Movie Ticket Booking System

1. Title : Movie ticket booking system

2. Problem statement :

“ Browse movies/shows, view seat layout, and book tickets. ”

The objective of this system is to provide users with a convenient and user-friendly way to browse movies or shows, check seat availability, and book tickets online. The system should simulate the real-world process of selecting a movie, choosing seats, and confirming a booking.

3. Introduction :

The **Movie Ticket Booking System** serves as a foundational console-based application that mimics the core operations of popular online ticketing platforms. Developed in Java, the system enables users to interactively explore show listings, view seat availability, and perform booking and cancellation operations in real-time. The application is initialized with a predefined list of movie shows across various theaters, each associated with essential metadata such as title, showtime, language, certificate type, and ticket price. For each show, a standardized seating layout is created using a 2D-style structure (seats A1 to E5), which simplifies both seat visualization and management.

When booking a ticket, users are prompted to enter a valid Show ID and choose from available seats. The system validates user input to ensure the seat exists and hasn't already been booked, thus preventing double bookings. Bookings are stored in an in-memory `Map`, where the user's ID is linked to their chosen seat. Similarly, ticket cancellation is handled by checking if the user has an existing booking, and if so, their seat is released back into the available pool.This application uses basic data structures such as `List`, `Map`, and custom `Show` objects, demonstrating principles of object-oriented programming, encapsulation, and list manipulation. The design is modular, with separate methods for listing shows, booking, cancelling, and displaying seats. While the system currently runs in-memory without persistent storage or concurrency handling, it lays the groundwork for enhancements such as database integration, user authentication, seat class categorization (e.g., Premium, Standard), and multi-user support with thread-safe booking mechanisms.

In summary, this project provides a simplified yet insightful look into how movie ticketing systems operate under the hood. It's particularly useful for students or developers learning Java and system design, and can be expanded into a more robust application with additional features like payment integration, QR code generation for tickets, and a graphical user interface.

4. Objectives :

- Develop a console-based application for movie ticket booking using Java.

- Allow users to view available movie shows with details like title, time, language, and price.
- Enable users to select and book specific seats from a predefined layout (A1–E5).
- Provide functionality to cancel booked tickets and update seat availability accordingly.
- Implement object-oriented programming principles for better structure and modularity.
- Use data structures like `List` and `Map` to manage shows, seats, and bookings in memory.
- Ensure input validation and prevent duplicate or invalid bookings.
- Maintain real-time seat availability updates after booking or cancellation.
- Create a simple, interactive menu-driven user interface in the console.
- Lay the foundation for future enhancements like database integration, UI improvements, or payment systems.

5. Methedology :

The **Movie Ticket Booking System** is designed as a console-based Java application that simulates a real-world ticket reservation experience. This methodology outlines the step-by-step approach taken to analyze, design, develop, and test the system using fundamental software engineering principles. The methodology follows the **waterfall model**, ensuring each phase is completed before moving to the next, maintaining structure and clarity throughout development.

i) Requirement Analysis

The first step involved identifying and defining the **functional and non-functional requirements** of the system.

Functional Requirements:-

- List all available movie shows by city.
- Display seat availability for a selected show.
- Book a ticket by selecting a seat and entering user details.
- Cancel an existing booking by user ID.

Non-Functional Requirements:

- System should be menu-driven and easy to use via console.
- Validate seat inputs and prevent duplicate bookings.
- Maintain internal data integrity (available vs. booked seats).
- Modular, scalable, and maintainable code using Java OOP.

ii) System Design

The system design phase involves the architectural planning of components and data structures to support core features like booking and cancellation.

Architecture :-

| User Interface | ← Console I/O (Main Menu)

↓

| Application | ← Main Logic (book/cancel/list)

↓

| Data Management | ← Show objects, seat list, bookings map

iii) Modular Design Breakdown

- **Main Class (`practise`)**

  - ✓ Acts as the entry point of the application.
  - ✓ Manages user interaction through a text-based menu.
  - ✓ Contains the main loop to list shows, book, cancel, or exit.

