

**TALLER DE PROGRAMACIÓN GUÍA 6**  
**CLASE THREAD E INTERFACE RUNNABLE**

**MIGUEL ANGEL ZAMBRANO**  
**HUGO ANDRES FORERO**  
**SAMUEL ANTONIO TARAZONA**

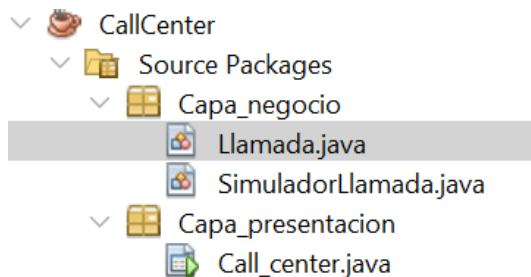
**TALLER DE PROGRAMACIÓN**  
**UNIVERSIDAD MANUELA BELTRAN**

**2024**

## Códigos

1.

a. La clase Llamada en Java representa una llamada telefónica en un simulador de Call Center, encapsulando información como el nombre del cliente, su número de teléfono y el operador que atiende la llamada. Incluye atributos privados para proteger los datos, un constructor para inicializarlos y métodos "getter" para acceder a la información, siguiendo así las buenas prácticas de programación orientada a objetos.



```
package Capa_negocio;
```

```
public class Llamada {
    private String nombre;
    private String telefono;
    private String operador;

    public Llamada(String nombre, String telefono, String operador) {
        this.nombre = nombre;
        this.telefono = telefono;
        this.operador = operador;
    }

    public String getNombre() {
        return nombre;
    }

    public String getTelefono() {
        return telefono;
    }

    public String getOperador() {
        return operador;
    }
}
```

b. La clase `SimuladorLlamada` implementa la interfaz `Runnable` para permitir que cada instancia se ejecute en un hilo separado, lo que es crucial en un simulador de Call center, ya que evita bloquear la interfaz de usuario mientras se simula una llamada. En su método `run()`, se simula el proceso de una llamada mostrando el estado en un `JTextArea`, se espera 3 segundos para representar la duración de la llamada y se determina aleatoriamente si la llamada fue contestada. Al finalizar, se actualiza el estado en el `JTextArea` y se registra la llamada en un `DefaultTableModel`, lo que permite mantener un historial de las llamadas realizadas, mejorando así la experiencia del usuario en la simulación.

```
package Capa_negocio;

import javax.swing.JTextArea;
import javax.swing.table.DefaultTableModel;

public class SimuladorLlamada implements Runnable {
    private String nombre;
    private String telefono;
    private String operador;
    private JTextArea txArea_EstadoLlamada;
    private DefaultTableModel modelRegistro; // Modelo de la tabla de registro

    public SimuladorLlamada(String nombre, String telefono, String operador, JTextArea txArea_EstadoLlamada, DefaultTableModel modelRegistro) {
        this.nombre = nombre;
        this.telefono = telefono;
        this.operador = operador;
        this.txArea_EstadoLlamada = txArea_EstadoLlamada;
        this.modelRegistro = modelRegistro; // Inicializa el modelo de registro
    }

    @Override
    public void run() {
        // Simulación de llamada
        txArea_EstadoLlamada.append("Estado de la Simulación: | | - " + nombre + " llamando al número " + telefono + "... | |\n");
        try {
            Thread.sleep(3000); // Simula el tiempo que dura la llamada (3 segundos)
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt(); // Manejar la interrupción del hilo
        }

        // Lógica para determinar si la llamada fue contestada
        boolean contesto = Math.random() > 0.5; // Simula un 50% de probabilidad de contestar
        String estado = contesto ? "Contesto" : "No contesto";

        // Actualiza el JTextArea
        txArea_EstadoLlamada.append("Estado de la Simulación: | | - " + nombre + " ha terminado la llamada. " + estado + " | |\n");

        // Agrega la llamada al registro en la tabla correspondiente
        modelRegistro.addRow(new Object[]{nombre, telefono, operador, estado}); // Agregar al registro
    }
}
```

c. La clase `Call_center`, que extiende `JFrame`, representa la interfaz gráfica de un simulador de Call center en Java. En su constructor, se inicializan los componentes y se configuran dos modelos de tabla con `DefaultTableModel`: uno para mostrar información de llamadas (nombre, teléfono, operador y estado) y otro para registrar las llamadas realizadas. Esto permite a los usuarios gestionar y visualizar las interacciones de llamadas de manera organizada.

```
package Capa_presentacion;

import Capa_negocio.Llamada;
import Capa_negocio.SimuladorLlamada;
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Call_center extends JFrame {

    private DefaultTableModel model;
    private DefaultTableModel modelRegistro;

    public Call_center() {
        initComponents();
        // Cambia el modelo para tener 4 columnas: Nombre, Teléfono, Operador, Llamada
        model = new DefaultTableModel(new Object[][] {}, new String[] {"Nombre", "Teléfono", "Operador", "Llamada"});
        table_datos.setModel(dataModel:model);

        modelRegistro = new DefaultTableModel(new Object[][] {}, new String[] {"Nombre", "Teléfono", "Operador", "Estado"});
        tabla_registro.setModel(dataModel:modelRegistro);
    }
}
```

