



BIRMINGHAM CITY
University

Name: Sadikshya Ghimire/Prashamsa Rijal

Student No: 23140745 / 23140743

Date: March 12, 2024

Module: Object Oriented Programming

Module Code: CMP5332

CourseWork: Flight Booking System

Tutor: Sumanta Silwal

Word Count: 2998 words

Table of Content

Introduction.....	4
Purpose and Functionality.....	4
Key Features:.....	4
Functionality Overview:.....	5
Main.java.....	6
• Description:.....	6
• Output:.....	7
Commands.....	8
1. Addcustomer.....	8
• Description:.....	8
• Output:.....	9
2. ListCustomers.....	9
• Description:.....	10
• Output:.....	10
3.ShowCustomer.....	11
• Description:.....	11
4) ShowFlight.....	12
• Description:.....	12
5) AddBooking.....	13
• Description:.....	13
• Description :.....	14
• Output:.....	15
• Description:.....	16
• Output:.....	17
8) CancelBooking.....	17
• Description:.....	18
• Output:.....	18
Data of txt file.....	19
1) Bookings.txt:.....	19
2) Customers.txt.....	19
3) Flights.txt.....	19
1. Flight.java.....	20
2. Customer.java.....	20
4. Flight Booking System.....	21
Junit Testing:.....	22
Graphical User Interface (GUI) :.....	23
1) Flight.....	24
• Add.....	24
• Delete.....	24

• View.....	25
• Passenger List.....	25
2) Customer.....	26
• Add.....	26
• Delete.....	26
• View.....	27
3) Booking.....	27
• Add.....	27
• Cancel.....	28
• View All.....	28
Java Documentation:.....	29
Conclusion :.....	29

Introduction

The Flight Booking System stands as a fully functional and sophisticated solution, engineered to revolutionize the management of flight bookings within the airline industry. Acting as a centralized hub, our system offers a seamless experience for both customers and airline administrators alike. For customers, it serves as a user-friendly portal where they can effortlessly explore a wide array of available flights, make bookings in a few simple clicks, and tailor their travel plans according to their preferences. On the other hand, airline administrators leverage the system's robust suite of tools to efficiently orchestrate flight schedules, monitor seat availability, and effectively manage customer bookings with precision and ease. By providing a comprehensive set of features within a single, integrated platform, our Flight Booking System enhances operational efficiency, fosters smoother communication between airlines and passengers, and ultimately elevates the overall quality of the booking experience within the dynamic aviation landscape.

Purpose and Functionality

The primary purpose of the Flight Booking System is to simplify and automate the booking process for both customers and airline staff. It allows customers to search for flights based on various criteria such as destination, departure date, and flight preferences. Once a suitable flight is found, customers can proceed to book their tickets securely through the system.

For airline administrators, the system provides tools to manage flight schedules, aircraft availability, customer bookings, and other operational aspects. They can add new flights, update existing schedules, view passenger lists, and make necessary adjustments to accommodate changing demands or unforeseen circumstances.

Key Features:

1. ***Flight Management:*** The system maintains a database of available flights, including details such as flight numbers, origins, destinations, departure dates, and seat availability.
2. ***Customer Management:*** Customers can create accounts, view their booking history, and manage their reservations. Administrators can also access customer information and assist with booking inquiries.
3. ***Booking Processing:*** The system facilitates the booking process by allowing customers to search for flights, select preferred options, and make reservations securely.

Functionality Overview:

- **Add Flight:** This feature empowers administrators to seamlessly add new flights to the system. By inputting essential details such as flight number, origin, destination, departure date, number of seats, and price, airlines can efficiently expand their flight offerings and enhance customer accessibility.
- **View Flight :** Users can easily browse through available flights, examine crucial details including flight number, origin, destination
- **Delete Flight:** This functionality enables administrators to remove unwanted or discontinued flights from the system effortlessly. By inputting the flight ID or other identifying details, airlines can efficiently manage their flight schedules, ensuring that only relevant and operational flights are displayed to customers.
- **Add Booking:** With the "Add Booking" feature, customers can effortlessly reserve seats on desired flights. By providing their details along with flight preferences, users can secure their bookings seamlessly, enhancing the overall booking experience and ensuring a hassle-free travel journey.
- **Cancel Booking:** The "Cancel Booking" functionality empowers customers to manage their reservations flexibly. Whether due to change of plans or unforeseen circumstances, users can easily cancel their bookings, freeing up seats for other potential passengers and optimizing flight occupancy.
- **Add Customer:** This feature allows administrators to onboard new customers into the system seamlessly. By capturing essential customer information such as name, contact details, and email address, airlines can efficiently expand their customer base and tailor services to individual preferences.
- **View Customer:** The "View Customer" functionality provides administrators with a comprehensive overview of customer profiles. By accessing customer details such as name, contact information, booking history, and preferences, airlines can personalize services and enhance customer satisfaction.
- **Delete Customer:** This functionality enables administrators to remove redundant or inactive customer profiles from the system effortlessly. By streamlining customer databases, airlines can optimize resource allocation and ensure data integrity within the flight booking system.

Main.java

```
package bcu.cmp5332.bookingsystem.main;

import bcu.cmp5332.bookingsystem.data.FlightBookingSystemData;

public class Main {

    public static void main(String[] args) throws IOException, FlightBookingSystemException {

        FlightBookingSystem fbs = FlightBookingSystemData.Load();

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        System.out.println("Flight Booking System");
        System.out.println("Enter 'help' to see a list of available commands.");
        while (true) {
            System.out.print("> ");
            String line = br.readLine();
            if (line.equals("exit")) {
                break;
            }
            try {
                Command command = CommandParser.parse(line, fbs);
                command.execute(fbs);
            } catch (FlightBookingSystemException ex) {
                System.out.println(ex.getMessage());
            }
        }
        FlightBookingSystemData.store(fbs);
        System.exit(0);
    }
}
```

- **Description:**

This Java code represents the core functionality of a flight booking system application. At its essence, it orchestrates the interaction between users and the flight booking system through a command-line interface. Initially, it loads pre-existing data from storage, likely containing information about available flights, bookings, and other relevant details. Subsequently, it establishes a means for users to input commands via the console, enabling them to interact with the system. Users are presented with a clear prompt, indicating where they can input commands. The system then continuously reads the process and executes corresponding actions within the flight booking system. These actions could involve booking or canceling flights, retrieving flight details, or performing administrative tasks. Exception handling is incorporated to manage any errors that might occur during the execution of commands, ensuring the robustness of the system. The loop continues until the user decides to exit by entering the "exit" command, at which point any modifications to the data are saved back to storage. A javadoc can also be generated in all command classes as it has been integrated

- **Output:**

```
Flight Booking System
Enter 'help' to see a list of available commands.
> help
Commands:
    listflights                print all flights
    listcustomers              print all customers
    addflight                  add a new flight
    addcustomer                add a new customer
    showflight [flight id]     show flight details
    showcustomer [customer id] show customer details
    addbooking [customer id] [flight id] add a new booking
    cancelbooking [customer id] [flight id] cancel a booking
    editbooking [booking id] [flight id] update a booking
    loadgui                    loads the GUI version of the app
    help                      prints this help message
    exit                      exits the program
>
```

