Module: 5COSC019C Object-Oriented Programming
Practical Name: Ticket Management System

Weighting: 50%
Qualifying Mark: 30%

Description:
This practical involves designing and implementing a **Command-Line Interface (CLI)** and a **Graphical User Interface (GUI)** using JavaFX for managing ticket distribution and customer retrieval operations. The system employs **object-oriented programming principles**, **multi-threading**, **file handling**, and **logging** to simulate ticket vending and customer purchase functionalities in a synchronized and safe environment.
Students are required to implement essential functionalities, including system initialization, ticket management, vendor-consumer simulations, and real-time system visualization through a GUI.

Learning Outcomes Covered:
This assignment contributes to the following Learning Outcomes (LOs):
- LO2 Acquired detailed knowledge of concepts of object-oriented programming and apply characteristics, tools and environments to adapt to new computational environments and programming languages which are based on object- oriented principles within a time constraint
- LO3 Design implement applications based on an object-oriented programming language, given a set of functional requirements. Use APIs which have not been exposed to previously, in order to develop an application requiring specialised functionality.
- LO4 Design and Implement graphical interfaces using an object-oriented programming language.
- LO5 Apply appropriate techniques for evaluation and testing and adapt the performance accordingly.

Provided System:
A basic Java project template will be provided for students to expand upon. The provided files include:

- **CLI Components**: Initialization, configuration, and basic operations.
- **GUI Components**: User-friendly JavaFX-based interface for advanced interaction and visualization.

Project Files:

- **TicketManagementSystem.java**: Entry point of the system.
- **Configuration.java**: Manages system configuration and settings.
- **TicketPool.java**: Handles shared ticket management with thread-safe access.
- **Vendor.java**: Simulates ticket producers using threads.
- **Customer.java**: Simulates ticket consumers using threads.
- **Logger.java**: Logs activities for tracking and debugging.
- **JavaFXInterface.java:** GUI Interface

Task Instructions

CLI: System Initialization and Configuration

1. **Develop a CLI for Configuration**:
    - Prompt users to enter system parameters:
        - Total Number of Tickets (totalTickets)
        - Ticket Release Rate (ticketReleaseRate)
        - Customer Retrieval Rate (customerRetrievalRate)
        - Maximum Ticket Capacity (maxTicketCapacity)
    - Validate the inputs to ensure they are within acceptable ranges (e.g., positive integers). If invalid, prompt the user to re-enter the value.
2. **Configuration Management**:
    - Design a Configuration class to store system parameters.
    - Implement functionality to **save** and **load settings**:
        - Use **JSON** for serialization (via a library like Gson).
        - Alternatively, save the settings in a plain text file.

GUI: System Interaction and Visualization

1. **Design the JavaFX GUI**:
    - Create an interface to visualize ticket availability, logs, and system status.
    - Include input fields for configuration settings.
2. **Components of the GUI**:
    - **Labels**: Display current ticket pool status.
    - **TextFields**: Allow input of configuration parameters.
    - **Buttons**: Start/stop the system.
    - **ListView/TableView**: Show ticket pool logs and activities.
3. **Event Handling**:
    - Assign button actions to start and stop the system.
    - Use Platform.runLater() for thread-safe updates to the GUI.

Functional Requirements

4. Implementing the Ticket Vendor (Producer)

1. **Vendor Class**:
    - Implement the Runnable interface for multi-threading.
    - Simulate vendors releasing tickets into the shared pool.
2. **Multi-threading**:
    - Create multiple vendor threads, each releasing tickets independently.
3. **Synchronization**:
    - Use synchronized blocks or methods to ensure thread-safe access to the addTickets() method in TicketPool.

---

5. Implementing the Customer (Consumer)

1. **Customer Class**:
    - Implement the Runnable interface for multi-threading.
    - Simulate customers purchasing tickets from the shared pool.
2. **Multi-threading**:
    - Create multiple customer threads, each attempting to retrieve tickets independently.

