Travel system project

Dr/ Mahmoud Bassioni

Teaching Assistant / Eng.Esraa Mohsen

Center: Fayoum

Team members:

1- Mario Ibrahem Nassef	21-02020
2- Mario Mena Gabala	21-01022
3- Abanoub Yousry	21-01809
4- Rana Abdelrahman Abdelfattah	21-01470
5- Shahd Wael Fathy	21-02283
6- Maria Fadl Naguib	21-01042

Introduction :

The Travel Project is a software solution designed to simplify and streamline the process of booking and managing travel packages. This project leverages key object-oriented programming principles and design patterns to build a scalable, flexible, and maintainable system for managing travel-related services such as flights and trip packages.

We used 5 design patterns (Singleton - Factory - Observer - Prototype - Builder)

1 - Singleton Design Pattern:

Core Idea :

The Singleton Pattern ensures that only one instance of a class exists throughout the lifecycle of the application. It allows global access to that instance, ensuring that all parts of the program use the same object.

When to Use:

- When you need to ensure that there is only one instance of a class (e.g., managing a database connection, global system settings, or a logging service).

- When you need to share a common resource across different parts of the system.

Key Benefits:

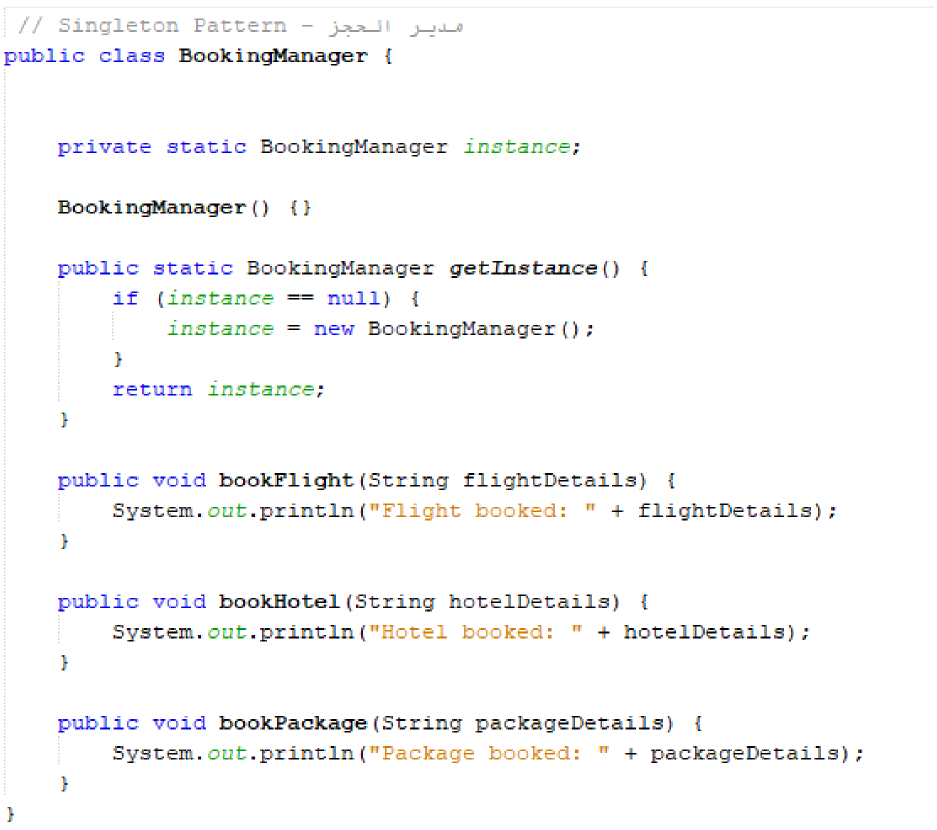
1. Unified State: There's only one instance of the class, ensuring consistency across the system.