Commands

1. Addcustomer

```
package bcu.cmp5332.bookingsystem.commands;

import java.io.BufferedWriter; ...

public class AddCustomer implements Command {

    private final String name;
    private final String phone;
    private final String email;

    public AddCustomer( String name, String phone, String email) {
        this.name = name;
        this.phone = phone;
        this.email = email;
    }

    @Override
    public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
        int maxId = 0;
        if (fbs.getCustomers().size() > 0) {
            int lastIndex = fbs.getCustomers().size() - 1;
            maxId = fbs.getCustomers().get(lastIndex).getId();
        }

        Customer customer = new Customer(++maxId, name, phone, email);
        fbs.addCustomer(customer);
        System.out.println("Customer #" + customer.getId() + " added.");

        // Write customer data to a file
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("resources/data/customers.txt", true))) {
            writer.write(customer.getId() + "," + customer.getName() + "," + customer.getPhone() + "," + customer.getEmail());
            writer.newLine();
        } catch (IOException e) {
            throw new FlightBookingSystemException("Error writing to customers.txt: " + e.getMessage());
        }
    }
}
```

- **Description:**

This Java code defines a command class named `AddCustomer`, which implements the `Command` interface. This class represents a command to add a new customer to the flight booking system. The class has three instance variables representing the customer's name, phone number, and email. The constructor initializes these variables with the provided parameters. The `execute` method is overridden from the `Command` interface and is responsible for adding the customer to the flight booking system. It performs input validation by checking if the customer's name contains any digits, throwing an exception if it does. Then, it retrieves the next available customer ID from the flight booking system, creates a new `Customer` object with the provided details (including the email), adds the customer to the system, and prints a success message indicating the addition of the customer along with their details and assigned ID.

- **Output:**

```
> addcustomer
Customer Name: Binayak Bhattarai
Customer Phone: 9876230987
Customer Email: binayak@gmail.com
Customer #1 added.
> addcustomer
Customer Name: Sarita Poudel
Customer Phone: 987622309
Customer Email: sarita@gmail.com
Customer #2 added.
> addcustomer
Customer Name: Prashamsa Rijal
Customer Phone: 98456732100
Customer Email: prashamsarijal@gmail.com
Customer #3 added.
> addcustomer
Customer Name: Sadikshya Ghimire
Customer Phone: 9821769923
Customer Email: sadikshyaghimire@gmail.com
Customer #4 added.
```

2. ListCustomers

```
package bcu.cmp5332.bookingsystem.commands;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException; ...

public class ListCustomers implements Command {

    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        List<Customer> customers = readCustomersFromFile("resources/data/customers.txt");
        for (Customer customer : customers) {
            System.out.println(customer.getDetailsShort());
        }
        System.out.println(customers.size() + " customer(s)");
    }

    private List<Customer> readCustomersFromFile(String filename) throws FlightBookingSystemException {
        List<Customer> customers = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(",");
                int id = Integer.parseInt(parts[0]);
                String name = parts[1];
                String phone = parts[2];
                String email = parts[3];
                Customer customer = new Customer(id, name, phone, email);
                customers.add(customer);
            }
        } catch (IOException | NumberFormatException e) {
            throw new FlightBookingSystemException("Error reading customers from file: " + e.getMessage());
        }
        return customers;
    }
}
```

- **Description:**

This Java code defines a command class named ListCustomers, which implements the Command interface. The purpose of this class is to execute a command to list all customers stored in the flight booking system. The execute method is overridden from the Command interface and is responsible for retrieving customer data from a file (customers.txt), parsing each line to extract customer information such as ID, name, phone number, and email (if available), and printing this information to the console. The method utilizes a BufferedReader to read data from the file in a line-by-line manner. Then, it extracts the ID, name, and phone number from the parsed parts. If the line contains an email address (determined by the length of the split parts), it extracts and includes it in the output as well. Finally, the method catches any IOException that might occur during file reading and throws a custom FlightBookingSystemException, providing an error message indicating the issue encountered while reading the customer data file. Overall, this command facilitates the listing of customers stored in the flight booking system, aiding in system management and oversight.

- **Output:**

```
> listcustomers
Customer #1 - Binayak Bhattarai - 9876230987 - binayak@gmail.com
Customer #2 - Sarita Poudel - 987622309 - sarita@gmail.com
Customer #3 - Prashamsa Rijal - 98456732100 - prashamsarijal@gmail.com
Customer #4 - Sadikshya Ghimire - 9821769923 - sadikshyaghimire@gmail.com
4 customer(s)
```

3.ShowCustomer

```
1 package bcu.cmp5332.bookingsystem.commands;
2
30 import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;
11
12 public class ShowCustomer implements Command {
13
14     private final int customerId;
15
16     public ShowCustomer(int customerId) {
17         this.customerId = customerId;
18     }
19
20     @Override
21     public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
22         Customer customer = fbs.getCustomerByID(customerId);
23         if (customer == null) {
24             throw new FlightBookingSystemException("Customer with ID " + customerId + " not found.");
25         }
26
27         System.out.println("Customer ID: " + customer.getId());
28         System.out.println("Name: " + customer.getName());
29         System.out.println("Phone: " + customer.getPhone());
30         System.out.println("Email: " + customer.getEmail());
31
32         List<Booking> bookings = customer.getBookings();
33         if (bookings.isEmpty()) {
34             System.out.println("This customer has not made any bookings.");
35         } else {
36             System.out.println("Bookings:");
37             for (Booking booking : bookings) {
38                 Flight flight = booking.getFlight();
39                 System.out.println("Booking ID: " + booking.getId());
40                 System.out.println("Flight Number: " + flight.getFlightNumber());
41                 System.out.println("Origin: " + flight.getOrigin());
42                 System.out.println("Destination: " + flight.getDestination());
43                 System.out.println("Date: " + flight.getDepartureDate().format(DateTimeFormatter.ofPattern("yyyy-MM-dd")));
44                 System.out.println("Price: " + booking.getPrice());
45                 System.out.println();
46             }
47         }
48     }
49 }
50
```

- **Description:**

This Java class represents a command called 'ShowCustomer', which is a part of a flight booking system application. The purpose of this command is to display detailed information about a specific customer, including their ID, name, contact details, and any bookings they have made. If the customer has made bookings, the command lists each booking along with details such as flight number, origin, destination, date, and price. If the customer has not made any bookings, it indicates so in the output. This command helps users to view customer details and their associated bookings within the system.