- **Show Class**

  - ✓ Represents a movie show with fields: `showId`, `title`, `time`, `theater`, `language`, `certificate`, `price`, and `availableSeats`.
  - ✓ Initializes seat layout (A1 to E5) using nested loops.
  - ✓ Encapsulates show-specific seat data.

- **addShows() Method**

  - ✓ Adds pre-defined shows to the system.
  - ✓ Initializes the `shows` list with sample movie data.

- **listShows(String city) Method**

  - ✓ Displays available shows.
  - ✓ (Currently not filtering based on city — future enhancement needed.)

- **bookTicket(Scanner sc) Method**

  - ✓ Allows a user to select a show and book a seat.
  - ✓ Validates seat format and availability.
  - ✓ Updates `availableSeats` and `bookings` map.

- **cancelTicket(Scanner sc) Method**

  - ✓ Cancels a booking based on the user ID.
  - ✓ Returns the seat to the available list.

- **displaySeats(Show show) Method**

✓ Displays seat layout in a structured grid format (5 seats per row).
✓ Only shows currently available seats.

- **isValidSeat(String seat) Method**

✓ Uses regular expression to validate seat ID format (`A1` to `E5`).

iv) Data Structures and Logic

Core Data Structures

✓ List<Show> shows → Stores all movie show details.
✓ Map<String, String> bookings → Maps each userId to the booked seat.
✓ List<String> availableSeats → Seat availability for each show (per instance).

Booking Flow

✓ User selects **Book Ticket** from the main menu.
✓ System prompts for a valid **Show ID**.
✓ System displays **available seats** for the selected show.
✓ User inputs **seat ID** (e.g., A3) and **User ID**.
✓ System validates seat format and availability.
✓ Seat is marked as **booked** (removed from the list).
✓ Booking is stored in the bookings map with userId as the key.

Cancellation Flow

✓ User selects **Cancel Ticket** from the menu.
✓ System prompts for **User ID**.
✓ If a booking is found, the seat is added back to the availableSeats list.
✓ Booking is removed from the bookings map.

Seat Layout Implementation

The seat layout is generated as a **5x5 grid**:

1)Rows: A to E

2)Columns: 1 to 5

3)Seat IDs: A1 to E5

V) Programming Language & Tools & Flow

✓ The project is developed using the Java programming language.

- ✓ Object-Oriented Programming (OOP) principles are followed for modular design.
- ✓ The development environment used is IntelliJ IDEA or Eclipse.
- ✓ User input is handled through the Scanner class for console-based interaction.
- ✓ Output and messages are displayed using System.out.println() statements.
- ✓ The application is executed entirely via the command-line interface (CLI).
- ✓ No GUI framework is used; the system runs purely in text mode.
- ✓ There is no external data persistence; all data is stored in-memory.
- ✓ On program termination, all data (shows, bookings, seat availability) is lost.
- ✓ Java's built-in data structures like List and Map are used for storing shows and bookings.
- ✓ List<Show> is used to store all movie show details.
- ✓ Map<String, String> is used to track seat bookings against user IDs.
- ✓ The logic is implemented through clearly defined methods for booking, canceling, and listing.
- ✓ The system is single-threaded and designed for single-user use cases.

Booking Flow

- ✓ User selects **Book Ticket** from the main menu.
- ✓ System prompts for a valid **Show ID**.
- ✓ System displays **available seats** for the selected show.
- ✓ User inputs **seat ID** (e.g., A3) and **User ID**.
- ✓ System validates seat format and availability.
- ✓ Seat is marked as **booked** (removed from the list).
- ✓ Booking is stored in the bookings map with userId as the key.