2. Global Access: The instance is accessible from anywhere in the program.

3. Control over Creation: The instance is created only once, often lazily when needed.

- In Code :

This code defines a BookingManager class implementing the Singleton Design Pattern to ensure only one instance of the class is created during runtime.
The class ensures that all booking operations are managed centrally, avoiding the need to create multiple instances.



```java
// Singleton Pattern - مدير الحجز
public class BookingManager {


    private static BookingManager instance;

    BookingManager() {}

    public static BookingManager getInstance() {
        if (instance == null) {
            instance = new BookingManager();
        }
        return instance;
    }

    public void bookFlight(String flightDetails) {
        System.out.println("Flight booked: " + flightDetails);
    }

    public void bookHotel(String hotelDetails) {
        System.out.println("Hotel booked: " + hotelDetails);
    }

    public void bookPackage(String packageDetails) {
        System.out.println("Package booked: " + packageDetails);
    }
}
```

```java
// Singleton Pattern - مدير بيانات المستخدم
public class UserProfileManager {
   private static UserProfileManager instance;

    private String preferences;
    private int loyaltyPoints;

    UserProfileManager() {}

    public static UserProfileManager getInstance() {
        if (instance == null) {
            instance = new UserProfileManager();
        }
        return instance;
    }

    public void updatePreferences(String preferences) {
        this.preferences = preferences;
    }

    public String getPreferences() {
        return preferences;
    }

    public void addLoyaltyPoints(int points) {
        loyaltyPoints += points;
    }

    public int getLoyaltyPoints() {
        return loyaltyPoints;
    }

    void manageUserProfile() {
        throw new UnsupportedOperationException("Not supported yet.");
//To change body of generated methods, choose Tools | Templates.
    }
}
```

2- Observer Design Pattern:

Core Idea:
The **Observer Pattern** defines a one-to-many dependency between objects. When the state of a subject object changes, all its dependent observers are notified and updated automatically. This decouples the subject from the observers, allowing them to stay updated without needing to be directly linked.

When to Use:

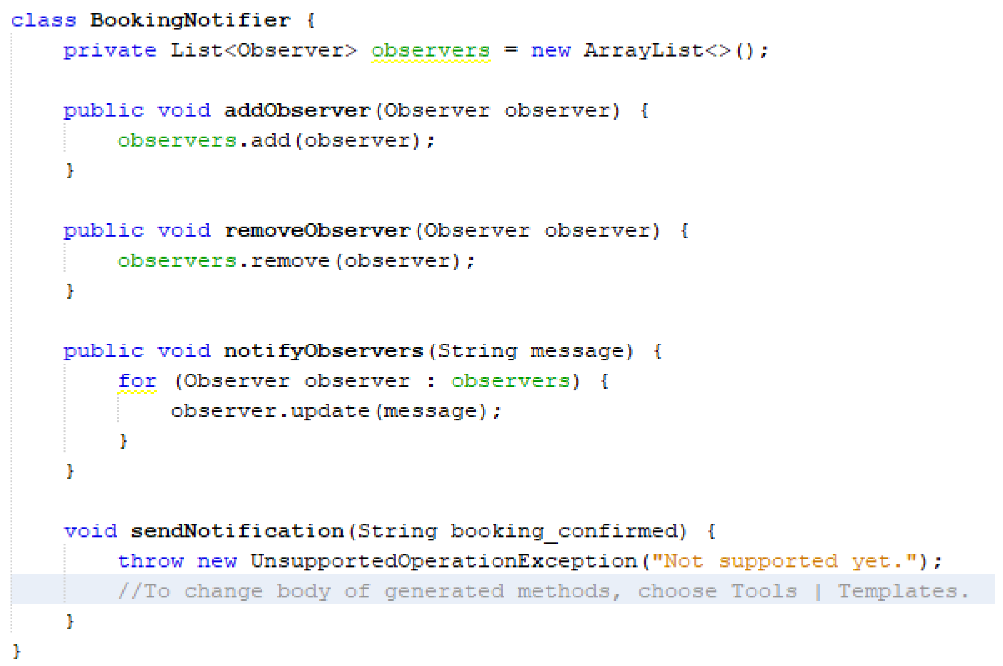
- When one object needs to notify many other objects about changes in its state.
- When you want to maintain the consistency of several dependent objects without tightly coupling them.