- **Output:**

```
> showcustomer 2
Customer ID: 2
Name: Sita Pandey
Phone: 987654321
Email: sitapandey@gmail.com
Bookings:
Booking ID: 5
Flight Number: MNO345
Origin: Sydney
Destination: Melbourne
Date: 2024-03-15
Price: 20200.0
```

4) ShowFlight

```
1 package bcu.cmp5332.bookingsystem.commands;
2
3 import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;
4
5 public class ShowFlight implements Command {
6
7     private final int flightId;
8
9     public ShowFlight(int flightId) {
10         this.flightId = flightId;
11     }
12
13     @Override
14     public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
15         Flight flight = fbs.getFlightById(flightId);
16         if (flight == null) {
17             throw new FlightBookingSystemException("Flight with ID " + flightId + " not found.");
18         }
19
20         System.out.println("Flight Number: " + flight.getFlightNumber());
21         System.out.println("Origin: " + flight.getOrigin());
22         System.out.println("Destination: " + flight.getDestination());
23         System.out.println("Departure Date: " + flight.getDepartureDate());
24         System.out.println("Number of Seats: " + flight.getNumberOfSeats());
25         System.out.println("Price: " + flight.getPrice());
26
27         List<Customer> passengers = flight.getPassengers();
28         if (passengers.isEmpty()) {
29             System.out.println("No passengers booked for this flight.");
30         } else {
31             System.out.println("Passengers:");
32             for (Customer passenger : passengers) {
33                 System.out.println("Name: " + passenger.getName());
34                 System.out.println("Phone Number: " + passenger.getPhone());
35                 System.out.println();
36             }
37         }
38     }
39 }
40 }
```

- **Description:**

This Java class represents a command called 'ShowFlight', which is part of a flight booking system application. The purpose of this command is to display detailed information about a specific flight, including its ID, origin, destination, departure date, number of seats, and price. Additionally, it lists the passengers booked for the flight, including their names and phone numbers. If there are no passengers booked for the flight, it indicates so in the output. This command helps users to view flight details and the passengers associated with each flight within the system.

- **Output:**

```
> showflight 5
Flight Number: MNO345
Origin: Sydney
Destination: Melbourne
Departure Date: 2024-03-15
Number of Seats: 2
Price: 10000.0
Passengers:
Name: Sita Pandey
Phone Number: 987654321

Name: Ram Sharma
Phone Number: 123456789
```

5) AddBooking

```
package bcu.cmp5332.bookingsystem.commands;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

public class AddBooking implements Command {

    private final int customerId;
    private final int flightId;

    public AddBooking(int customerId, int flightId, LocalDate localDate) {
        this.customerId = customerId;
        this.flightId = flightId;
    }

    @Override
    public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
        LocalDate today = LocalDate.now();
        LocalDate twoYearsFromToday = today.plusYears(2);

        if (today.isAfter(twoYearsFromToday)) {
            throw new FlightBookingSystemException("Bookings more than 2 years in advance are not allowed.");
        }

        Customer customer = fbs.getCustomerByID(customerId);
        if (customer == null) {
            throw new FlightBookingSystemException("Customer with ID " + customerId + " not found.");
        }

        Flight flight = fbs.getFlightByID(flightId);
        if (flight == null) {
            throw new FlightBookingSystemException("Flight with ID " + flightId + " not found.");
        }

        // Check if the flight has available seats
        if (flight.getPassengers().size() >= flight.getNumberOfSeats()) {
            throw new FlightBookingSystemException("The flight is full. Booking cannot be made.");
        }

        // Calculate the price for the booking
        int price = flight.calculatePrice(today);

        int bookingId = fbs.generateBookingId();
        LocalDate bookingDate = LocalDate.now();
        Booking booking = new Booking(bookingId, customer, flight, bookingDate, price);
        customer.addBooking(booking);
        flight.addPassenger(customer);

        // Write booking data to the file
        if (!booking.isCancelled()) {
            try (BufferedWriter writer = new BufferedWriter(new FileWriter("resources/data/bookings.txt", true))) {
                writer.write(booking.getId() + "," + customer.getId() + "," + flight.getId() + "," + booking.getBookingDate() + "," + booking.getPrice());
                writer.newLine();
            } catch (IOException e) {
                throw new FlightBookingSystemException("Error writing to bookings.txt: " + e.getMessage());
            }
        }

        System.out.println("Booking was issued successfully to the customer.");
    }
}
```

- **Description:**

The 'AddBooking' class is a command in a flight booking system that enables the addition of bookings for customers on specific flights. It ensures that bookings are made within two years from the current date and checks for seat availability on the specified flight. Upon validation, it calculates the booking price, generates a unique booking ID, creates a new booking object, associates it with the customer and flight, and updates their respective records. Additionally, it writes the booking details to a file for persistence. This command streamlines the process of adding bookings to the system while maintaining data integrity and providing feedback on successful booking issuance.

Output:

```
> addbooking 2 1
Customer ID: 2
Flight ID: 1
Booking was issued successfully to the customer.
>
```

6) AddFlight

```
1 package bcu.cmp5332.bookingsystem.commands;
2
3 import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;
4
5 public class AddFlight implements Command {
6
7     private final String flightNumber;
8     private final String origin;
9     private final String destination;
10    private final LocalDate departureDate;
11    private final int numberOfSeats;
12    private final double price;
13
14    public AddFlight(String flightNumber, String origin, String destination, LocalDate departureDate, int numberOfSeats, double price) {
15        this.flightNumber = flightNumber;
16        this.origin = origin;
17        this.destination = destination;
18        this.departureDate = departureDate;
19        this.numberOfSeats = numberOfSeats;
20        this.price = price;
21    }
22
23    @Override
24    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
25        LocalDate currentDate = LocalDate.now();
26        if (departureDate.isBefore(currentDate)) {
27            throw new FlightBookingSystemException("Cannot add flight with a departure date in the past.");
28        }
29
30        int maxId = flightBookingSystem.getFlights().stream().mapToInt(Flight::getId).max().orElse(0);
31
32        Flight flight = new Flight(++maxId, flightNumber, origin, destination, departureDate, numberOfSeats, price);
33        flightBookingSystem.addFlight(flight);
34        System.out.println("Flight #" + flight.getId() + " added.");
35
36        // Write the new flight data to the flights.txt file
37        try (BufferedWriter writer = new BufferedWriter(new FileWriter("resources/data/flights.txt", true))) {
38            DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd");
39            String formattedDate = departureDate.format(dtf);
40            writer.write(flight.getId() + "," + flight.getFlightNumber() + "," + flight.getOrigin()
41                + "," + flight.getDestination() + "," + formattedDate + "," + flight.getNumberOfSeats() + "," + flight.getPrice());
42            writer.newLine();
43        } catch (IOException ex) {
44            throw new FlightBookingSystemException("Error writing to flights.txt: " + ex.getMessage());
45        }
46    }
47
48 }
```

- **Description :**

The 'AddFlight' class represents a command in a flight booking system application that allows the addition of new flights to the system. It takes parameters such as flight number, origin, destination, departure date, number of seats, and price to create a new flight object. The command validates that the departure date of the flight is not in the past before adding the flight. Upon successful addition, it updates the flight booking system with the new flight and prints a confirmation message. Additionally, it writes the details of the new flight to a file named 'flights.txt' for persistence, ensuring that the flight data is stored for future reference. This command facilitates the management of flights within the booking system by providing a mechanism to add new flights with relevant information.