3. **Synchronization**:
    - Use synchronized blocks or methods to ensure thread-safe access to the removeTicket() method in TicketPool.

---

6. Ticket Management

1. **TicketPool Class**:
    - Use a thread-safe data structure (e.g., Vector or Collections.synchronizedList).
2. **Methods**:
    - addTickets(): Allows vendors to add tickets.
    - removeTicket(): Allows customers to purchase tickets.
3. **Synchronization**:
    - Ensure thread safety with synchronized methods or blocks.

---

7. Logging and Error Handling

1. **Logging**:
    - Log activities such as ticket additions, removals, and system errors.
    - Implement both console logging (System.out.println) and file logging (using BufferedWriter or java.util.logging).
2. **Error Handling**:
    - Use try-catch blocks to handle exceptions.
    - Display meaningful error messages to inform users and suggest solutions.

Assessment Day Instructions:

1. **Permitted Resources**:
    - Access to the Blackboard course material.
    - Personal notes and textbooks.
2. **Prohibited Activities**:
    - No access to external websites, search engines, communication platforms, or generative AI tools (e.g., ChatGPT, Bard).
    - Violations may result in penalties per academic integrity policies.

Project Structure

```
TicketManagementSystem/
├── src/
│   ├── config/
│   │   └── Configuration.java
│   ├── core/
│   │   ├── TicketPool.java
│   │   ├── TicketOperation.java
│   │   └── AbstractTicketHandler.java
│   ├── exceptions/
│   │   └── InvalidConfigurationException.java
│   ├── logging/
│   │   └── Logger.java
│   ├── threads/
│   │   ├── Vendor.java
│   │   └── Customer.java
│   ├── ui/
│   │   ├── CommandLineInterface.java
│   │   └── JavaFXInterface.java
│   └── Main.java
├── resources/
│   ├── config.json
│   └── logs.txt
└── README.md
```

Class Implementations

1. Configuration.java (Encapsulation and File Handling)

```
package config;

import com.google.gson.Gson;
import java.io.*;

public class Configuration {
   private int totalTickets;
   private int ticketReleaseRate;
   private int customerRetrievalRate;
   private int maxTicketCapacity;

   public Configuration(int totalTickets, int ticketReleaseRate, int
customerRetrievalRate, int maxTicketCapacity) {
      this.totalTickets = totalTickets;
      this.ticketReleaseRate = ticketReleaseRate;
      this.customerRetrievalRate = customerRetrievalRate;
      this.maxTicketCapacity = maxTicketCapacity;
   }
```

```
    public int getTotalTickets() { return totalTickets; }
    public int getTicketReleaseRate() { return ticketReleaseRate; }
    public int getCustomerRetrievalRate() { return customerRetrievalRate; }
    public int getMaxTicketCapacity() { return maxTicketCapacity; }

    public static Configuration loadFromFile(String filePath) throws IOException {
        Gson gson = new Gson();
        try (Reader reader = new FileReader(filePath)) {
            return gson.fromJson(reader, Configuration.class);
        }
    }

    public void saveToFile(String filePath) throws IOException {
        Gson gson = new Gson();
        try (Writer writer = new FileWriter(filePath)) {
            gson.toJson(this, writer);
        }
    }
}
```

---

2. AbstractTicketHandler.java (Abstraction and Inheritance)

```
package core;

public abstract class AbstractTicketHandler {
    protected TicketPool ticketPool;

    public AbstractTicketHandler(TicketPool ticketPool) {
        this.ticketPool = ticketPool;
    }

    public abstract void handleTickets();
}
```

---

3. TicketOperation.java (Interface and Polymorphism)

java
Copy code

```
package core;

public interface TicketOperation {
    void addTickets(String ticket);
    String removeTicket();
}
```