Key Benefits:

1. **Automatic Updates:** Observers are automatically notified when the state of the subject changes.
2. **Decoupling:** The subject does not need to know much about the observers, promoting flexibility.
3. **Scalability:** You can add as many observers as needed without affecting the subject.

- in code :

This code defines a BookingNotifier class implementing the Observer Design Pattern to manage notifications sent to multiple observers. The class facilitates communication between a subject (BookingNotifier) and its observers by notifying them whenever there's an update (e.g., booking confirmations). This ensures loose coupling and real-time updates for all registered observers.



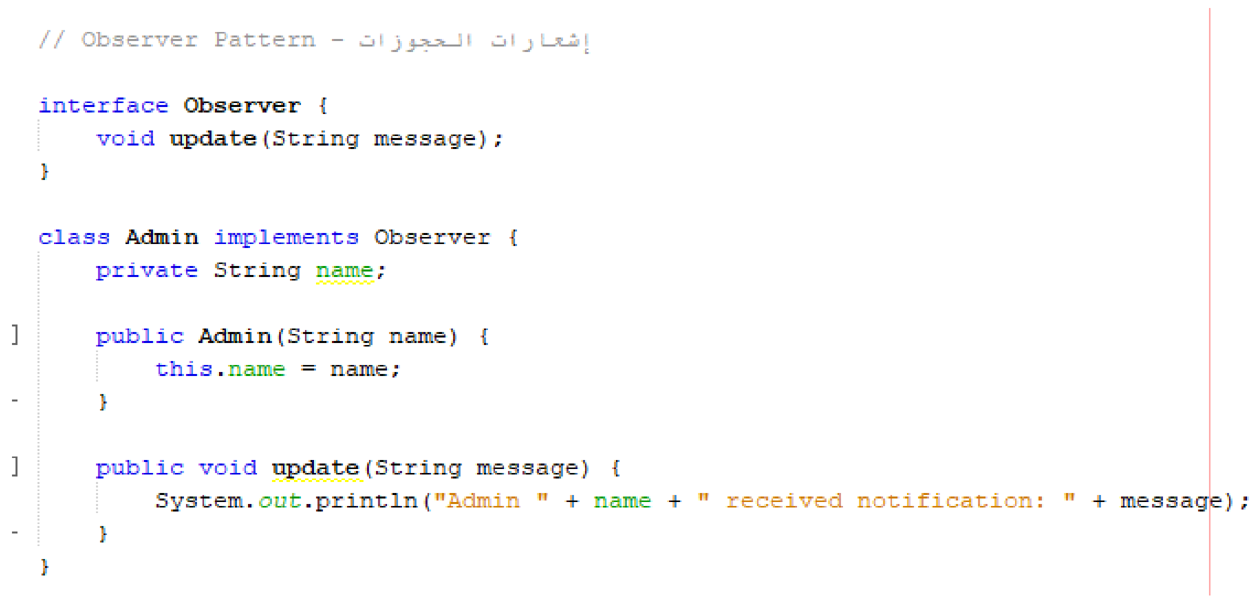
```java
class BookingNotifier {
    private List<Observer> observers = new ArrayList<>();

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }

    void sendNotification(String booking_confirmed) {
        throw new UnsupportedOperationException("Not supported yet.");
        //To change body of generated methods, choose Tools | Templates.
    }
}
```



```java
// Observer Pattern - إشعارات الحجوزات

interface Observer {
    void update(String message);
}

class Admin implements Observer {
    private String name;

    public Admin(String name) {
        this.name = name;
    }

    public void update(String message) {
        System.out.println("Admin " + name + " received notification: " + message);
    }
}
```

3-Prototype Design Pattern:

Core Idea:
The **Prototype Pattern** is used to create duplicate objects by cloning an existing object, called the prototype, instead of creating new instances from scratch.

Purpose:
It allows the creation of new objects by cloning a prototype, which can be more efficient when generating objects with similar structures. The prototype object typically has a `clone()` method to create an exact copy.

When to Use:

- When the process of creating an object is costly or complex.
- When you need to generate new objects in various configurations or variations quickly by copying an existing object.

