Travel Management System

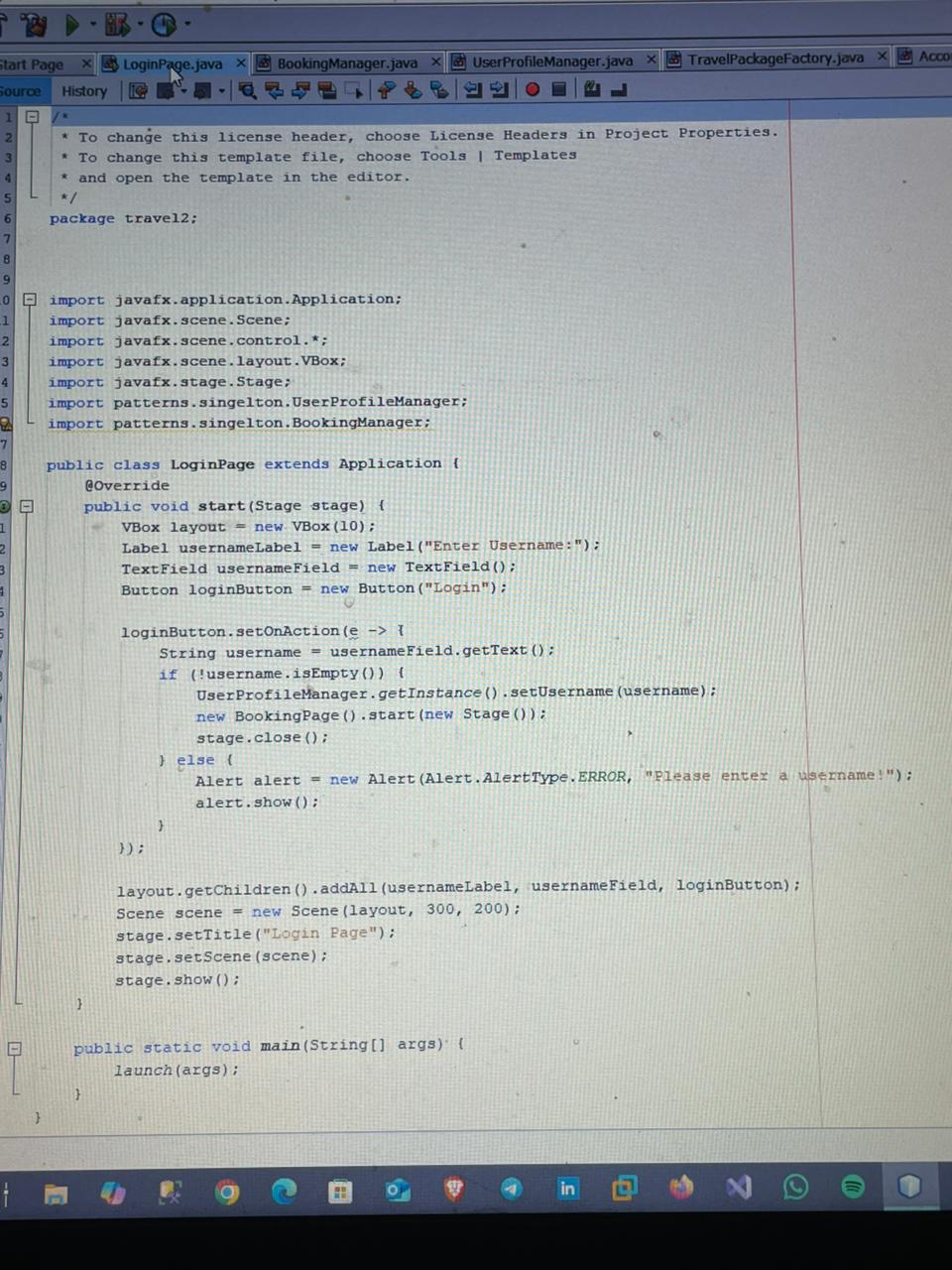
Project team: Yara Essam Khalaf 21-02141

