

Documentation

1. Singleton Pattern :

Main Points and Key Implementation Details

Overview

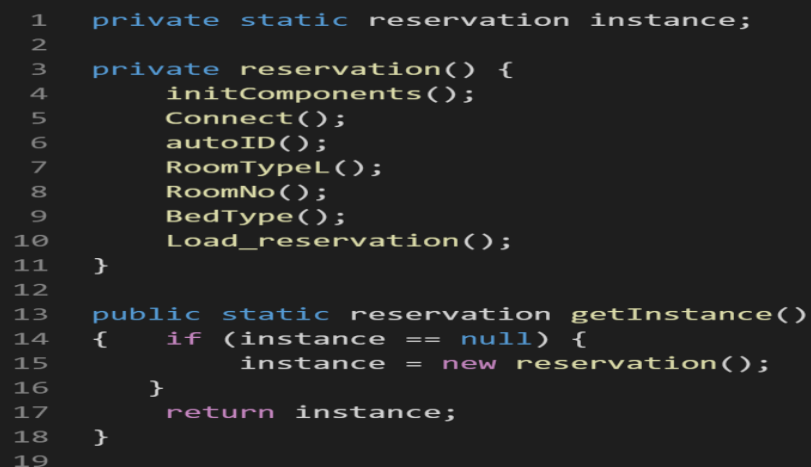
- Implements Singleton Design Pattern to ensure a single instance of the reservation class.
- Manages hotel reservations with MySQL integration.
- Provides dynamic loading for room details and reservation display.

Key Features

1. Singleton Design Pattern:
 - Lazy and Eager Initialization approaches to control instance creation.
 - Thread-safe and optimized for memory usage.
2. Database Integration:
 - Establishes a MySQL connection.
 - Implements methods for dynamic data retrieval and manipulation.
3. Auto-Generated Reservation IDs:
 - Dynamically generates unique IDs based on existing database records.
4. Dynamic UI Updates:
 - Populates dropdowns (room types, room numbers, bed types) and tables with real-time data.
5. Reservation Management:
 - Adds and loads reservations while resetting fields for future inputs.

Singleton Pattern Implementation

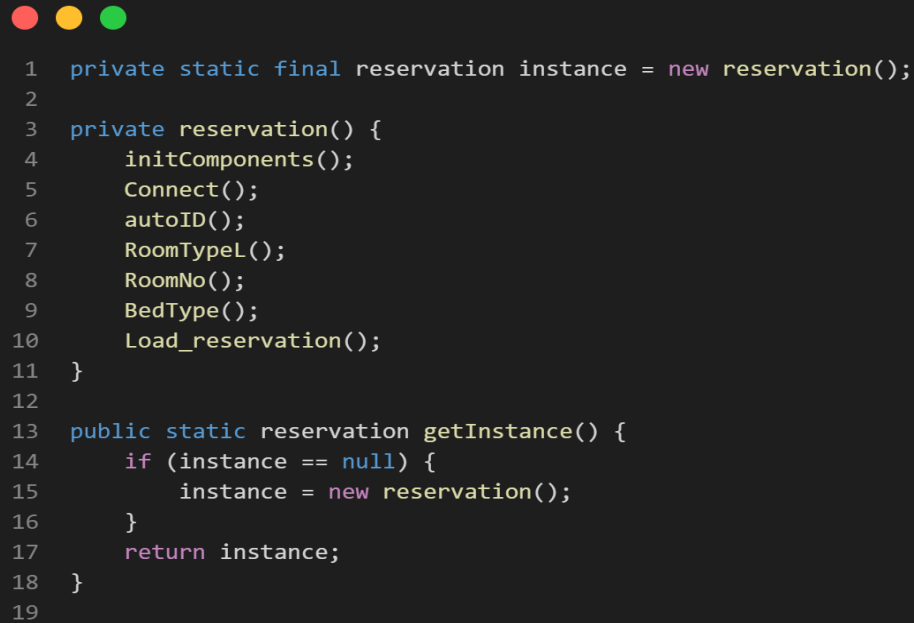
1. Lazy Initialization
 - Ensures the instance is created only when needed.



```
1  private static reservation instance;
2
3  private reservation() {
4      initComponents();
5      Connect();
6      autoID();
7      RoomTypeL();
8      RoomNo();
9      BedType();
10     Load_reservation();
11 }
12
13 public static reservation getInstance()
14 {
15     if (instance == null) {
16         instance = new reservation();
17     }
18     return instance;
19 }
```

2. Eager Initialization

- Preloads the instance at class loading for simplicity.



```
1 private static final reservation instance = new reservation();
2
3 private reservation() {
4     initComponents();
5     Connect();
6     autoID();
7     RoomTypeL();
8     RoomNo();
9     BedType();
10    Load_reservation();
11 }
12
13 public static reservation getInstance() {
14     if (instance == null) {
15         instance = new reservation();
16     }
17     return instance;
18 }
19
```

2. Factory Pattern :

Overview:

- The DatabaseConnectionFactory class implements the Factory Pattern to handle the creation of database connections. It abstracts the process of establishing a connection to a MySQL database, ensuring a consistent and centralized approach to managing database connections across the application.