Key Benefits:

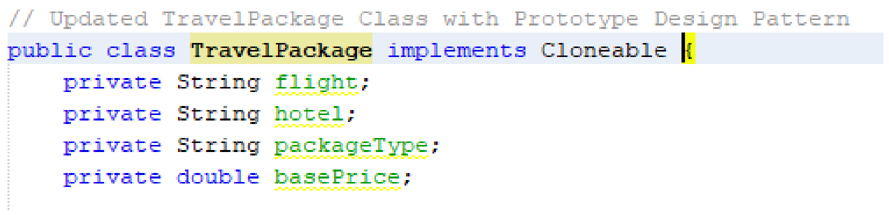
1. **Efficient Object Creation:** Cloning can be more efficient than creating a new object from scratch each time.
2. **Flexible Customization:** Once the prototype is cloned, it can be customized as needed.
3. **Avoid Redundant Creation:** It avoids repeated creation of identical objects.

-In Code :

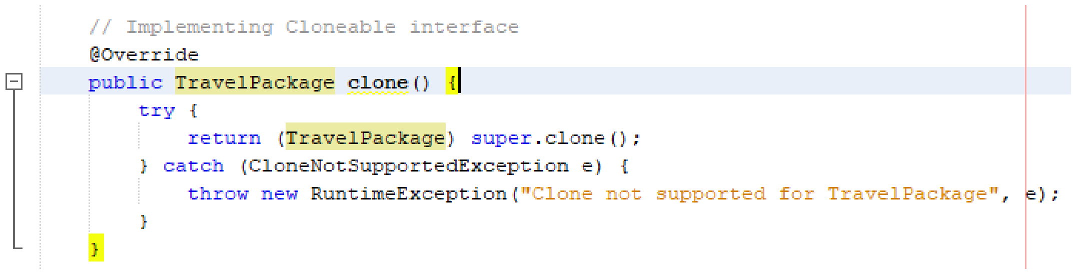
The class implements the Cloneable interface.

The clone() method allows creating a duplicate of an existing TravelPackage object efficiently without reconstructing it from scratch.

Allows efficient duplication of packages using the Prototype Pattern.



```java
// Updated TravelPackage Class with Prototype Design Pattern
public class TravelPackage implements Cloneable {
    private String flight;
    private String hotel;
    private String packageType;
    private double basePrice;
```



```java
// Implementing Cloneable interface
@Override
public TravelPackage clone() {
    try {
        return (TravelPackage) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new RuntimeException("Clone not supported for TravelPackage", e);
    }
}
```

```java
public interface Prototype {
    Prototype clone();
    // يجب أن تعود بنسخة طبق الأصل من الكائن.
}
```

4-Factory Design Pattern:

Core Idea:

The **Factory Pattern** provides a way to create objects without specifying the exact class of object that will be created. Instead, a factory class decides which type of object to create based on the input or certain conditions.

When to Use:

- When you have multiple related classes that share common behavior but differ in implementation.

- When you want to delegate the creation logic to a factory class instead of handling it in multiple places in your code.

Key Benefits:

1. Hides Complex Creation Logic:The Factory abstracts the instantiation logic and hides the complexity.

2. Flexibility: You can easily add new types of objects without modifying the existing code.

3. Extensibility:New types of objects can be added to the system with minimal changes to the codebase.

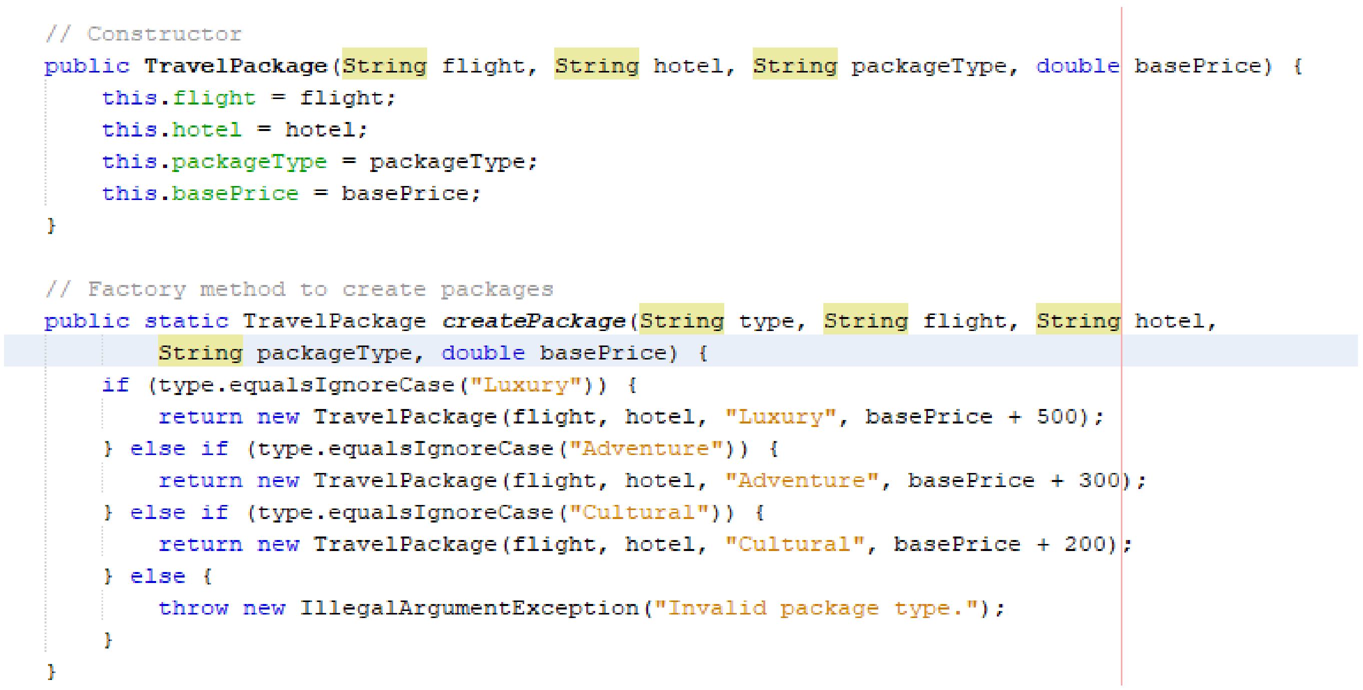
- In Code :

The createPackage(String type, ...) static method acts as a factory to create predefined package types: Luxury, Adventure, and Cultural.

Based on the type, it adjusts the basePrice to reflect the package type's additional cost.