Zahraa Samir Abdrhman 21-01861

Hajar Husni Hussien 21-02128

Abdullah Mohamed Kamel 21-01960

Ahmed Mohamed Ahmed 21-01841



**Singleton Pattern**

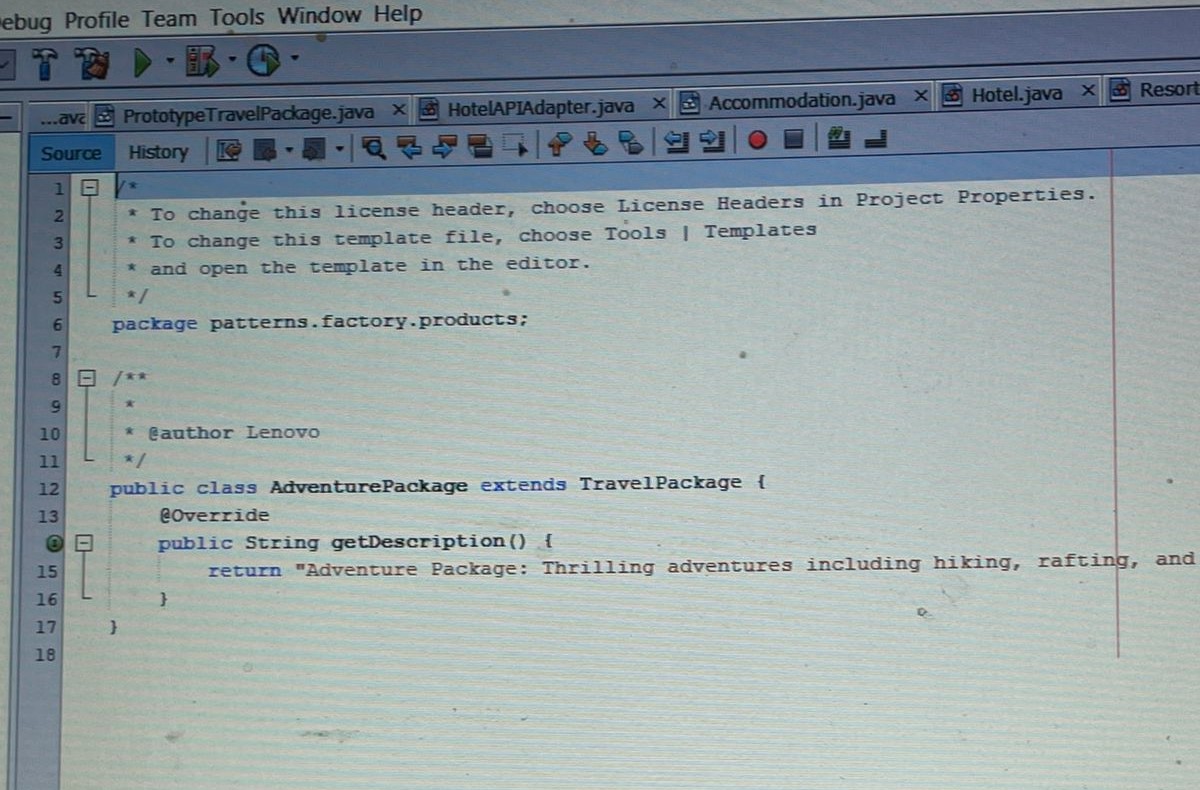
The Singleton design pattern ensures that a class has only one instance and provides a global point of access to that instance. It is often used to manage shared resources or to coordinate actions that must happen only once.

In the code snippet:

1. **UserProfileManager**:
   * The UserProfileManager.getInstance() method is called to obtain the single instance of the UserProfileManager class. This suggests that the UserProfileManager class follows the Singleton design pattern.
   * The instance is used to store the username (setUsername(username)), ensuring that only one object manages the user's profile throughout the application.
2. **BookingManager**:
   * Similarly, patterns.singleton.BookingManager is likely another class that implements the Singleton pattern. Although it is not directly used in the visible part of the code, its inclusion indicates that the application design relies on Singleton objects to manage bookings and profiles.

