DAW

Proyecto Laberinto

Juan Felipe Mena - Nicolás Furnieles - Gonzalo Márquez - Javier Maroto

BASADO EN CONSOLA

¡VAMOS!

```java
package BBDD;

import java.sql.*;
import java.util.Scanner;
import java.util.List;
import java.util.ArrayList;
import java.util.Random;

public class JugadorLaberinto {
    private static Connection conexion;
    private static Scanner scn = new Scanner(System.in);
    private static int vidaJugador = 100;
    private static int posX, posY;
    private static char[][] laberinto;
    private static boolean[][] visitado;
    private static int tamaño;
    private static String usuario;
    private static int laberintoSeleccionado;
    private static int disposicionSeleccionada;
    private static int dmgCocodrilo = 25;
    private static int vidaBotiquin = 20;
    private static Random random = new Random();

    public static void main(String[] args) {
        conectarBD();
        if (conexion != null) {
            mostrarLaberintosDisponibles();
            jugar();
            cerrarConexion();
        }
    }

    private static void conectarBD() {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            conexion = DriverManager.getConnection(
                "jdbc:mysql://localhost/laberintobueno",
                "root",
                "mysql"
            );
            System.out.println("Conexión con BD establecida");
        } catch (Exception e) {
            System.err.println(" Error al conectar con la BD: " + e.getMessage());
            System.exit(1);
        }
    }

    private static void mostrarLaberintosDisponibles() {
        try (Statement stmt = conexion.createStatement()) {
            ResultSet rs = stmt.executeQuery(
                "SELECT id, dimension1, cocodrilos, botiquines, dmgCocodrilo FROM Laberintos"
            );

            System.out.println("\n🏰 === LABERINTOS DISPONIBLES ===");
            System.out.println("┌────┬──────────┬────────────┬────────────┬──────────────┐");
            System.out.println("│ ID │ Tamaño   │ Cocodrilos │ Botiquines │ Daño Cocodr. │");
            System.out.println("├────┼──────────┼────────────┼────────────┼──────────────┤");

            while (rs.next()) {
                System.out.printf(
```

¿Por qué en consola?

Swing habría añadido complejidad innecesaria con contenedores y eventos gráficos. Nuestra aplicación de consola es más liviana y eficiente al no tener dependencias gráficas.

Aparte de que queríamos hacer algo diferente al resto ya que suponemos que la mayoría lo iba a hacer con interfaz gráfica.

Índice

YAAAAAA AAAS!!!

Funcionamiento

Base de Datos

MySQL

Código

Java

Funcionamiento

Jugador

Administrador

Funcionamiento del jugador

Objetivo del Juego

El jugador comienza en la esquina superior izquierda del laberinto (marcada con *) y debe llegar a la esquina inferior derecha (marcada con =) para ganar. El desafío está en encontrar el camino correcto mientras se mantiene con vida.

Mecánicas de Juego

Sistema de Vida: Cada jugador comienza con 100 puntos de vida. Si la vida llega a cero, el juego termina y el jugador pierde.

Movimiento: El jugador se mueve usando las teclas W, A , S y D. Solo puede moverse por caminos libres, no puede atravesar paredes.

Exploración Limitada: El jugador solo puede ver las áreas que ya ha visitado. Las zonas no exploradas aparecen como signos de interrogación.

Elementos del Laberinto

Cocodrilos: Enemigos que quitan vida al jugador cuando los encuentra. Cada cocodrilo causa un daño específico que varía según el laberinto elegido.

Botiquines: Objetos que restauran 20 puntos de vida cuando el jugador los encuentra. Son esenciales para sobrevivir en laberintos más difíciles.

Paredes y Caminos: Las paredes bloquean el movimiento, mientras que los caminos libres permiten el paso del jugador.

