Documentation

1. **Prototype Pattern**:

Ticket Interface

2. Purpose:
   - Defines a blueprint for ticket objects.
   - Enforces consistency across ticket implementations.
3. Methods:
   - clone():
   - Returns a duplicate instance of the ticket object.
   - Ensures that tickets can be cloned easily.
   - getPrice():
   - Returns the ticket's price as an integer.
   - getType():
   - Returns the ticket's type as a string.
4. Use Cases:
   - Provides flexibility for implementing various ticket types.
   - Supports the Prototype design pattern via the clone() method.

```java
public interface Ticket {
    Ticket clone();  // This allows cloning of tickets

    int getPrice();  // Returns the price of the ticket

    String getType();  // Returns the type of the ticket as a Strin
}

```

ConcreteTicket Class

1. Purpose:
   - A concrete implementation of the Ticket interface.
   - Represents a specific type of ticket with an associated price.

2. Attributes:
    - type:
    - Immutable string that describes the type of the ticket (e.g., "VIP", "Regular").
    - price:
    - Immutable integer representing the cost of the ticket.
3. Constructor:
    - ConcreteTicket(String type, int price):
    - Initializes a ConcreteTicket object with the given type and price.
    - Ensures immutability by using final fields.
4. Implemented Methods:
    - getType():
    - Returns the ticket's type.
    - getPrice():
    - Returns the ticket's price.
    - clone():
    - Creates and returns a new ConcreteTicket instance with the same type and price as the original object.
    - Implements the Prototype design pattern to allow object duplication.
5. Key Features:
    - Immutability ensures the integrity of type and price once the object is created.
    - Simple and efficient cloning through the clone() method.
    - Adheres to the Ticket interface, guaranteeing consistency.
6. Example Usage:

```java
public class ConcreteTicket implements Ticket {
    private final String type;  // Immutable
    private final int price;  // Immutable

    // Constructor to initialize type and price
    public ConcreteTicket(String type, int price) {
        this.type = type;
        this.price = price;
    }

    // Getter for ticket type
    @Override
    public String getType() {
        return type;
    }

    // Getter for ticket price
    @Override
    public int getPrice() {
        return price;
    }

    // Clone method to create a new instance of ConcreteTicket with the same type and pric
e   @Override
    public Ticket clone() {
        return new ConcreteTicket(this.type, this.price);
    }
}
```

7. Design Patterns:
    - Implements the Prototype pattern through the clone() method.
    - Ensures immutability of data, adhering to best practices in object design.

---

1. Factory Pattern :

1. Purpose of the Class:
   The TicketFactory class is designed to create ticket objects dynamically based on the specified type.

---

2. Factory Method:

Method Name: createTicket

Parameter: A String representing the ticket type.

Functionality: Uses a switch statement to match the ticket type and create a corresponding ConcreteTicket object.

---

3. Supported Ticket Types:

"ODC" with a price of 300.

"Balcony" with a price of 350.

"Box" with a price of 350.

"Super Balcony" with a price of 500.

---

4. Error Handling:
   If the ticket type doesn't match any of the predefined types, the method returns null.

---

5. Flexibility:
   The factory pattern ensures easy extensibility. New ticket types can be added by extending the switch statement or modifying the logic.

```java
public class TicketFactory {

    // Method to create tickets based on type
    public Ticket createTicket(String type) {
        switch (type.toLowerCase()) {
            case "odc":
                return new ConcreteTicket("ODC", 300);
            case "balcony":
                return new ConcreteTicket("Balcony", 350);
            case "box":
                return new ConcreteTicket("Box", 350);
            case "super balcony":
                return new ConcreteTicket("Super Balcony", 500);
            default:
                return null;  // Return null if type doesn't match any predefined ticket types
        }
    }
}
```

Singleton pattern :

1. Purpose of the Class:
The TicketFactory class follows the Singleton design pattern. Its purpose is to ensure that only one instance of the TicketFactory class exists throughout the application's lifecycle.

2. Singleton Method:

- Method Name: getInstance
- Parameter: None.
- Functionality: This method checks if the `instance` of the TicketFactory class is null. If it is null, it creates a new instance. Otherwise, it returns the already created instance. This ensures that no matter how many times getInstance is called, it always returns the same object.

3. Instance Management:

- The class has a private static member called `instance`, which holds the single instance of the class.

- The constructor is private to prevent creating multiple objects from outside the class.