- **Output:**

```
> addflight
Flight Number: 23
Origin: Kathamandu
Destination: Biratnagar
Departure Date ("YYYY-MM-DD" format): 2024-02-05
Number of Seats: 20
Price: 4500
Flight #1 added.
> addflight
Flight Number: 34
Origin: Texas
Destination: New York
Departure Date ("YYYY-MM-DD" format): 2024-03-01
Number of Seats: 10
Price: 50000
Flight #2 added.
```

7) ListFlights

```
package bcu.cmp5332.bookingssystem.commands;

import bcu.cmp5332.bookingssystem.main.FlightBookingSystemException;

public class ListFlights implements Command {

    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        List<Flight> flights = readFlightsFromFile("resources/data/flights.txt");
        LocalDate today = LocalDate.now();
        flights = filterFlights(flights, today);
        for (Flight flight : flights) {
            System.out.println(flight.getDetailsShort());
        }
        System.out.println(flights.size() + " flight(s)");
    }

    private List<Flight> readFlightsFromFile(String filename) throws FlightBookingSystemException {
        List<Flight> flights = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(",");
                int id = Integer.parseInt(parts[0]);
                String flightNumber = parts[1];
                String origin = parts[2];
                String destination = parts[3];
                LocalDate departureDate = LocalDate.parse(parts[4]);
                int numberOfSeats = Integer.parseInt(parts[5]);
                double price = Double.parseDouble(parts[6]);

                Flight flight = new Flight(id, flightNumber, origin, destination, departureDate, numberOfSeats, price);
                flights.add(flight);
            }
        } catch (IOException | NumberFormatException e) {
            throw new FlightBookingSystemException("Error reading flights from file: " + e.getMessage());
        }
        return flights;
    }

    private List<Flight> filterFlights(List<Flight> flights, LocalDate today) {
        return flights.stream()
            .filter(flight -> flight.getDepartureDate().isAfter(today))
            .collect(Collectors.toList());
    }
}
```

- **Description:**

The 'ListFlights' class implements a command in a flight booking system that lists all available flights. Upon execution, it reads flight information from a file named 'flights.txt', parsing each line to create 'Flight' objects. These flights are then filtered based on whether their departure date is after the current date, ensuring that only future flights are included in the list. After filtering, the details of each flight, obtained using 'getDetailsShort()' method, are printed to the console along with the total count of flights. If any error occurs during file reading or parsing, a 'FlightBookingSystemException' is thrown with an appropriate error message. This command provides users with an overview of all upcoming flights in the system, aiding in flight selection and management within the booking system.

- Output:

```

Enter help to see a list of available commands.
> listflights
Flight #3 - 45 - UK to Nepal on 04/04/2024 - Price: $80000.0 - Seats: 60
Flight #6 - 34 - Kathmandu to Biratnagar on 09/09/2024 - Price: $2500.0 - Seats: 3
Flight #7 - G34 - Kathmandu to Biratnagar on 06/05/2024 - Price: $7000.0 - Seats: 4
3 flight(s)
>

```

8) CancelBooking

```

package bcu.cmp5332.bookingsystem.commands;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

public class CancelBooking implements Command {

    private final int customerId;
    private final int flightId;

    public CancelBooking(int customerId, int flightId) {
        this.customerId = customerId;
        this.flightId = flightId;
    }

    @Override
    public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
        Customer customer = fbs.getCustomerByID(customerId);
        if (customer == null) {
            throw new FlightBookingSystemException("Customer not found for ID: " + customerId);
        }

        Flight flight = fbs.getFlightByID(flightId);
        if (flight == null) {
            throw new FlightBookingSystemException("Flight not found for ID: " + flightId);
        }

        Booking booking = null;
        for (Booking b : customer.getBookings()) {
            if (b.getFlight().getId() == flightId) {
                booking = b;
                break;
            }
        }

        if (booking == null) {
            throw new FlightBookingSystemException("No booking found for customer ID: " + customerId + " and flight ID: " + flightId);
        }

        booking.cancelBooking();

        BookingDataManager dataManager = new BookingDataManager();
        try {
            dataManager.storeData(fbs);
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.println("Booking successfully canceled for customer ID: " + customerId + " and flight ID: " + flightId);
    }
}

```

- **Description:**

The CancelBooking class implements the Command interface to execute a command canceling a booking for a specific customer on a particular flight within the flight booking system. It takes parameters including the customer and flight IDs. The execute method first retrieves the customer and flight objects based on the provided IDs. If either the customer or flight is not found, it throws a FlightBookingSystemException. It then attempts to find the booking associated with the customer and flight. If the booking is not found, it throws an exception as well. Otherwise, it removes the booking from the customer's booking list and removes the customer from the flight's passenger list. Additionally, it updates the booking data file by removing the canceled booking entry. Exception handling is included for potential IO errors during file operations. Overall, this command facilitates the cancellation of bookings, allowing for efficient.

- **Output:**

```
> cancelbooking 4 2
Customer ID: 4
Flight ID: 2
Booking successfully canceled for customer
ID: 4 and flight ID: 2
```

Data of txt file

All the information are extracted from their respective classes and stored in text files accordingly

1) Bookings.txt:

```
3,2,4,2024-03-11,19050.0,cancelled
1,3,2,2024-03-11,16000.0,cancelled
2,4,2,2024-03-11,16000.0
4,1,5,2024-03-11,20150.0
5,2,5,2024-03-11,20200.0
```

2) Customers.txt

```
1,Ram Sharma,123456789,ramsharma@gmail.com
2,Sita Pandey,987654321,sitapandey@gmail.com
3,Gopal Rajkot,555555555,gopalrajakot@gmail.com
4,Amit Kafle,777777777,amitkafe@gmail.com
5,Rajeshwari Rajput,888888888,rajeshwarirajput@gmail.com
6,Prashamsa Rijal,9876309843,prashamsarijal@gmail.com
7,Sita Rijal,9883246253,sita@gmail.com
```

3) Flights.txt

```
1,ABC123,London,Paris,2024-03-11,10,7500.0
2,DEF456,Paris,London,2024-03-12,8,8000.0
3,GHI789,New York,Los Angeles,2024-03-13,6,9000.0
4,JKL012,Tokyo,Beijing,2024-03-14,4,9500.0
5,MNO345,Sydney,Melbourne,2024-03-15,2,10000.0
```

Model

1. **Flight.java**

The `Flight` class is the backbone of the flight booking system, representing individual flights with properties like flight number, origin, destination, departure date, number of seats, and price. It facilitates essential functionalities such as adding and removing passengers, managing bookings, and calculating booking prices based on various factors like the number of days until departure and available seats. The class ensures data integrity by encapsulating its properties and providing methods to access and modify them appropriately. Overall, the `Flight` class serves as a central component in the system, enabling efficient management of flights and bookings while maintaining consistency and reliability in flight-related operations.