Funcionamiento del jugador

Selección de Laberinto

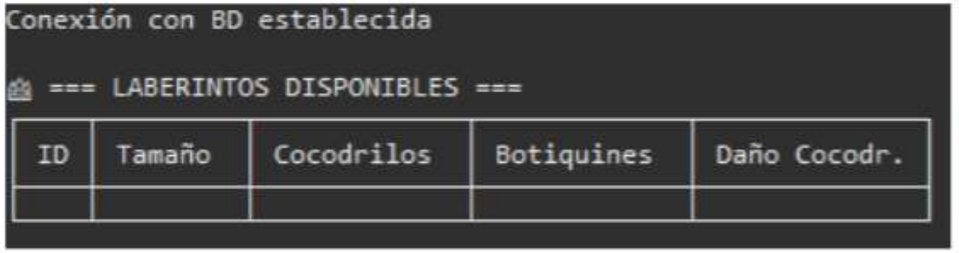
Al iniciar, el jugador puede elegir entre diferentes laberintos, cada uno con:

- Tamaños variables.
- Diferente cantidad de cocodrilos y botiquines.
- Distintos niveles de daño por cocodrilo.

Sistema de Pistas

El juego proporciona pistas útiles como:

- Indicaciones sobre paredes cercanas.
- Distancia aproximada a la salida.
- Sensaciones sobre qué tan cerca está el objetivo.



Competencia y Ranking

Los jugadores que completan exitosamente un laberinto quedan registrados en un ranking que muestra los mejores puntajes basados en la vida restante al finalizar. Esto fomenta la competencia para completar los laberintos con la mayor cantidad de vida posible.

Funcionamiento del Admin

Funciona como un creador de laberintos donde el admin puede diseñar su propio laberinto paso a paso.

Configuración inicial: Primero te pide que le des un nombre a tu laberinto y que elijas qué tan grande quieres que sea (mínimo 5x5 casillas). También puedes añadir elementos especiales como botiquines que curan vida y cocodrilos que hacen daño.

Creación del laberinto: El programa crea una cuadrícula vacía donde:
- La esquina superior izquierda (0,0) es la entrada, marcada con *
- La esquina inferior derecha es la salida, marcada con =
- Todo lo demás son muros, marcados con |

Diseño de la ruta principal: El programa te muestra el laberinto actual y te pide que vayas eligiendo casilla por casilla para crear el camino principal desde la entrada hasta la salida. Solo puedes moverte a casillas vecinas y no puedes volver atrás. Estos caminos se marcan con -.

Caminos alternativos: Una vez que terminas la ruta principal, puedes añadir caminos extra que se conecten con la ruta principal. Estos caminos alternativos se marcan con / y te permiten crear un laberinto más interesante con múltiples rutas.

Funcionamiento del Admin

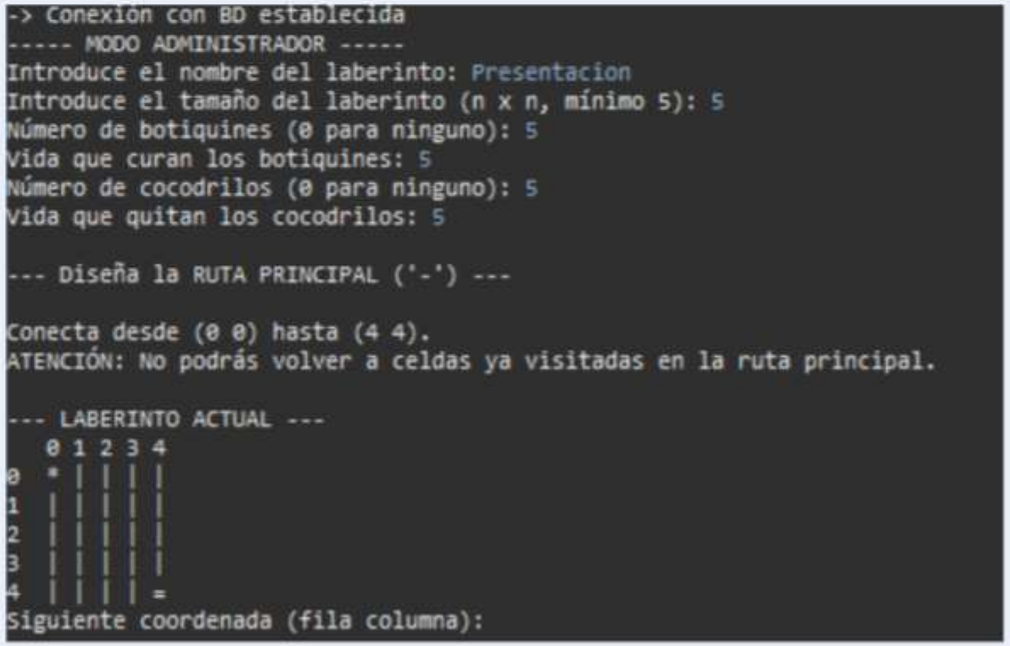
Guardado en base de datos: Cuando terminas de diseñar, el programa guarda automáticamente tu laberinto en una base de datos para que pueda ser usado después.

