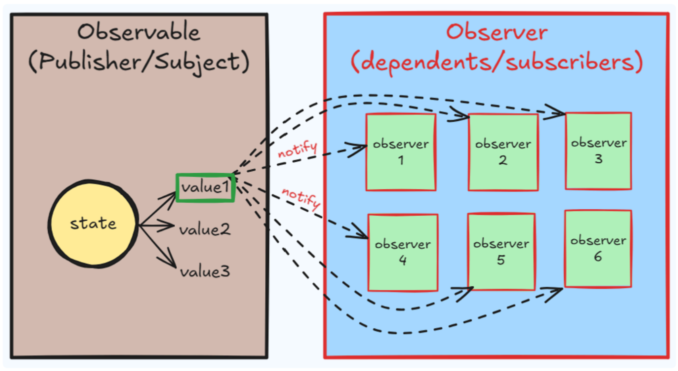
Observer Pattern

Resources

- Video link → 41. All Behavioral Design Patterns | Strategy, Observer, State, Template, Command, Visitor, Memento
- Video link → 3. Observer Design Pattern Explanation (Hindi) | Design Interview Question | LLD System Design

Definition

> ***The Observer Pattern is a behavioral design pattern where an object (aka the "subject" or "observable" or “publisher”) maintains a list of dependents (called "observers") and automatically notifies them whenever there is a change in its state. The pattern also allows addition and removal of observers at runtime.***

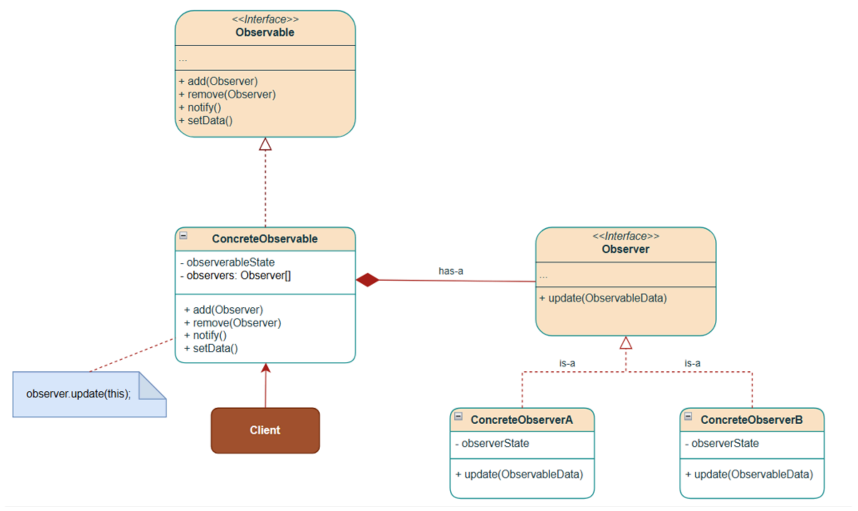


Real-life Examples

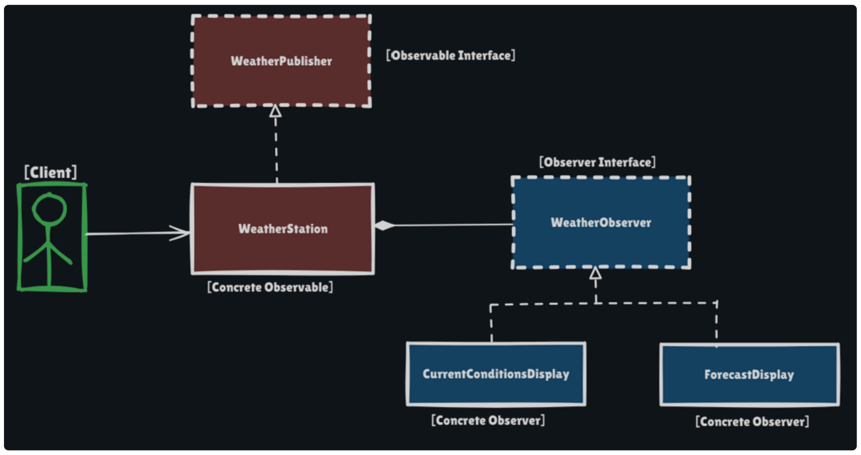
Real-life examples of the Observer pattern include:

- **Weather Applications:** Where multiple devices receive updates from a weather station.
- **Social Media Feeds:** When we follow someone on Instagram, Facebook, or Twitter, we become observer of their profile. When they post new content, we are automatically notified.
- **Subscription Services:** YouTube subscriptions, where viewers are notified of new videos, or Content magazine/newspaper/newsletter subscriptions, where publishers send new issues to subscribers.
- **Stock Market Trackers:** When the price of a stock (or state) changes, the stock's market system (the observable) sends out notifications to all interested investors (the observers).

Class Diagram



Structure of the Observer Pattern



Let's understand the Structure of the Observer Pattern using the Weather Station example:

- **Observable Interface (**or Subject Interface, i.e., `WeatherObservable`**)**
  - Defines methods for adding, removing, and notifying observers.
  - The weather station implements this interface.
- **Observer Interface (`WeatherObserver`)**
  - Defines the `update()` method that all concrete observers must implement.
  - Called by the observable when there is a change in its state.
- **Concrete Observable (or Concrete Subject, i.e., `WeatherStation`)**
  - Maintains a list of observers.
  - Holds the weather data, i.e., **observable data** (temperature, humidity, pressure).
  - Notifies all observers when measurements(state) change.
- **Concrete Observers (`ForecastDisplay` & `CurrentConditionsDisplay`)**
  - Each display has different behavior when updated.
  - `CurrentConditionsDisplay`: Shows current weather on gadgets like TV or mobile.

- `ForecastDisplay` : Predicts weather based on pressure changes.

Implementation

1. Example: Weather Station

```java
// Observable(Subject) interface
// Defines methods for managing observers and notifying them of changes
public interface WeatherObservable {

    void addObserver(WeatherObserver observer);

    void removeObserver(WeatherObserver observer);

    void notifyObservers();

    void setWeatherReadings(float temperature, float humidity, float pressure);
}
```

```java
// Concrete Observable (Subject)
// WeatherStation - the concrete observable class that holds weather data
public class WeatherStation implements WeatherObservable {
    // List of observers registered for updates
    private final List<WeatherObserver> observers;
    // Observable Data
    private float temperature;
    private float humidity;
    private float pressure;

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

    @Override
    public void addObserver(WeatherObserver observer) {
        observers.add(observer);
        System.out.println("[+] Observer registered: " + observer.getClass().getSimpleName());
    }

    @Override
    public void removeObserver(WeatherObserver observer) {
        observers.remove(observer);
        System.out.println("[-] Observer removed: " + observer.getClass().getSimpleName());
    }

    @Override
    public void notifyObservers() {
        for (WeatherObserver observer : observers) {
            // Notify each observer about the change in weather data(state)
            observer.update(); // Observer will update its state based on the new data and respond accordingly
        }
    }

    // Method to update weather measurements
    public void setWeatherReadings(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        notifyObservers();
    }

    // Getters for observers to access weather data
    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }

    @Override
    public String toString() {
        return "WeatherStation{" +
                "temperature=" + temperature +
                ", humidity=" + humidity +
                ", pressure=" + pressure +
                '}';
    }
}
```