d. El método `bt_guardarActionPerformed` se ejecuta cuando se hace clic en el botón "Guardar". En este método, se recopilan los valores ingresados por el usuario en los campos de texto `tx_nombre` y `tx_telefono`, así como el operador seleccionado del combo box `combox_operador`. Luego, se agrega una nueva fila a la tabla asociada al modelo `model`, que incluye el nombre, el teléfono, el operador y un estado inicial de "No" en la columna de llamadas. Finalmente, el método limpia los campos de texto y restablece la selección del combo box a su estado inicial, permitiendo al usuario ingresar nuevos datos sin problemas.

```
private void bt_guardarActionPerformed(java.awt.event.ActionEvent evt) {
    String nombre = tx_nombre.getText();
    String telefono = tx_telefono.getText();
    String operadorSeleccionado = (String) combox_operador.getSelectedItem();

    // Agregar una nueva fila a la tabla, con "No" en la columna Llamada
    model.addRow(new Object[]{nombre, telefono, operadorSeleccionado, "No"});

    // Limpiar los campos de texto
    tx_nombre.setText("");
    tx_telefono.setText("");
    combox_operador.setSelectedIndex(0);
}
```

e. El método `bt_eliminarActionPerformed` se ejecuta al hacer clic en el botón "Eliminar". Verifica si hay una fila seleccionada en la tabla `table_datos` y, si es así, la elimina del modelo `model`. Si no hay una fila seleccionada, muestra un mensaje de advertencia para informar al usuario que debe seleccionar una fila antes de eliminarla. Esto asegura que la eliminación se realice de manera controlada.

```
private void bt_eliminarActionPerformed(java.awt.event.ActionEvent evt) {
    int filaSeleccionada = table_datos.getSelectedRow();

    if (filaSeleccionada != -1) { // Verificar si hay una fila seleccionada
        model.removeRow(row: filaSeleccionada); // Eliminar la fila seleccionada
    } else {
        // Mostrar un mensaje si no hay fila seleccionada
        JOptionPane.showMessageDialog(parentComponent: this, message: "Por favor, seleccione una fila para eliminar.", title: "Advertencia", messageType: JOptionPane.WARNING_MESSAGE);
    }
}
```

f. El método `bt_LlamarActionPerformed` se ejecuta al hacer clic en el botón "Iniciar Llamada". Primero, verifica si hay una fila seleccionada en la tabla `table_datos`. Si se seleccionó una fila, obtiene los valores de nombre, teléfono y operador de esa fila y actualiza la columna "Llamada" a "Sí". Luego, crea un nuevo hilo utilizando la clase `SimuladorLlamada`, que simula el proceso de la llamada, y lo inicia. Si no hay ninguna fila seleccionada, muestra un mensaje de advertencia indicando que se debe seleccionar una fila para iniciar la llamada.

```
private void bt_LlamarActionPerformed(java.awt.event.ActionEvent evt) {
    int filaSeleccionada = table_datos.getSelectedRow();

    if (filaSeleccionada != -1) { // Verificar si hay una fila seleccionada
        String nombre = (String) model.getValueAt(row: filaSeleccionada, column: 0);
        String telefono = (String) model.getValueAt(row: filaSeleccionada, column: 1);
        String operador = (String) model.getValueAt(row: filaSeleccionada, column: 2); // Obtén el operador

        // Cambiar el valor de la columna "Llamada" a "Si"
        model.setValueAt(value: "Si", row: filaSeleccionada, column: 3);

        // Crear y ejecutar el hilo, pasando el modelo de registro
        SimuladorLlamada simulador = new SimuladorLlamada(nombre, telefono, operador, txArea_EstadoLlamada, modelRegistro);
        Thread hilo = new Thread(thread: simulador);
        hilo.start(); // Inicia el hilo
    } else {
        // Mostrar un mensaje si no hay fila seleccionada
        JOptionPane.showMessageDialog(parentComponent: this, message: "Por favor, seleccione una fila para iniciar la llamada.", title: "Advertencia", messageType: JOptionPane.WARNING_MESSAGE);
    }
}
```

g. El método `bt_reiniciarActionPerformed` se activa al hacer clic en "Reiniciar" y se encarga de limpiar la interfaz: elimina todas las filas de las tablas de datos y registro, vacía el área de texto `txArea_EstadoLlamada`, y restablece los campos de entrada de nombre y teléfono, así como el combo box de operadores a su estado predeterminado. Esto permite iniciar una nueva sesión en la aplicación.


```
private void bt_reiniciarActionPerformed(java.awt.event.ActionEvent evt) {
    model.setRowCount(rowCount: 0);

    // Limpiar la tabla de datos
    DefaultTableModel modelRegistro = (DefaultTableModel) tabla_registro.getModel();
    modelRegistro.setRowCount(rowCount: 0);

    // Limpiar el área de texto
    txArea_EstadoLlamada.setText("");

    // Limpiar campos de texto
    tx_nombre.setText("");
    tx_telefono.setText("");
    combobox_operador.setSelectedIndex(selectedIndex: 0);
}
```

Salida:


— □ ×

## Simulacion Call Center

Nombre: 
 Telefono: 
 Operador: Seleccionar ▼

Guardar
Eliminar

Nombre	Teléfono	Operador	Llamada
Miguel	1234567	Tigo	Sí
Angel	7654321	Movistar	Sí
3421	123422131	Tigo	Sí

