OOAD ASSIGNMENT

Kushagra Agarwal - PES2UG22CS275

1. **Singleton Pattern – Database Connection Manager Concept:**
   Ensures a single instance of a class, providing a global access point.

Questions:

1. Explain how the Singleton pattern can be used to manage database connections in an application.

Solution: This could be done using :

Single Instance, Shared Resource: Only one database connection is created for the entire application, ensuring all components share the same connection instance.
Global Access, Unified Point: Provides a central, globally accessible point to interact with the single database connection.
Optimized Resource Usage: Prevents the needless creation of multiple database connections, conserving resources and improving application performance.
Centralized Management: Connection settings and credentials are managed in a single, central location, streamlining configuration and maintenance.

2. Implement a DatabaseConnection class using the Singleton pattern. Ensure that only one connection instance is created.

Solution:

```
public class DatabaseConnection {

    private static DatabaseConnection instance;
    private Connection connection; // java.sql.Connection (import it!)
    private String dbUrl;
    private String username;
    private String password;

    // Private constructor to prevent external instantiation
    private DatabaseConnection(String dbUrl, String username, String password) {
        this.dbUrl = dbUrl;
        this.username = username;
```

```
        this.password = password;
        try {
            // Load the JDBC driver (replace with your specific driver)
            Class.forName("com.mysql.cj.jdbc.Driver"); // Example: MySQL
            // Establish the connection
            this.connection = DriverManager.getConnection(dbUrl, username, password);
            System.out.println("Database connection established!");
        } catch (ClassNotFoundException e) {
            System.err.println("JDBC Driver not found: " + e.getMessage());
            // Handle the exception appropriately (e.g., throw a runtime exception)
            throw new RuntimeException("Failed to load JDBC driver", e);
        } catch (SQLException e) {
            System.err.println("Database connection failed: " + e.getMessage());
            // Handle the exception appropriately (e.g., throw a runtime exception)
            throw new RuntimeException("Failed to connect to the database", e);
        }
    }

    // Public static method to get the instance
    public static synchronized DatabaseConnection getInstance(String dbUrl, String username,
String password) {
        if (instance == null) {
            instance = new DatabaseConnection(dbUrl, username, password);
        }
        return instance;
    }

    // Public method to get the connection
    public Connection getConnection() {
        return connection;
    }

    // Method to close the connection (call when the application shuts down)
    public void closeConnection() {
        try {
            if (connection != null && !connection.isClosed()) {
                connection.close();
```

```
            System.out.println("Database connection closed.");
        }
    } catch (SQLException e) {
        System.err.println("Error closing database connection: " + e.getMessage());
    }
}

// Example usage (for demonstration)
public static void main(String[] args) {
    // Get the Singleton instance (with your actual credentials)
    DatabaseConnection db = DatabaseConnection.getInstance("jdbc:mysql://localhost:3306/
mydatabase", "myuser", "mypassword");

    // Get the connection
    Connection conn = db.getConnection();

    // Now you can use the 'conn' object to execute queries, etc.
    // Example (very basic):
    try {
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM mytable");
        while (rs.next()) {
            System.out.println(rs.getString("column1")); // Example column
        }
    } catch (SQLException e) {
        System.err.println("Error executing query: " + e.getMessage());
    } finally {
        // close resources
    }

    // Close the connection when you're done (usually on application shutdown)
    db.closeConnection();
}
}
```

3. Modify the DatabaseConnection class to support lazy initialization.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;

public class DatabaseConnection {

    private static DatabaseConnection instance;
    private Connection connection;
    private String dbUrl;
    private String username;
    private String password;

    // Private constructor
    private DatabaseConnection(String dbUrl, String username, String password) {
        this.dbUrl = dbUrl;
        this.username = username;
        this.password = password;
    }

    // Public static method to get the instance (Lazy Initialization)
    public static DatabaseConnection getInstance(String dbUrl, String username, String password)
{
        if (instance == null) {
            synchronized (DatabaseConnection.class) { // Double-checked locking
                if (instance == null) {
                    instance = new DatabaseConnection(dbUrl, username, password);
                }
            }
        }
        return instance;
    }