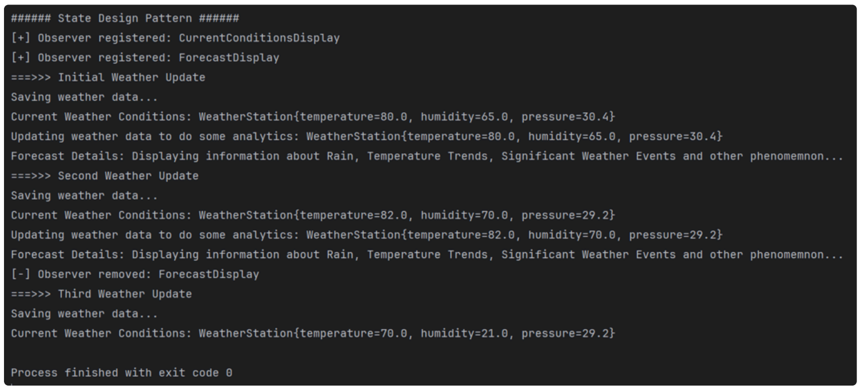
```java
// Observer interface - defines the update method
// Concrete observers implement this interface to update their state
// and respond to changes in its OWN way
public interface WeatherObserver {
    void update();
}
```

```java
// Concrete Observer 1 - Current Conditions Display (on TV or Mobile)
public class CurrentConditionsDisplay implements WeatherObserver {
    private final WeatherObservable weatherStation;

    public CurrentConditionsDisplay(WeatherObservable weatherStation)
{
        this.weatherStation = weatherStation;
        weatherStation.addObserver(this);
    }

    // CurrentConditionsDisplay implements the update method in its
own way
    @Override
    public void update() {
        System.out.println("Saving weather data... ");
        display();
    }

    // Display the current weather conditions
    public void display() {
        System.out.println("Current Weather Conditions: " +
weatherStation.toString());
    }
}
```

```java
// Concrete Observer 4 - Forecast Display - Predicts weather based on
pressure changes
public class ForecastDisplay implements WeatherObserver {
    private final WeatherObservable weatherStation;

    public ForecastDisplay(WeatherObservable weatherStation) {
        this.weatherStation = weatherStation;
        weatherStation.addObserver(this);
    }

    // ForecastDisplay implements the update method in its own way
    @Override
    public void update() {
        System.out.println("Updating weather data to do some analytics: " + weatherStation.toString());
        display();
    }

    // Display the forecast based on the current pressure
    public void display() {
        System.out.println("Forecast Details: Displaying information about Rain, " +
                "Temperature Trends, Significant Weather Events and other phenomemnon...");
    }
}

```

```java
// Client code to demonstrate the Observer Pattern
public class WeatherStationApp {
    public static void main(String[] args) {
        System.out.println("###### State Design Pattern ######");
        // Create the weather station (observable/subject)
        WeatherObservable weatherStation = new WeatherStation();

        // Create displays (observers)
        CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay(weatherStation);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherStation);

        System.out.println("===>>> Initial Weather Update");
        weatherStation.setWeatherReadings(80, 65, 30.4f);

        System.out.println("===>>> Second Weather Update");
        weatherStation.setWeatherReadings(82, 70, 29.2f);

        // Remove forecast display
        weatherStation.removeObserver(forecastDisplay);

        System.out.println("===>>> Third Weather Update");
        weatherStation.setWeatherReadings(70, 21, 29.2f);
        // Forecast display will not be notified
    }
}
```

Output



```
###### State Design Pattern ######
[+] Observer registered: CurrentConditionsDisplay
[+] Observer registered: ForecastDisplay
===>>> Initial Weather Update
Saving weather data...
Current Weather Conditions: WeatherStation{temperature=80.0, humidity=65.0, pressure=30.4}
Updating weather data to do some analytics: WeatherStation{temperature=80.0, humidity=65.0, pressure=30.4}
Forecast Details: Displaying information about Rain, Temperature Trends, Significant Weather Events and other phenomemnon...
===>>> Second Weather Update
Saving weather data...
Current Weather Conditions: WeatherStation{temperature=82.0, humidity=70.0, pressure=29.2}
Updating weather data to do some analytics: WeatherStation{temperature=82.0, humidity=70.0, pressure=29.2}
Forecast Details: Displaying information about Rain, Temperature Trends, Significant Weather Events and other phenomemnon...
[-] Observer removed: ForecastDisplay
===>>> Third Weather Update
Saving weather data...
Current Weather Conditions: WeatherStation{temperature=70.0, humidity=21.0, pressure=29.2}

Process finished with exit code 0
```