Lo que hace por dentro:

El programa se conecta a una base de datos y organiza la información en tres tablas: una para los datos generales del laberinto (tamaño, número de elementos), otra para agrupar las disposiciones, y una tercera donde guarda cada casilla del camino con sus coordenadas.

Elementos del laberinto:

* = Entrada del laberinto
= = Salida del laberinto
- = Camino principal
/ = Caminos alternativos
| = Muros (no se puede pasar)

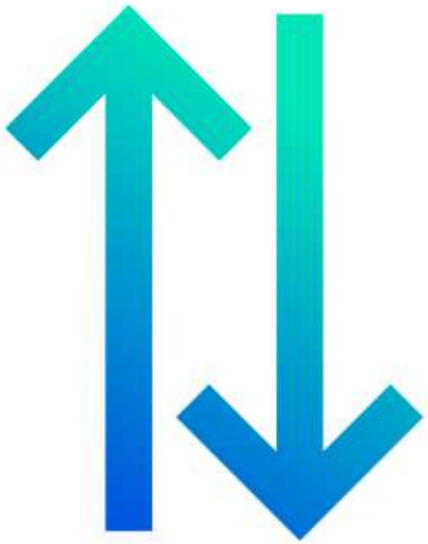


YAAAAAA AAAS!!!

Base de Datos

Código

Conexión



Código

```sql
CREATE TABLE Laberintos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    dimension1 INT NOT NULL,
    dimension2 INT NOT NULL,
    cocodrilos INT DEFAULT 0,
    botiquines INT DEFAULT 0,
    dmgCocodrilo INT DEFAULT 25
);
```

Finalidad: Define las características de un laberinto.

Columnas:
id: Identificador único (clave primaria, auto incremental).
dimension1 y dimension2: Tamaño del laberinto (ancho x alto).
cocodrilos: Número de cocodrilos en el laberinto.
botiquines: Cantidad de botiquines disponibles.
dmgCocodrilo: Daño que inflige un cocodrilo.

```sql
CREATE TABLE Disposiciones (
    id INT AUTO_INCREMENT PRIMARY KEY,
    id_laberinto INT NOT NULL,
    FOREIGN KEY (id_laberinto) REFERENCES Laberintos(id) ON DELETE CASCADE
);
```

Finalidad: Define una disposición del laberinto.

Columnas:
id: Identificador único.
id_laberinto: Asocia esta disposición a un laberinto específico.
Restricción: Si se elimina un laberinto, sus disposiciones también se eliminan automáticamente (ON DELETE CASCADE).

Código

Finalidad: Define las puertas del laberinto dentro de una disposición.