    // Public method to get the connection (Lazy Initialization)
```

```
public Connection getConnection() {
    if (connection == null) {
        synchronized (this) {
            if (connection == null) {
                try {
                    // Load the JDBC driver
                    Class.forName("com.mysql.cj.jdbc.Driver"); // Replace with your driver
                    // Establish the connection
                    connection = DriverManager.getConnection(dbUrl, username, password);
                    System.out.println("Database connection established!");
                } catch (ClassNotFoundException e) {
                    System.err.println("JDBC Driver not found: " + e.getMessage());
                    throw new RuntimeException("Failed to load JDBC driver", e);
                } catch (SQLException e) {
                    System.err.println("Database connection failed: " + e.getMessage());
                    throw new RuntimeException("Failed to connect to the database", e);
                }
            }
        }
    }
    return connection;
}

// Method to close the connection
public void closeConnection() {
    try {
        if (connection != null && !connection.isClosed()) {
            connection.close();
            System.out.println("Database connection closed.");
            connection = null; // Important: set to null after closing
        }
    } catch (SQLException e) {
        System.err.println("Error closing database connection: " + e.getMessage());
    }
}

// Example usage
```

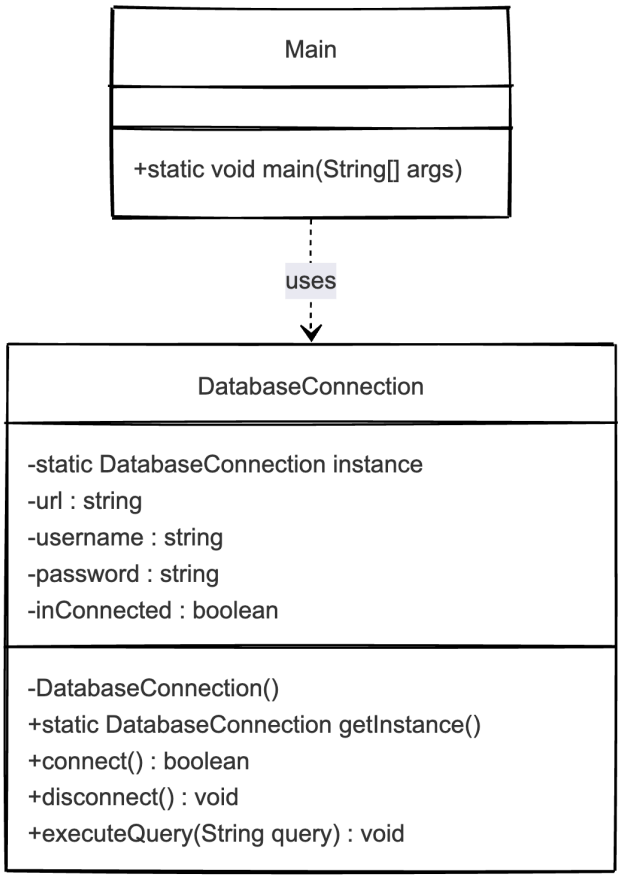
```
    public static void main(String[] args) {
        // Get the Singleton instance (connection is not yet created)
        DatabaseConnection db = DatabaseConnection.getInstance("jdbc:mysql://localhost:3306/
mydatabase", "myuser", "mypassword");

        // Get the connection (connection is created here, on first access)
        Connection conn = db.getConnection();

        // Example query
        try (Statement stmt = conn.createStatement(); //Try with resources
             ResultSet rs = stmt.executeQuery("SELECT * FROM mytable")) {  //Try with resources
            while (rs.next()) {
                System.out.println(rs.getString("column1"));
            }
        } catch (SQLException e) {
            System.err.println("Error executing query: " + e.getMessage());
        }

        // Close the connection
        db.closeConnection();
    }
}
```

4.What are the potential issues with using the Singleton pattern in a multi-threaded environment, and how can they be resolved?

Solution:
Race Conditions (Instance Creation): Multiple threads might create multiple Singleton instances simultaneously.
Data Corruption (Instance Methods): Concurrent access to instance methods can lead to data corruption if not synchronized.
Visibility Issues: Changes to Singleton state might not be visible to all threads.
Serialization Issues: Serialization/deserialization can create multiple instances.
Reflection Issues: Reflection can bypass the private constructor and create new instances.

5. Represent using class diagram

2.



Factory Pattern – Notification System Concept:
Provides an interface for creating different types of objects dynamically.

Questions:
1. How does the Factory pattern improve code maintainability and flexibility?
Solution:
● Decoupling client code from specific implementations
● Centralizing object creation logic
● Simplifying adding new notification types
● Enabling consistent error handling
● Supporting dependency injection
● Facilitating unit testing with mock objects

2. Implement a Notification interface with a sendNotification() method.
Create EmailNotification, SMSNotification, and PushNotification classes that implement this interface.

Solution:

```
// Notification Interface
interface Notification {
    void sendNotification(String message, String recipient);
}

// Email Notification Class
class EmailNotification implements Notification {
    @Override
    public void sendNotification(String message, String recipient) {
        System.out.println("Sending email to: " + recipient);
        System.out.println("Subject: Notification");
        System.out.println("Message: " + message);
        System.out.println("Email sent successfully!\n");
    }
}

// SMS Notification Class
class SMSNotification implements Notification {
    @Override
```

```
    public void sendNotification(String message, String recipient) {
        System.out.println("Sending SMS to: " + recipient);
        System.out.println("Message: " + message);
        System.out.println("SMS sent successfully!\n");
    }
}

// Push Notification Class
class PushNotification implements Notification {
    @Override
    public void sendNotification(String message, String recipient) {
        System.out.println("Sending push notification to device ID: " + recipient);
        System.out.println("Message: " + message);
        System.out.println("Push notification sent successfully!\n");
    }
}

// Example Usage
public class NotificationExample {
    public static void main(String[] args) {
        Notification emailNotification = new EmailNotification();
        Notification smsNotification = new SMSNotification();
        Notification pushNotification = new PushNotification();

        emailNotification.sendNotification("Your order has been shipped!", "user@example.com");
        smsNotification.sendNotification("Reminder: Your appointment is tomorrow.",
"+15551234567");
        pushNotification.sendNotification("New message received!", "device12345");
    }
}
```

3. Design a NotificationFactory that returns the correct notification object based on user input (e.g., "email", "sms", or "push").

Solution:

```
// Notification Interface (same as before)
interface Notification {
```

```
    void sendNotification(String message, String recipient);
}

// Concrete Notification Classes (same as before)
class EmailNotification implements Notification {
    @Override
    public void sendNotification(String message, String recipient) {
        System.out.println("Sending email to: " + recipient);
        System.out.println("Subject: Notification");
        System.out.println("Message: " + message);
        System.out.println("Email sent successfully!\n");
    }
}

class SMSNotification implements Notification {
    @Override
    public void sendNotification(String message, String recipient) {
        System.out.println("Sending SMS to: " + recipient);
        System.out.println("Message: " + message);
        System.out.println("SMS sent successfully!\n");
    }
}

class PushNotification implements Notification {
    @Override
    public void sendNotification(String message, String recipient) {
        System.out.println("Sending push notification to device ID: " + recipient);
        System.out.println("Message: " + message);
        System.out.println("Push notification sent successfully!\n");
    }
}

// Notification Factory
class NotificationFactory {
    public Notification createNotification(String channel) {
        if (channel == null || channel.isEmpty()) {
            return null;  // Or throw an exception, depending on desired behavior
```

```
        }
        switch (channel.toLowerCase()) {
            case "email":
                return new EmailNotification();
            case "sms":
                return new SMSNotification();
            case "push":
                return new PushNotification();
            default:
                // Handle unknown notification type.  Either:
                // 1. Return null, or
                // 2. Throw an IllegalArgumentException or custom exception.
                // I'll throw an exception here for clarity.
                throw new IllegalArgumentException("Unknown notification channel: " + channel);
        }
    }
}

// Example Usage
public class FactoryExample {
    public static void main(String[] args) {
        NotificationFactory notificationFactory = new NotificationFactory();

        // Get notification type from user input (e.g., command line, web form)
        String channel = "email"; // Example input

        try {
            Notification notification = notificationFactory.createNotification(channel);

            if (notification != null) {
                notification.sendNotification("Hello!", "recipient@example.com");
            } else {
                System.out.println("Invalid notification channel.");
            }
        } catch (IllegalArgumentException e) {
            System.err.println("Error: " + e.getMessage()); // Properly handle the exception.
        }
```