2. Example: E-commerce “Notify Me” feature

```java
// Observable interface
public interface StockAvailabilityObservable {
    void addStockObserver(StockNotificationObserver observer);

    void removeStockObserver(StockNotificationObserver observer);

    void notifyStockObservers();

    boolean purchase(int quantity);

    void restock(int quantity);
}
```

```java
// Concrete Observable
public class IphoneProductObservable implements StockAvailabilityObservable {
    private final String productId;
    private final String productName;
    private final double price;
    private final List<StockNotificationObserver> stockObservers;
    private int stockQuantity;

    public IphoneProductObservable(String productId, String productName, double price, int stockQuantity) {
        this.productId = productId;
        this.productName = productName;
        this.price = price;
        this.stockQuantity = stockQuantity;
        this.stockObservers = new ArrayList<>();
    }

    @Override
    public void addStockObserver(StockNotificationObserver observer) {
        stockObservers.add(observer);
        System.out.println("[+]" + observer.getUserId() + " subscribed for notifications on " + productName);

    }

    @Override
    public void removeStockObserver(StockNotificationObserver observer) {
        stockObservers.remove(observer);
        System.out.println("[-]" + observer.getUserId() + " unsubscribed for notifications on " + productName);
    }

    @Override
    public void notifyStockObservers() {
        if (stockQuantity > 0 && !stockObservers.isEmpty()) {
            System.out.println("Notifying " + stockObservers.size() + " subscribers... ");

            // Create a copy to avoid concurrent modification
            List<StockNotificationObserver> observersToNotify = new ArrayList<>(stockObservers);

            for (StockNotificationObserver observer : observersToNotify) {
                observer.update();
            }
        }
    }

    // Method to restock items
    @Override
    public void restock(int quantity) {
        boolean wasOutOfStock = (stockQuantity == 0);
        stockQuantity += quantity;
        System.out.println("RESTOCKED: " + productName + " - Added " + quantity + " items " + " | " + "Current stock: " + stockQuantity);
        // Only notify if product was previously out of stock
        if (wasOutOfStock && stockQuantity > 0) {
            notifyStockObservers();
        }
    }

    // Method to purchase items
    @Override
    public boolean purchase(int quantity) {
        if (stockQuantity >= quantity) {
            stockQuantity -= quantity;
            System.out.println("PURCHASE SUCCESS: " + quantity + " units of " + productName + " | " + "Remaining stock: " + stockQuantity);
            return true;
        } else {
            System.out.println("PURCHASE FAILED: " + productName + " is out of stock! | " + "Available Quantity: " + stockQuantity);
            return false;
        }
    }

    // Getters
    public String getProductId() {
        return productId;
    }

    public String getProductName() {
        return productName;
    }

    public double getPrice() {
        return price;
    }

    public int getStockQuantity() {
        return stockQuantity;
    }
}