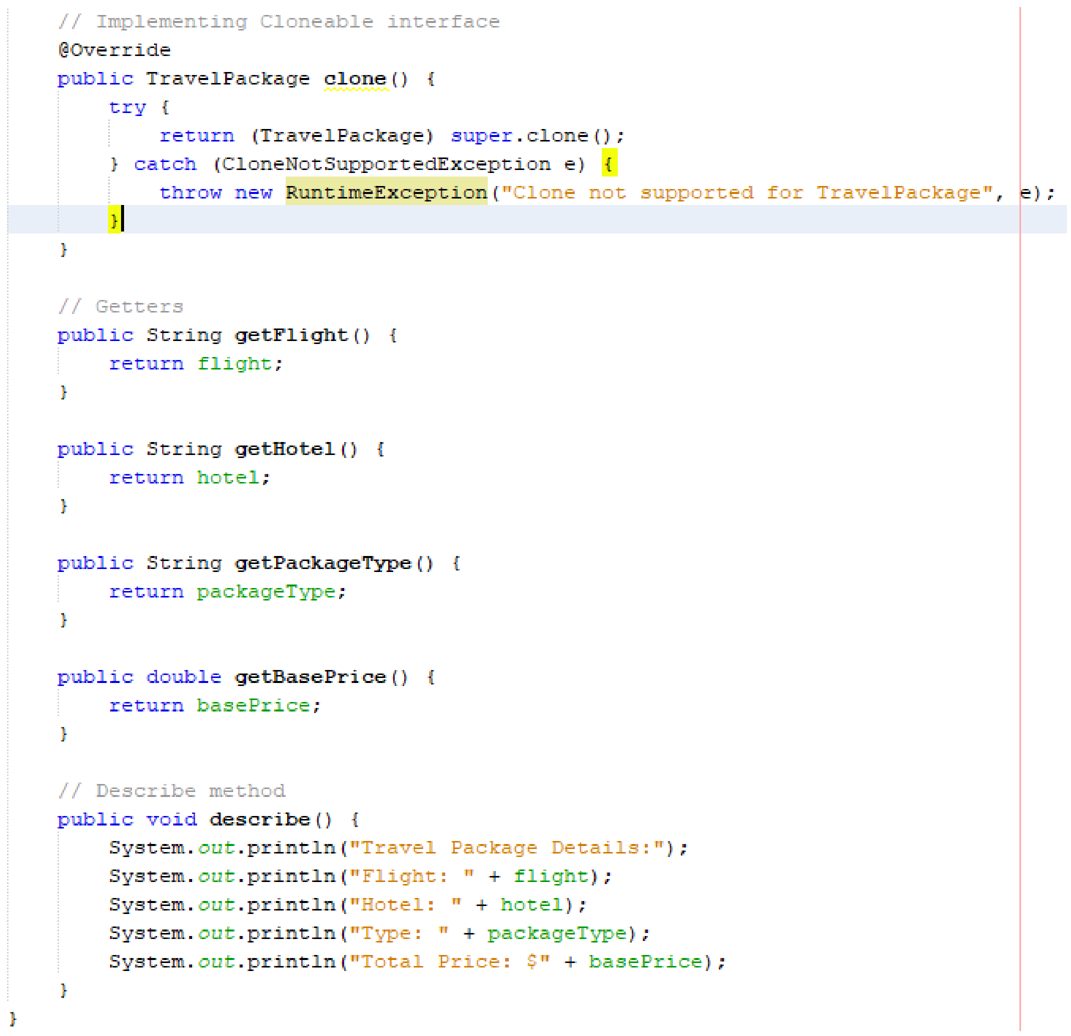
Provides a mechanism to easily create and manage travel packages using the Factory Pattern.

Ensures scalability for adding new package types or extending its functionality.



```java
// Constructor
public TravelPackage(String flight, String hotel, String packageType, double basePrice) {
    this.flight = flight;
    this.hotel = hotel;
    this.packageType = packageType;
    this.basePrice = basePrice;
}

// Factory method to create packages
public static TravelPackage createPackage(String type, String flight, String hotel,
        String packageType, double basePrice) {
    if (type.equalsIgnoreCase("Luxury")) {
        return new TravelPackage(flight, hotel, "Luxury", basePrice + 500);
    } else if (type.equalsIgnoreCase("Adventure")) {
        return new TravelPackage(flight, hotel, "Adventure", basePrice + 300);
    } else if (type.equalsIgnoreCase("Cultural")) {
        return new TravelPackage(flight, hotel, "Cultural", basePrice + 200);
    } else {
        throw new IllegalArgumentException("Invalid package type.");
    }
}
```



```java
    // Implementing Cloneable interface
    @Override
    public TravelPackage clone() {
        try {
            return (TravelPackage) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException("Clone not supported for TravelPackage", e);
        }
    }

    // Getters
    public String getFlight() {
        return flight;
    }

    public String getHotel() {
        return hotel;
    }

    public String getPackageType() {
        return packageType;
    }

    public double getBasePrice() {
        return basePrice;
    }

    // Describe method
    public void describe() {
        System.out.println("Travel Package Details:");
        System.out.println("Flight: " + flight);
        System.out.println("Hotel: " + hotel);
        System.out.println("Type: " + packageType);
        System.out.println("Total Price: $" + basePrice);
    }
}
```

5-Builder Design Pattern:

Core Idea:
The **Builder Pattern** is used to construct complex objects step by step. It separates the construction of an object from its representation, allowing the same construction process to create different representations.