Columnas:

id: Identificador único.

id_disposicion: A qué disposición pertenece esta puerta.

coord1, coord2: Coordenadas de la puerta en el mapa.

posicion: Indica en qué lado de la celda está la puerta (por ejemplo, 0 = arriba, 1 = derecha, etc.).

```
CREATE TABLE Puertas (
    id INT AUTO_INCREMENT PRIMARY KEY,
    id_disposicion INT NOT NULL,
    coord1 INT NOT NULL,
    coord2 INT NOT NULL,
    posicion INT NOT NULL,
    FOREIGN KEY (id_disposicion) REFERENCES Disposiciones(id) ON DELETE CASCADE
);
```

```
CREATE TABLE Ranking (
    id INT AUTO_INCREMENT PRIMARY KEY,
    usuario VARCHAR(100) UNIQUE NOT NULL,
    vida INT,
    laberinto INT,
    disposicion INT,
    salida TINYINT(1),
    FOREIGN KEY (laberinto) REFERENCES Laberintos(id),
    FOREIGN KEY (disposicion) REFERENCES Disposiciones(id)
);
```

Finalidad: Guarda los resultados de los jugadores.

Columnas:

id: Identificador único.

usuario: Nombre del jugador (único).

vida: Vida restante al final del juego.

laberinto: Referencia al laberinto jugado.

disposicion: Disposición utilizada.

salida: Valor booleano (0 o 1) indicando si el jugador salió del laberinto con éxito.

Conexión

Establecimiento de conexión: DriverManager.getConnection() crea la conexión usando la URL, usuario y contraseña.

Manejo de excepciones: Captura errores específicos como driver no encontrado o problemas de conexión.

```java
public AdministradorBBDD() {
    try {
        Class.forName("com.mysql.cj.jdbc.Driver");
        conexion = DriverManager.getConnection(url, login, pwd);
        System.out.println("-> Conexión con BD establecida");
    } catch (ClassNotFoundException e) {
        System.out.println("Driver JDBC No encontrado");
        e.printStackTrace();
    } catch (SQLException e) {
        System.out.println("Error al conectarse a la BD");
        e.printStackTrace();
    } catch (Exception e) {
        System.out.println("Error general de Conexión");
        e.printStackTrace();
    }
```

```java
public static void guardarLaberintoEnBD() {
    try {
        conexion.setAutoCommit(false); // Iniciar transacción

        // 1. Insertar en tabla Labecintos
        int laberintoId = insertarLaberinto();

        // 2. Insertar en tabla Disposiciones
        int disposicionId = insertarDisposicion(laberintoId);

        // 3. Insertar puertas en tabla Puertas
        insertarPuertas(disposicionId);

        conexion.commit(); // Confirmar transacción
        System.out.println("Laberinto guardado correctamente en la base de datos");

    } catch (SQLException e) {
```

1. Inicia una transacción.
2. Inserta el laberinto, su disposición y sus puertas.
3. Si todo sale bien, confirma los cambios.
4. Si ocurre un error, revierte todo para no dejar datos inconsistentes.

Conexión

1. Prepara una consulta para insertar un nuevo laberinto con sus atributos.
2. Usa PreparedStatement para evitar inyecciones SQL.
3. Asigna valores (incluyendo valores por defecto si no se configuraron algunos).
4. Ejecuta la inserción.
5. Si la inserción fue exitosa, obtiene y devuelve el ID generado automáticamente por la base de datos.
6. Si falla, lanza una excepción.

```java
private static int insertarLaberinto() throws SQLException {
    // CORREGIDO: Usar valores por defecto cuando no hay elementos configurados
    String query = "INSERT INTO Laberintos (dimension1, dimension2, cocodrilos, botiquines, dmgCocodrilo) " +
                   "VALUES (?, ?, ?, ?, ?)";

    try (PreparedStatement pstmt = conexion.prepareStatement(query, Statement.RETURN_GENERATED_KEYS)) {
        pstmt.setInt(1, tamaño);
        pstmt.setInt(2, tamaño);
        pstmt.setInt(3, numCocodrilos);
        pstmt.setInt(4, numBotiquines);
        // CORREGIDO: Usar valor por defecto si no hay cocodrilos configurados
        pstmt.setInt(5, vidaCocodrilos > 0 ? vidaCocodrilos : 25);
        // CORREGIDO: Agregar valores por defecto para las nuevas columnas

        int filasAfectadas = pstmt.executeUpdate();
        if (filasAfectadas > 0) {
            try (ResultSet generatedKeys = pstmt.getGeneratedKeys()) {
                if (generatedKeys.next()) {
                    int laberintoId = generatedKeys.getInt(1);
                    System.out.println("Laberinto insertado con ID: " + laberintoId);
                    return laberintoId;
                }
```

