Se utilizan **aserciones** (`assert`) para garantizar invariantes durante el desarrollo:

java

```
public JuegoAhorcado(String palabra, int intentosIniciales) {
    assert intentosIniciales > 0 : "Los intentos deben ser > 0";
    // ... inicialización
}
```

3. Justificación Técnica - RA03: Estructuras de Control

a) Selección (if/else, switch)

Switch: Utilizado en `AsciiArt` para seleccionar el dibujo y en `DiccionarioSencillo` para elegir la palabra según el número aleatorio.

If/Else: Fundamental en la lógica del juego (acierto vs fallo) y en la validación de entradas.

b) Repetición (Bucles)

Bucle Principal (`while`): En Main, mantiene la partida activa mientras `juego.getEstado() == EstadoPartida.EN_CURSO`.

Menú: Un bucle while(!salir) permite jugar múltiples partidas sin reiniciar el programa.

c) Saltos (break, return)

Break: Imprescindible en los bloques `switch` para evitar el *fall-through*.

Return: Usado en validaciones (guard clauses) para salir tempranamente de un método si los argumentos son inválidos.

d) Excepciones (Try/Catch)

El manejo de errores es robusto. El bucle de juego envuelve la lógica en un bloque `try-catch` para capturar excepciones específicas:

```
try {
    procesarEntrada(juego, entrada);
} catch (LetraRepetidaException e) {
    System.out.println(">>> AVISO: " + e.getMessage());
```

```
} catch (IntentoInvalidoException e) {
    System.out.println(">>> ERROR: " + e.getMessage());
}
```

Esto evita que el programa colapse ante entradas incorrectas del usuario.

e) Flujo Completo

El juego implementa un ciclo de vida completo:

1. Configuración: Selección de dificultad.
2. Juego: Bucle de turnos (mostrar estado -> pedir entrada -> procesar).
3. Fin:Detección de Victoria/Derrota y muestra de resultados y tiempo.

f) Pruebas y Depuración (Bug Report)

Bug Detectado y Corregido

Problema:Al usar "PISTA", si la letra revelada completaba la palabra, el juego no detectaba la victoria inmediatamente.

Causa: Falta de verificación de `todasLetrasAcertadas()` dentro del método `pedirPista()`.

Solución:Se añadió la llamada a la verificación de victoria justo después de revelar la letra.

g) Excepciones Propias

Se han creado `LetraRepetidaException` e `IntentoInvalidoException` para modelar errores de dominio específicos, mejorando la semántica del código frente a excepciones genéricas.

4. Extensiones Implementadas (Mejoras)

Para superar los requisitos básicos y optar a la máxima calificación, se han añadido las siguientes funcionalidades:

1. Marcador de Tiempo (Duración)

Se utiliza `java.time.Duration` para calcular el tiempo exacto que el jugador tarda en resolver el puzzle.

Al inicio: `inicioPartida = LocalDateTime.now();`

Al final: `Duration.between(inicioPartida, LocalDateTime.now())`

* El resultado se formatea en minutos y segundos para el usuario.

2. Niveles de Dificultad

Se ha mejorado `DiccionarioSencillo` para soportar niveles:

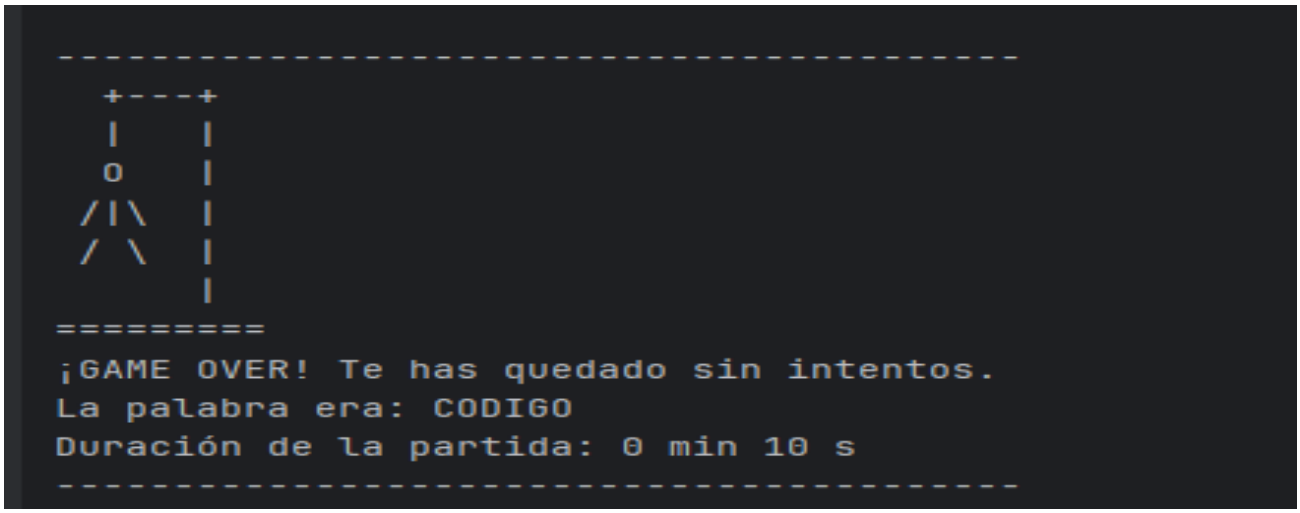
Nivel 1 (Básico) : Palabras cortas y comunes (ej. JAVA, RED).

Nivel 2 (Avanzado): Términos técnicos más largos (ej. POLIMORFISMO, ENCAPSULAMIENTO).

El usuario selecciona el nivel antes de comenzar la partida aleatoria.

Capturas de pantalla solicitadas:

Partida perdida.



Partida ganada

```
----------------------------------------
  +---+
  |   |
      |
      |
      |
      |
=========
¡FELICIDADES! Has ganado.
La palabra era: JAVA
Duración de la partida: 0 min 5 s
----------------------------------------

--- MENÚ PRINCIPAL ---
1. Jugar (Modo Aleatorio)
2. Jugar (Modo 2 Jugadores)
3. Salir
Selecciona una opción:
```