4. Thread Safety and Flexibility:
This basic Singleton implementation isn't thread-safe. For a multithreaded environment, additional mechanisms like `synchronized` or double-checked locking can be used to ensure thread safety.

Code:

```java
private static TicketFactory instance;

private TicketFactory() {
}

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

Builder Pattern Class

The `MovieTicket` class represents a ticket for a movie. It includes attributes such as the movie name, genre, showtime, theater, seat number, and optional attributes like 3D glasses and snacks.

Attributes

- `movieName` (String): The name of the movie.
- `genre` (String): The genre of the movie.
- `showTime` (String): The time of the movie show.
- `theater` (String): The name of the theater.
- `seatNumber` (String): The seat number.
- `has3DGlasses` (boolean): Whether the ticket includes 3D glasses.
- `hasSnacks` (boolean): Whether the ticket includes snacks.

Constructor

The constructor is private and is used by the `MovieTicketBuilder` class to create instances of `MovieTicket`.

MovieTicketBuilder Class

The `MovieTicketBuilder` class is used to construct instances of `MovieTicket` by setting its attributes step-by-step.

Attributes

- `movieName` (String): The name of the movie.
- `genre` (String): The genre of the movie.
- `showTime` (String): The time of the movie show.
- `theater` (String): The name of the theater.
- `seatNumber` (String): The seat number.
- `has3DGlasses` (boolean): Whether the ticket includes 3D glasses (default: false).
- `hasSnacks` (boolean): Whether the ticket includes snacks (default: false).

Methods

- `MovieTicketBuilder(String movieName, String showTime)`: Constructor to initialize `movieName` and `showTime`.
- `setGenre(String genre)`: Sets the genre of the movie.
- `setTheater(String theater)`: Sets the theater name.
- `setSeatNumber(String seatNumber)`: Sets the seat number.
- `add3DGlasses()`: Adds 3D glasses to the ticket.
- `addSnacks()`: Adds snacks to the ticket.
- `build()`: Builds and returns an instance of `MovieTicket`.

Example Usage

```
MovieTicket ticket = new MovieTicket.MovieTicketBuilder("Avatar", "7:00 PM")
                                        setGenre("Sci-Fi").
                                        setTheater("IMAX").
                                      setSeatNumber("A12").
                                            ()add3DGlasses.
                                               ()addSnacks.
                                                  ;()build.

                                                ;System.out.println(ticket)
```

toString() Method

The `toString()` method provides a string representation of the `MovieTicket` object, which includes all its attributes.

Proxy Pattern

Overview

The Movie Ticket project demonstrates the use of the Proxy Pattern to manage the creation of different types of tickets while incorporating additional logic, such as determining whether a ticket is available before creating it.

TicketFactoryProxy Class

The `TicketFactoryProxy` class acts as a proxy to the `TicketFactory`, adding an extra layer of control over the ticket creation process.

Attributes

- `realTicketFactory` (TicketFactory): The actual factory that creates ticket objects.

Methods

- `TicketFactoryProxy()`: Constructor that initializes the real ticket factory.
- `createTicket(String type)`: Creates and returns a ticket of the specified type if the ticket is available.

Implementation

```
                                                Proxy class for TicketFactory //
                                                      } class TicketFactoryProxy
                                   ;private TicketFactory realTicketFactory

                                                 } ()public TicketFactoryProxy
                  ;()this.realTicketFactory = TicketFactory.getInstance
                                                                                {

                                  } public Ticket createTicket(String type)
rintln("Proxy: Checking access to create a ticket of type: " + type)

                                 Add validation or logging if needed //
                                  } if (type == null || type.isEmpty())
    ;System.out.println("Proxy: Invalid ticket type requested!")
                                                            ;return null
                                                                          {

                   Delegates the ticket creation to the real factory //
                  ;Ticket ticket = realTicketFactory.createTicket(type)
rintln("Proxy: Ticket of type '" + type + "' created successfully.")
                                                             ;return ticket
                                                                                {
                                                                                    {
```

Usage Example

Here's an example of how you can use the `TicketFactoryProxy`:

```
 ;()TicketFactoryProxy ticketFactoryProxy = new TicketFactoryProxy

        ;Ticket odcTicket = ticketFactoryProxy.createTicket("ODC")
;Ticket balconyTicket = ticketFactoryProxy.createTicket("Balcony")

                                          ;System.out.println(odcTicket)
                                       ;System.out.println(balconyTicket)
```