Key Components:

1. createConnection Method:
2. Loading the MySQL JDBC Driver:
3. Establishing the Database Connection:
4. Error Handling:
5. Returning the Connection:

This class follows the Factory design pattern, providing a clean, reusable method for establishing a connection to the MySQL database, centralizing connection management, and improving maintainability.

```
1 public class DatabaseConnectionFactory {
2     public static Connection createConnection() {
3         Connection con = null;
4         try {
5             // Load MySQL JDBC driver
6             Class.forName("com.mysql.jdbc.Driver");
7             // Create connection to database
8             con = DriverManager.getConnection("jdbc:mysql://localhost/hotelmanagement", "root", "");
9         } catch (ClassNotFoundException | SQLException ex) {
10             ex.printStackTrace();
11         }
12         return con;
13     }
14 }
```

3. Bulider Pattern with Factory Pattern :

1. createHotelRoom Method:

- Purpose: Creates a new HotelRoom object using the builder pattern.
- Parameters: Room number, room type, bed type, and amount.
- Returns: A HotelRoom object with the provided details.

```
1 // Create a HotelRoom object using the Builder
2 public static HotelRoom createHotelRoom(String roomNo, String roomType, String bedType, int amount) {
3     return new HotelRoom.HotelRoomBuilder()
4         .setRoomNo(roomNo)
5         .setRoomType(roomType)
6         .setBedType(bedType)
7         .setAmount(amount)
8         .build();
9 }
10
```

2. fetchHotelRoomFromDatabase Method:

- Purpose: Fetches a HotelRoom from the database based on the provided room number.
- Parameters: Connection object (for database access) and room number.
- Returns: A HotelRoom object populated with the data from the database or null if not found.

```
1 // Fetch a HotelRoom from the database
2 public static HotelRoom fetchHotelRoomFromDatabase(Connection con, String roomNo) {
3     try {
4         PreparedStatement pst = con.prepareStatement("SELECT * FROM room WHERE rid = ?");
5         pst.setString(1, roomNo);
6         ResultSet rs = pst.executeQuery();
7
8         if (rs.next()) {
9             return new HotelRoom.HotelRoomBuilder()
10                 .setRoomNo(rs.getString("rid"))
11                 .setRoomType(rs.getString("rtype"))
12                 .setBedType(rs.getString("btype"))
13                 .setAmount(rs.getInt("amount"))
14                 .build();
15         }
16     } catch (SQLException ex) {
17         ex.printStackTrace();
18     }
19     return null;
20 }
21
```

3. loadRoomsIntoTable Method:

- Purpose: Loads all hotel room data from the database and displays it in a JTable.
- Parameters: Connection object and a JTable to display the data.
- Returns: None, but updates the JTable with room details.

```
1 // Load all rooms into a JTable
2 public static void loadRoomsIntoTable(Connection con, JTable table) {
3     DefaultTableModel model = (DefaultTableModel) table.getModel();
4     try {
5         PreparedStatement pst = con.prepareStatement("SELECT * FROM room");
6         ResultSet rs = pst.executeQuery();
7
8         model.setRowCount(0); // Clear existing rows
9         while (rs.next()) {
10             model.addRow(new Object[]{
11                 rs.getString("rid"),
12                 rs.getString("rtype"),
13                 rs.getString("btype"),
14                 rs.getInt("amount")
15             });
16         }
17     } catch (SQLException ex) {
18         ex.printStackTrace();
19     }
20 }
21
```

4. saveHotelRoomToDatabase Method:

- Purpose: Saves a HotelRoom object to the database.
- Parameters: Connection object and the HotelRoom to be saved.
- Returns: None, but shows a success message after saving.

```
1 // Save a HotelRoom object to the database
2 public static void saveHotelRoomToDatabase(Connection con, HotelRoom room) {
3     try {
4         PreparedStatement pst = con.prepareStatement(
5             "INSERT INTO room(rid, rtype, btype, amount) VALUES (?, ?, ?, ?)"
6         );
7         pst.setString(1, room.getRoomNo());
8         pst.setString(2, room.getRoomType());
9         pst.setString(3, room.getBedType());
10        pst.setInt(4, room.getAmount());
11        pst.executeUpdate();
12        JOptionPane.showMessageDialog(null, "Room Added Successfully!");
13    } catch (SQLException ex) {
14        ex.printStackTrace();
15    }
16 }
17
```

5. addHotelRoom Method:

- Purpose: Adds a new hotel room to the database with the provided details.
- Parameters: Connection object and room details (room number, room type, bed type, and amount).
- Returns: None, but shows a success message after adding the room.

```
1 // Save a HotelRoom object to the database
2 public static void saveHotelRoomToDatabase(Connection con, HotelRoom room) {
3     try {
4         PreparedStatement pst = con.prepareStatement(
5             "INSERT INTO room(rid, rtype, btype, amount) VALUES (?, ?, ?, ?)"
6         );
7         pst.setString(1, room.getRoomNo());
8         pst.setString(2, room.getRoomType());
9         pst.setString(3, room.getBedType());
10        pst.setInt(4, room.getAmount());
11        pst.executeUpdate();
12        JOptionPane.showMessageDialog(null, "Room Added Successfully!");
13    } catch (SQLException ex) {
14        ex.printStackTrace();
15    }
16 }
17
```

6. updateHotelRoom Method:

- Purpose: Updates the details of an existing hotel room in the database.
- Parameters: Connection object, room number, and updated room details.
- Returns: None, but shows a success message after updating the room.

```
1 // Update a HotelRoom in the database
2 public static void updateHotelRoom(Connection con, String roomno, String roomtype, String bedtype, String amount) {
3     try {
4         PreparedStatement pst = con.prepareStatement(
5             "UPDATE room SET rtype = ?, btype = ?, amount = ? WHERE rid = ?"
6         );
7         pst.setString(1, roomtype);
8         pst.setString(2, bedtype);
9         pst.setInt(3, Integer.parseInt(amount)); // Convert amount to integer
10        pst.setString(4, roomno);
11        pst.executeUpdate();
12        JOptionPane.showMessageDialog(null, "Room Updated Successfully!");
13    } catch (SQLException ex) {
14        ex.printStackTrace();
15    }
16 }
17
```

7. deleteHotelRoom Method:

- Purpose: Deletes a hotel room from the database based on the provided room number.
- Parameters: Connection object and room number.
- Returns: None, but shows a success message after deletion.

```
1 // Delete a HotelRoom from the database
2 public static void deleteHotelRoom(Connection con, String roomno) {
3     try {
4         PreparedStatement pst = con.prepareStatement(
5             "DELETE FROM room WHERE rid = ?"
6         );
7         pst.setString(1, roomno);
8         pst.executeUpdate();
9         JOptionPane.showMessageDialog(null, "Room Deleted Successfully!");
10    } catch (SQLException ex) {
11        ex.printStackTrace();
12    }
13 }
14
```

8. generateAutoID Method:

- Purpose: Generates a new room ID based on the highest existing ID in the database.
- Parameters: Connection object and a JLabel to display the generated room ID.
- Returns: None, but updates the JLabel with the generated ID (e.g., R0001, R0002, etc.).

```
1 // Generate Auto ID for Room
2 public static void generateAutoID(Connection con, JLabel label) {
3     try {
4         PreparedStatement pst = con.prepareStatement("SELECT MAX(rid) FROM room");
5         ResultSet rs = pst.executeQuery();
6
7         if (rs.next()) {
8             String lastID = rs.getString(1);
9             if (lastID == null) {
10                label.setText("R0001");
11            } else {
12                int newID = Integer.parseInt(lastID.substring(1)) + 1;
13                label.setText("R" + String.format("%04d", newID)); // Format to R0002, R0003, etc.
14            }
15        }
16    } catch (SQLException ex) {
17        ex.printStackTrace();
18    }
19 }
20
```

Class Overview

- The HotelRoom class demonstrates the implementation of the Builder Pattern to simplify the creation of HotelRoom objects with a flexible and readable approach.