Purpose:
The Builder pattern provides an organized and flexible way to build complex objects, especially when the object has many components or optional configurations.

When to Use:

- When you need to create an object with many optional parts or configurations.
- When the construction of the object involves several steps or has complex logic that can be broken down into separate phases.

Key Benefits:

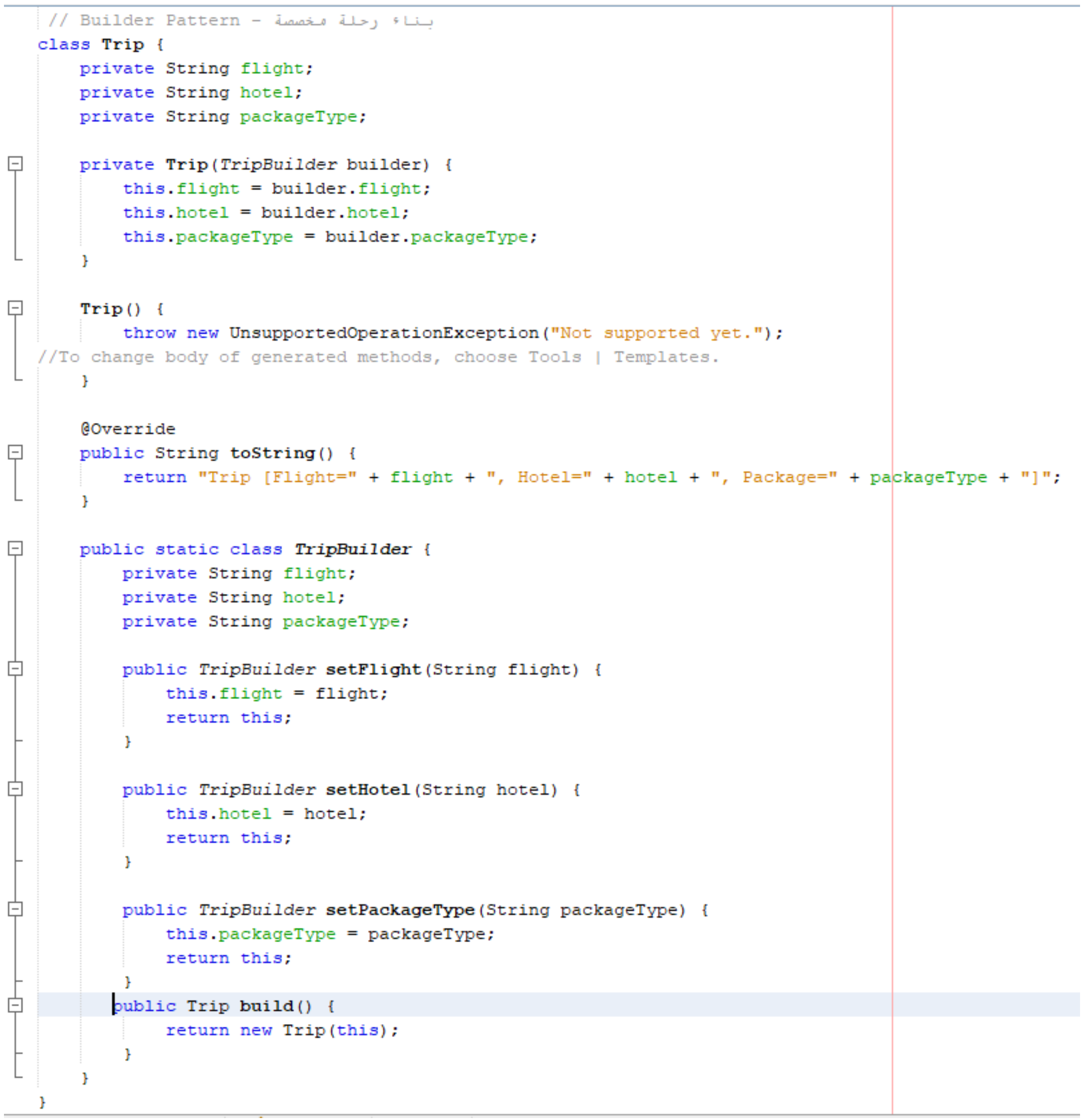
1. **Step-by-Step Construction:** Allows for building an object incrementally, adding parts as needed.
2. **Customizable Objects:** It enables creating different representations of the same object by specifying different configurations during the building process.
3. **Separation of Construction Logic:** Keeps the construction logic separate from the actual object.