---

4. TicketPool.java (Java Collections and Synchronization)

```
package core;

import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

public class TicketPool implements TicketOperation {
  private final List<String> tickets = Collections.synchronizedList(new LinkedList<>());

  @Override
  public synchronized void addTickets(String ticket) {
    tickets.add(ticket);
  }

  @Override
  public synchronized String removeTicket() {
    return tickets.isEmpty() ? null : tickets.remove(0);
  }

  public int getTicketCount() {
    return tickets.size();
  }
}
```

---

5. Vendor.java (Multi-threading and Inheritance)

```
package threads;

import core.AbstractTicketHandler;
import logging.Logger;

public class Vendor extends AbstractTicketHandler implements Runnable {
  private final int ticketReleaseRate;

  public Vendor(TicketPool ticketPool, int ticketReleaseRate) {
    super(ticketPool);
    this.ticketReleaseRate = ticketReleaseRate;
  }

  @Override
  public void run() {
    for (int i = 0; i < ticketReleaseRate; i++) {
      String ticket = "Ticket-" + System.nanoTime();
      ticketPool.addTickets(ticket);
      Logger.log("Vendor added: " + ticket);
```

```
        try {
          Thread.sleep(500);
        } catch (InterruptedException e) {
          Logger.log("Vendor interrupted.");
        }
      }
    }

    @Override
    public void handleTickets() {
      run();
    }
}
```

---

6. Customer.java (Multi-threading and Polymorphism)

```
package threads;

import core.AbstractTicketHandler;
import logging.Logger;

public class Customer extends AbstractTicketHandler implements Runnable {
    public Customer(TicketPool ticketPool) {
      super(ticketPool);
    }

    @Override
    public void run() {
      while (true) {
        String ticket = ticketPool.removeTicket();
        if (ticket != null) {
          Logger.log("Customer retrieved: " + ticket);
        } else {
          Logger.log("Customer found no tickets available.");
          break;
        }
        try {
          Thread.sleep(500);
        } catch (InterruptedException e) {
          Logger.log("Customer interrupted.");
        }
      }
    }

    @Override
    public void handleTickets() {
      run();
```

```
    }
}
```

---

7. Logger.java (Encapsulation and File Handling)

```
package logging;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.time.LocalDateTime;

public class Logger {
    private static final String LOG_FILE = "resources/logs.txt";

    public static void log(String message) {
        String timeStampedMessage = LocalDateTime.now() + ": " + message;
        System.out.println(timeStampedMessage);
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(LOG_FILE, true))) {
            writer.write(timeStampedMessage);
            writer.newLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

---

8. CommandLineInterface.java (CLI with Validation)

```
package ui;

import config.Configuration;
import logging.Logger;

import java.util.Scanner;

public class CommandLineInterface {
    public static Configuration configureSystem() {
        Scanner scanner = new Scanner(System.in);
        Logger.log("Starting system configuration...");

        int totalTickets = getInput(scanner, "Enter Total Tickets: ");
        int ticketReleaseRate = getInput(scanner, "Enter Ticket Release Rate: ");
        int customerRetrievalRate = getInput(scanner, "Enter Customer Retrieval Rate: ");
        int maxTicketCapacity = getInput(scanner, "Enter Max Ticket Capacity: ");

        Logger.log("System configured successfully.");
```

```
        return new Configuration(totalTickets, ticketReleaseRate, customerRetrievalRate,
maxTicketCapacity);
    }

    private static int getInput(Scanner scanner, String prompt) {
        int value;
        while (true) {
            System.out.print(prompt);
            try {
                value = Integer.parseInt(scanner.nextLine());
                if (value > 0) {
                    return value;
                } else {
                    System.out.println("Value must be positive. Try again.");
                }
            } catch (NumberFormatException e) {
                System.out.println("Invalid input. Please enter a number.");
            }
        }
    }
}
```

Main.java (Integration)

```
package main;

import config.Configuration;
import core.TicketPool;
import logging.Logger;
import threads.Customer;
import threads.Vendor;
import ui.CommandLineInterface;

public class Main {
    public static void main(String[] args) {
        Configuration config = CommandLineInterface.configureSystem();
        TicketPool ticketPool = new TicketPool();