Key Features

1. Encapsulation of Construction Logic:
2. Static Inner Builder Class:
3. Immutability of Built Objects:
4. Separation of Concerns:

```
1 // For Builder Pattern
2
3 public class HotelRoom {
4     private String roomNo;
5     private String roomType;
6     private String bedType;
7     private int amount;
8
9     // Private constructor
10    private HotelRoom(HotelRoomBuilder builder) {
11        this.roomNo = builder.roomNo;
12        this.roomType = builder.roomType;
13        this.bedType = builder.bedType;
14        this.amount = builder.amount;
15    }
16
17    // Getters
18    public String getRoomNo() { return roomNo; }
19    public String getRoomType() { return roomType; }
20    public String getBedType() { return bedType; }
21    public int getAmount() { return amount; }
22
23    // Static Inner Builder Class
24    public static class HotelRoomBuilder {
25        private String roomNo;
26        private String roomType;
27        private String bedType;
28        private int amount;
29
30        public HotelRoomBuilder setRoomNo(String roomNo) {
31            this.roomNo = roomNo;
32            return this;
33        }
34
35        public HotelRoomBuilder setRoomType(String roomType) {
36            this.roomType = roomType;
37            return this;
38        }
39
40        public HotelRoomBuilder setBedType(String bedType) {
41            this.bedType = bedType;
42            return this;
43        }
44
45        public HotelRoomBuilder setAmount(int amount) {
46            this.amount = amount;
47            return this;
48        }
49
50        public HotelRoom build() {
51            return new HotelRoom(this);
52        }
53    }
54 }
```


4. Prototype Pattern :

Class Overview

- This code demonstrates the Prototype Design Pattern, allowing the creation of new objects by copying existing ones instead of creating them from scratch. It is useful when object creation is resource-intensive.

Key Features

1. **Prototype Interface:**
 - The Prototype interface defines the contract for cloning objects with a single method, `clone()`.
 - This ensures that all implementing classes can provide a mechanism to duplicate their instances.
2. **Implementation in UserPrototype:**
 - The UserPrototype class implements the Prototype interface and provides the logic for creating a copy of its instances through the `clone()` method.
3. **Deep Copy Mechanism:**
 - The `clone()` method in UserPrototype creates a deep copy, meaning a new UserPrototype object is instantiated with the same field values, ensuring independence from the original object.
4. **Encapsulation of Object Duplication:**
 - The cloning logic is encapsulated within the class, providing an easy and consistent way to duplicate objects without exposing internal details.
5. **Usage of Getters and Setters:**
 - The class provides getters and setters for its fields (name, username, and password), enabling controlled access and modifications.
6. **Constructor for Initialization:**
 - The UserPrototype class includes a constructor to initialize the fields during object creation.

```
1  //ProtoType Pattern
2
3
4  public class UserPrototype implements Prototype {
5      private String name;
6      private String username;
7      private String password;
8
9      // Constructor
10     public UserPrototype(String name, String username, String password) {
11         this.name = name;
12         this.username = username;
13         this.password = password;
14     }
15
16     // Getters and Setters
17     public String getName() {
18         return name;
19     }
20 }
```

5. Proxy Pattern :

Class Overview

- The AuthenticationProxy class demonstrates the Proxy Design Pattern, which provides a surrogate or placeholder for another object to control access, add functionality, or enhance behavior.

Key Features

- 1 Proxy Role:
 - AuthenticationProxy serves as an intermediary between the client and the RealAuthenticationService, implementing the same AuthenticationService interface.
- 2 Encapsulation of Real Object:
 - The AuthenticationProxy encapsulates the RealAuthenticationService and delegates the authenticate method call to it.
- 3 Enhancing Functionality:
 - The proxy adds pre-processing logic (e.g., logging, input validation) before forwarding requests to the real service.

- 4 Access Control:
 - Includes checks (e.g., ensuring non-null credentials) to restrict or validate access before invoking the real service.
- 5 Interchangeability:
 - The AuthenticationProxy implements the AuthenticationService interface, making it interchangeable with the RealAuthenticationService in client code.
- 6 Separation of Concerns:
 - Logging and input validation are handled by the proxy, while the real service focuses solely on actual authentication logic.

```
1 // Proxy Pattern
2
3 public class AuthenticationProxy implements AuthenticationService {
4     private RealAuthenticationService realService;
5
6     public AuthenticationProxy() {
7         this.realService = new RealAuthenticationService();
8     }
9
10    @Override
11    public boolean authenticate(String username, String password) {
12        // Add logging or access control
13        System.out.println("Authentication attempt for user: " + username);
14        if (username == null || password == null) {
15            System.out.println("Invalid credentials provided.");
16            return false;
17        }
18        return realService.authenticate(username, password);
19    }
20 }
21
```