```

```java
// Observer interface for stock availability notifications
public interface StockNotificationObserver {
    void update();

    String getNotificationMethod();

    String getUserId();
}
```

```java
// Concrete observer for email notifications
public class EmailNotificationObserver implements StockNotificationObserver {
    private final String userId;
    private final String emailAddress;

    public EmailNotificationObserver(String userId, String emailAddress) {
        this.userId = userId;
        this.emailAddress = emailAddress;
    }

    @Override
    public void update() {
        sendEmail();
    }

    private void sendEmail() {
        System.out.println("!! EMAIL SENT to: " + emailAddress + " - " + "Product is back in stock! Hurry Up!!");
    }

    @Override
    public String getNotificationMethod() {
        return "Email";
    }

    @Override
    public String getUserId() {
        return userId;
    }
}
```

```java
// Concrete observer for push notifications
public class PushNotificationObserver implements StockNotificationObserver {
    private final String userId;
    private final String deviceToken;

    public PushNotificationObserver(String userId, String deviceToken) {
        this.userId = userId;
        this.deviceToken = deviceToken;
    }

    @Override
    public void update() {
        sendPushNotification();
    }

    private void sendPushNotification() {
        System.out.println("!! PUSH NOTIFICATION SENT to: " + deviceToken + " - " + "Product is back in stock! Hurry Up!!");
    }

    @Override
    public String getNotificationMethod() {
        return "Push Notification";
    }

    @Override
    public String getUserId() {
        return userId;
    }
}
```

```java
public class ECommerceStockNotificationApp {
        System.out.println("-------------------------------------------------------------------------------");
        System.out.println("###### E-commerce Store - Stock Availability Notification Feature Demo ######");

        // Create an iPhone product - stock available = 10 units
        StockAvailabilityObservable iphoneProduct = new IphoneProductObservable("ip15", "iphone 15", 1250, 10);

        // Create observers
        StockNotificationObserver John_PUSH = new PushNotificationObserver("John123", "JohnDeviceP1");
        StockNotificationObserver Katy_PUSH = new PushNotificationObserver("Katy678", "KatyDeviceP2");
        StockNotificationObserver Jane_EMAIL = new EmailNotificationObserver("Jane783", "jane783@gmail.com");
        StockNotificationObserver George_EMAIL = new EmailNotificationObserver("George993", "george993@gmail.com");

        // Black Friday Sale - Purchase all 10 units
        iphoneProduct.purchase(10);

        // Stock unavailability leads to users subscribing to notifications
        boolean success = iphoneProduct.purchase(1); // Failed purchase
        if (!success) {
            // Register observers - John, Katy, Jane, George subscribe for notifications upon stock availability
            iphoneProduct.addStockObserver(John_PUSH); // John
            iphoneProduct.addStockObserver(Katy_PUSH); // Katy
            iphoneProduct.addStockObserver(Jane_EMAIL); // Jane
            iphoneProduct.addStockObserver(George_EMAIL); // George
        }

        // Restock 20 units of iPhone 15
        iphoneProduct.restock(20); // All 4 observers are notified

        // Users purchase upon receiving notifications
        iphoneProduct.purchase(1); // John purchases 1 unit
        iphoneProduct.purchase(1); // Katy purchases 1 unit

        // John & Katy unsubscribe from notifications
        iphoneProduct.removeStockObserver(John_PUSH);
        iphoneProduct.removeStockObserver(Katy_PUSH);

        // NYE Sale - All 18 units sold
        iphoneProduct.purchase(18);
        iphoneProduct.restock(5); // Only Jane & George are notified

        iphoneProduct.purchase(1); // Jane purchases 1 unit
        iphoneProduct.purchase(1); // George purchases 1 unit

        // Jane & George unsubscribe from notifications
        iphoneProduct.removeStockObserver(Jane_EMAIL);
        iphoneProduct.removeStockObserver(George_EMAIL);
    }
}
```

Output

```
-----------------------------------------------------------------------------
###### E-commerce Store - Stock Availability Notification Feature Demo ######
PURCHASE SUCCESS: 10 units of iphone 15 | Remaining stock: 0
PURCHASE FAILED: iphone 15 is out of stock! | Available Quantity: 0
[+]John123 subscribed for notifications on iphone 15
[+]Katy678 subscribed for notifications on iphone 15
[+]Jane783 subscribed for notifications on iphone 15
[+]George993 subscribed for notifications on iphone 15
RESTOCKED: iphone 15 - Added 20 items  | Current stock: 20
Notifying 4 subscribers...
!! PUSH NOTIFICATION SENT to: JohnDeviceP1 - Product is back in stock! Hurry Up!!
!! PUSH NOTIFICATION SENT to: KatyDeviceP2 - Product is back in stock! Hurry Up!!
!! EMAIL SENT to: jane783@gmail.com - Product is back in stock! Hurry Up!!
!! EMAIL SENT to: george993@gmail.com - Product is back in stock! Hurry Up!!
PURCHASE SUCCESS: 1 units of iphone 15 | Remaining stock: 19
PURCHASE SUCCESS: 1 units of iphone 15 | Remaining stock: 18
[-]John123 unsubscribed for notifications on iphone 15
[-]Katy678 unsubscribed for notifications on iphone 15
PURCHASE SUCCESS: 18 units of iphone 15 | Remaining stock: 0
RESTOCKED: iphone 15 - Added 5 items  | Current stock: 5
Notifying 2 subscribers...
!! EMAIL SENT to: jane783@gmail.com - Product is back in stock! Hurry Up!!
!! EMAIL SENT to: george993@gmail.com - Product is back in stock! Hurry Up!!
PURCHASE SUCCESS: 1 units of iphone 15 | Remaining stock: 4
PURCHASE SUCCESS: 1 units of iphone 15 | Remaining stock: 3
[-]Jane783 unsubscribed for notifications on iphone 15
[-]George993 unsubscribed for notifications on iphone 15
```