Statement.RETURN_GENERATED_KEYS es una constante de JDBC que se utiliza para recuperar automáticamente las claves primarias generadas cuando se ejecuta una sentencia INSERT en una base de datos.

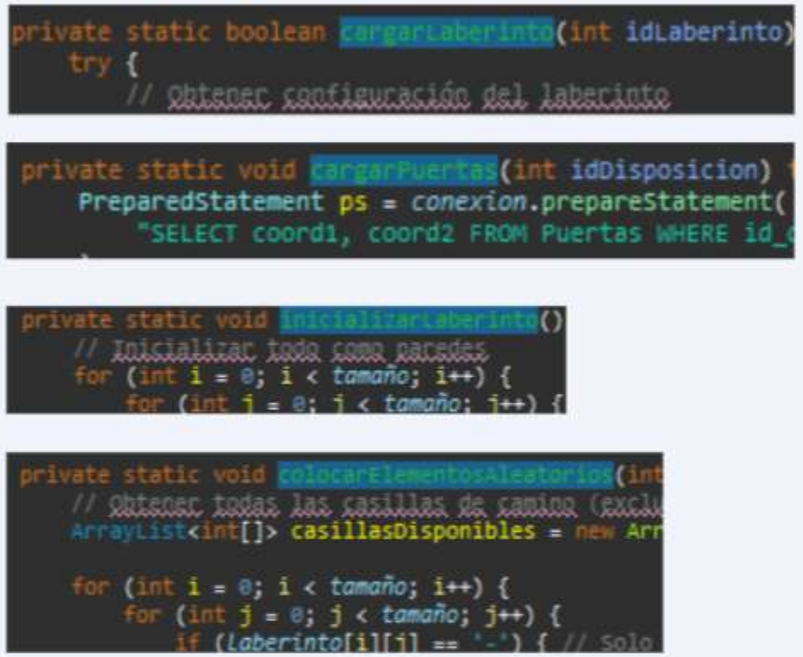
YAAAAAA
AAAS!!!

Código

Jugador

Administrador

Código de Jugador



Collections.shuffle() en Java sirve para barajar o permutar aleatoriamente los elementos de una lista especificada. Es útil para desordenar una lista de forma aleatoria.

cargarLaberinto(int id): utiliza PreparedStatements para obtener la configuración específica de un laberinto seleccionado, incluyendo dimensiones, número de elementos y parámetros de daño.

cargarPuertas(int idDisposicion): mapea la estructura del laberinto cargando las coordenadas de los caminos libres desde la tabla Puertas.

inicializarLaberinto(): crea las matrices bidimensionales necesarias: una para el mapa del laberinto y otra para el estado de exploración. Inicializa todas las posiciones como paredes y establece la entrada y salida en coordenadas específicas.

colocarElementosAleatorios(): distribuye cocodrilos y botiquines de forma aleatoria en las casillas de camino disponibles, utilizando Collections.shuffle() para garantizar una distribución impredecible.