2. **Customer.java**

The `Customer` class represents a customer within the flight booking system, storing information such as their unique ID, name, phone number, email address, and a list of bookings they have made. The class provides methods to access and modify these properties, including getters and setters for customer details and methods to manage bookings, such as adding, removing, and retrieving bookings. Additionally, the class includes functionality to determine whether a customer has been deleted from the system and if any of their bookings have been cancelled. Overall, the `Customer` class serves as a fundamental component of the booking system, enabling the storage and management of customer information and their associated bookings.

3. **Booking.java**

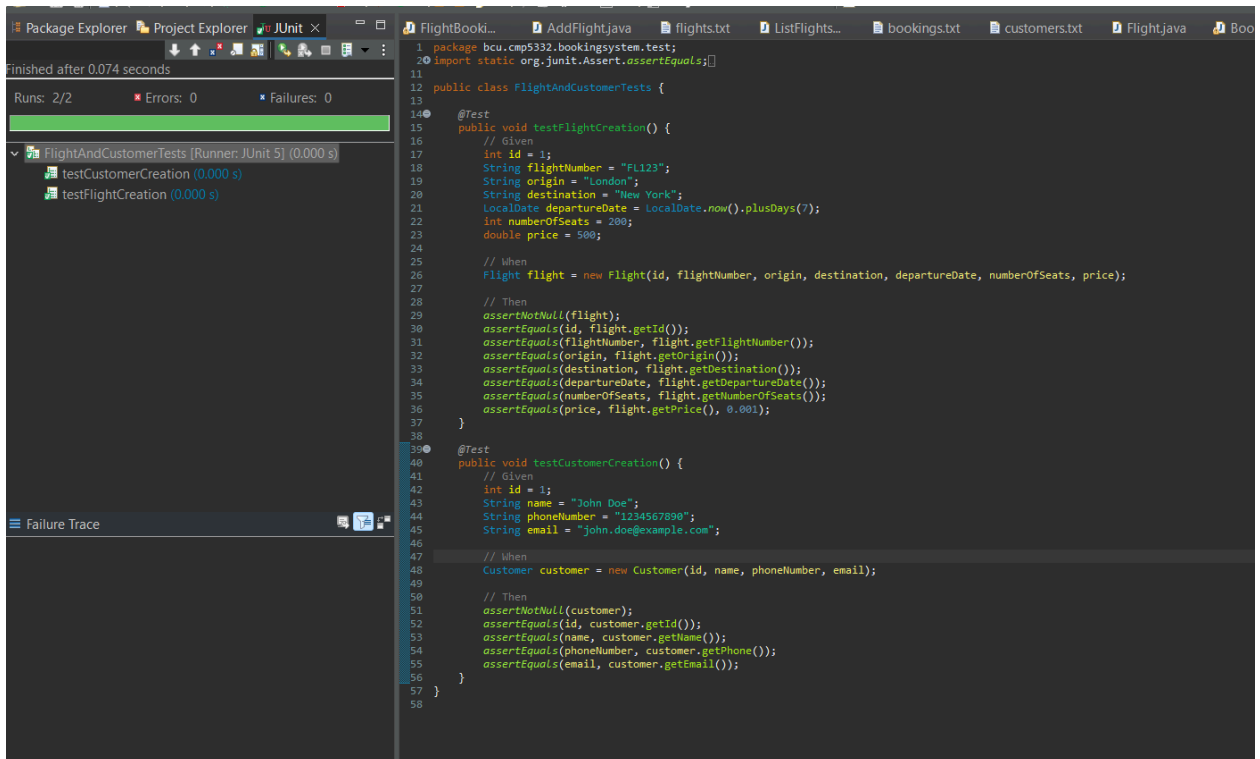
The `Booking` class represents a booking made by a customer for a specific flight within the flight booking system. It contains properties such as an ID, customer, flight, price, booking date, and a flag indicating whether the booking has been cancelled. This class encapsulates essential functionalities for managing bookings, including getters and setters for accessing and modifying booking properties. Each booking is associated with a customer and a flight, providing a link between the customer and the booked flight. The booking date indicates when the booking was made, while the price represents the cost of the booking. Additionally, the class includes methods to check whether a booking has been cancelled and to update its cancellation status. Overall, the `Booking` class plays a crucial role in the flight booking system by representing individual bookings and facilitating their management and tracking throughout the system.

4. Flight Booking System

The `FlightBookingSystem` class serves as the central component of the flight booking system, managing flights, customers, and bookings. It maintains collections of flights, customers, and bookings using maps and provides methods for adding, retrieving, and deleting these entities. Additionally, the class offers functionalities such as generating unique booking IDs, getting flights, customers, and bookings by their respective IDs, and retrieving bookings associated with specific customers or flights. It also includes methods for deleting flights and customers from the system along with their associated bookings. Furthermore, it contains utility methods for retrieving bookings by customer and flight IDs, as well as for removing bookings. Overall, the `FlightBookingSystem` class encapsulates the core functionality of the flight booking system, facilitating the management and interaction of flights, customers, and bookings.

JUnit Testing:

The `FlightAndCustomerTests` class contains JUnit tests to verify the creation of `Flight` and `Customer` objects in the flight booking system. In the `testFlightCreation` method, it creates a new `Flight` object with given parameters such as ID, flight number, origin, destination, departure date, number of seats, and price. Then, it asserts that the created `Flight` object is not null and that its attributes match the provided values. Similarly, in the `testCustomerCreation` method, it creates a new `Customer` object with given parameters such as ID, name, phone number, and email. It then asserts that the created `Customer` object is not null and that its attributes match the provided values. These tests ensure that the constructors of the `Flight` and `Customer` classes correctly initialize objects with the specified attributes.



The screenshot displays an IDE with the JUnit test runner interface on the left and the source code of the `FlightAndCustomerTests` class on the right.