```
    }
}
```

4. Modify the factory to include an additional method for sending a batch of notifications of different types.

Solution:

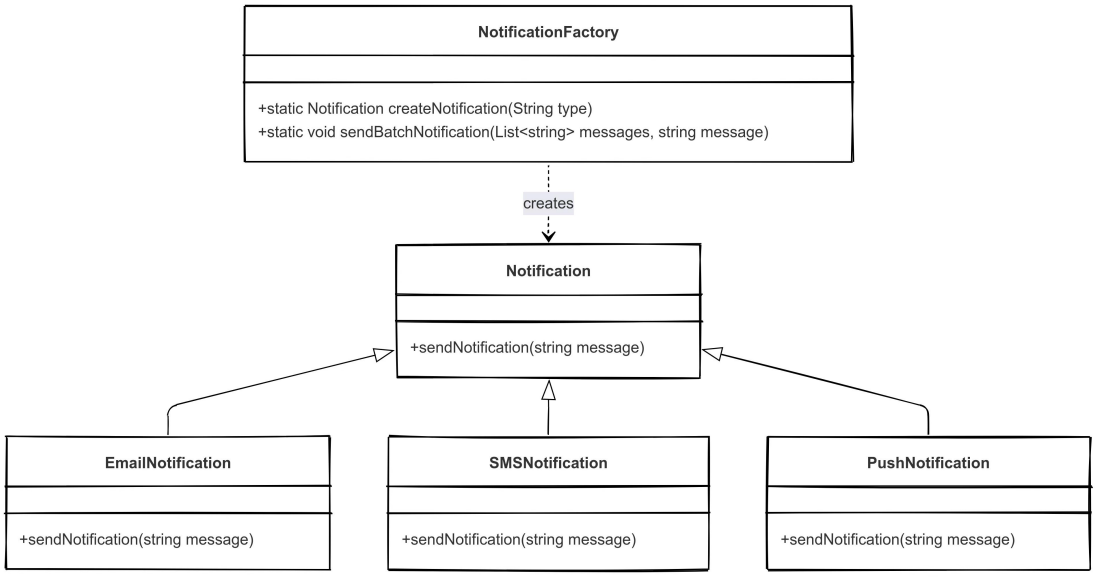
```
import java.util.List;

// Notification Interface (Same as before)
interface Notification {
    void sendNotification(String message, String recipient);
}

// Email Notification Class (Same as before)
class EmailNotification implements Notification {
    @Override
    public void sendNotification(String message, String recipient) {
        System.out.println("Sending email to: " + recipient);
        System.out.println("Subject: Notification");
        System.out.println("Message: " + message);
        System.out.println("Email sent successfully!\n");
    }
}

// SMS Notification Class (Same as before)
class SMSNotification implements Notification {
    @Override
    public void sendNotification(String message, String recipient) {
        System.out.println("Sending SMS to: " + recipient);
        System.out.println("Message: " + message);
        System.out.println("SMS sent successfully!\n");
    }
}

// Push Notification Class (Same as before)
class PushNotification implements Notification {
```

```
    @Override
    public void sendNotification(String message, String recipient) {
        System.out.println("Sending push notification to device ID: " + recipient);
        System.out.println("Message: " + message);
        System.out.println("Push notification sent successfully!\n");
    }
}

// Data structure to hold notification details
class NotificationDetails {
    private String type;
    private String message;
    private String recipient;

    public NotificationDetails(String type, String message, String recipient) {
        this.type = type;
        this.message = message;
        this.recipient = recipient;
    }

    public String getType() {
        return type;
    }

    public String getMessage() {
        return message;
    }

    public String getRecipient() {
        return recipient;
    }
}


// Notification Factory Class
class NotificationFactory {
```

```
    public Notification createNotification(String channel) {
        if (channel == null || channel.isEmpty()) {
            throw new IllegalArgumentException("Notification type cannot be empty or null.");
        }
        switch (channel.toLowerCase()) {
            case "email":
                return new EmailNotification();
            case "sms":
                return new SMSNotification();
            case "push":
                return new PushNotification();
            default:
                throw new IllegalArgumentException("Unknown channel " + channel);
        }
    }

