Index Number: EG/2020/3833

Q1.

1. Explain the concept of the Singleton pattern and how it addresses the requirement for a single instance of a class.

   The Singleton Pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance.

   It involves a class responsible for instantiating itself while making sure that only one instance of the class exists. This pattern is useful when exactly one object is needed to coordinate actions across the system, such as managing resources or providing a single point of control.

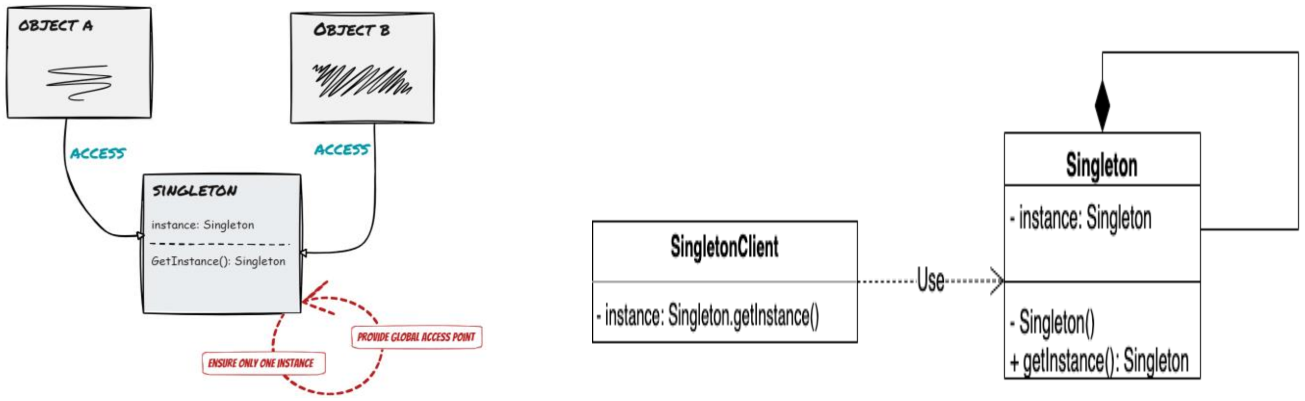


Figure 1: UML and graphical representation of the Singleton pattern

Key concepts of the Singleton pattern:

- Single Instance: The Singleton pattern guarantees that only one instance of a class can be created. This is achieved by restricting instantiation to a single object and providing a global point of access to that object.

- Global Access Point: The Singleton pattern provides a globally accessible reference to the single instance of the class. This reference is typically accessed through a static method or property on the class itself, allowing clients to easily obtain an instance of the class.

- Thread Safety: When multiple threads are involved, the Singleton pattern must ensure that only one instance of the class is created and that access to the instance is synchronized. This is critical to prevent race conditions and maintain the integrity of the single instance.