        Thread vendor = new Thread(new Vendor(ticketPool,
config.getTicketReleaseRate()));
        Thread customer = new Thread(new Customer(ticketPool));

        vendor.start();
        customer.start();

        try {
            vendor.join();
```

```
            customer.join();
        } catch (InterruptedException e) {
            Logger.log("Main thread interrupted.");
        }

        Logger.log("System terminated.");
    }
}
```

JavaFX-Based GUI Code
JavaFXInterface.java

```
package ui;

import config.Configuration;
import core.TicketPool;
import logging.Logger;
import threads.Customer;
import threads.Vendor;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class JavaFXInterface extends Application {

    private TextField totalTicketsField;
    private TextField ticketReleaseRateField;
    private TextField customerRetrievalRateField;
    private TextField maxTicketCapacityField;
    private Label statusLabel;
    private TableView<String> logTable;
    private TicketPool ticketPool;
    private Thread vendorThread;
    private Thread customerThread;

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Ticket Management System");

        // Configuration input fields
        GridPane gridPane = new GridPane();
        gridPane.setVgap(10);
        gridPane.setHgap(10);
```

```
totalTicketsField = new TextField();
ticketReleaseRateField = new TextField();
customerRetrievalRateField = new TextField();
maxTicketCapacityField = new TextField();

gridPane.add(new Label("Total Tickets:"), 0, 0);
gridPane.add(totalTicketsField, 1, 0);
gridPane.add(new Label("Ticket Release Rate:"), 0, 1);
gridPane.add(ticketReleaseRateField, 1, 1);
gridPane.add(new Label("Customer Retrieval Rate:"), 0, 2);
gridPane.add(customerRetrievalRateField, 1, 2);
gridPane.add(new Label("Max Ticket Capacity:"), 0, 3);
gridPane.add(maxTicketCapacityField, 1, 3);

// Buttons
Button startButton = new Button("Start");
Button stopButton = new Button("Stop");

gridPane.add(startButton, 0, 4);
gridPane.add(stopButton, 1, 4);

// Status Label
statusLabel = new Label("System Status: Stopped");
gridPane.add(statusLabel, 0, 5, 2, 1);

// Log Table
logTable = new TableView<>();
logTable.setPlaceholder(new Label("Logs will appear here"));
gridPane.add(new Label("System Logs:"), 0, 6);
gridPane.add(logTable, 0, 7, 2, 1);

// Start button event
startButton.setOnAction(event -> {
  try {
    startSystem();
  } catch (Exception e) {
    Logger.log("Error starting system: " + e.getMessage());
    updateStatus("Error: Check input values.");
  }
});

// Stop button event
stopButton.setOnAction(event -> stopSystem());

// Scene setup
Scene scene = new Scene(gridPane, 600, 400);
primaryStage.setScene(scene);
```

```
        primaryStage.show();
    }

    private void startSystem() throws Exception {
        // Validate and parse input
        int totalTickets = validateInput(totalTicketsField.getText(), "Total Tickets");
        int ticketReleaseRate = validateInput(ticketReleaseRateField.getText(), "Ticket
Release Rate");
        int customerRetrievalRate = validateInput(customerRetrievalRateField.getText(),
"Customer Retrieval Rate");
        int maxTicketCapacity = validateInput(maxTicketCapacityField.getText(), "Max
Ticket Capacity");

        // Initialize configuration and ticket pool
        Configuration config = new Configuration(totalTickets, ticketReleaseRate,
customerRetrievalRate, maxTicketCapacity);
        ticketPool = new TicketPool();