Código de Jugador

```
private static void iniciarJuego() {
    System.out.println("\n¡COMIENZA EL JUEGO!");
    System.out.println("Objetivo: Llegar desde la entrada (*) hasta la salida (=)");
    System.out.println("Cuidado con los cocodrilos (C) y busca los botiquines (B)");
    System.out.println("Solo conoces tu entorno inmediato...");

    while (vidaJugador > 0) {
        mostrarEstadoJuego();
        procesarMovimiento();
        verificarCasillaActual();

        if (haGanado()) {
            System.out.println("\n¡FELICIDADES! ¡HAS LLEGADO A LA SALIDA!");
            System.out.printf("Vida restante: %d puntos%n", vidaJugador);
            guardarResultado(true);
            mostrarRanking();
            return;
        }
    }

    System.out.println("\n¡HAS PERDIDO TODA TU VIDA!");
    System.out.println("¡Inténtalo de nuevo!");
    guardarResultado(false);
}
```

iniciarJuego(): contiene el bucle principal que controla el flujo del juego, gestionando la secuencia de mostrar estado, procesar movimiento, verificar eventos y evaluar condiciones de victoria o derrota.

procesarMovimiento(): interpreta la entrada del usuario (WASD) y actualiza las coordenadas del jugador, validando el movimiento.

esMovimientoValido(): verifica si una posición específica es accesible, comprobando límites del array y tipo de casilla (distinguiendo entre paredes y caminos transitables).

verificarCasillaActual(): ejecuta la lógica de eventos según el tipo de casilla donde se encuentra el jugador, aplicando daño por cocodrilos o curación por botiquines, y actualizando el estado de la casilla después del encuentro.

haGanado(): evalúa la condición de victoria verificando si el jugador ha alcanzado las coordenadas de la salida del laberinto.

Código de Jugador

mostrarEstadoJuego(): renderiza el estado completo del juego, incluyendo el mapa explorado, vida actual y movimientos disponibles.

mostrarMapaExplorado(): visualiza únicamente las áreas que el jugador ha visitado.

darPistasUbicacion(): proporciona información contextual sobre la posición actual del jugador y la distancia estimada a la salida y pistas ambientales.

guardarResultado(): almacena los resultados de la partida en la base de datos.

mostrarRanking(): consulta y presenta el top 10 de jugadores que han completado exitosamente laberintos, ordenados por vida restante al finalizar.

esAdyacente(): calcula si dos coordenadas están a distancia, se utiliza para determinar qué casillas son visibles desde la posición actual del jugador.


```
private static void mostrarRanking() {
    try (Statement stmt = conexion.createStatement()) {
        ResultSet rs = stmt.executeQuery(
            "SELECT usuario, vida, salida FROM Ranking WHERE salida = 1 ORDER BY vida DESC LIMIT 10"
        );

        System.out.println("\n🏆 === TOP 10 JUGADORES (VICTORIOSOS) ===");
        System.out.println("┌─────┬──────────────────┬──────┬─────────────┐");
        System.out.println("│ Pos │     Usuario      │ Vida │ Completado  │");
        System.out.println("├─────┼──────────────────┼──────┼─────────────┤");

        int posicion = 1;
        while (rs.next()) {
            String emoji = posicion == 1 ? "🥇" : posicion == 2 ? "🥈" : posicion == 3 ? "🥉" : "🏅";
            System.out.printf(
                "|%s %-2d | %-16s | %-4d |     %-3s     |%n",
                emoji,
                posicion,
                rs.getString("usuario"),
                rs.getInt("vida"),
                rs.getBoolean("salida") ? "Sí" : "No"
            );
            posicion++;
        }
        System.out.println("└─────┴──────────────────┴──────┴─────────────┘");
```

Código de Admin

laberinto char[][]: Array que representa visual y lógicamente el laberinto (paredes, caminos, entrada, salida, etc.).
puertas List<Puerta>: Lista de caminos válidos en el laberinto (entrada, salida, ruta principal y caminos alternativos).
celdasVisitadas List<String>: Evita reutilizar celdas en la ruta principal.

```
public class AdministradorBBDD {
    // Variables para el laberinto
    private static char[][] laberinto;
    private static int tamaño;
    private static Scanner scn = new Scanner(System.in);
    private static List<String> celdasVisitadas = new ArrayList<>();
    private static String nombreLaberinto;
    private static int numBotiquines;
    private static int numCocodrilos;
    private static int vidaBotiquines;
    private static int vidaCocodrilos;

    // Variables para almacenar las puertas del laberinto
    private static List<Puerta> puertas = new ArrayList<>();
```

```
public void terminar() {
```

```
private static String solicitarNombreLaberinto() {
    System.out.print("Introduce el nombre del laberi
```

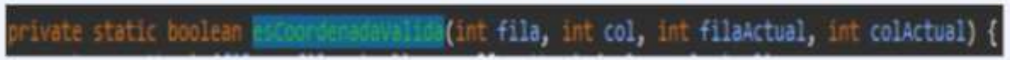
```
rivate static int solicitarTamañoLaberinto() {
    while (true) {
        try {
```

```
private static void solicitarElementos() {
    numBotiquines = solicitarNumero("Número
```

terminar(): Cierra la conexión a la base de datos correctamente.
solicitarNombreLaberinto(): Pide el nombre del laberinto al usuario.
solicitarTamañoLaberinto(): Valida e introduce el tamaño de la matriz del laberinto (mínimo 5x5).
solicitarElementos(): Solicita número y características de botiquines y cocodrilos.

Código de Admin

solicitarNumero(...): Método para leer números positivos (controla errores de entrada).
inicializarLaberinto(): Llena la matriz con muros.
disenarRutaPrincipal(): Permite al administrador definir una ruta principal (-) desde la entrada a la salida.
esCoordenadaValidaRutaPrincipal(...): Verifica que la siguiente celda de la ruta principal sea válida y adyacente.



añadirCaminosAlternativos(): Permite crear rutas alternativas (/) desde la ruta principal.
esCoordenadaValida(...): Verifica si la coordenada es adyacente a otra.
extraerPuertasDelLaberinto(): Extrae las coordenadas de celdas accesibles (*, -, /, =) y las guarda como "puertas".
mostrarLaberintoActual(): Muestra por consola el laberinto durante la edición.

Código de Admin

```
private static void mostrarLaberintoFinal() {
    System.out.println("\n--- LABERINTO FINAL:
    System.out.println("Botiquines: " + numBoti
    System.out.println("Cocodrilos: " + numCoco

    System.out.print("    ");
    for (int j = 0; j < tamaño; j++) {
```

mostrarLaberintoFinal(): Muestra la versión definitiva del laberinto y activa el guardado en BD.

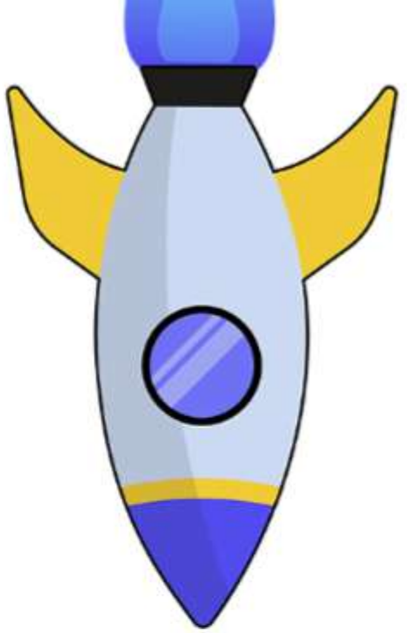
insertarLaberinto(): Inserta información general del laberinto en la tabla Laberintos.

insertarDisposicion(int): Crea una nueva disposición (versión del laberinto) en Disposiciones.

insertarPuertas(int): Inserta las puertas del laberinto en la tabla Puertas.

```
private static void insertarPuertas(int disposicionId) throws SQLException
    String query = "INSERT INTO Puertas (id_disposicion, coord1, coord2, p

    try (PreparedStatement pstmt = conexion.prepareStatement(query)) {
        for (Puerta puerta : puertas) {
            pstmt.setInt(1, disposicionId);
```



DEMO