    // Method to send a batch of notifications
    public void sendBatchNotifications(List<NotificationDetails> notifications) {
        for (NotificationDetails notificationDetail : notifications) {
            try {
                Notification notification = createNotification(notificationDetail.getType());
                notification.sendNotification(notificationDetail.getMessage(),
notificationDetail.getRecipient());
            } catch (IllegalArgumentException e) {
                System.err.println("Error processing notification of type " +
notificationDetail.getType() + ": " + e.getMessage());
            } catch (Exception e) {
                System.err.println("Unexpected error sending notification of type " +
notificationDetail.getType() + ": " + e.getMessage());
            }
        }
    }
}

// Example Usage
public class NotificationExample {
    public static void main(String[] args) {
```

```
        NotificationFactory notificationFactory = new NotificationFactory();

        // Create a list of notification details
        List<NotificationDetails> notificationBatch = List.of(
                new NotificationDetails("email", "Welcome!", "user1@example.com"),
                new NotificationDetails("sms", "Your code is 5678", "+15552223333"),
                new NotificationDetails("push", "New update available", "device456"),
                new NotificationDetails("invalid", "This will cause an error", "someRecipient") //
Invalid type
        );

        // Send the batch of notifications
        notificationFactory.sendBatchNotifications(notificationBatch);
    }
}
```

5. Represent using class diagram



3. Builder Pattern – Computer Assembly System Concept:
Separates object construction from its representation, useful for complex object creation.
Questions:
1. Why is the Builder pattern useful when creating complex objects like a computer system?
Solution:
● Encapsulation of Construction Logic: Keeps object creation separate from representation.
● Step-by-Step Configuration: Users can configure a computer piece by piece
● Immutable Final Object: Ensures that once a Computer object is built, it cannot be modified.
● Predefined Configurations: The ComputerDirector can create standard builds
2. Implement a Computer class with attributes: processor, RAM, storage, and graphicsCard.
Solution:

```
public class Computer {

    private String processor;
    private int ram;        // RAM in GB
    private int storage;    // Storage in GB
    private String graphicsCard;

    // Constructor
    public Computer(String processor, int ram, int storage, String graphicsCard) {
        this.processor = processor;
        this.ram = ram;
        this.storage = storage;
        this.graphicsCard = graphicsCard;
    }

    // Getters (Optional but Recommended)
    public String getProcessor() {
        return processor;
    }

    public int getRam() {
        return ram;
    }

    public int getStorage() {
```

```
    return storage;
  }

  public String getGraphicsCard() {
    return graphicsCard;
  }

  // Setters (Optional - Use only if you need to modify the attributes after creation)
  public void setProcessor(String processor) {
    this.processor = processor;
  }

  public void setRam(int ram) {
    this.ram = ram;
  }

  public void setStorage(int storage) {
    this.storage = storage;
  }

  public void setGraphicsCard(String graphicsCard) {
    this.graphicsCard = graphicsCard;
  }

  // Method to display the computer's specifications
  public void displaySpecs() {
    System.out.println("Computer Specifications:");
    System.out.println("Processor: " + processor);
    System.out.println("RAM: " + ram + " GB");
    System.out.println("Storage: " + storage + " GB");
    System.out.println("Graphics Card: " + graphicsCard);
  }

  // Example usage
  public static void main(String[] args) {
    Computer myComputer = new Computer("Intel Core i7", 16, 512, "NVIDIA GeForce RTX
3060");
```

```
        myComputer.displaySpecs();

        //Another example using setters
        Computer laptop = new Computer("", 0, 0, ""); //Create empty Computer object
        laptop.setProcessor("AMD Ryzen 5");
        laptop.setRam(8);
        laptop.setStorage(256);
        laptop.setGraphicsCard("AMD Radeon Vega 8");
        System.out.println("Laptop Specifications:");
        laptop.displaySpecs();

    }
}
```

3. Create a ComputerBuilder class to allow step-by-step customization of the computer's components.

Solution:

```
public class Computer {

    private String processor;
    private int ram;
    private int storage;
    private String graphicsCard;