**Why Use Singleton in This Context?**

* **Centralized Resource Management**: Singleton ensures there is a single point to manage user profile information and booking data.
* **Consistency**: Since only one instance exists, all parts of the application will reference the same user profile or booking data, reducing the risk of inconsistencies.
* **Ease of Access**: The Singleton instance can be accessed globally without passing it as a parameter between methods or classes.



**Factory Pattern**

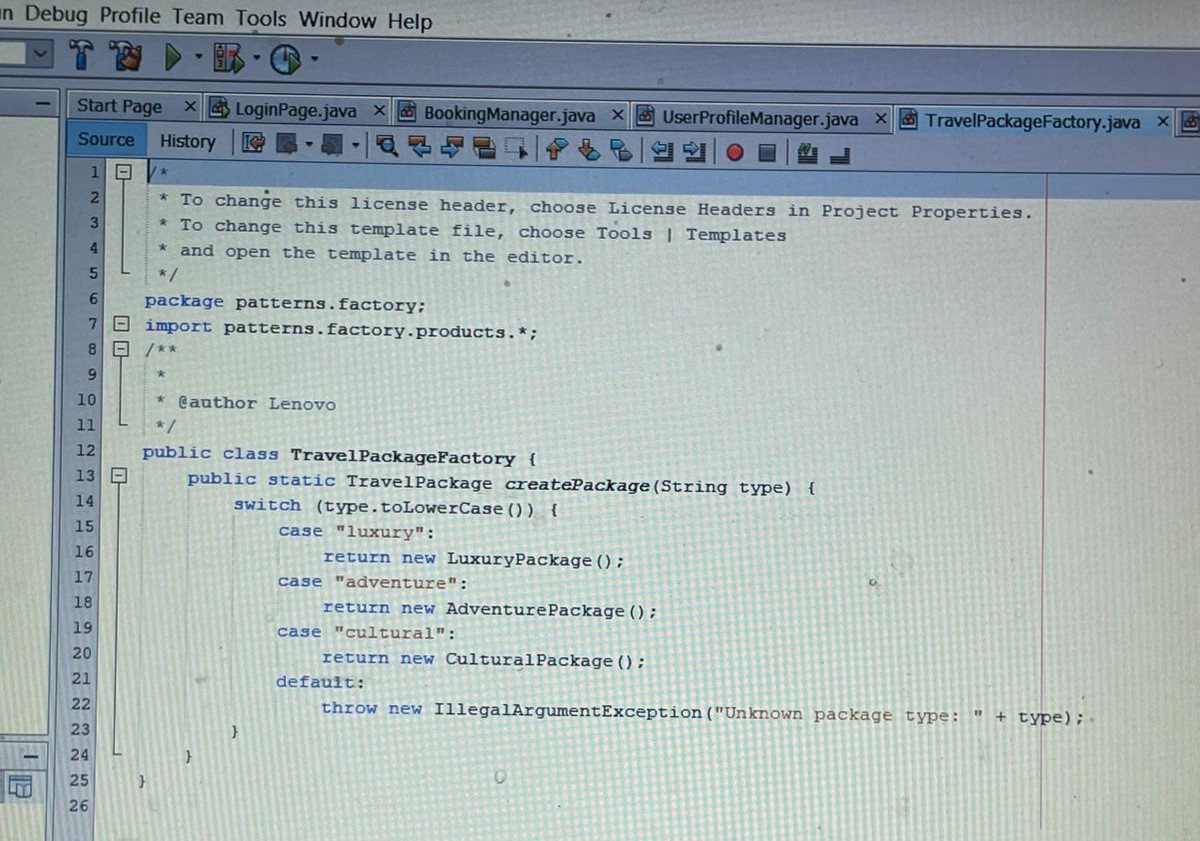
The Factory design pattern is a **creational design pattern** that provides an interface for creating objects in a super class but allows subclasses to alter the type of objects that will be created. This promotes loose coupling in the code.