        // Start threads
        vendorThread = new Thread(new Vendor(ticketPool, config.getTicketReleaseRate()));
        customerThread = new Thread(new Customer(ticketPool));
        vendorThread.start();
        customerThread.start();

        updateStatus("System Running...");
    }

    private void stopSystem() {
        if (vendorThread != null && customerThread != null) {
            vendorThread.interrupt();
            customerThread.interrupt();
        }
        updateStatus("System Stopped.");
    }

    private void updateStatus(String status) {
        Platform.runLater(() -> statusLabel.setText("System Status: " + status));
    }

    private int validateInput(String input, String fieldName) throws Exception {
        try {
            int value = Integer.parseInt(input);
            if (value <= 0) {
                throw new Exception(fieldName + " must be positive.");
            }
            return value;
        } catch (NumberFormatException e) {
```

```
            throw new Exception(fieldName + " must be a valid integer.");
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

How the GUI Works

1. **Input Fields**:
    - Users input the configuration parameters (e.g., total tickets, release rate) in the text fields.
2. **Buttons**:
    - **Start**: Initializes the system, starts threads for vendors and customers, and begins the ticket operations.
    - **Stop**: Stops all running threads and halts the system.
3. **Status Label**:
    - Displays the current system status (e.g., "Running", "Stopped").
4. **Log Table**:
    - Shows logs generated during the system's operation, including ticket additions and retrievals.
5. **Thread-Safe Updates**:
    - Uses Platform.runLater() to ensure thread-safe updates to the JavaFX UI.

Sample Question for Lab Based Test

1. Add Logging to Monitor Ticket Operations
Question:
Modify the Logger class to log all ticket operations (addTickets and removeTicket) in a standardized format that includes:

- The ticket ID.
- The type of operation (add/remove).
- The timestamp of the operation.

Update the TicketPool class to use the Logger for every ticket operation. Write a test scenario to verify the logs.

---

2. Extending Customer Functionality with Interfaces
Question:
Refactor the system to allow customers to retrieve tickets either:

- By ticket priority.
- By ticket ID.

Create an interface TicketRetrievalStrategy with methods for both retrieval approaches. Implement two concrete classes, PriorityRetrieval and IDRetrieval, and modify the Customer class to use these strategies. Write a test case to demonstrate both retrieval methods.

---

3. Exception Handling for Invalid Configurations
Question:
Create a custom exception InvalidConfigurationException for handling invalid system configurations (e.g., negative ticket capacity or retrieval rates).

- Add validation in the Configuration class to check for invalid values.
- If validation fails, throw the exception with a descriptive message.
  Update the CLI to handle this exception and prompt the user to enter valid inputs.

---

4. Enhancing Vendor Behavior Using Polymorphism
Question:
Extend the Vendor class to support different types of vendors:

- FastVendor: Adds tickets at double the standard rate.
- SlowVendor: Adds tickets at half the standard rate.

Use polymorphism to integrate these behaviors seamlessly into the system. Write a test scenario to demonstrate both vendor types operating concurrently.

---

5. Periodic Ticket Statistics Reporting
Question:
Add a feature to the system to periodically report statistics about the TicketPool:

- Total tickets added.
- Total tickets removed.
- Current tickets in the pool.

Implement this functionality in a separate thread that runs every 5 seconds and logs the statistics using the Logger. Write a test case to verify the periodic reporting.

6. GUI for Dynamic Ticket Pool Visualization

Question:

Enhance the JavaFX GUI to display a dynamic visualization of the ticket pool:

- Add a **ListView** to display all tickets currently in the pool.
- Update the view in real-time as tickets are added or removed. Use JavaFX's Platform.runLater() to ensure thread-safe updates to the GUI. Write a test case to verify real-time updates.

Time Allocation Summary

Task	**Time (minutes)**	**Marks (Points)**
Logging for Ticket Operations	15	/15
Extending Customer Functionality	15	/15
Exception Handling for Configurations	10	/10
Enhancing Vendor Behavior	15	/15
Periodic Ticket Statistics Reporting	15	/15
GUI for Dynamic Visualization	20	/20
Total	**90**	**/90**

Total Marks = Total / 90 * 100