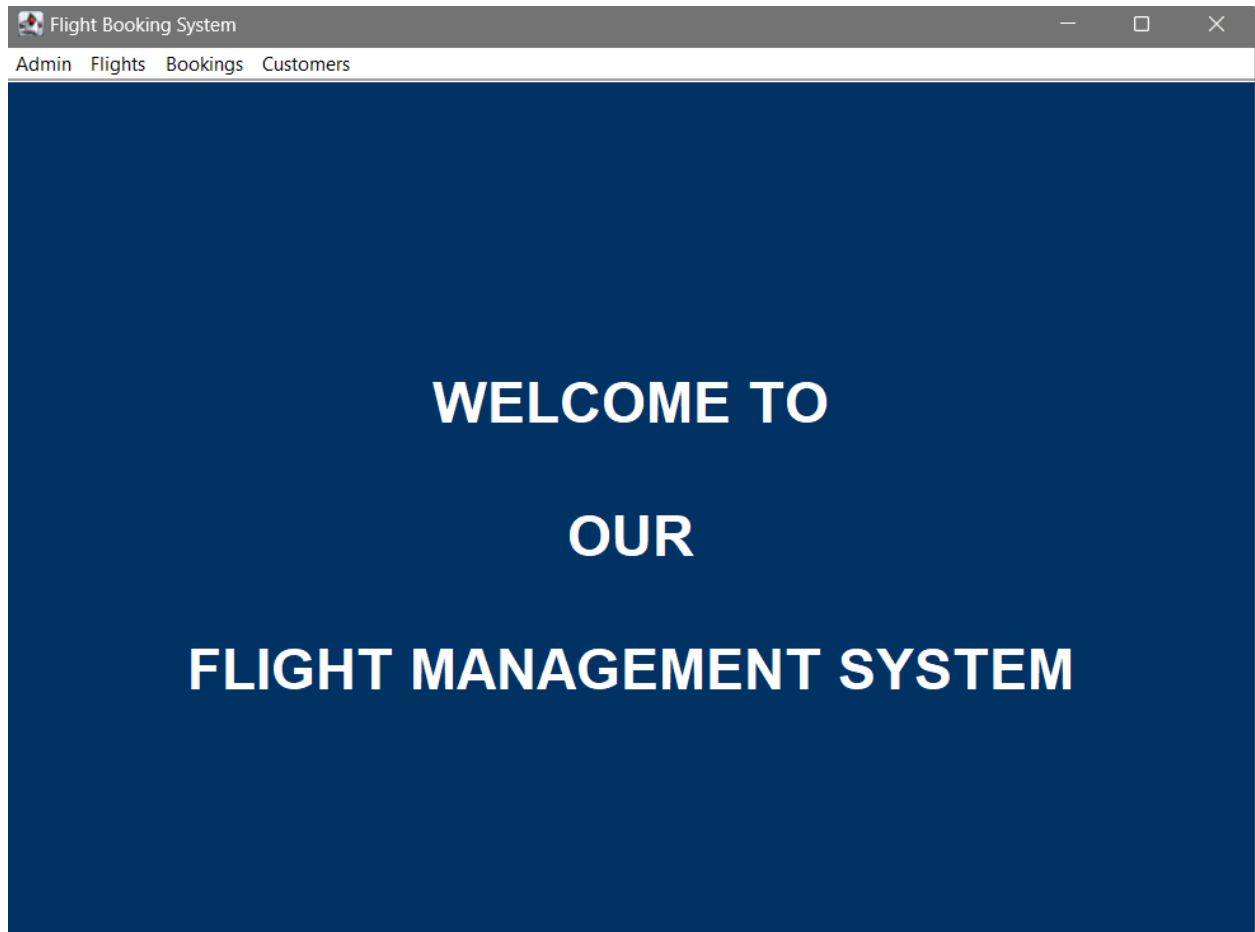
JUnit Test Runner Interface (Left):

- Package Explorer: Shows the project structure.
- Project Explorer: Shows the project structure.
- JUnit: Shows the test results.
- Test Results: Shows the test results for the `FlightAndCustomerTests` class.
- Test Results Summary: Shows the test results summary.
- Test Results Details: Shows the test results details.
- Test Results Table: Shows the test results table.
- Test Results Log: Shows the test results log.

Source Code (Right):

```
1 package bcu.cmp5332.bookingssystem.test;
2 import static org.junit.Assert.assertEquals;
3
4 public class FlightAndCustomerTests {
5
6     @Test
7     public void testFlightCreation() {
8         // Given
9         int id = 1;
10        String flightNumber = "FL123";
11        String origin = "London";
12        String destination = "New York";
13        LocalDate departureDate = LocalDate.now().plusDays(7);
14        int numberOfSeats = 200;
15        double price = 500;
16
17        // When
18        Flight flight = new Flight(id, flightNumber, origin, destination, departureDate, numberOfSeats, price);
19
20        // Then
21        assertNotNull(flight);
22        assertEquals(id, flight.getId());
23        assertEquals(flightNumber, flight.getFlightNumber());
24        assertEquals(origin, flight.getOrigin());
25        assertEquals(destination, flight.getDestination());
26        assertEquals(departureDate, flight.getDepartureDate());
27        assertEquals(numberOfSeats, flight.getNumberOfSeats());
28        assertEquals(price, flight.getPrice(), 0.001);
29    }
30
31    @Test
32    public void testCustomerCreation() {
33        // Given
34        int id = 1;
35        String name = "John Doe";
36        String phoneNumber = "1234567890";
37        String email = "john.doe@example.com";
38
39        // When
40        Customer customer = new Customer(id, name, phoneNumber, email);
41
42        // Then
43        assertNotNull(customer);
44        assertEquals(id, customer.getId());
45        assertEquals(name, customer.getName());
46        assertEquals(phoneNumber, customer.getPhone());
47        assertEquals(email, customer.getEmail());
48    }
49 }
```

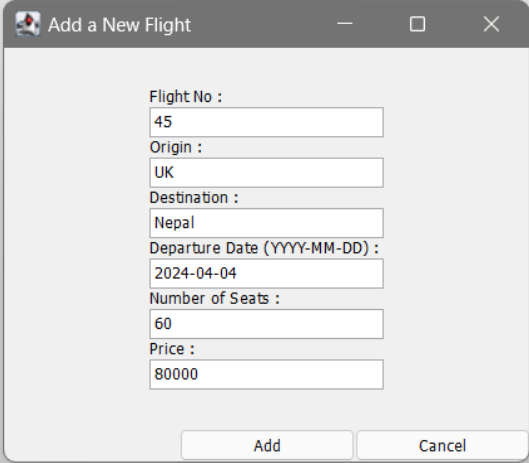
Graphical User Interface (GUI) :



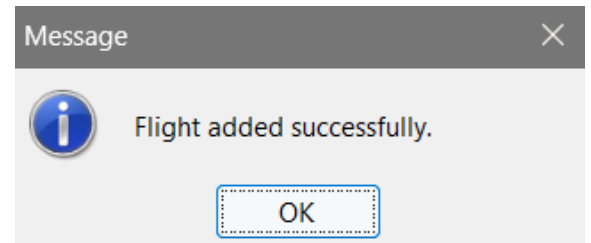
This is the first interface of our GUI. Users can go to this UI and then go to various sections such as Admins, Flights, Bookings and Customers. They can book flights, add, and cancel according to their choices.

1) Flight

- Add

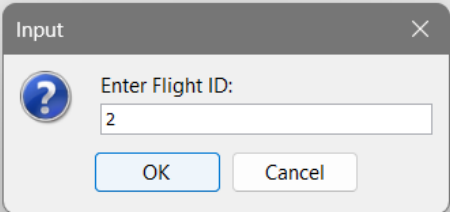


A dialog box titled "Add a New Flight" with a close button (X) in the top right corner. It contains several input fields for flight details: Flight No (45), Origin (UK), Destination (Nepal), Departure Date (YYYY-MM-DD) (2024-04-04), Number of Seats (60), and Price (80000). At the bottom, there are "Add" and "Cancel" buttons.