**How it is implemented in the provided code:**

1. **Class TravelPackageFactory**:
   * The class contains a static method createPackage(String type) that takes a String as input (representing the type of travel package) and returns an object of type TravelPackage.
   * Depending on the value of type, the method creates and returns different subclasses of TravelPackage (LuxuryPackage, AdventurePackage, CulturalPackage).
2. **Switch Statement**:
   * The switch block dynamically determines which subclass to instantiate based on the type parameter:
     + "luxury" → Creates and returns an instance of LuxuryPackage.
     + "adventure" → Creates and returns an instance of AdventurePackage.
     + "cultural" → Creates and returns an instance of CulturalPackage.
   * If the type is not recognized, it throws an IllegalArgumentException with an appropriate error message.
3. **Loose Coupling**:
   * The client code does not need to know which specific subclass of TravelPackage is being instantiated. It simply calls the factory method (createPackage) with the required type, making the code more maintainable and flexible.

**Why Use the Factory Pattern Here?**

* **Centralized Object Creation**: The creation logic for TravelPackage objects is centralized in the TravelPackageFactory, making it easy to modify or extend without changing client code.
* **Encapsulation of Object Creation**: The details of how LuxuryPackage, AdventurePackage, and CulturalPackage are instantiated are hidden from the client code.
* **Flexibility**: It is easy to add a new package type (e.g., "beach") without altering existing client code. You only need to update the factory to handle the new case.



**Prototype Pattern**

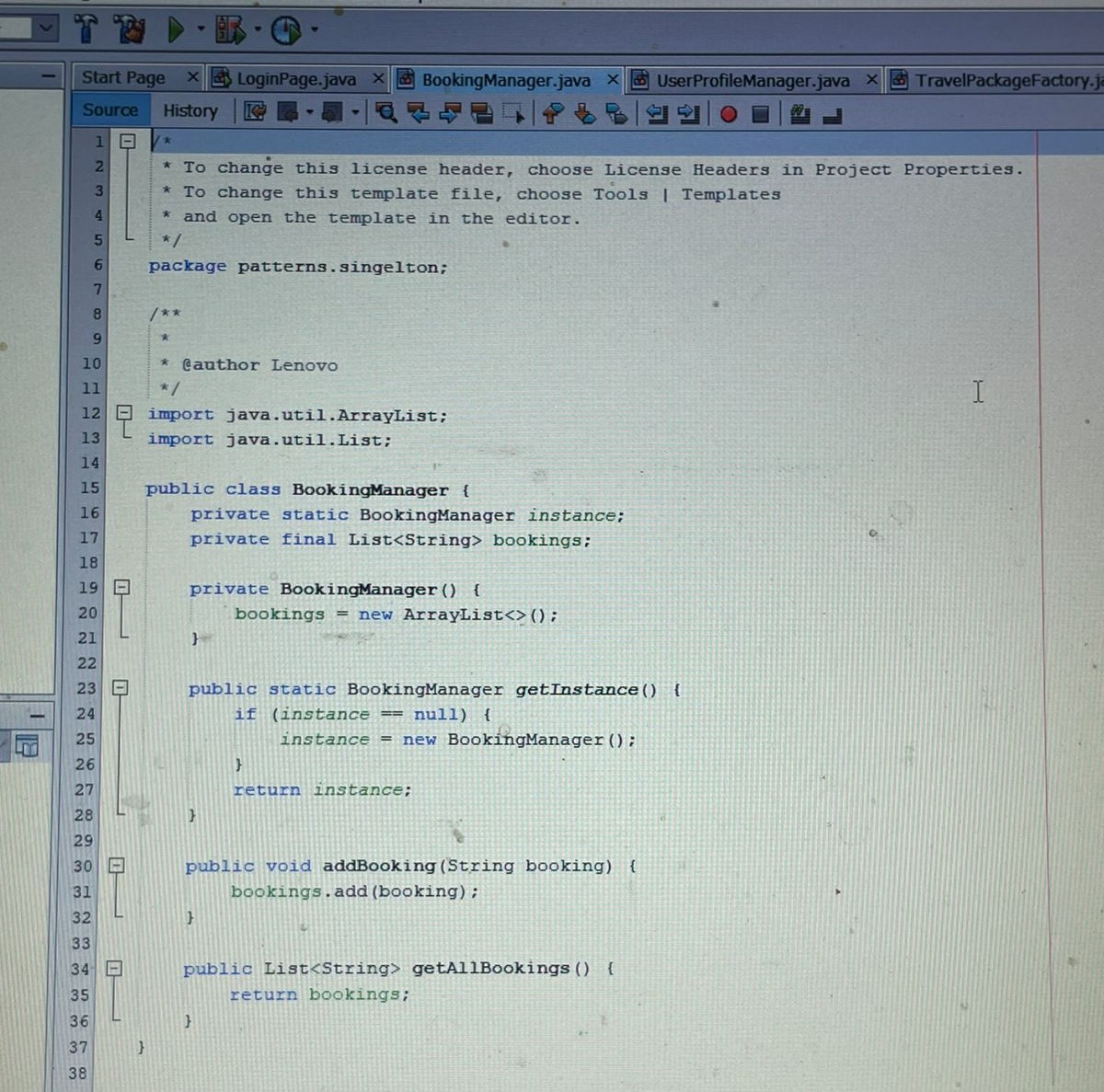
The Prototype pattern is a **creational design pattern** used to create new objects by copying (or "cloning") an existing object. This is useful when object creation is expensive (e.g., in terms of time or resources) or when specific objects need to be replicated with their current state.