    // Private constructor (used by the builder)
    private Computer(String processor, int ram, int storage, String graphicsCard) {
        this.processor = processor;
        this.ram = ram;
        this.storage = storage;
        this.graphicsCard = graphicsCard;
    }

    public String getProcessor() {
        return processor;
    }

    public int getRam() {
```

```
    return ram;
}

public int getStorage() {
    return storage;
}

public String getGraphicsCard() {
    return graphicsCard;
}

public void displaySpecs() {
    System.out.println("Computer Specifications:");
    System.out.println("Processor: " + processor);
    System.out.println("RAM: " + ram + " GB");
    System.out.println("Storage: " + storage + " GB");
    System.out.println("Graphics Card: " + graphicsCard);
}

// Builder Class
public static class ComputerBuilder {
    private String processor;
    private int ram;
    private int storage;
    private String graphicsCard;

    public ComputerBuilder() {
        // You can initialize with default values here if needed.  For example:
        // this.ram = 8;
    }

    public ComputerBuilder processor(String processor) {
        this.processor = processor;
        return this;  // Return the builder for chaining
    }

    public ComputerBuilder ram(int ram) {
```

```
            this.ram = ram;
            return this;
        }

        public ComputerBuilder storage(int storage) {
            this.storage = storage;
            return this;
        }

        public ComputerBuilder graphicsCard(String graphicsCard) {
            this.graphicsCard = graphicsCard;
            return this;
        }

        public Computer build() {
            return new Computer(processor, ram, storage, graphicsCard);
        }
    }

    // Example usage
    public static void main(String[] args) {
        Computer myComputer = new ComputerBuilder()
                .processor("Intel Core i7")
                .ram(16)
                .storage(512)
                .graphicsCard("NVIDIA GeForce RTX 3060")
                .build();

        myComputer.displaySpecs();

        Computer basicComputer = new ComputerBuilder()
                .ram(8)
                .storage(256)
                .build();
        basicComputer.displaySpecs(); //Will print nulls for other values as they were not specified.
    }
}
```

4. Add a ComputerDirector class that provides predefined configurations like "Gaming PC" and "Office PC".

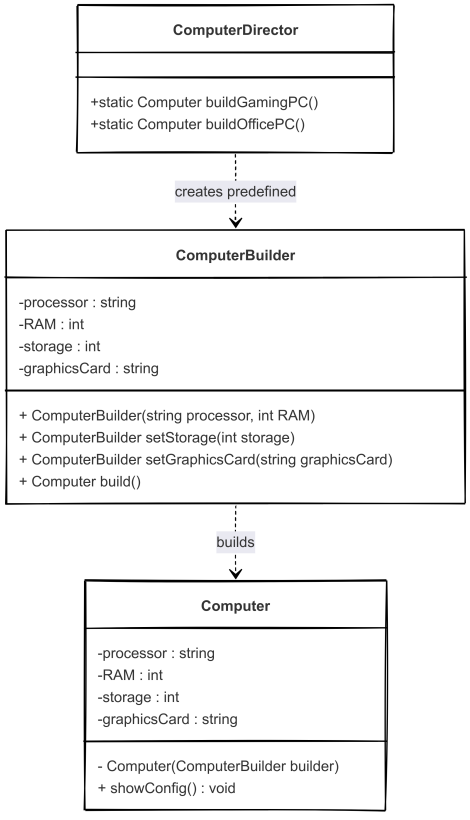
```
public class Computer {

    private String processor;
    private int ram;
    private int storage;
    private String graphicsCard;

    private Computer(String processor, int ram, int storage, String graphicsCard) {
        this.processor = processor;
        this.ram = ram;
        this.storage = storage;
        this.graphicsCard = graphicsCard;
    }

    public String getProcessor() {
        return processor;
    }

    public int getRam() {
        return ram;
    }

    public int getStorage() {
        return storage;
    }

    public String getGraphicsCard() {
        return graphicsCard;
    }

    public void displaySpecs() {
        System.out.println("Computer Specifications:");
        System.out.println("Processor: " + processor);
        System.out.println("RAM: " + ram + " GB");
```

```
    System.out.println("Storage: " + storage + " GB");
    System.out.println("Graphics Card: " + graphicsCard);
}

public static class ComputerBuilder {
    private String processor;
    private int ram;
    private int storage;
    private String graphicsCard;

    public ComputerBuilder() {}

    public ComputerBuilder processor(String processor) {
        this.processor = processor;
        return this;
    }

    public ComputerBuilder ram(int ram) {
        this.ram = ram;
        return this;
    }

    public ComputerBuilder storage(int storage) {
        this.storage = storage;
        return this;
    }

    public ComputerBuilder graphicsCard(String graphicsCard) {
        this.graphicsCard = graphicsCard;
        return this;
    }

    public Computer build() {
        return new Computer(processor, ram, storage, graphicsCard);
    }
}
```