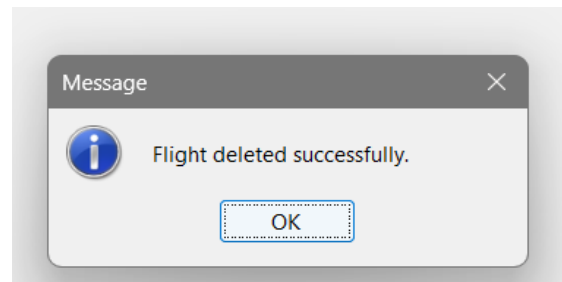


- Delete

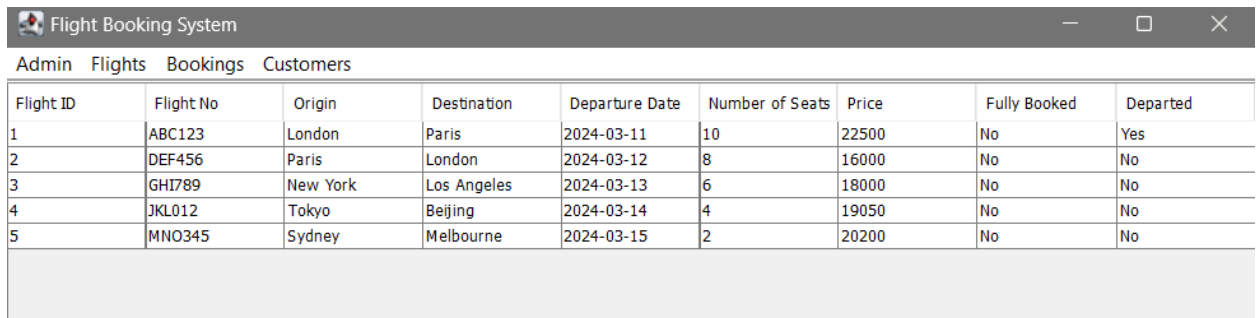
Deleting the flight will only delete the details of flight from the view but won't delete from the 'flight.txt' file.



An input dialog box titled "Input" with a close button (X) in the top right corner. It features a question mark icon (?) and the text "Enter Flight ID:". Below the text is an input field containing the number "2". At the bottom, there are "OK" and "Cancel" buttons.



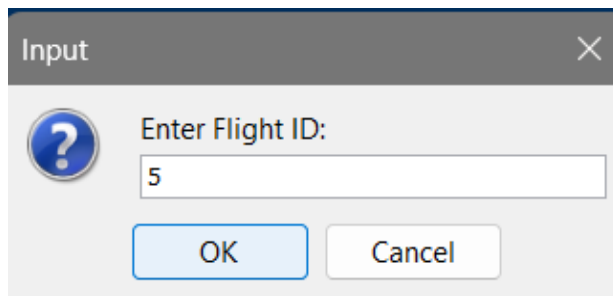
- View



Flight ID	Flight No	Origin	Destination	Departure Date	Number of Seats	Price	Fully Booked	Departed
1	ABC123	London	Paris	2024-03-11	10	22500	No	Yes
2	DEF456	Paris	London	2024-03-12	8	16000	No	No
3	GHI789	New York	Los Angeles	2024-03-13	6	18000	No	No
4	JKL012	Tokyo	Beijing	2024-03-14	4	19050	No	No
5	MNO345	Sydney	Melbourne	2024-03-15	2	20200	No	No

- Passenger List

When we enter a certain flight number after clicking in the passenger list sub menu the list of passengers of that particular flight will be displayed to us.

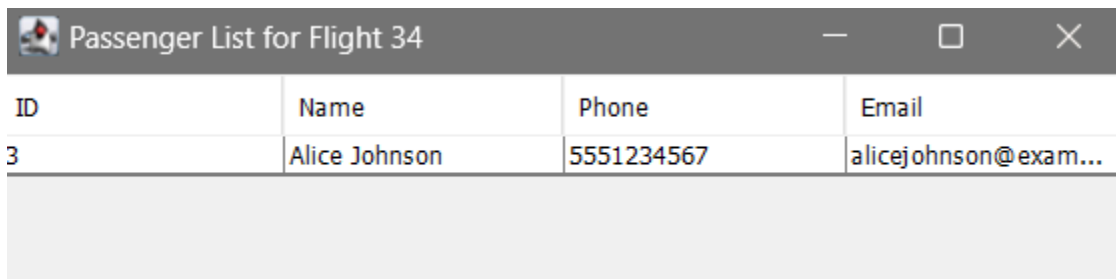


Input

Enter Flight ID:

5

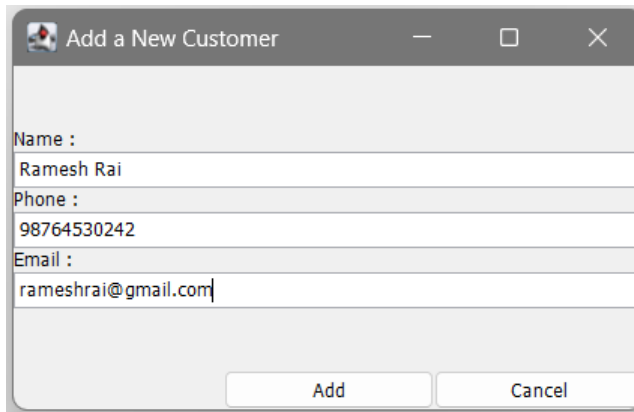
OK Cancel



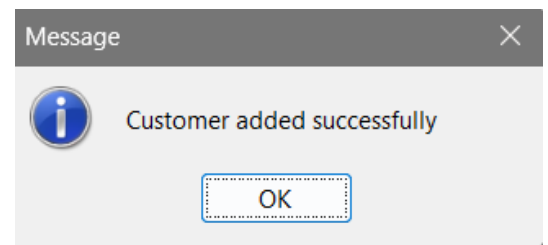
ID	Name	Phone	Email
3	Alice Johnson	5551234567	alicejohnson@exam...

2) Customer

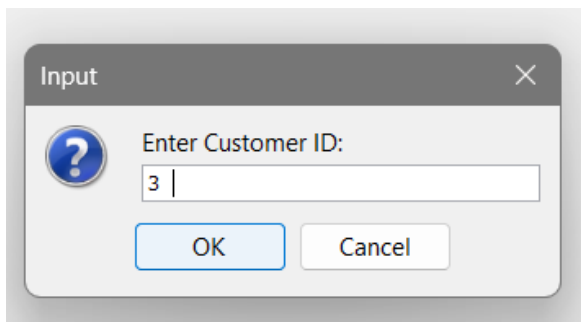
- Add



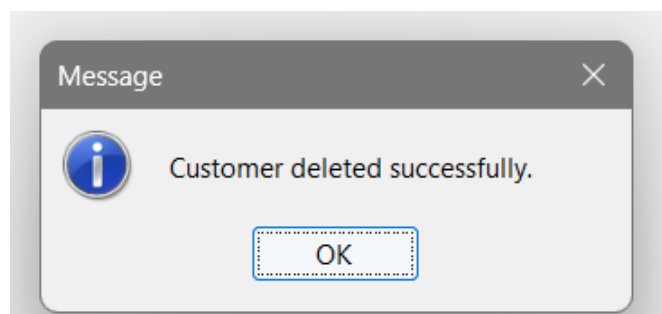
A dialog box titled "Add a New Customer" with a close button (X) in the top right corner. It contains four input fields: "Name :" with the text "Ramesh Rai", "Phone :" with the text "98764530242", and "Email :" with the text "rameshrai@gmail.com". At the bottom, there are two buttons: "Add" and "Cancel".