**How it is implemented in this code:**

1. **Abstract Class PrototypeTravelPackage**:
   * The class implements the Cloneable interface, which allows objects of this class (or its subclasses) to be cloned.
   * The clone() method is overridden to return a deep copy of the current object.
2. **Key Methods**:
   * getDescription():
     + An abstract method, meaning each subclass of PrototypeTravelPackage must provide its own implementation of how to describe itself.
   * clone():
     + This method uses super.clone() to create and return a copy of the object. It throws a CloneNotSupportedException if the object cannot be cloned.
3. **Usage**:
   * Subclasses of PrototypeTravelPackage can define specific types of travel packages. For example:
     + A LuxuryPackage subclass might have a unique description or properties.
     + An AdventurePackage subclass might represent adventure-specific details.
   * When a new instance of a travel package is needed, the existing prototype object is cloned instead of creating a new object from scratch.

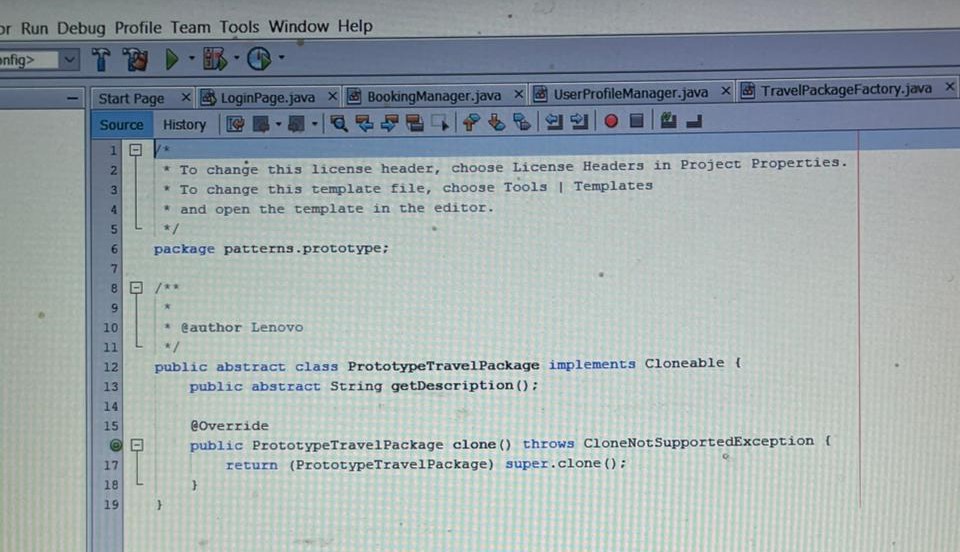
**Advantages of the Prototype Pattern:**

* **Efficiency**:
  + Cloning an object is typically faster than instantiating a new one, especially when the object creation process involves complex logic.
* **Dynamic Object Creation**:
  + New objects can be created at runtime by cloning a prototype, without requiring hard-coded logic for object creation.
* **Preserves State**:
  + The cloned object retains the state of the prototype, which can be useful in applications where state consistency is required.