Iniciar Llamada

Estado de la Simulación: | | - Miguel llamando al número 1234567... | |  
 Estado de la Simulación: | | - Miguel ha terminado la llamada. Contesto | |  
 Estado de la Simulación: | | - Angel llamando al número 7654321... | |  
 Estado de la Simulación: | | - Angel ha terminado la llamada. Contesto | |  
 Estado de la Simulación: | | - Angel llamando al número 7654321... | |

Registro de Llamadas:

Nombre	Teléfono	Operador	Estado
Miguel	1234567	Tigo	Contesto
Angel	7654321	Movistar	Contesto
Angel	7654321	Movistar	Contesto
3421	123422131	Tigo	No contesto

Reiniciar

2. Desarrolle el siguiente ejercicio: “simular el proceso de cobro de un supermercado; es decir, unos clientes van con un carro lleno de productos y una cajera les cobra los productos, pasándolos uno a uno por el escáner de la caja registradora. En este caso la cajera debe de procesar la compra cliente a cliente.



```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package super_mercado;

import javax.swing.JTextArea;

/**
 *
 * @author FORERO
 */
class Cliente implements Runnable {
    private final String nombre;
    private final int productos;
    private final JTextArea areaTexto;

    public Cliente(String nombre, int productos, JTextArea areaTexto) {
        this.nombre = nombre;
        this.productos = productos;
        this.areaTexto = areaTexto;
    }

    @Override
    public void run() {
        try {
            areaTexto.append(nombre + " está comprando " + productos + " productos.\n");
            Thread.sleep(1000 * productos); // Simula el tiempo que tarda en comprar
            areaTexto.append(nombre + " ha terminado de comprar.\n");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public String getNombre() {
        return nombre;
    }

    public int getProductos() {
        return productos;
    }
}
```

```

/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package super_mercado;

import javax.swing.JTextArea;

/**
 *
 * @author FORERO
 */
class Cajera implements Runnable {
    private final Cliente cliente;
    private final JTextArea areaTexto;

    public Cajera(Cliente cliente, JTextArea areaTexto) {
        this.cliente = cliente;
        this.areaTexto = areaTexto;
    }

    @Override
    public void run() {
        try {
            areaTexto.append("Cajera comenzando a procesar la compra de " + cliente.getNombre() + ".\n");
            for (int i = 1; i <= cliente.getProductos(); i++) {
                areaTexto.append("Escaneando producto " + i + " de " + cliente.getNombre() + ".\n");
                Thread.sleep(1000); // Simula el tiempo de escaneo por producto
            }
            areaTexto.append("Cajera ha terminado de procesar la compra de " + cliente.getNombre() + ".\n");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

```

```

/*
package super_mercado;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.SwingUtilities;

/**
 *
 * @author FORERO
 */
// Clase principal para ejecutar la simulación
public class Super_mercado {
    private JTextArea areaTexto;

    public Super_mercado() {
        JFrame frame = new JFrame("Simulador de Supermercado");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 400);
        frame.setLayout(new BorderLayout());

        areaTexto = new JTextArea();
        areaTexto.setEditable(false);
        JScrollPane scrollPane = new JScrollPane(areaTexto);
        frame.add(scrollPane, BorderLayout.CENTER);

        JButton botonNuevoCliente = new JButton("Nuevo Cliente");
        frame.add(botonNuevoCliente, BorderLayout.SOUTH);

        botonNuevoCliente.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                agregarNuevoCliente();
            }
        });
    }
}

```



```

        frame.setVisible(true);
    }

    private void agregarNuevoCliente() {
        Random random = new Random();
        String nombreCliente = "Cliente " + (areaTexto.getLineCount() + 1);
        int productos = random.nextInt(5) + 1; // Cada cliente compra entre 1 y 5 productos
        Cliente cliente = new Cliente(nombreCliente, productos, areaTexto);

        Thread hiloCliente = new Thread(cliente);
        hiloCliente.start();

        // Esperar a que el cliente termine antes de procesar la compra
        try {
            hiloCliente.join(); // Esperar a que el cliente termine de comprar
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        // Crear y ejecutar hilo de la cajera para este cliente
        Cajera cajera = new Cajera(cliente, areaTexto);
        Thread hiloCajera = new Thread(cajera);
        hiloCajera.start();
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(Super_mercado::new);
    }
}

```

---

Cliente 2 está comprando 2 productos.  
 Cliente 2 ha terminado de comprar.  
 Cajera comenzando a procesar la compra de Cliente 2.  
 Escaneando producto 1 de Cliente 2.  
 Escaneando producto 2 de Cliente 2.  
 Cajera ha terminado de procesar la compra de Cliente 2.  
 Cliente 8 está comprando 1 productos.  
 Cliente 8 ha terminado de comprar.  
 Cajera comenzando a procesar la compra de Cliente 8.  
 Escaneando producto 1 de Cliente 8.  
 Cliente 12 está comprando 5 productos.  
 Cajera ha terminado de procesar la compra de Cliente 8.  
 Cliente 12 ha terminado de comprar.  
 Cajera comenzando a procesar la compra de Cliente 12.  
 Escaneando producto 1 de Cliente 12.  
 Escaneando producto 2 de Cliente 12.  
 Escaneando producto 3 de Cliente 12.  
 Escaneando producto 4 de Cliente 12.  
 Escaneando producto 5 de Cliente 12.  
 Cajera ha terminado de procesar la compra de Cliente 12.  
 Cliente 22 está comprando 3 productos.  
 Cliente 22 ha terminado de comprar.  
 Cajera comenzando a procesar la compra de Cliente 22.  
 Escaneando producto 1 de Cliente 22.  
 Escaneando producto 2 de Cliente 22.  
 Escaneando producto 3 de Cliente 22.  
 Cajera ha terminado de procesar la compra de Cliente 22.

3.

a) productor/ consumidor que trabaje con 2 Threads, el primer Thread generará números aleatorios entre 1 y 100 que serán leídos y multiplicados por 2 en el segundo Thread el cual imprimirá el resultado.

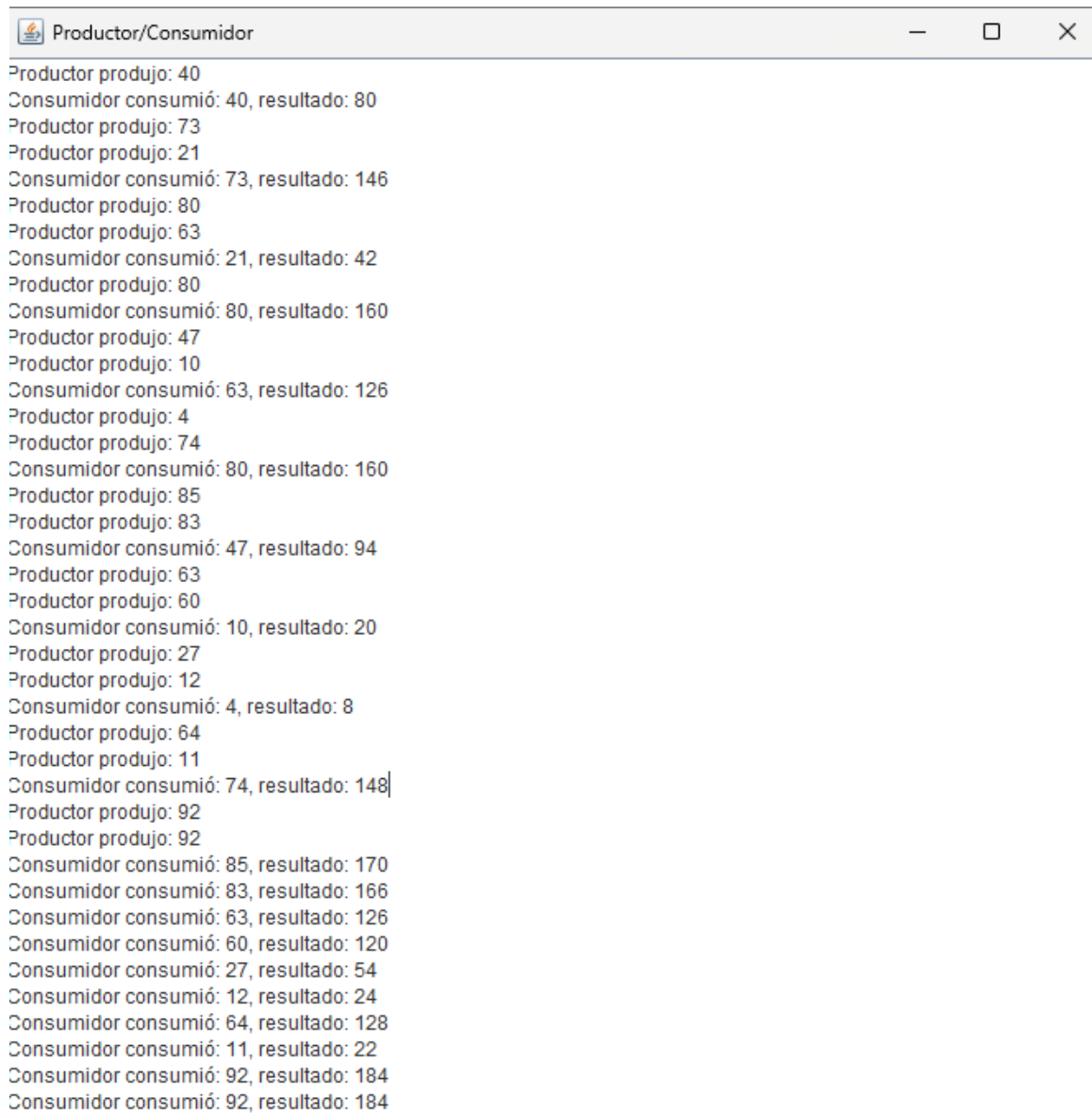
```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.util.Random;
4  import java.util.concurrent.ArrayBlockingQueue;
5
6  class Productor implements Runnable {
7      private ArrayBlockingQueue<Integer> queue;
8      private final int maxProducciones;
9      private JTextArea textArea; // Área de texto para mostrar la salida
10
11     public Productor(ArrayBlockingQueue<Integer> queue, int maxProducciones, JTextArea textArea) {
12         this.queue = queue;
13         this.maxProducciones = maxProducciones;
14         this.textArea = textArea;
15     }
16
17     @Override
18     public void run() {
19         Random random = new Random();
20         try {
21             for (int i = 0; i < maxProducciones; i++) {
22                 int number = random.nextInt(100) + 1;
23                 queue.put(number);
24                 textArea.append("Productor produjo: " + number + "\n");
25                 Thread.sleep(500);
26             }
27             queue.put(-1); // Señal para el consumidor
28         } catch (InterruptedException e) {
29             Thread.currentThread().interrupt();
30         }
31     }
32 }
33
34 class Consumidor implements Runnable {
35     private ArrayBlockingQueue<Integer> queue;
36     private JTextArea textArea; // Área de texto para mostrar la salida
37
38     public Consumidor(ArrayBlockingQueue<Integer> queue, JTextArea textArea) {
39         this.queue = queue;
40         this.textArea = textArea;
41     }
42
43     @Override
44     public void run() {
45         try {
46             while (true) {
47                 int number = queue.take();
48                 if (number == -1) {
49                     break; // Salir si recibe la señal de terminación
```

```

49         break; // Salir si recibe la señal de terminación
50     }
51     int result = number * 2;
52     textArea.append("Consumidor consumió: " + number + ", resultado: " + result + "\n");
53     Thread.sleep(1000);
54 }
55 } catch (InterruptedException e) {
56     Thread.currentThread().interrupt();
57 }
58 }
59 }
60
61 public class ProductorConsumidorGUI extends JFrame {
62     public ProductorConsumidorGUI() {
63         setTitle("Productor/Consumidor");
64         setSize(400, 300);
65         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
66         setLayout(new BorderLayout());
67
68         JTextArea textArea = new JTextArea();
69         textArea.setEditable(false);
70         JScrollPane scrollPane = new JScrollPane(textArea);
71         add(scrollPane, BorderLayout.CENTER);
72
73         ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<>(10);
74         int maxProducciones = 20;
75         Thread productorThread = new Thread(new Productor(queue, maxProducciones, textArea));
76         Thread consumidorThread = new Thread(new Consumidor(queue, textArea));
77
78         productorThread.start();
79         consumidorThread.start();
80     }
81
82     public static void main(String[] args) {
83         SwingUtilities.invokeLater(() -> {
84             ProductorConsumidorGUI frame = new ProductorConsumidorGUI();
85             frame.setVisible(true);
86         });
87     }
88 }

```

Salida:



```
Productor produjo: 40
Consumidor consumió: 40, resultado: 80
Productor produjo: 73
Productor produjo: 21
Consumidor consumió: 73, resultado: 146
Productor produjo: 80
Productor produjo: 63
Consumidor consumió: 21, resultado: 42
Productor produjo: 80
Consumidor consumió: 80, resultado: 160
Productor produjo: 47
Productor produjo: 10
Consumidor consumió: 63, resultado: 126
Productor produjo: 4
Productor produjo: 74
Consumidor consumió: 80, resultado: 160
Productor produjo: 85
Productor produjo: 83
Consumidor consumió: 47, resultado: 94
Productor produjo: 63
Productor produjo: 60
Consumidor consumió: 10, resultado: 20
Productor produjo: 27
Productor produjo: 12
Consumidor consumió: 4, resultado: 8
Productor produjo: 64
Productor produjo: 11
Consumidor consumió: 74, resultado: 148
Productor produjo: 92
Productor produjo: 92
Consumidor consumió: 85, resultado: 170
Consumidor consumió: 83, resultado: 166
Consumidor consumió: 63, resultado: 126
Consumidor consumió: 60, resultado: 120
Consumidor consumió: 27, resultado: 54
Consumidor consumió: 12, resultado: 24
Consumidor consumió: 64, resultado: 128
Consumidor consumió: 11, resultado: 22
Consumidor consumió: 92, resultado: 184
Consumidor consumió: 92, resultado: 184
```

b) Implemente un programa secuencial que calcule el producto de dos grandes matrices. Después modifíquelo para que esta tarea se realice entre cuatro Threads, cada uno ocupado de un subconjunto de la matriz resultado. Mida el tiempo que emplea cada una de las versiones.

```

1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5  import java.util.Random;
6
7  public class ProductoMatricesGUI extends JFrame {
8      private JTextField txtDimension;
9      private JTextArea txtResultado;
10     private JButton btnCalcularSecuencial;
11     private JButton btnCalcularConThreads;
12
13     public ProductoMatricesGUI() {
14         setTitle("Producto de Matrices");
15         setSize(400, 300);
16         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17         setLayout(new BorderLayout());
18
19         // Panel para ingresar la dimensión de las matrices
20         JPanel panelDimension = new JPanel();
21         panelDimension.add(new JLabel("Dimensión de las matrices:"));
22         txtDimension = new JTextField(5);
23         panelDimension.add(txtDimension);
24         add(panelDimension, BorderLayout.NORTH);
25
26         // Panel para mostrar el resultado
27         txtResultado = new JTextArea(10, 30);
28         txtResultado.setEditable(false);
29         JScrollPane scrollPane = new JScrollPane(txtResultado);
30         add(scrollPane, BorderLayout.CENTER);
31
32         // Panel para los botones de cálculo
33         JPanel panelBotones = new JPanel();
34         btnCalcularSecuencial = new JButton("Calcular Secuencial");
35         btnCalcularSecuencial.addActionListener(new ActionListener() {
36             @Override
37             public void actionPerformed(ActionEvent e) {
38                 calcularSecuencial();
39             }
40         });
41         panelBotones.add(btnCalcularSecuencial);
42
43         btnCalcularConThreads = new JButton("Calcular con Threads");
44         btnCalcularConThreads.addActionListener(new ActionListener() {
45             @Override
46             public void actionPerformed(ActionEvent e) {
47                 calcularConThreads();
48             }
49         });