- Delete



An input dialog box titled "Input" with a close button (X) in the top right corner. It features a question mark icon (?) on the left and the text "Enter Customer ID:" in the center. Below the text is a text input field containing the number "3". At the bottom, there are two buttons: "OK" and "Cancel".



- **View**

Once we click on any customer we can see the bookings that they had made till date.

Flight Booking System				
Admin Flights Bookings Customers				
ID	Name	Phone	Email	Number of Active Bookings
1	Ram Sharma	123456789	ramsharma@gmail.com	1
2	Sita Pandey	987654321	sitapandey@gmail.com	1
3	Gopal Rajkot	555555555	gopalrajakot@gmail.com	0
4	Amit Kafle	777777777	amitkafe@gmail.com	0
5	Rajeshwari Rajput	888888888	rajeshwarirajput@gmail.com	0
6	Prashamsa Rijal	9876309843	prashamsarijal@gmail.com	0
7	Sita Rijal	9883246253	sita@gmail.com	0

For example, this is the info displayed for one of the passengers above.

Booking Details					
Booking ID	Flight	Booking Date	Price	Cancellation Fee	Rebook Fee
3	JKL012	2024-03-11	19050.0	1905.0	0.0
5	MNO345	2024-03-11	20200.0	0.0	0.0

3) Booking

- **Add**

Add Booking

Customer ID :

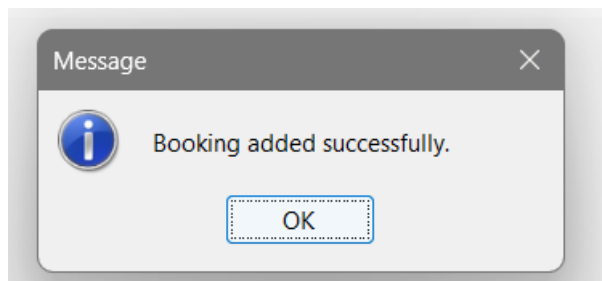
2

Flight ID :

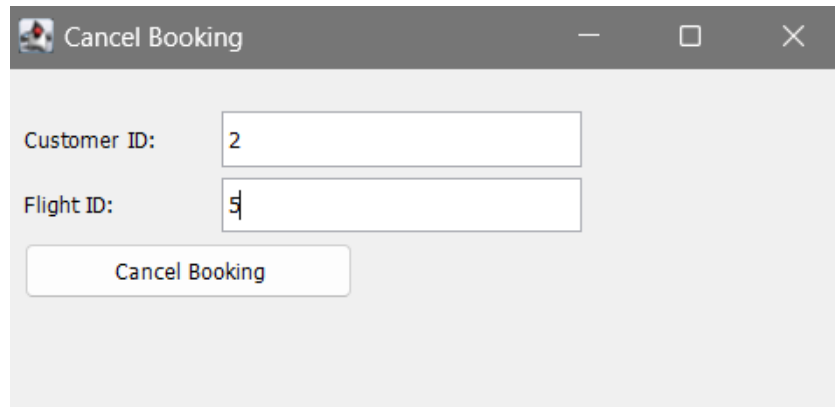
1

Add

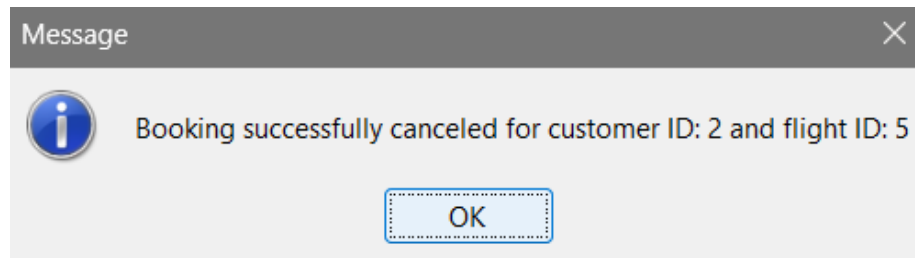
Cancel



- **Cancel**



A dialog box titled "Cancel Booking" with a close button (X) in the top right corner. It contains two input fields: "Customer ID:" with the value "2" and "Flight ID:" with the value "5". Below the input fields is a button labeled "Cancel Booking".



A message dialog box titled "Message" with a close button (X) in the top right corner. It features an information icon (i) on the left and the text "Booking successfully canceled for customer ID: 2 and flight ID: 5". At the bottom center is an "OK" button.

- **View All**

When we click on the view all sub menu of the Booking menu all the data from the bookings.txt is displayed to us.

Booking ID	Customer Name	Flight Number	Booking Date	Price	Status
4	Customer 1	Flight 5	2024-03-11	20150.0	Active
3	Customer 2	Flight 4	2024-03-11	19050.0	Cancelled
5	Customer 2	Flight 5	2024-03-11	20200.0	Cancelled
1	Customer 3	Flight 2	2024-03-11	16000.0	Cancelled
2	Customer 4	Flight 2	2024-03-11	16000.0	Cancelled

Java Documentation:

The JavaDocs for our Flight Booking System provide comprehensive documentation of the system's functionality, architecture, and usage. Each class, method, and variable is meticulously documented, offering clear explanations of their purpose, parameters, return types, and possible exceptions. The documentation provides valuable insights into the inner workings of the system, enabling developers to understand and utilize the codebase effectively. Additionally, the JavaDocs include detailed explanations of the command-line interface, outlining available commands and their respective functionalities. This documentation serves as a vital resource for developers, facilitating the development, maintenance, and troubleshooting of the Flight Booking System. With its clear and concise explanations, the JavaDocs ensure transparency and ease of comprehension, empowering developers to build upon and enhance the system with confidence.

Conclusion :

In summary, the flight booking system provides a user-friendly platform for managing flight reservations efficiently. By offering a range of functionalities such as adding, listing, and canceling flights, along with booking management for customers, the system caters to diverse user needs. Each command within the system is meticulously crafted to execute specific actions with ease, ensuring a seamless user experience. Furthermore, the system's robust error handling mechanism enhances its reliability and stability, safeguarding against potential disruptions. Leveraging file input/output operations contributes to the system's data persistence and integrity, enhancing its overall reliability. With its intuitive interface and comprehensive features, the flight booking system serves as a valuable tool for both customers and administrators in the airline industry, simplifying the reservation process and streamlining operations effectively.