**Singleton Pattern**

The Singleton design pattern ensures that a class has only one instance and provides a global point of access to that instance. It is often used to manage shared resources or to coordinate actions that must happen only once.



**Prototype Pattern**

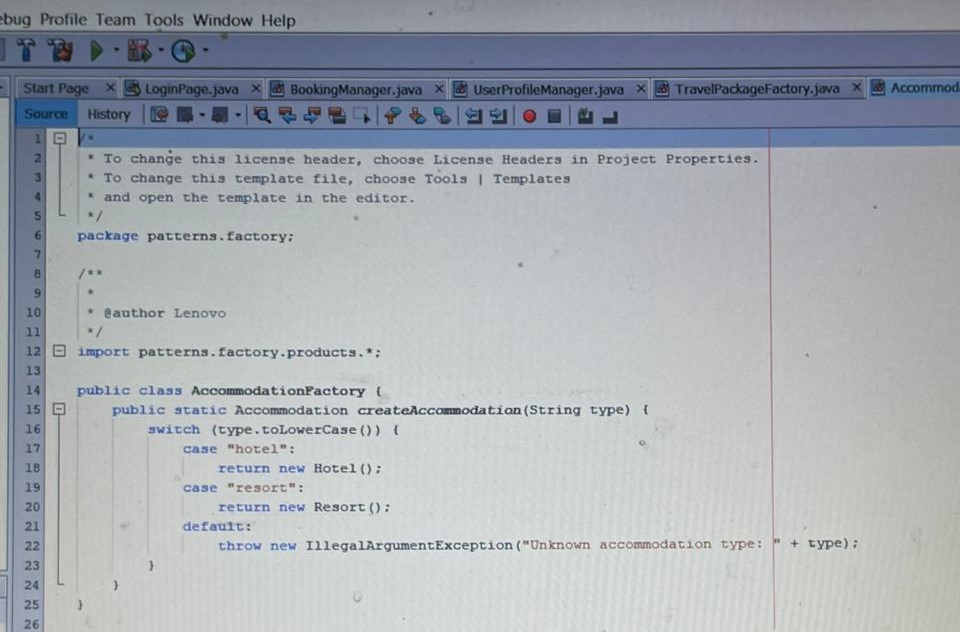
The Prototype pattern is a **creational design pattern** used to create new objects by copying (or "cloning") an existing object. This is useful when object creation is expensive (e.g., in terms of time or resources) or when specific objects need to be replicated with their current state.

**How it is implemented in this code:**

1. **Abstract Class PrototypeTravelPackage**:
   * The class implements the Cloneable interface, which allows objects of this class (or its subclasses) to be cloned.
   * The clone() method is overridden to return a deep copy of the current object.
2. **Key Methods**:
   * getDescription():
     + An abstract method, meaning each subclass of PrototypeTravelPackage must provide its own implementation of how to describe itself.
   * clone():
     + This method uses super.clone() to create and return a copy of the object. It throws a CloneNotSupportedException if the object cannot be cloned.
3. **Usage**:
   * Subclasses of PrototypeTravelPackage can define specific types of travel packages. For example:
     + A LuxuryPackage subclass might have a unique description or properties.
     + An AdventurePackage subclass might represent adventure-specific details.
   * When a new instance of a travel package is needed, the existing prototype object is cloned instead of creating a new object from scratch.

**Advantages of the Prototype Pattern:**

* **Efficiency**:
  + Cloning an object is typically faster than instantiating a new one, especially when the object creation process involves complex logic.
* **Dynamic Object Creation**:
  + New objects can be created at runtime by cloning a prototype, without requiring hard-coded logic for object creation.
* **Preserves State**:
  + The cloned object retains the state of the prototype, which can be useful in applications where state consistency is required.



**Factory Pattern**

The Factory design pattern is a **creational design pattern** that provides an interface for creating objects in a super class but allows subclasses to alter the type of objects that will be created. This promotes loose coupling in the code.