Cancellation Flow

- ✓ User selects **Cancel Ticket** from the menu.
- ✓ System prompts for **User ID**.
- ✓ If a booking is found, the seat is added back to the availableSeats list.
- ✓ Booking is removed from the bookings map.

vi) Testing & Validation

- ✓ Functional testing was conducted to ensure the core features work as expected.
- ✓ Booking a valid seat correctly removes it from the available list and confirms success.
- ✓ Attempting to book an already booked seat results in an error message.
- ✓ Invalid seat IDs like Z1 or A9 are rejected through regex validation.
- ✓ Cancelling a booking with a valid user ID returns the seat back to the available list.
- ✓ Cancelling with an invalid or unbooked user ID displays an appropriate error message.
- ✓ Boundary testing was performed using the first (A1) and last (E5) seats in the layout.

- ✓ Tests confirmed that duplicate bookings using the same user ID are not allowed.
- ✓ Inputs such as non-numeric show IDs and empty strings are handled to avoid crashes.
- ✓ Display of available seats is tested for correct row/column layout.
- ✓ System stability is verified through multiple consecutive bookings and cancellations.
- ✓ Manual verification ensured that all bookings are user-specific and do not overlap.
- ✓ No concurrency testing was performed, as the application is single-threaded.
- ✓ Edge cases like booking all seats in a show were tested successfully.
- ✓ The in-memory storage was validated by checking runtime behavior (data resets on restart).

vii) Limitations

- ✓ **No city-based filtering**: The city input is taken but not used in filtering logic.
- ✓ **One booking per user**: User can only book one seat at a time.
- ✓ **No data persistence**: All data is stored in-memory and lost when the app stops.
- ✓ **Single-user flow**: No multi-threading or concurrent bookings supported.

6 .Outcomes :

The **Movie Ticket Booking System** project achieved its primary objective of simulating a functional ticket booking experience using Java. Through this project, we successfully developed a console-based application that allows users to browse available movie shows, view seat availability, and perform ticket bookings and cancellations. The system was designed with simplicity, modularity, and user interaction in mind, making it an effective prototype of a real-world application.

- ➢ A fully functional **menu-driven application** was implemented using Java.
- ➢ Core features like **show listing, seat selection, booking, and cancellation** were successfully integrated.
- ➢ The project demonstrated effective use of **Object-Oriented Programming (OOP)** concepts such as class creation, encapsulation, and modular design.
- ➢ Java's **Collections Framework** (`List`, `Map`) was efficiently used to manage dynamic data like seats and bookings.
- ➢ Seat layout was **automatically generated** in a 5x5 format and displayed in real-time for each show.
- ➢ Input validation and error handling were implemented to ensure **data integrity and user-friendly interaction**.
- ➢ Conducted extensive **functional and boundary testing** to verify the correctness and robustness of the system.

- Gained practical experience in **building console-based applications** that simulate real-life systems.
- Laid the foundation for future expansion, including **GUI development, database integration**, and multi-user handling.

