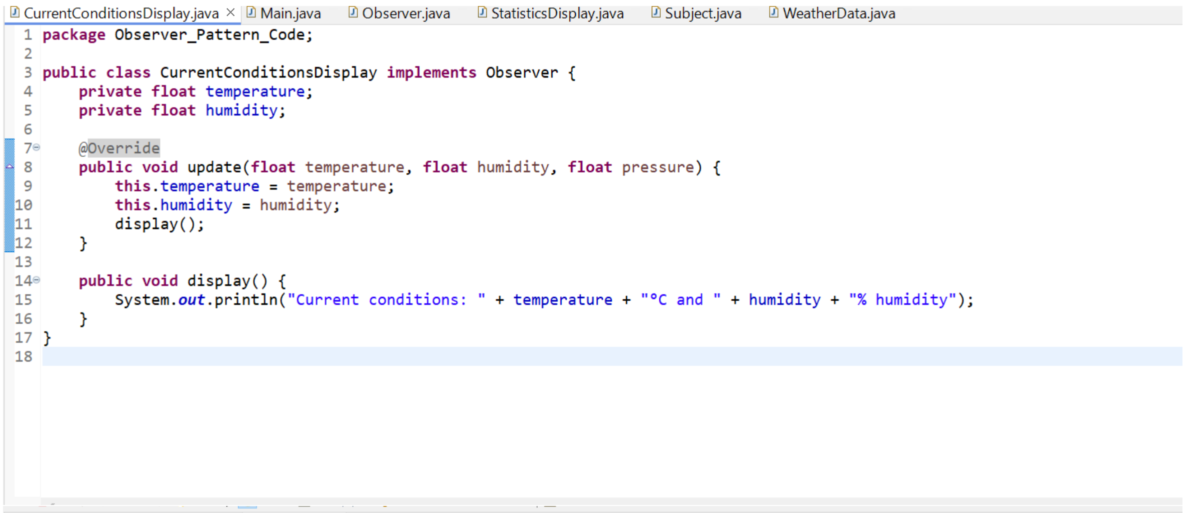
Documentation for Observer Pattern Code

Introduction

This document provides a comprehensive overview of a Java implementation of the Observer Pattern used to update and notify multiple displays based on changes in weather data. The Observer Pattern defines a one-to-many dependency between objects, allowing multiple observers to be notified of state changes in a subject.

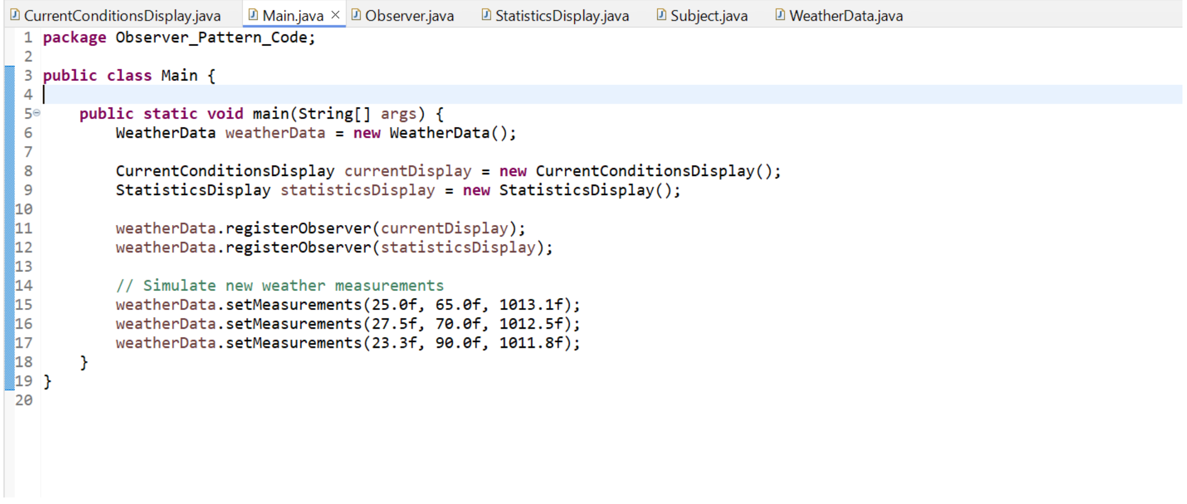
General Explanation

The code demonstrates the Observer Pattern applied to a weather monitoring system. The WeatherData class acts as the subject, maintaining and notifying observers of weather changes. Observers such as CurrentConditionsDisplay and StatisticsDisplay respond to these changes and display the updated information.



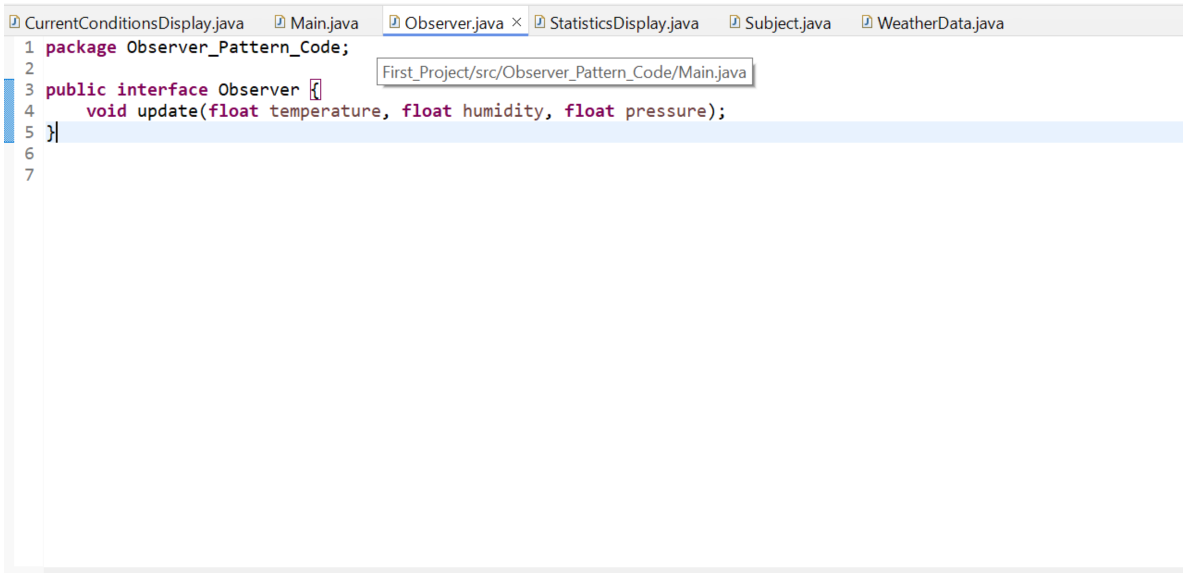
CurrentConditionsDisplay.java | Main.java | Observer.java | StatisticsDisplay.java | Subject.java | WeatherData.java

```java
package Observer_Pattern_Code;

public class CurrentConditionsDisplay implements Observer {
    private float temperature;
    private float humidity;

    @Override
    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature + "°C and " + humidity + "% humidity");
    }
}

```



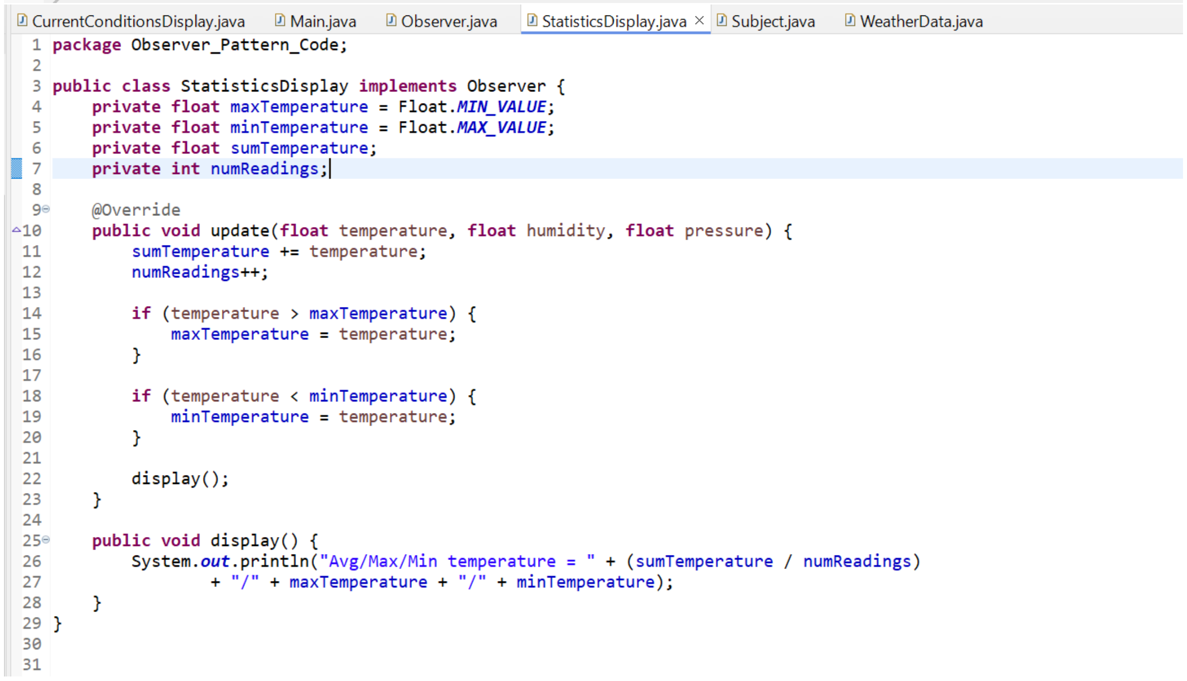
CurrentConditionsDisplay.java | Main.java | Observer.java | StatisticsDisplay.java | Subject.java | WeatherData.java

```java
package Observer_Pattern_Code;

public class Main {

    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay();
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay();

        weatherData.registerObserver(currentDisplay);
        weatherData.registerObserver(statisticsDisplay);

        // Simulate new weather measurements
        weatherData.setMeasurements(25.0f, 65.0f, 1013.1f);
        weatherData.setMeasurements(27.5f, 70.0f, 1012.5f);
        weatherData.setMeasurements(23.3f, 90.0f, 1011.8f);
    }
}

```

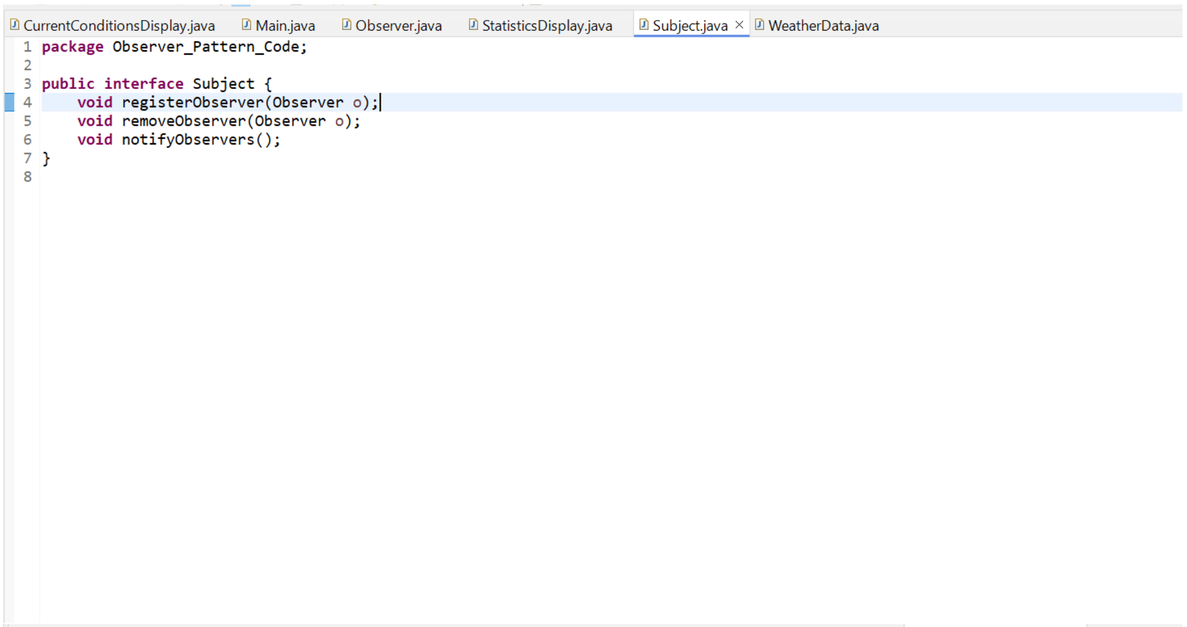

CurrentConditionsDisplay.java   Main.java   Observer.java ×   StatisticsDisplay.java   Subject.java   WeatherData.java

First_Project/src/Observer_Pattern_Code/Main.java

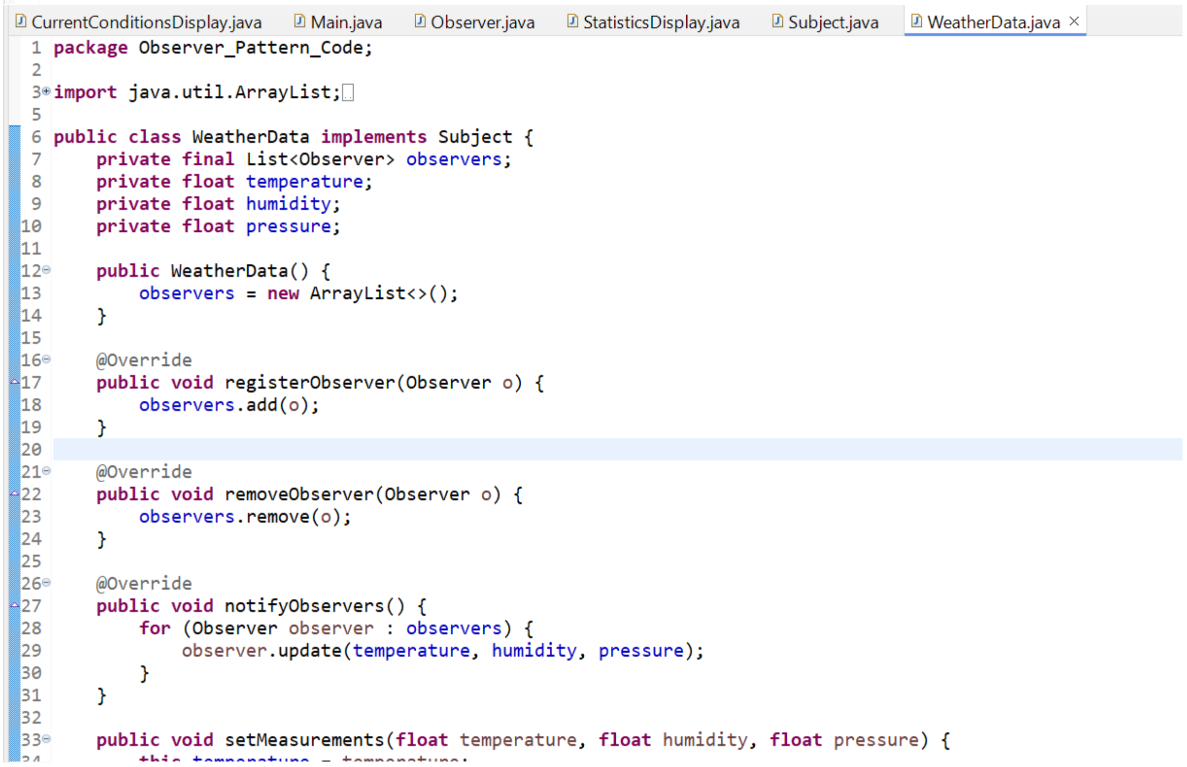
```java
package Observer_Pattern_Code;

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


```


CurrentConditionsDisplay.java   Main.java   Observer.java   StatisticsDisplay.java ×   Subject.java   WeatherData.java

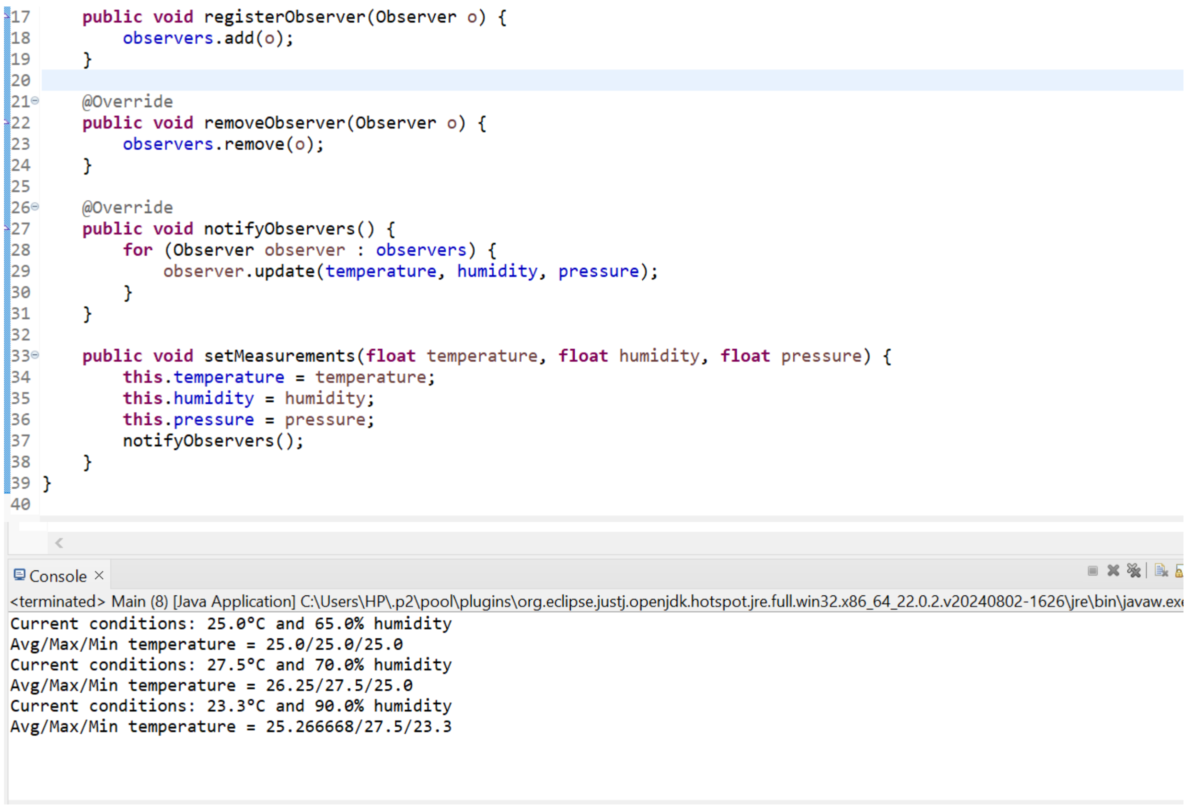
```java
package Observer_Pattern_Code;

public class StatisticsDisplay implements Observer {
    private float maxTemperature = Float.MIN_VALUE;
    private float minTemperature = Float.MAX_VALUE;
    private float sumTemperature;
    private int numReadings;

    @Override
    public void update(float temperature, float humidity, float pressure) {
        sumTemperature += temperature;
        numReadings++;

        if (temperature > maxTemperature) {
            maxTemperature = temperature;
        }

        if (temperature < minTemperature) {
            minTemperature = temperature;
        }

        display();
    }

    public void display() {
        System.out.println("Avg/Max/Min temperature = " + (sumTemperature / numReadings)
                + "/" + maxTemperature + "/" + minTemperature);
    }
}


```



```java
package Observer_Pattern_Code;

public interface Subject {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}

```



```java
package Observer_Pattern_Code;

import java.util.ArrayList;

public class WeatherData implements Subject {
    private final List<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

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

    @Override
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
```



```java
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        notifyObservers();
    }
}
```

```
<terminated> Main (8) [Java Application] C:\Users\HP\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_22.0.2.v20240802-1626\jre\bin\javaw.ex
Current conditions: 25.0°C and 65.0% humidity
Avg/Max/Min temperature = 25.0/25.0/25.0
Current conditions: 27.5°C and 70.0% humidity
Avg/Max/Min temperature = 26.25/27.5/25.0
Current conditions: 23.3°C and 90.0% humidity
Avg/Max/Min temperature = 25.266668/27.5/23.3
```

Key Classes

- **Observer Interface:** Defines the contract for observer objects. It includes the update method, which is called by the subject to notify the observer of changes in weather data.
- **Subject Interface:** Defines the contract for subject objects. It includes methods for registering, removing, and notifying observers.
- **WeatherData Class:** Implements the Subject interface and maintains a list of observers. It updates the observers when weather measurements change by invoking their update methods.
- **CurrentConditionsDisplay Class:** Implements the Observer interface and displays the current weather conditions (temperature and humidity). It updates its display whenever new weather data is received.
- **StatisticsDisplay Class:** Implements the Observer interface and tracks and displays average, maximum, and minimum temperatures. It updates its statistics whenever new weather data is received.
- **Main Class:** Demonstrates the use of the observer pattern. It creates a WeatherData instance and registers two observers (CurrentConditionsDisplay and StatisticsDisplay). It simulates new weather measurements to show how the observers are updated.

Methods Used

- **update(float temperature, float humidity, float pressure)**: Part of the Observer interface. Implemented by observer classes to receive and process updates from the subject.
- **display()**: Custom method in CurrentConditionsDisplay and StatisticsDisplay classes to show the current weather conditions and statistical information, respectively.
- **registerObserver(Observer o)**: Part of the Subject interface. Adds an observer to the list of observers in the WeatherData class.
- **removeObserver(Observer o)**: Part of the Subject interface. Removes an observer from the list of observers in the WeatherData class.
- **notifyObservers()**: Part of the Subject interface. Notifies all registered observers of changes in the WeatherData class.
- **setMeasurements(float temperature, float humidity, float pressure)**: Updates weather measurements in the WeatherData class and triggers the notification of observers.

Key Points

1. **Observer Pattern:** Establishes a one-to-many dependency between objects, allowing a subject to notify multiple observers of changes. This pattern is useful for scenarios where an object’s state changes and multiple dependent objects need to be updated.
2. **Decoupling:** The pattern promotes loose coupling between the subject and its observers. Observers are updated automatically when the subject changes, without the need for the subject to know specific details about the observers.
3. **Dynamic Updates:** Observers can be added or removed at runtime. This provides flexibility in managing which components receive updates from the subject.
4. **Responsibility Separation:** The pattern allows for the separation of concerns, with observers handling different aspects of the subject's state. For example, CurrentConditionsDisplay focuses on displaying current weather conditions, while StatisticsDisplay calculates and shows statistical information.