```

```

49     });
50     panelBotones.add(btnCalcularConThreads);
51     add(panelBotones, BorderLayout.SOUTH);
52 }
53
54 private void calcularSecuencial() {
55     int n = Integer.parseInt(txtDimension.getText());
56     double[][] A = new double[n][n];
57     double[][] B = new double[n][n];
58     double[][] C = new double[n][n];
59
60     Random rand = new Random();
61     for (int i = 0; i < n; i++) {
62         for (int j = 0; j < n; j++) {
63             A[i][j] = rand.nextDouble();
64             B[i][j] = rand.nextDouble();
65         }
66     }
67
68     long inicio = System.nanoTime();
69     for (int i = 0; i < n; i++) {
70         for (int j = 0; j < n; j++) {
71             for (int k = 0; k < n; k++) {
72                 C[i][j] += A[i][k] * B[k][j];
73             }
74         }
75     }
76     long fin = System.nanoTime();
77
78     txtResultado.setText("Tiempo empleado en el cálculo secuencial: " + (fin - inicio) / 1_000_000.0 + " milisegundos");
79 }
80
81 private void calcularConThreads() {
82     int n = Integer.parseInt(txtDimension.getText());
83     double[][] A = new double[n][n];
84     double[][] B = new double[n][n];
85     double[][] C = new double[n][n];
86
87     Random rand = new Random();
88     for (int i = 0; i < n; i++) {
89         for (int j = 0; j < n; j++) {
90             A[i][j] = rand.nextDouble();
91             B[i][j] = rand.nextDouble();
92         }
93     }
94 }

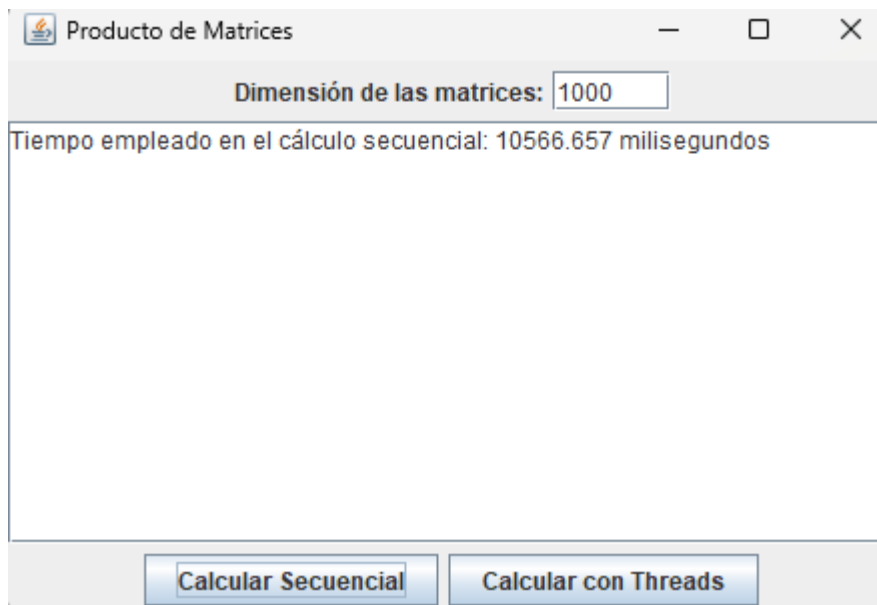
```


```

105     long inicio = System.nanoTime();
106     Thread[] threads = new Thread[4];
107     for (int i = 0; i < 4; i++) {
108         final int inicioFila = i * n / 4;
109         final int finFila = (i + 1) * n / 4;
110         threads[i] = new Thread(new Runnable() {
111             @Override
112             public void run() {
113                 for (int i = inicioFila; i < finFila; i++) {
114                     for (int j = 0; j < n; j++) {
115                         for (int k = 0; k < n; k++) {
116                             C[i][j] += A[i][k] * B[k][j];
117                         }
118                     }
119                 }
120             }
121         });
122         threads[i].start();
123     }
124     for (Thread thread : threads) {
125         try {
126             thread.join();
127         } catch (InterruptedException e) {
128             Thread.currentThread().interrupt();
129         }
130     }
131     long fin = System.nanoTime();
132     txtResultado.setText("Tiempo empleado en el cálculo con threads: " + (fin - inicio) / 1_000_000.0 + " milisegundos");
133 }
134
135 public static void main(String[] args) {
136     SwingUtilities.invokeLater(() -> {
137         ProductoMatricesGUI frame = new ProductoMatricesGUI();
138         frame.setVisible(true);
139     });
140 }
141 }

```

Salida:



 Producto de Matrices

— □ ×

Dimensión de las matrices:

Tiempo empleado en el cálculo con threads: 2364.8888 milisegundos

Calcular Secuencial

Calcular con Threads



## **Preguntas Orientadoras**

### **Análisis de Datos Obtenidos**

### **Aprendizajes Obtenidos**

#### **1. Concurrencia en Java:**

- Comprendí cómo funcionan las clases Thread e Runnable, cruciales para gestionar tareas simultáneas en aplicaciones, como en un call center.
- Relación Profesional: Fundamental para desarrollar aplicaciones eficientes y de alto rendimiento.

#### **2. Patrones de Diseño:**

- Aprendí sobre patrones como productor-consumidor, mejorando la estructura y escalabilidad del código.
- Relación Profesional: Mejora la mantenibilidad del software, una habilidad valorada en la industria.

#### **3. Resolución de Problemas Concurrentes:**

- Experimenté con condiciones de carrera y sincronización, aprendiendo a manejar el acceso a recursos compartidos.
- Relación Profesional: Prepara para enfrentar desafíos en el desarrollo de aplicaciones multihilo.

### **Dificultades Encontradas**

Tuve dificultades para entender la interacción entre hilos y cómo prevenir condiciones de carrera.

### **Estrategias de Solución**

1. Investigación: Leí documentación y ejemplos sobre concurrencia en Java.
2. Práctica: Implementé proyectos de prueba para simular problemas de concurrencia.
3. Colaboración: Participé en foros y grupos de estudio para discutir experiencias y soluciones.

## **Actividad de Trabajo Autónomo**

### **1. Paralelismo de Datos**

El paralelismo de datos se refiere a la técnica de procesar múltiples datos simultáneamente, dividiendo un conjunto de datos en partes más pequeñas que pueden ser procesadas en paralelo. Esta técnica es común en aplicaciones que requieren el procesamiento de grandes volúmenes de datos, como en el análisis de datos, procesamiento de imágenes, y simulaciones científicas.

- Ejemplo: En un algoritmo que debe realizar una operación matemática sobre una matriz, el paralelismo de datos permite que varias filas o columnas se procesen al mismo tiempo en diferentes núcleos de un procesador o en múltiples procesadores.
- Ventajas:
  - Aumento significativo en la velocidad de procesamiento.
  - Mejora en la eficiencia del uso de recursos del sistema.
  - Reducción del tiempo de ejecución para tareas que pueden dividirse.

### **2. Paralelismo de Tareas**

El paralelismo de tareas se refiere a la técnica de ejecutar múltiples tareas o hilos de ejecución al mismo tiempo. En lugar de centrarse en la división de datos, el paralelismo de tareas se enfoca en descomponer una tarea en subtareas que pueden ser ejecutadas de manera concurrente.

- Ejemplo: En una aplicación que realiza procesamiento de imágenes, una tarea puede ser cargar una imagen, mientras que otra puede ser aplicar un filtro a la imagen. Ambas tareas pueden ejecutarse simultáneamente en diferentes hilos.
- Ventajas:
  - Mejora en la capacidad de respuesta de aplicaciones interactivas.
  - Utilización efectiva de recursos multicore, permitiendo que múltiples tareas se ejecuten sin esperar a que otras terminen.
  - Permite la ejecución de tareas independientes que pueden no depender de los mismos datos.

## Ejemplo de Paralelismo de Datos:

```
import java.util.concurrent.RecursiveAction;
import java.util.concurrent.ForkJoinPool;

public class DataParallelismExample {
    // Clase que extiende RecursiveAction para realizar el cálculo en paralelo
    static class SquareTask extends RecursiveAction {
        private static final int THRESHOLD = 4; // Umbral para dividir la tarea
        private int[] data;
        private int start;
        private int end;

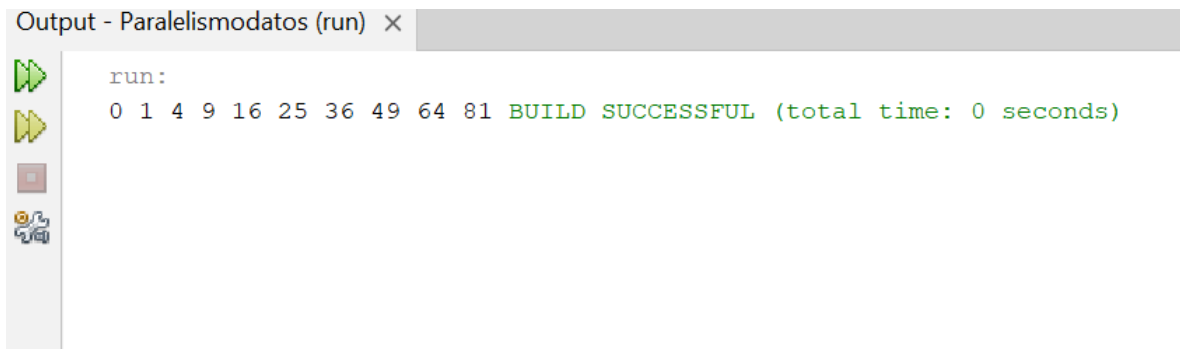
        public SquareTask(int[] data, int start, int end) {
            this.data = data;
            this.start = start;
            this.end = end;
        }

        @Override
        protected void compute() {
            if (end - start <= THRESHOLD) {
                // Cálculo secuencial si el tamaño es menor o igual al umbral
                for (int i = start; i < end; i++) {
                    data[i] = data[i] * data[i];
                }
            } else {
                // Dividir la tarea en subtareas
                int mid = (start + end) / 2;
                invokeAll(new SquareTask(data, start, end: mid), new SquareTask(data, start: mid, end));
            }
        }
    }

    public static void main(String[] args) {
        int[] data = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

        ForkJoinPool pool = new ForkJoinPool();
        pool.invoke(new SquareTask(data, start: 0, end: data.length));

        // Imprimir los resultados
        for (int num : data) {
            System.out.print(num + " "); // Salida: 0 1 4 9 16 25 36 49 64 81
        }
    }
}
```



```
run:
0 1 4 9 16 25 36 49 64 81 BUILD SUCCESSFUL (total time: 0 seconds)
```

### Explicación:

- **ForkJoinPool:** Esta clase proporciona un marco de trabajo para el procesamiento paralelo. Permite dividir tareas en subtareas más pequeñas y ejecutarlas en diferentes hilos.
- **RecursiveAction:** Esta es una clase abstracta que se utiliza para tareas que no retornan un resultado. Se extiende para definir cómo se deben realizar los cálculos.
- **SquareTask:**
  - Esta clase define la tarea de calcular el cuadrado de un arreglo de enteros.
  - THRESHOLD es un umbral que determina cuándo dividir la tarea. Si la cantidad de elementos a procesar es menor o igual a THRESHOLD, se procesan secuencialmente. Si es mayor, se divide en dos subtareas.
  - El método compute() es el núcleo del procesamiento. Si la tarea se puede ejecutar secuencialmente (es decir, el tamaño del rango es menor o igual al umbral), se calcula el cuadrado de cada número. De lo contrario, se divide la tarea en dos subtareas y se invocan en paralelo.
- **Main:**
  - Se crea un arreglo de enteros del 0 al 9.
  - Se crea un ForkJoinPool y se invoca la tarea SquareTask.
  - Finalmente, se imprimen los resultados. Los números originales se han transformado en sus cuadrados.

## Ejemplo de Paralelismo de Tareas:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TaskParallelismExample {
    // Función que simula la descarga de datos
    public static void downloadData(String url) {
        System.out.println("Descargando datos de " + url + "...");
        try {
            Thread.sleep(millis: 2000); // Simular tiempo de descarga
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        System.out.println("Datos descargados de " + url + ".");
    }

    // Función que simula el procesamiento de datos
    public static void processData() {
        System.out.println("Procesando datos...");
        try {
            Thread.sleep(millis: 3000); // Simular tiempo de procesamiento
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        System.out.println("Datos procesados.");
    }

    public static void main(String[] args) {
        String url = "http://example.com/data";

        ExecutorService executor = Executors.newFixedThreadPool(nThreads: 2); // Pool de hilos

        // Enviar las tareas al executor
        executor.submit(() -> downloadData(url));
        executor.submit(TaskParallelismExample::processData);

        // Cerrar el executor
        executor.shutdown();
        while (!executor.isTerminated()) {
            // Esperar a que todas las tareas terminen
        }
    }
}
```

```
run:
Procesando datos...
Descargando datos de http://example.com/data...
Datos descargados de http://example.com/data.
Datos procesados.
BUILD SUCCESSFUL (total time: 3 seconds)
```

### Explicación:

- **ExecutorService:** Este es un marco de trabajo que permite gestionar un grupo de hilos (pool) y ejecutar tareas en paralelo sin necesidad de gestionar directamente los hilos.
- **downloadData(String url):** Esta función simula la descarga de datos. Utiliza `Thread.sleep(2000)` para simular el tiempo que tomaría la descarga, y luego imprime un mensaje.
- **processData():** Esta función simula el procesamiento de datos. De forma similar, usa `Thread.sleep(3000)` para simular el tiempo de procesamiento.
- **Main:**
  - Se define una URL de ejemplo.
  - Se crea un `ExecutorService` con un pool de 2 hilos, lo que permite ejecutar hasta 2 tareas en paralelo.
  - Se envían dos tareas al executor: una para descargar datos y otra para procesarlos.
  - Después de enviar las tareas, se cierra el executor con `executor.shutdown()`, lo que indica que no se enviarán más tareas y se espera a que todas terminen.

## **Potenciales Problemas en el Paralelismo de Datos y Tareas**

### **1. Paralelismo de Datos:**

- Dependencias de Datos: Si las tareas dependen de datos que se están modificando, puede provocar errores.
- Sobrecarga de Comunicación: En entornos distribuidos, la transferencia de datos puede ser costosa.
- Desbalance de Carga: Algunas tareas pueden ser más pesadas que otras, lo que resulta en un uso ineficiente de los recursos.
- Aumento de Complejidad: La lógica de paralelismo puede aumentar la dificultad de mantenimiento y depuración.

### **2. Paralelismo de Tareas:**

- Condiciones de Carrera: Acceder a recursos compartidos sin sincronización puede causar inconsistencias.
- Bloqueos (Deadlocks): Situaciones donde dos o más hilos esperan indefinidamente entre sí.
- Complejidad de Sincronización: Sincronizar el acceso a recursos puede ser difícil y afectar el rendimiento.
- Costos de Creación de Hilos: Crear y destruir hilos puede ser costoso; un número excesivo de hilos puede reducir el rendimiento.

**Link códigos:**

<https://drive.google.com/drive/u/0/folders/1nggGT3a1i0TPMkAbl9q7TXGcqeCP-TBQ>

**Link GitHub:**

<https://github.com/Miguecapi/TallerProgramacionGuias>