7 . Implementation code :

```
import java.util.*;

public class practise {

    static List<Show> shows = new ArrayList<>();

    static Map<String, String> bookings = new HashMap<>(); // userId -> seat

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        addShows();

        while (true) {

            System.out.println("==== Movie Ticket Booking ====");

            System.out.println("1. List shows by city");

            System.out.println("2. Book ticket");

            System.out.println("3. Cancel ticket");

            System.out.println("4. Exit");

            System.out.print("Enter choice: ");

            int choice = sc.nextInt();

            sc.nextLine();

            switch (choice) {

                case 1:

                    System.out.print("Enter city: ");

                    String city = sc.nextLine();

                    listShows(city);

                    break;

                case 2:

                    bookTicket(sc);

                    break;
```

```
            case 3:
                cancelTicket(sc);
                break;
            case 4:
                System.out.println("Thank you !");
                return;
            default:
                System.out.println("Invalid choice!");
        }
    }
}
static void addShows() {
    shows.add(new Show(1, "Madharaasi", "24-08-2025 12:13 pm", "PVR Cinemas - Audi1", "Tamil", "U/A", 180));
    shows.add(new Show(2, "Coolie", "24-08-2025 03:30 pm", "INOX - Gold", "Tamil", "A", 200));
    shows.add(new Show(3, "Jananayagan", "24-08-2025 06:00 pm", "Kasi Cinemas - Audi3", "Tamil", "U/A", 150));
    shows.add(new Show(4, "Thug Life", "24-08-2025 09:15 am", "DMAX", "Tamil", "A", 220));
    shows.add(new Show(5, "Avatar 2 :The Way of Water (3-D)", "24-08-2025 11:45 am", "IMAX - Dolby Atmos", "English",
"U/A", 350));
    shows.add(new Show(6, "Deadpool and Wolverine", "24-08-2025 05:20 pm", "Sathyam Cinemas - Audi5", "English", "A",
300));
    shows.add(new Show(7, "Avengers : Doomsday (3-D)", "24-08-2025 08:30 pm", "Escape Cinemas - Audi2", "English",
"U/A", 320));
}
static void listShows(String city) {
    System.out.println("Available shows in " + city + ":");
    for (Show show : shows) {
        System.out.println("ShowID: " + show.showId + " | " + show.title + " | " + show.time + " | " +
                show.theater + " | Language: " + show.language + " | Cert: " + show.certificate + " | Price: ₹" + show.price);
    }
}
static void bookTicket(Scanner sc) {
    System.out.print("Enter ShowID: ");
```

```
int showId = sc.nextInt();
sc.nextLine();
Show selectedShow = null;
for (Show show : shows) {
    if (show.showId == showId) {
        selectedShow = show;
        break;
    }
}
if (selectedShow == null) {
    System.out.println("Invalid ShowID!");
    return;
}
System.out.println("Available seats:");
displaySeats(selectedShow);
System.out.print("Enter seat (e.g., A1): ");
String seat = sc.nextLine().toUpperCase();
if (!isValidSeat(seat)) {
    System.out.println("Please enter a correct seat ID like A1, B2, etc.");
    return;
}
if (!selectedShow.availableSeats.contains(seat)) {
    System.out.println("Seat not available!");
    return;
}
System.out.print("Enter UserID: ");
String userId = sc.nextLine();
selectedShow.availableSeats.remove(seat);
bookings.put(userId, seat);
// Updated booking success message
```

```
        System.out.println("Booking successful for movie: " + selectedShow.title
                + " | Seat: " + seat
                + " | Language: " + selectedShow.language
                + " | Cert: " + selectedShow.certificate
                + " | Price: " + selectedShow.price);
    }
    static void cancelTicket(Scanner sc) {
        System.out.print("Enter UserID: ");
        String userId = sc.nextLine();
        if (!bookings.containsKey(userId)) {
            System.out.println("No booking found for this UserID!");
            return;
        }
        String seat = bookings.remove(userId);
        for (Show show : shows) {
            if (!show.availableSeats.contains(seat)) {
                show.availableSeats.add(seat);
                break;
            }
        }
        System.out.println("Ticket cancelled successfully!");
    }
    static void displaySeats(Show show) {
        int count = 0;
        for (String seat : show.availableSeats) {
            System.out.print("[" + seat + "] ");
            count++;
            if (count % 5 == 0) System.out.println();
        }
        System.out.println();
```

```
    }
    static boolean isValidSeat(String seat) {
        return seat.matches("^[A-E][1-5]$");
    }
}
class Show {
    int showId;
    String title;
    String time;
    String theater;
    String language;
    String certificate;
    int price;
    List<String> availableSeats;
    Show(int showId, String title, String time, String theater, String language, String certificate, int price) {
        this.showId = showId;
        this.title = title;
        this.time = time;
        this.theater = theater;
        this.language = language;
        this.certificate = certificate;
        this.price = price;
        this.availableSeats = new ArrayList<>();
        // Generate seats (A1-E5)
        char[] rows = {'A', 'B', 'C', 'D', 'E'};
        for (char row : rows) {
            for (int i = 1; i <= 5; i++) {
                availableSeats.add(row + String.valueOf(i));
            }
      }
```

}}

8 . Sample Screenshot