- In Code :

This code uses the **Builder Design Pattern** to create a Trip object in a flexible and modular way.
Simplifies Complex Object Creation: Allows step-by-step configuration without using multiple constructors.

Readable and Maintainable: Makes code cleaner and easier to extend.

This implements the **Builder Design Pattern** to ensure flexible, modular, and controlled object creation.



```java
// Builder Pattern - بناء رحلة مخصصة
class Trip {
    private String flight;
    private String hotel;
    private String packageType;

    private Trip(TripBuilder builder) {
        this.flight = builder.flight;
        this.hotel = builder.hotel;
        this.packageType = builder.packageType;
    }

    Trip() {
        throw new UnsupportedOperationException("Not supported yet.");
//To change body of generated methods, choose Tools | Templates.
    }

    @Override
    public String toString() {
        return "Trip [Flight=" + flight + ", Hotel=" + hotel + ", Package=" + packageType + "]";
    }

    public static class TripBuilder {
        private String flight;
        private String hotel;
        private String packageType;

        public TripBuilder setFlight(String flight) {
            this.flight = flight;
            return this;
        }

        public TripBuilder setHotel(String hotel) {
            this.hotel = hotel;
            return this;
        }

        public TripBuilder setPackageType(String packageType) {
            this.packageType = packageType;
            return this;
        }
       public Trip build() {
            return new Trip(this);
        }
    }
}
```

Travel Booking System GUI:

The provided Java code implements a simple Travel Booking System GUI using Swing.

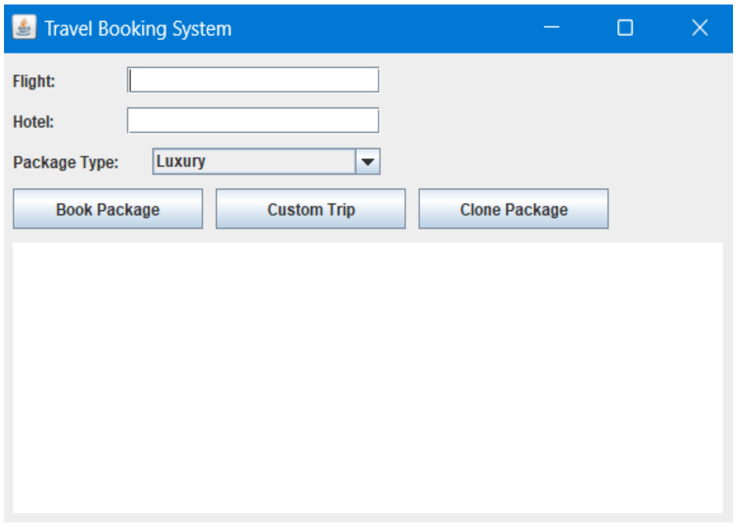
Main Workflow

- **Startup**: The Travel_GUI class initializes the GUI and registers admins as observers.
- **User Interaction**: Users interact with the GUI to book or manage travel packages.
- **Output**: Results of actions (e.g., bookings, cloned packages) are displayed in the outputArea.

This code demonstrates the combination of Swing-based GUI and multiple design patterns for better modularity and functionality in a travel booking application.

Design Patterns Used in GUI

1. **Singleton**:
   - BookingManager and UserProfileManager are implemented as singletons to ensure only one instance exists.
2. **Observer**:
   - BookingNotifier notifies registered admins when a package is booked.
3. **Builder**:
   - The Trip class uses a builder pattern for constructing custom trips.
4. **Prototype**:
   - The TravelPackage class allows cloning of existing packages.


Travel Booking System
Flight:
Hotel:
Package Type:
Luxury
Book Package
Custom Trip
Clone Package