| Experiment No. 12 |
| --- |
| To implement interfaces in object-oriented programming |
| Date of Performance:12/09/24 |
| Date of Submission:19/09/24 |
| KARAN PAWAR COMPS SE-2 |

**Aim:-** To implement interfaces in object-oriented programming

**Objective:-** Develop a Java application that demonstrates the implementation of interfaces in object-oriented programming within a travel booking system. Create two interfaces, Bookable with a method book and Cancellable with a method cancel. Then, create classes Flight and Hotel that implement both interfaces, providing concrete implementations for these methods and adding additional attributes like flightNumber, hotelName, bookingStatus, and cancellationPolicy. The program should instantiate objects of both classes, set values for their attributes, invoke the interface methods, and display the results to illustrate the concept and benefits of using interfaces for defining contracts in a travel booking context.

**Theory:-** an interface is a reference type, similar to a class, that is used to specify a set of abstract methods that any class implementing the interface must provide. Interfaces provide a way to achieve abstraction and multiple inheritance in Java.

Key Concepts:

Definition and Declaration:

An interface is declared using the interface keyword.

Methods in an interface are implicitly public and abstract. This means that they must be implemented by any class that implements the interface.

Fields in an interface are implicitly public, static, and final, meaning they are constants that cannot be changed.

Purpose and Usage:

Interfaces are used to define a contract for what a class can do, without specifying how it does it.

They are ideal for defining common behaviors that can be shared across different classes.

Implementing Interfaces:

A class implements an interface using the implements keyword.

The class must provide concrete implementations for all abstract methods defined in the interface.

A class can implement multiple interfaces, allowing for multiple inheritance of method signatures.

Default Methods:

Introduced in Java 8, default methods allow interfaces to provide concrete methods with a default implementation.

This feature enables interfaces to evolve without breaking existing implementations.

Static Methods:

Interfaces can also include static methods, which belong to the interface itself rather than any instance.

Static methods can provide utility functions related to the interface.

Benefits of Interfaces:

Abstraction: Interfaces provide a way to define what methods a class should have, without dictating how those methods should be implemented.

Loose Coupling: By programming to an interface rather than a specific implementation, code becomes more flexible and easier to maintain.

Multiple Inheritance: Interfaces allow a class to inherit method signatures from multiple sources, promoting code reuse and modularity.

Real-World Application:

Interfaces are commonly used in large systems to define the roles that various components play.

For example, in a travel booking system, interfaces like Bookable and Cancellable can define common behaviors for different entities like flights and hotels.

**Code:**

interface Bookable {

void book();

}

interface Cancellable {

void cancel();

}

class Flight implements Bookable, Cancellable {

private String flightNumber;

private String bookingStatus;

private String cancellationPolicy;

public Flight(String flightNumber, String cancellationPolicy) {

this.flightNumber = flightNumber;

this.cancellationPolicy = cancellationPolicy;

this.bookingStatus = "Not Booked";

}

@Override

public void book() {

this.bookingStatus = "Booked";

System.out.println("Flight " + flightNumber + " has been booked.");

}

@Override

public void cancel() {

if ("Booked".equals(bookingStatus)) {

this.bookingStatus = "Cancelled";

System.out.println("Flight " + flightNumber + " has been cancelled. Cancellation Policy: " + cancellationPolicy);

} else {

System.out.println("Flight " + flightNumber + " cannot be cancelled as it is not booked.");

}

}

public String getBookingStatus() {

return bookingStatus;

}

}

class Hotel implements Bookable, Cancellable {

private String hotelName;

private String bookingStatus;

private String cancellationPolicy;

public Hotel(String hotelName, String cancellationPolicy) {

this.hotelName = hotelName;

this.cancellationPolicy = cancellationPolicy;

this.bookingStatus = "Not Booked";

}

@Override

public void book() {

this.bookingStatus = "Booked";

System.out.println("Hotel " + hotelName + " has been booked.");

}

@Override

public void cancel() {

if ("Booked".equals(bookingStatus)) {

this.bookingStatus = "Cancelled";

System.out.println("Hotel " + hotelName + " has been cancelled. Cancellation Policy: " + cancellationPolicy);

} else {

System.out.println("Hotel " + hotelName + " cannot be cancelled as it is not booked.");

}

}

public String getBookingStatus() {

return bookingStatus;

}

}

public class TravelBookingSystem {

public static void main(String[] args) {

Flight flight = new Flight("AI202", "24-hour notice required for cancellation.");

Hotel hotel = new Hotel("Seaside Resort", "No refund after check-in.");

flight.book();

flight.cancel();

System.out.println("Current Flight Booking Status: " + flight.getBookingStatus());

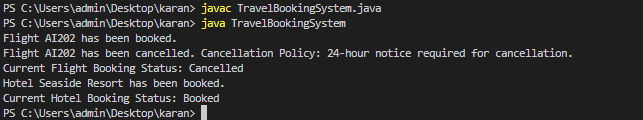
hotel.book();

System.out.println("Current Hotel Booking Status: " + hotel.getBookingStatus());

}

}

**Output:**

****

**Conclusion:**

Interfaces in OOP provide abstraction, enable polymorphism, and promote decoupling. They allow multiple inheritance, standardize behaviors across classes, enhance flexibility and scalability, and facilitate testing by enabling mock objects, ultimately leading to cleaner, maintainable, and robust software design.