Day 23:
Task 1: Singleton
Implement a Singleton class that manages database connections. Ensure the class adheres strictly to the singleton pattern principles.

Answer:

Singleton and Database Connections:

- A Singleton class ensures that only one instance of the class exists throughout the application's lifecycle.
- While it's tempting to make a database connection class a Singleton, it's not always the best approach.

Reasons to Avoid Singleton for DB Connections:

- Thread Safety: Many database drivers are not thread-safe. Using a Singleton means that multiple threads will share the same connection, which can lead to issues.
- No Inherent Thread Safety: The Singleton pattern doesn't inherently provide thread safety; it merely allows easy sharing of a global instance.
- Consider Connection Pooling: Instead of a Singleton, consider using a database connection pool. Connection pooling allows efficient reuse of connections while maintaining thread safety.
- Sample pool libraries: Apache Commons DBCP and HikariCP.

Implementing a Connection Pool (Object Pool Pattern):

- Create a ConnectionPool class that manages a pool of database connections.
- Methods in the ConnectionPool class:
- getConnection(): Borrow a connection from the pool.
- returnConnection(): Return a connection back to the pool after use.

To implement a Singleton class that manages database connections in Java, you need to ensure that the class strictly adheres to the Singleton design pattern principles. This involves making sure that only one instance of the class is created and providing a global point of access to that instance.

Here is a step-by-step implementation:

1. **Private Constructor**: Prevent instantiation of the class from other classes.
2. **Private Static Instance**: Hold the single instance of the class.
3. **Public Static Method**: Provide a global access point to get the instance.

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

public class DatabaseConnectionManager {
    // Step 2: Private static instance of the class
    private static volatile DatabaseConnectionManager instance;
    private Connection connection;

    // Database credentials
```

```
    private static final String URL = "jdbc:mysql://localhost:3306/your_database";
    private static final String USER = "your_username";
    private static final String PASSWORD = "your_password";

    // Step 1: Private constructor to prevent instantiation
    private DatabaseConnectionManager() {
        try {
            // Initialize the database connection
            this.connection = DriverManager.getConnection(URL, USER, PASSWORD);
        } catch (SQLException e) {
            e.printStackTrace(); // Handle exception appropriately in real scenarios
        }
    }

    // Step 3: Public static method to provide global access point
    public static DatabaseConnectionManager getInstance() {
        if (instance == null) {
            synchronized (DatabaseConnectionManager.class) {
                if (instance == null) { // Double-check locking
                    instance = new DatabaseConnectionManager();
                }
            }
        }
        return instance;
    }

    public Connection getConnection() {
        return connection;
    }
}
```

Task 2: Factory Method
Create a ShapeFactory class that encapsulates the object creation logic of different Shape objects like Circle, Square, and Rectangle.

- Define a common interface or abstract class for Shape.
- Implement the Shape interface with concrete classes like Circle, Square, and Rectangle.
- Implement the ShapeFactory class with a method that creates and returns instances of the different shapes based on input parameters.

Step 1: Define the `Shape` Interface

```
public interface Shape {
    void draw();
}
```

Step 2: Implement Concrete Shape Classes

```
public class Circle implements Shape {
   @Override
   public void draw() {
      System.out.println("Drawing a Circle");
   }
}

public class Square implements Shape {
   @Override
   public void draw() {
      System.out.println("Drawing a Square");
   }
}

public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Drawing a Rectangle");
   }
}
```

Step 3: Implement the Shape Factory Class

```
public class ShapeFactory {
   // Factory method to create shapes
   public Shape getShape(String shapeType) {
      if (shapeType == null) {
         return null;
      }
      if (shapeType.equalsIgnoreCase("CIRCLE")) {
         return new Circle();
      } else if (shapeType.equalsIgnoreCase("SQUARE")) {
         return new Square();
      } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
         return new Rectangle();
      }
      return null;
   }
}
```

```
public class FactoryPatternDemo {
   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();
```

```
        // Get an object of Circle and call its draw method
        Shape shape1 = shapeFactory.getShape("CIRCLE");
        shape1.draw();

        // Get an object of Square and call its draw method
        Shape shape2 = shapeFactory.getShape("SQUARE");
        shape2.draw();

        // Get an object of Rectangle and call its draw method
        Shape shape3 = shapeFactory.getShape("RECTANGLE");
        shape3.draw();
    }
}
```

Output:

```
Drawing a Circle
Drawing a Square
Drawing a Rectangle
```

Task 3: Proxy
Create a proxy class for accessing a sensitive object that contains a secret key. The proxy should only allow access to the secret key if a correct password is provided.

SensitiveObject.java

```
public class SensitiveObject {
    private String secretKey;

    public SensitiveObject(String secretKey) {
        this.secretKey = secretKey;
    }

    public String getSecretKey() {
        return secretKey;
    }
}
```

ProxyObject.java

```
public class ProxyObject {
    private SensitiveObject sensitiveObject;
    private String password;

    public ProxyObject(String secretKey, String password) {
        this.sensitiveObject = new SensitiveObject(secretKey);
        this.password = password;
    }
```

```
    public String getSecretKey() {
        if ("mySecretPassword".equals(password)) {
            return sensitiveObject.getSecretKey();
        } else {
            return "Access denied!";
        }
    }

    public static void main(String[] args) {
        // Example usage
        ProxyObject proxy = new ProxyObject("superSecretKey", "mySecretPassword");
        System.out.println(proxy.getSecretKey()); // Prints the secret key
        System.out.println(proxy.getSecretKey()); // Prints "Access denied!"
    }
}
```

- SensitiveObject represents the actual sensitive data (the secret key).
- ProxyObject acts as a proxy to SensitiveObject.
- The getSecretKey() method in ProxyObject checks if the provided password matches the correct password before allowing access to the secret key.

Task 4: Strategy
Develop a Context class that can use different SortingStrategy algorithms interchangeably to sort a collection of numbers.

- Define a SortingStrategy interface that declares a method for sorting.
- Implement different sorting strategies (e.g., BubbleSort, QuickSort, MergeSort).
- Create a Context class that uses a SortingStrategy to sort a collection of numbers.

Step 1 : Define the SortingStrategy Interface

```
public interface SortingStrategy {
    void sort(int[] numbers);
}
```

Step 2: Implement Different Sorting Strategies

BubbleSort

```
public class BubbleSort implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        int n = numbers.length;
        for (int i = 0; i < n - 1; i++) {
```

```
            for (int j = 0; j < n - i - 1; j++) {
                if (numbers[j] > numbers[j + 1]) {
                    // Swap numbers[j] and numbers[j + 1]
                    int temp = numbers[j];
                    numbers[j] = numbers[j + 1];
                    numbers[j + 1] = temp;
                }
            }
        }
    }
}
```

QuickSort

```
public class QuickSort implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        quickSort(numbers, 0, numbers.length - 1);
    }

    private void quickSort(int[] array, int low, int high) {
        if (low < high) {
            int pi = partition(array, low, high);
            quickSort(array, low, pi - 1);
            quickSort(array, pi + 1, high);
        }
    }

    private int partition(int[] array, int low, int high) {
        int pivot = array[high];
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (array[j] <= pivot) {
                i++;
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;
        return i + 1;
    }
}
```

MergeSort

```
public class MergeSort implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        mergeSort(numbers, 0, numbers.length - 1);
    }

    private void mergeSort(int[] array, int left, int right) {
        if (left < right) {
            int middle = (left + right) / 2;
            mergeSort(array, left, middle);
            mergeSort(array, middle + 1, right);
            merge(array, left, middle, right);
        }
    }

    private void merge(int[] array, int left, int middle, int right) {
        int n1 = middle - left + 1;
        int n2 = right - middle;
        int[] L = new int[n1];
        int[] R = new int[n2];

        for (int i = 0; i < n1; ++i) L[i] = array[left + i];
        for (int j = 0; j < n2; ++j) R[j] = array[middle + 1 + j];

        int i = 0, j = 0;
        int k = left;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                array[k] = L[i];
                i++;
            } else {
                array[k] = R[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            array[k] = L[i];
            i++;
            k++;
        }

        while (j < n2) {
            array[k] = R[j];
            j++;
            k++;
        }
    }
}
```

Step 3: Create the Context Class

```
public class Context {
    private SortingStrategy strategy;

    // Constructor
    public Context(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    // Method to set the strategy at runtime
    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    // Method to sort using the current strategy
    public void sort(int[] numbers) {
        strategy.sort(numbers);
    }
}
```

StrategyPatternDemo.Java

```
public class StrategyPatternDemo {
    public static void main(String[] args) {
        int[] numbers = {5, 2, 9, 1, 5, 6};

        Context context = new Context(new BubbleSort());
        System.out.println("Using Bubble Sort:");
        context.sort(numbers);
        System.out.println(Arrays.toString(numbers));

        numbers = new int[]{5, 2, 9, 1, 5, 6};
        context.setStrategy(new QuickSort());
        System.out.println("Using Quick Sort:");
        context.sort(numbers);
        System.out.println(Arrays.toString(numbers));

        numbers = new int[]{5, 2, 9, 1, 5, 6};
        context.setStrategy(new MergeSort());
        System.out.println("Using Merge Sort:");
        context.sort(numbers);
        System.out.println(Arrays.toString(numbers));
    }
}
```

Output:

```
Using Bubble Sort:
[1, 2, 5, 5, 6, 9]
Using Quick Sort:
[1, 2, 5, 5, 6, 9]
Using Merge Sort:
[1, 2, 5, 5, 6, 9]
```

1. SortingStrategy Interface:
    - Declares a sort method for sorting arrays.
2. Concrete Sorting Strategies:
    - BubbleSort, QuickSort, and MergeSort implement the SortingStrategy interface.
    - Each class provides its own implementation of the sort method.
3. Context Class:
    - Holds a reference to a SortingStrategy object.
    - Allows setting or changing the sorting strategy at runtime using setStrategy.
    - Uses the current strategy to sort arrays via the sort method.
4. StrategyPatternDemo Class:
    - Demonstrates the use of different sorting strategies.
    - Changes the sorting strategy at runtime and sorts the array using the current strategy.
    - Prints the sorted array for each strategy used.