```
// ComputerDirector Class
public static class ComputerDirector {

    public Computer createGamingPC(ComputerBuilder builder) {
        return builder.processor("Intel Core i9")
                      .ram(32)
                      .storage(1000)
                      .graphicsCard("NVIDIA GeForce RTX 4080")
                      .build();
    }

    public Computer createOfficePC(ComputerBuilder builder) {
        return builder.processor("Intel Core i5")
                      .ram(16)
                      .storage(500)
                      .graphicsCard("Integrated Graphics")
                      .build();
    }

    public Computer createBasicPC(ComputerBuilder builder) {
        return builder.processor("Intel Celeron")
                .ram(8)
                .storage(256)
                .graphicsCard("Integrated Graphics")
                .build();
    }
}

// Example usage
public static void main(String[] args) {
    ComputerDirector director = new ComputerDirector();
    ComputerBuilder builder = new ComputerBuilder();

    Computer gamingPC = director.createGamingPC(builder);
    System.out.println("Gaming PC:");
    gamingPC.displaySpecs();
```

```
        builder = new ComputerBuilder(); // Reset the builder
        Computer officePC = director.createOfficePC(builder);
        System.out.println("\nOffice PC:");
        officePC.displaySpecs();

        builder = new ComputerBuilder(); // Reset the builder
        Computer basicPC = director.createBasicPC(builder);
        System.out.println("\nBasic PC:");
        basicPC.displaySpecs();

        //Or chain them on one line like this
        System.out.println("\nGaming PC:");
        director.createGamingPC(new ComputerBuilder()).displaySpecs();

    }
}
```

5. Represent using class diagram



4. Prototype Pattern – User Profile Cloning Concept:
Creates new objects by cloning existing ones instead of instantiating them from scratch.

Questions:
1. How does the Prototype pattern help in reducing object creation time
● Reduces initialization cost by avoiding expensive setup operations.
● Faster object creation by copying an existing object instead of rebuilding from scratch.
● Preserves object state to ensure new instances inherit attributes and configurations.
● Enhances flexibility by allowing easy modification of cloned objects without affecting the original.

2. Implement a UserProfile class with attributes: name, email, preferences, and implement a cloning mechanism.

Solution:

```
import java.util.HashMap;
import java.util.Map;

public class UserProfile implements Cloneable {

    private String name;
    private String email;
    private Map<String, String> preferences;  // Using Map for flexibility

    public UserProfile(String name, String email) {
        this.name = name;
        this.email = email;
        this.preferences = new HashMap<>();
    }

    // Getters and Setters
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Map<String, String> getPreferences() {
        //return preferences; //Consider returning a copy of the map
        return new HashMap<>(preferences); //Returns a deep copy, so modifications to the original
don't affect the clone.
    }

    public void setPreference(String key, String value) {
        this.preferences.put(key, value);
    }

    // Cloning Mechanism (Override clone() method) - Shallow Copy
    @Override
    public UserProfile clone() {
        try {
            return (UserProfile) super.clone();  // Uses the Object.clone() method (shallow copy)
        } catch (CloneNotSupportedException e) {
            System.err.println("Cloning not supported: " + e.getMessage());
            return null; // Or throw a RuntimeException
        }
    }

    // Deep Copy (Alternative to Shallow Copy)
    public UserProfile deepClone() {
        UserProfile clonedProfile = new UserProfile(this.name, this.email); // Copy basic attributes

        // Deep copy the preferences map
        for (Map.Entry<String, String> entry : this.preferences.entrySet()) {
```

```
        clonedProfile.setPreference(entry.getKey(), entry.getValue()); // Copy each entry
    }
    return clonedProfile;
}

// Display Profile Information
public void displayProfile() {
    System.out.println("User Profile:");
    System.out.println("Name: " + name);
    System.out.println("Email: " + email);
    System.out.println("Preferences:");
    for (Map.Entry<String, String> entry : preferences.entrySet()) {
        System.out.println("  " + entry.getKey() + ": " + entry.getValue());
    }
}