2. Provide a sample implementation of the Singleton pattern in the context of the library management system. Include relevant code snippets and explain how your implementation ensures a single instance of the database connection manager.

```java
public class DatabaseConnectionManager {
    private static DatabaseConnectionManager instance;


    private DatabaseConnectionManager() {

             System.out.println("Singleton instance created");


    }


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

If we try to create an instance of the database connection manager, compiler will throw an error message because of the private constructor. So we can't initiate any instance of the database connection manager. ultimately rather creating an object, we can access the getinstance() function because of the static type. This is the only possible way to get the single instance from database connection manager. this is how singleton pattern is achieved.

3. Discuss the potential advantages and disadvantages of using the Singleton Pattern in this scenario. Include considerations related to thread safety, lazy initialization, and any impact on unit testing.

- Advantages of singleton design pattern

I. **Single Instance:** The primary advantage is that the Singleton Pattern ensures there is only one instance of the class throughout the application.

II. **Global Access:** The singleton instance can be globally accessed, providing a convenient way to manage and access the shared resource from any part of the application.

III. **Lazy Initialization:** The Singleton Pattern allows for lazy initialization, meaning that the instance is created only when it is first requested.

IV. **Thread Safety:** When implemented correctly with proper synchronization mechanisms, the Singleton Pattern ensures thread safety.

- Disadvantages of singleton design pattern

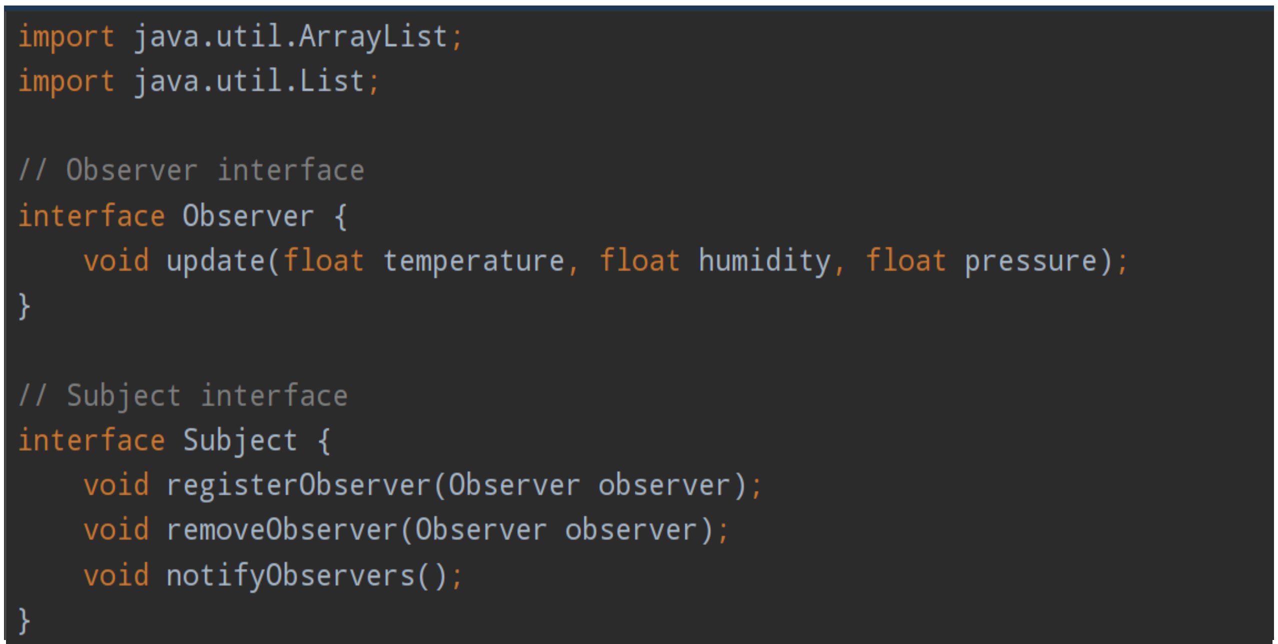
1. **Thread Safety Overhead:** Synchronization in the **getInstance** method for lazy initialization introduces a level of overhead, especially in high-concurrency scenarios. This can impact the performance of the application.

2. **Lazy Initialization Impact:** While lazy initialization is beneficial for resource efficiency, it may introduce a delay in accessing the singleton instance for the first time. In scenarios where the instance needs to be available immediately, eager initialization might be preferred.

3. **Impact on Unit Testing:** Testing can be impacted because the singleton instance retains its state between tests. Resetting the state for each test may be necessary, or alternative approaches (e.g., dependency injection) might be considered for better testability.

Q3.

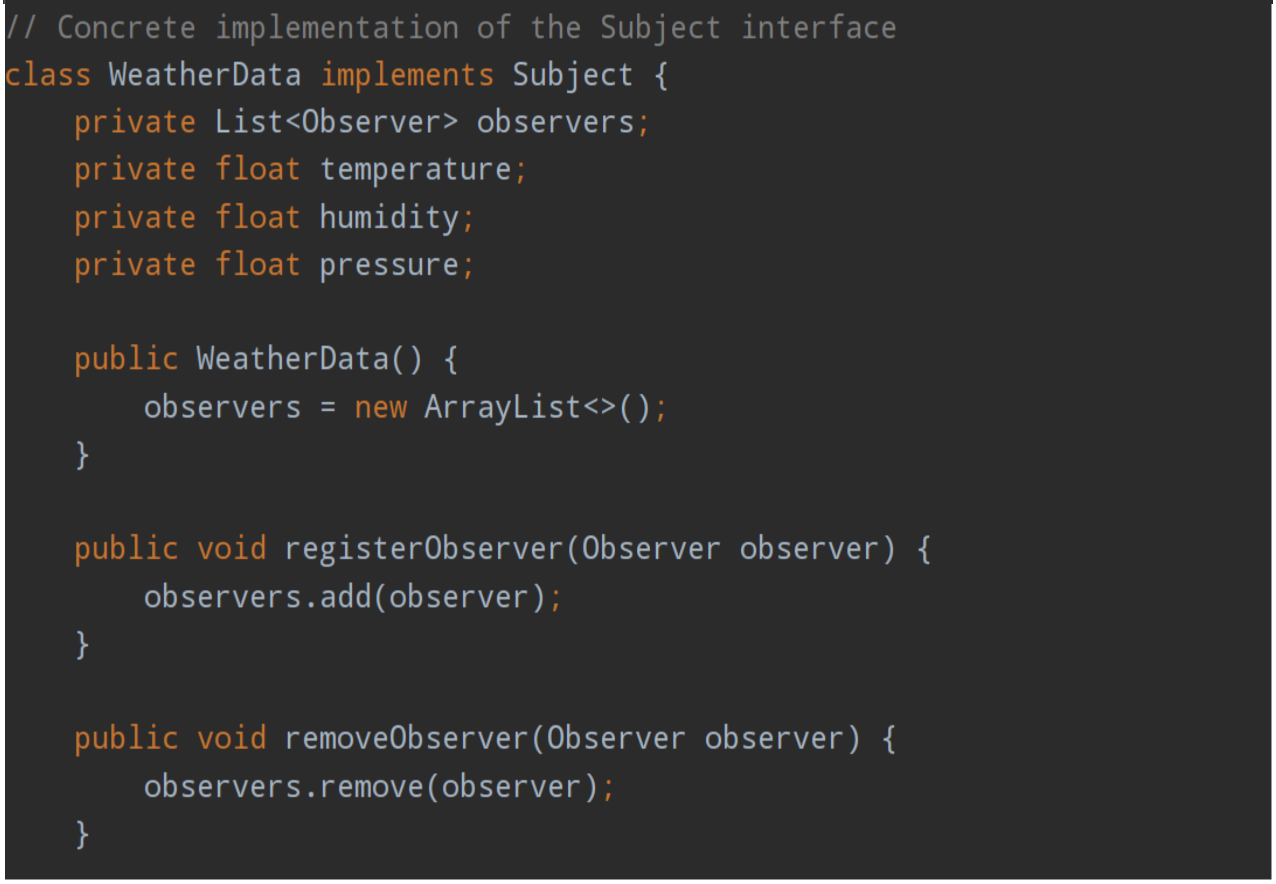
1. Provide a sample implementation of the Observer pattern in the context of the weather monitoring system.



```java
import java.util.ArrayList;
import java.util.List;

// Observer interface
interface Observer {
    void update(float temperature, float humidity, float pressure);
}

// Subject interface
interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}
```

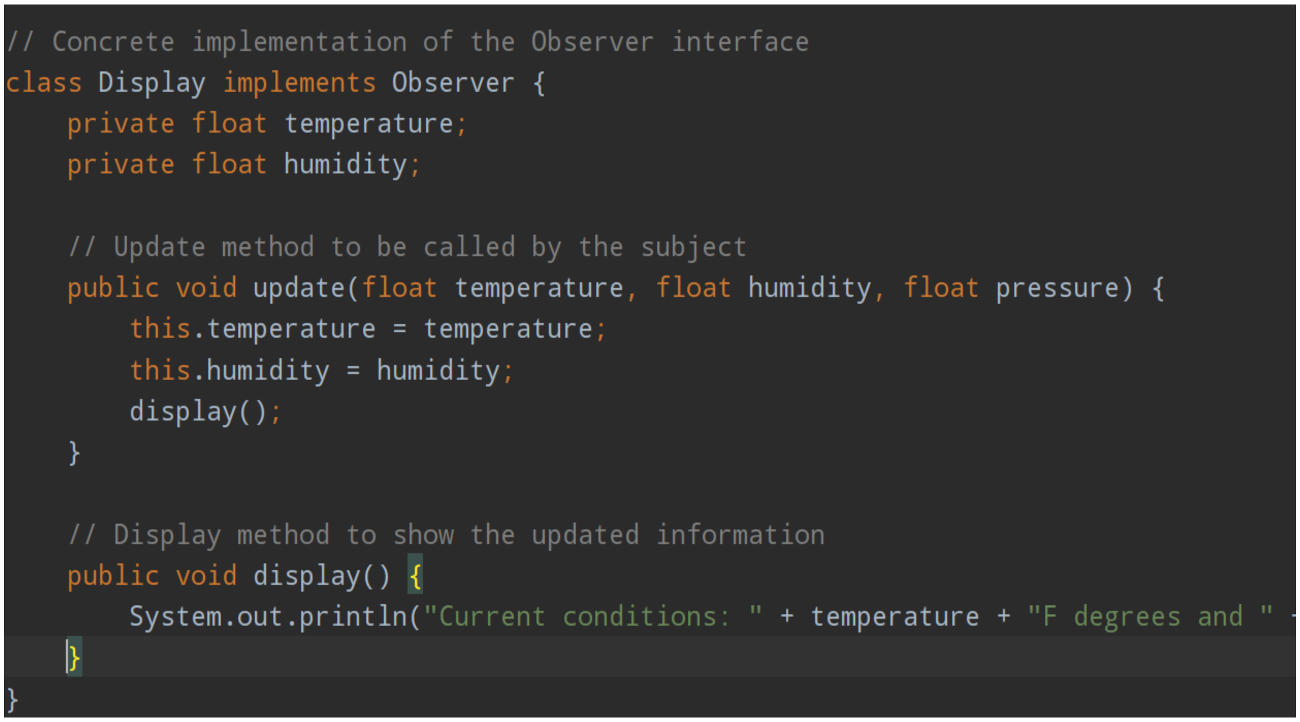


```java
// Concrete implementation of the Subject interface
class WeatherData implements Subject {
    private List<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<>();
    }

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
```



```java
// Concrete implementation of the Observer interface
class Display implements Observer {
    private float temperature;
    private float humidity;

    // Update method to be called by the subject
    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    // Display method to show the updated information
    public void display() {
        System.out.println("Current conditions: " + temperature + "F degrees and "
    }
}
```

2. Provide a class diagram for the weather monitoring system

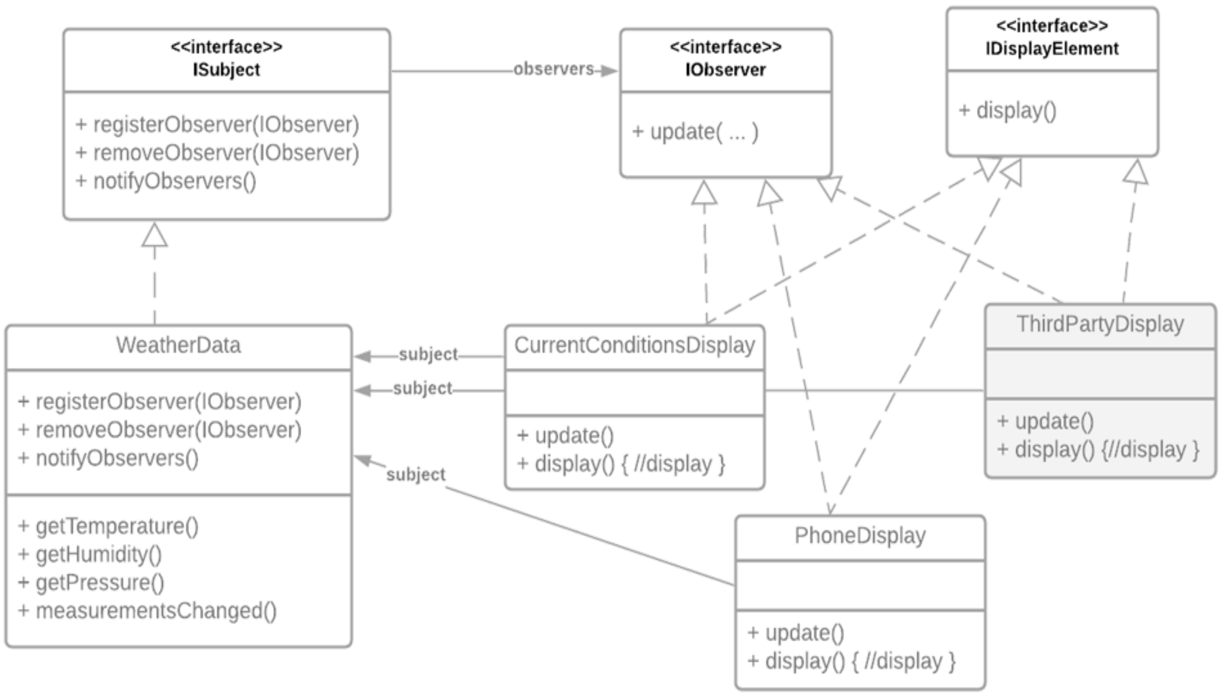


Figure 2:whetherforcast observer Pattern

Q4.

1. Explain the Adapter pattern and provide an example?

   The Adapter Pattern is a structural design pattern that allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces by providing a wrapper class (the adapter) that converts the interface of a class into another interface that a client expects. This pattern allows the integration of existing systems without modifying their source code.

   The main components in the Adapter Pattern are:

   I. **Target Interface:** The interface that the client expects.
   II. **Adaptee:** The existing class with an incompatible interface that needs to be integrated.
   III. **Adapter:** The class that implements the target interface and wraps an instance of the adaptee, translating calls from the target interface to the adaptee's interface.

2. Provide a sample implementation of the Adapter pattern?

```
public interface MediaPlayer {
public void play(String audioType, String fileName);}
public interface AdvancedMediaPlayer {
public void playVlc(String fileName);
public void playMp4(String fileName);
}
public class VlcPlayer implements AdvancedMediaPlayer{
   @Override
   public void playVlc(String fileName) {
      System.out.println("Playing vlc file. Name: "+ fileName);

   }

   @Override
   public void playMp4(String fileName) {
      //do nothing
   }
}
public class Mp4Player implements AdvancedMediaPlayer{

   @Override
   public void playVlc(String fileName) {
      //do nothing
   }

   @Override
   public void playMp4(String fileName) {
      System.out.println("Playing mp4 file. Name: "+ fileName);

   }
```

```
}
public class MediaAdapter implements MediaPlayer {

   AdvancedMediaPlayer advancedMusicPlayer;

   public MediaAdapter(String audioType){

      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();

      }else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }

   @Override
   public void play(String audioType, String fileName) {

      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }
      else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
}
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;

   @Override
   public void play(String audioType, String fileName) {

      //inbuilt support to play mp3 music files
      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: " +
fileName);
      }

      //mediaAdapter is providing support to play other file
formats
      else if(audioType.equalsIgnoreCase("vlc") ||
audioType.equalsIgnoreCase("mp4")){
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      }

      else{
         System.out.println("Invalid media. " + audioType + "
format not supported");
      }
   }
}
```

3. Provide a class diagram for the Adapter Pattern?

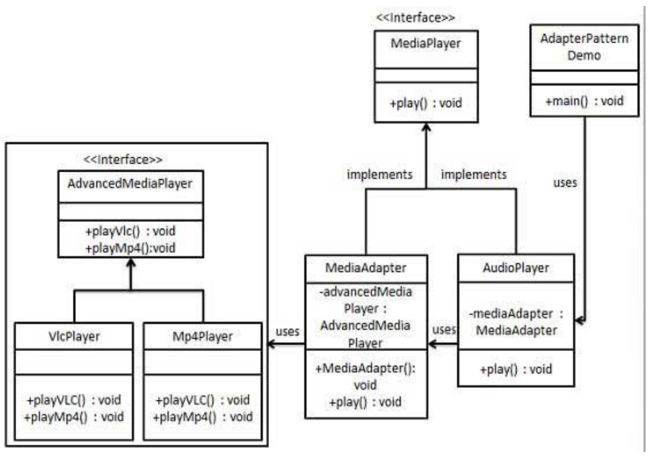


Figure 3: Adapter Pattern UML Diagram