The background is a dark navy blue. On the left, there is a large, semi-transparent circular image of a printed circuit board (PCB) with various electronic components. Overlaid on the top left of the PCB is a large, stylized geometric shape composed of two triangles: a blue one on the left and a light green one on the right, separated by a diagonal line. In the top right corner, there is a faint, light gray pattern of concentric, stepped lines resembling a maze or a topographical map.

Documentación Programa Laberinto

Mauricio Vela Ramirez

Antonio Rico Mendiola

Documentación de java sobre pilas

```
public class Stack<E>
```

```
extends Vector<E>
```

La clase representa un último en entrar, primero en salir (LIFO) pila de objetos. Extiende la clase **Vector** con cinco Operaciones que permiten tratar un vector como una pila. Se proporcionan las operaciones habituales de **push** y **pop**, así como un Método para **echar un vistazo** al elemento superior de la pila, un método para probar para saber si la pila está **vacía** y un método para **buscar** un elemento en la pila y descubrir qué tan lejos está de la parte superior. **Stack**

Cuando se crea una pila por primera vez, no contiene elementos.

Un conjunto más completo y consistente de operaciones de pila LIFO es proporcionado por la interfaz **Deque** y sus implementaciones, que debe utilizarse con preferencia a esta clase. Por ejemplo:

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

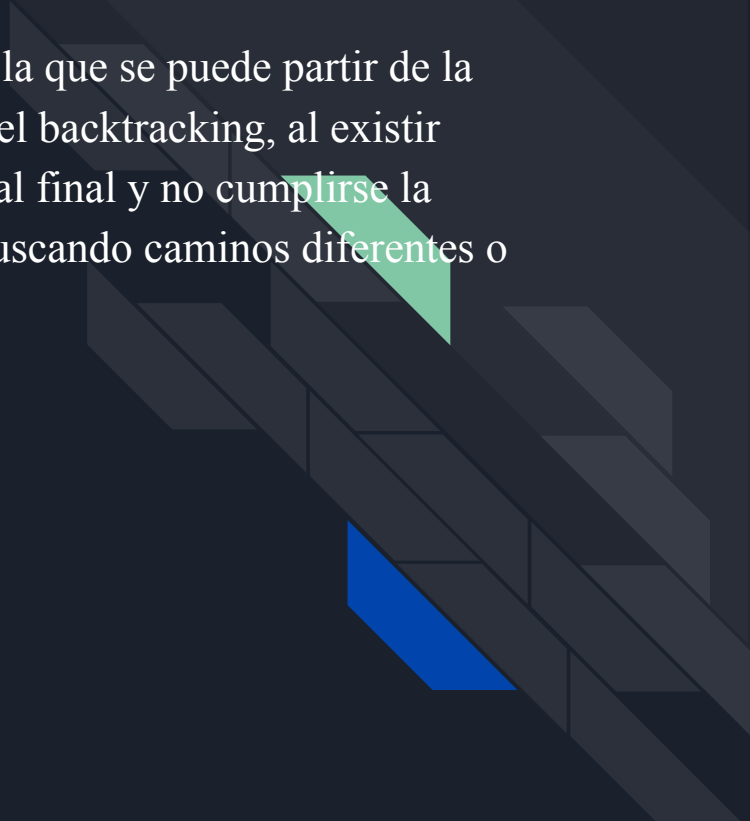
Métodos que usa en las pilas

Métodos

Modificador y tipo	Método y descripción
boolean	<code>empty()</code> Comprueba si esta pila está vacía.
E	<code>peek()</code> Mira el objeto en la parte superior de esta pila sin quitarlo de la pila.
E	<code>pop()</code> Quita el objeto en la parte superior de esta pila y devuelve que como el valor de esta función.
E	<code>push(E item)</code> Inserta un elemento en la parte superior de esta pila.
int	<code>search(Object o)</code> Devuelve la posición basada en 1 donde se encuentra un objeto en esta pila.

Back tracking en el uso de el programa

El **Backtracking** es una técnica de programación de la que se puede partir de la definición de Recursividad con la diferencia que, en el backtracking, al existir **varios caminos diferentes** a elegir, una vez llegado al final y no cumplirse la condición establecida, **volvemos atrás** para seguir buscando caminos diferentes o alternativos y posiblemente correctos.



Librerías utilizadas y variables globales

```
1
2 import javax.swing.JFrame;
3 import java.awt.Color;
4 import java.awt.Font;
5 import java.awt.GridLayout;
6 import java.awt.Point;
7 import java.util.Stack;
8 import javax.swing.BorderFactory;
9 import javax.swing.JFrame;
10 import javax.swing.JLabel;
11 import javax.swing.JPanel;
12
13 public class Laberinto extends JFrame {
14
15     private static final long serialVersionUID = 1L;
16     private final int filas = 10;
17     private final int columnas = 10;
18     private final JPanel panelLaberinto;
19     private final int[][] laberinto = new int[][]{
20         {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, //0.0-9
21         {1, 0, 1, 0, 0, 0, 0, 0, 0, 1},
22         {1, 0, 0, 0, 1, 0, 0, 1, 0, 1},
23         {1, 1, 1, 1, 0, 0, 1, 1, 1, 1},
24         {3, 0, 0, 0, 0, 1, 0, 0, 0, 1},
25         {1, 0, 1, 1, 1, 1, 0, 1, 0, 1},
26         {1, 0, 0, 0, 0, 1, 0, 1, 0, 1},
27         {1, 1, 0, 1, 1, 1, 0, 1, 0, 1},
28         {1, 0, 0, 0, 0, 0, 0, 1, 0, 5},
29         {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
30     private JLabel[][] etiquetas;
31     private Point posicionActual;
32     private int x, y;
33 }
```

Método sobrecargado de la clase

```
public Laberinto() {
    super("Laberinto");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // Crear el título del laberinto
    JLabel titulo = new JLabel("Laberinto", JLabel.CENTER);
    titulo.setFont(new Font("Arial", Font.BOLD, 24));
    titulo.setForeground(Color.WHITE);
    titulo.setOpaque(true);
    titulo.setBackground(Color.BLUE);

    // Crear el panel del laberinto
    JPanel panelLaberinto = new JPanel(new GridLayout(filas, columnas));
    panelLaberinto.setBackground(Color.BLACK);
    panelLaberinto.setBorder(BorderFactory.createLineBorder(Color.WHITE, 2));

    // Dibujar el laberinto en el panel
    dibujarLaberinto();

    // Agregar el panel y el título al JFrame
    add(titulo, "North");
    add(panelLaberinto, "Center");
    pack();
    setLocationRelativeTo(null);
}
```

Método encontrar Inicio;

```
59 public void encontrarInicio() {  
60     boolean encontrado = false;  
61     for (int i = 0; i < laberinto.length; i++) {  
62         for (int j = 0; j < laberinto[0].length; j++) {  
63             if (laberinto[i][j] == 3) {  
64                 x = i;  
65                 y = j;  
66                 // System.out.println(x + " " + y);  
67                 encontrado = true;  
68                 break;  
69             }  
70         }  
71         if (encontrado) {  
72             break;  
73         }  
74     }  
75 }  
76
```

Método Dibujar Laberito

```
private void dibujarLaberinto() {  
    etiquetas = new JLabel[filas][columnas];  
    for (int i = 0; i < filas; i++) {  
        for (int j = 0; j < columnas; j++) {  
            JLabel etiqueta = new JLabel();  
            etiqueta.setOpaque(true);  
            etiqueta.setHorizontalAlignment(JLabel.CENTER);  
            etiqueta.setVerticalAlignment(JLabel.CENTER);  
            etiqueta.setFont(new Font("Arial", Font.BOLD, 20));  
            etiqueta.setPreferredSize(new java.awt.Dimension(40, 40));  
            etiqueta.setBorder(BorderFactory.createLineBorder(Color.WHITE));  
            switch (laberinto[i][j]) {  
                case 0:  
                    etiqueta.setBackground(Color.WHITE);  
                    break;  
                case 1:  
                    etiqueta.setBackground(Color.BLACK);  
                    break;  
                case 5:  
                    etiqueta.setBackground(Color.GREEN);  
                    posicionActual = new Point(i, j);  
                    break;  
                case 3:  
                    etiqueta.setBackground(Color.RED);  
                    break;  
                default:  
                    break;  
            }  
            etiquetas[i][j] = etiqueta;  
            panellaberinto.add(etiqueta);  
        }  
    }  
}
```


Método Get Vecino

```
private int getVecino(int actualIndex, String direccion) {
    int filaActual = actualIndex / columnas;
    int columnaActual = actualIndex % columnas;
    switch (direccion) {
        case "arriba":
            if (filaActual > 0 && (laberinto[filaActual - 1][columnaActual] == 0 || laberinto[filaActual - 1][columnaActual] == 5)) {
                return (filaActual - 1) * columnas + columnaActual;
            }
            break;
        case "abajo":
            if (filaActual < filas - 1 && (laberinto[filaActual + 1][columnaActual] == 0 || laberinto[filaActual + 1][columnaActual] == 5)) {
                return (filaActual + 1) * columnas + columnaActual;
            }
            break;
        case "izquierda":
            if (columnaActual > 0 && laberinto[filaActual][columnaActual - 1] == 0 || laberinto[filaActual][columnaActual - 1] == 5) {
                return filaActual * columnas + columnaActual - 1;
            }
            break;
        case "derecha":
            if (columnaActual < columnas - 1 && (laberinto[filaActual][columnaActual + 1] == 0 || laberinto[filaActual][columnaActual + 1] == 5)) {
                return filaActual * columnas + columnaActual + 1;
            }
            break;
        default:
            break;
    }

    // Si llegamos aquí es porque no se encontró ningún vecino en esa dirección
    return -1;
}
```

Método recorrerLaberinto

```
public void recorrerLaberinto() {
    encontrarInicio();
    // Buscamos el inicio del recorrido
    int filaActual = x;
    int columnaActual = y;
    int actualIndex = filaActual * columnas + columnaActual;
    JLabel actual = (JLabel) panellaberinto.getComponent(actualIndex);
    actual.setBackground(Color.RED);
    actual.setForeground(Color.RED);

    // Creamos una pila para el backtracking y un conjunto para los nodos visitados
    Stack<Integer> pila = new Stack<>();
    boolean[][] visitados = new boolean[filas][columnas];
    boolean llegoMeta = false; // Variable para controlar si se llegó a la meta

    // Creamos un ciclo que recorre los ceros del laberinto hasta llega a la meta
    while (!llegoMeta) {

        if (laberinto[filaActual][columnaActual] == 5) {
            return;
        }

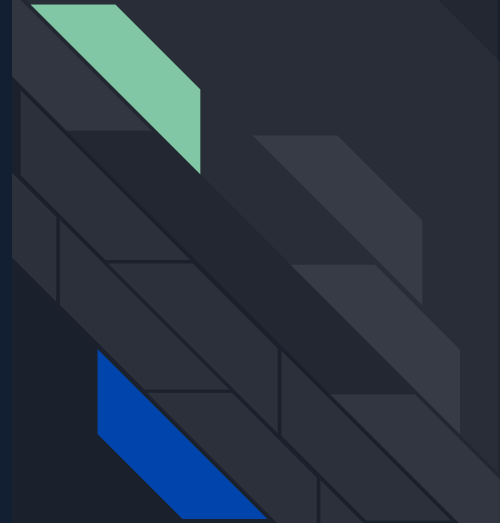
        try {
            // Agregamos una pausa de 500 milisegundos para que se vea el recorrido
```

Método recorrerLaberinto

```
try {  
    // Agregamos una pausa de 500 milisegundos para que se vea el recorrido  
    Thread.sleep(400);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}  
  
// Cambiamos el fondo y fuente de la casilla actual a blanco  
actual.setBackground(Color.WHITE);  
actual.setForeground(Color.WHITE);  
  
// Buscamos la siguiente casilla que sea cero  
int vecinoIndex = getVecino(actualIndex, "arriba");  
if (vecinoIndex != -1 && !visitados[vecinoIndex / columnas][vecinoIndex % columnas]) {  
    // Si encontramos un vecino no visitado, lo agregamos a la pila  
    pila.push(actualIndex);  
    visitados[vecinoIndex / columnas][vecinoIndex % columnas] = true;  
    actual = (JLabel) panelLaberinto.getComponent(vecinoIndex);  
    actual.setBackground(Color.RED);  
    actual.setForeground(Color.RED);  
    actualIndex = vecinoIndex;  
    filaActual = actualIndex / columnas;  
    columnaActual = actualIndex % columnas;
```

Método recorrerLaberinto

```
} else {  
    // Si no encontramos un vecino hacia arriba, buscamos hacia abajo  
    vecinoIndex = getVecino(actualIndex, "abajo");  
    if (vecinoIndex != -1 && !visitados[vecinoIndex / columnas][vecinoIndex % columnas]) {  
        // Si encontramos un vecino no visitado, lo agregamos a la pila  
        pila.push(actualIndex);  
        visitados[vecinoIndex / columnas][vecinoIndex % columnas] = true;  
        actual = (JLabel) panelLaberinto.getComponent(vecinoIndex);  
        actual.setBackground(Color.RED);  
        actual.setForeground(Color.RED);  
        actualIndex = vecinoIndex;  
        filaActual = actualIndex / columnas;  
        columnaActual = actualIndex % columnas;  
  
        // Si llegamos a la meta, terminamos el recorrido  
        if (filaActual == 8 && columnaActual == 8) {  
            llegoMeta = true;  
        }  
    } else {  
        // Si no encontramos un vecino hacia abajo, buscamos hacia la izquierda  
        vecinoIndex = getVecino(actualIndex, "izquierda");  
        if (vecinoIndex != -1 && !visitados[vecinoIndex / columnas][vecinoIndex % columnas]) {  
ncontramos un vecino no visitado, lo agregamos a la pila  
            pila.push(actualIndex);  
            visitados[vecinoIndex / columnas][vecinoIndex % columnas] = true;  
            actual = (JLabel) panelLaberinto.getComponent(vecinoIndex);  
            actual.setBackground(Color.RED);  
            actual.setForeground(Color.RED);  
            actualIndex = vecinoIndex;  
            filaActual = actualIndex / columnas;  
            columnaActual = actualIndex % columnas;
```



Método recorrerLaberinto

```
        } else {  
            // Si no encontramos un vecino hacia la izquierda, buscamos hacia la derecha  
            vecinoIndex = getVecino(actualIndex, "derecha");  
            if (vecinoIndex != -1 && !visitados[vecinoIndex / columnas][vecinoIndex % columnas]) {  
                // Si encontramos un vecino no visitado, lo agregamos a la pila  
                pila.push(actualIndex);  
                visitados[vecinoIndex / columnas][vecinoIndex % columnas] = true;  
                actual = (JLabel) panelLaberinto.getComponent(vecinoIndex);  
                actual.setBackground(Color.RED);  
                actual.setForeground(Color.RED);  
                actualIndex = vecinoIndex;  
                filaActual = actualIndex / columnas;  
                columnaActual = actualIndex % columnas;  
                // Si llegamos a la meta, terminamos el recorrido  
                if (filaActual == 8 && columnaActual == 8) {  
                    llegoMeta = true;  
                }  
            } else {  
                // Si no encontramos un vecino hacia la derecha, hacemos backtracking  
                actualIndex = pila.pop();  
                actual = (JLabel) panelLaberinto.getComponent(actualIndex);  
  
                actual.setBackground(Color.YELLOW);  
                actual.setForeground(Color.YELLOW);  
            }  
        }  
    }  
}  
}
```

Explicacion del metodo

Este algoritmo es una implementación del algoritmo Depth-First Search (DFS) o búsqueda en profundidad.

El algoritmo comienza encontrando el inicio del laberinto y marcándolo en rojo. A continuación, se crea una pila para el backtracking y un conjunto para los nodos visitados. Luego, se crea un ciclo que recorre los ceros del laberinto hasta llegar al final (en este caso, el número 5 en la matriz).

En cada iteración del ciclo, se realiza una pausa para que se pueda observar el recorrido, se cambia el fondo y fuente de la celda actual a blanco y se busca el siguiente vecino no visitado. Si se encuentra un vecino no visitado, se agrega su índice a la pila y se marca en rojo. Si se llega a la meta, se termina el recorrido.

Si no se encuentra un vecino no visitado, se realiza backtracking, es decir, se saca un elemento de la pila y se marca su celda correspondiente en amarillo. De esta manera, se retrocede hasta encontrar una celda vecina no visitada y se continúa el recorrido.

El algoritmo termina cuando se llega al final del laberinto o cuando la pila está vacía, lo que significa que no hay más vecinos por visitar.

Capturas del Programa ejecutándose