// Example Usage
public static void main(String[] args) {
    UserProfile originalProfile = new UserProfile("John Doe", "john.doe@example.com");
    originalProfile.setPreference("theme", "dark");
    originalProfile.setPreference("language", "en");

    System.out.println("Original Profile:");
    originalProfile.displayProfile();

    // Shallow Copy
    UserProfile clonedProfile = originalProfile.clone();
    clonedProfile.setName("Jane Doe");
    clonedProfile.setPreference("theme", "light");

    System.out.println("\nCloned Profile (Shallow Copy):");
    clonedProfile.displayProfile();
    System.out.println("\nOriginal Profile after shallow copy changes:");
    originalProfile.displayProfile(); //The theme is updated on this object as well.

    // Deep Copy
    UserProfile deepClonedProfile = originalProfile.deepClone();
```

```
        deepClonedProfile.setName("Peter Pan");
        deepClonedProfile.setPreference("language", "fr");

        System.out.println("\nDeep Cloned Profile:");
        deepClonedProfile.displayProfile();
        System.out.println("\nOriginal Profile after deep copy changes:");
        originalProfile.displayProfile(); //Original profile is untouched.
    }
}
```

3. Modify the UserProfile class to allow deep cloning of complex attributes such as preferences, which is a HashMap.

Solution:

```
import java.util.HashMap;
import java.util.Map;

public class UserProfile implements Cloneable {

    private String name;
    private String email;
    private Map<String, String> preferences;

    public UserProfile(String name, String email) {
        this.name = name;
        this.email = email;
        this.preferences = new HashMap<>(); // Initialize in the constructor
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
```

```
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Map<String, String> getPreferences() {
        return new HashMap<>(preferences);  // Return a copy to prevent external modification
    }

    public void setPreference(String key, String value) {
        this.preferences.put(key, value);
    }

    // Override clone() method for deep cloning
    @Override
    public UserProfile clone() {
        try {
            UserProfile clonedProfile = (UserProfile) super.clone(); // Shallow copy first

            // Deep copy the preferences Map
            clonedProfile.preferences = new HashMap<>(this.preferences);  // Create a new Map and
copy entries

            return clonedProfile;
        } catch (CloneNotSupportedException e) {
            System.err.println("Cloning not supported: " + e.getMessage());
            return null; // Or throw a RuntimeException
        }
    }

   // No longer need deepClone. It is handled in the clone method.
    public void displayProfile() {
        System.out.println("User Profile:");
        System.out.println("Name: " + name);
        System.out.println("Email: " + email);
        System.out.println("Preferences:");
```

```
        for (Map.Entry<String, String> entry : preferences.entrySet()) {
            System.out.println("  " + entry.getKey() + ": " + entry.getValue());
        }
    }

    // Example Usage
    public static void main(String[] args) {
        UserProfile originalProfile = new UserProfile("John Doe", "john.doe@example.com");
        originalProfile.setPreference("theme", "dark");
        originalProfile.setPreference("language", "en");

        System.out.println("Original Profile:");
        originalProfile.displayProfile();

        // Deep Copy
        UserProfile clonedProfile = originalProfile.clone();
        clonedProfile.setName("Jane Doe");
        clonedProfile.setPreference("theme", "light"); // Modifying cloned profile's preferences

        System.out.println("\nCloned Profile (Deep Copy):");
        clonedProfile.displayProfile();

        System.out.println("\nOriginal Profile (after modifying the cloned profile):");
        originalProfile.displayProfile(); // Original profile remains unchanged
    }
}
```

4. Discuss scenarios where Prototype might be a better choice than Factory or Builder.

Solution:

- Prototype excels at fast object duplication, especially when creation is costly, unlike Factory/Builder.
- It enables dynamic runtime object modification by cloning altered prototypes, which is less flexible in Factory/Builder.
- Prototype helps avoid a proliferation of subclasses by cloning pre-existing instances, simplifying complex hierarchies.
- It preserves the internal state of cloned objects, crucial for scenarios like game states or templates, whereas Factory/Builder create fresh instances.

● Prototype is most useful when existing object instances with their states need to be efficiently replicated.

5. Represent using class diagram
Solution: