

## LLD :

Concept & Coding : System Design RoadMap  
Monday, 26 December 2022 11:33 AM

### Low Level Design.

Low Level Design: Patterns to be covered		Low Level Design: Popular Interview Questions to be Covered	
✓ Strategy Pattern	✓	✓ S.O.L.I.D Principles	✓
✓ Observer Pattern	✓	✓ Design Notify-Me Button Functionality	✓
✓ Decorator Pattern	✓	✓ Design Pizza Billing System	✓
✓ Factory Pattern	✓	✓ Design Parking Lot	✓
✓ Abstract Factory Pattern	✓	✓ Design Snake n Ladder game	✓
✓ Chain of Responsibility Pattern	✓	✓ Design Elevator System	✓
✓ Proxy Pattern	✓	✓ Design Car Rental System	✓
✓ Null Object Pattern	✓	✓ Design Logging System	✓
✓ State Pattern	✓	✓ Design Tic-Tac-Toe game	✓
✓ Composite Pattern	✓	✓ Design BookMyShow & Concurrency handling	✓
✓ Adapter Pattern	✓	✓ Design Vending Machine	✓
✓ Singleton Pattern	✓	✓ Design ATM	✓
✓ Builder Pattern	✓	✓ Design Chess game	✓
✓ Prototype Pattern	✓	✓ Design File System	✓

- ① OOPS fundamental  
↓ C++, Java, Python
  - ② SOLID
  - ③ Design Pdd
- = } →

## 2) SOLID Principles

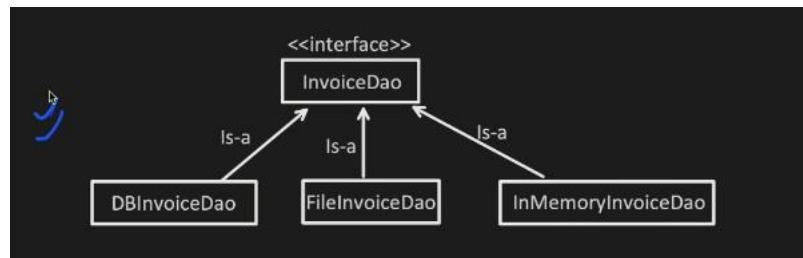
What does SOLID stand for?

- S - Single Responsibility Principle
- O - Open/Closed Principle
- L - Liskov Substitution Principle
- I - Interface Segregation Principle
- D - Dependency Inversion Principle

Advantages of using SOLID Design Principles

- Help us write better code ✓
- Avoid Duplicate Code
- Easy to Maintain
- Easy to Understand
- Flexible Software
- Reduce Complexity

## Open and Closed Principle



## Liskov Substitution Principle

## Code Example: Violating LSP

```
// BAD: This design violates LSP
public interface Bike {

    void turnOnEngine();

    void turnOffEngine();

    void accelerate();

    void applyBrakes();
}
```

```
// Subclass of Bike - implements all Bike class behavior
public class MotorCycle implements Bike {
    String company;
    boolean isEngineOn;
    int speed;

    public MotorCycle(String company, int speed) {
        this.company = company;
        this.speed = speed;
    }
}
```

```
// This class violates LSP!
public class Bicycle implements Bike {
    String brand;
    Boolean hasGears;
    int speed;

    public Bicycle(String brand, Boolean hasGears, int speed) {
        this.brand = brand;
        this.hasGears = hasGears;
        this.speed = speed;
    }
}
```

Is-a                          Is-a

```
int speed;

public MotorCycle(String company, int speed) {
    this.company = company;
    this.speed = speed;
}

@Override
public void turnOnEngine() {
    this.isEngineOn = true; // turn on the engine
    System.out.println("Engine is ON!");
}

@Override
public void turnOffEngine() {
    this.isEngineOn = false; // turn off the engine
    System.out.println("Engine is OFF!");
}

@Override
public void accelerate() {
    this.speed = this.speed + 10; // increase the speed
    System.out.println("MotorCycle Speed: " + this.speed);
}

@Override
public void applyBrakes() {
    this.speed = this.speed - 5; // decrease the speed
    System.out.println("MotorCycle Speed: " + this.speed);
}
```

```
int speed;

public Bicycle(String brand, Boolean hasGears, int speed) {
    this.brand = brand;
    this.hasGears = hasGears;
    this.speed = speed;
}

// LSP Violation: Strengthening preconditions
// Bicycle changes the behavior of turnOnEngine
@Override
public void turnOnEngine() {
    throw new AssertionError("Detail Message: Bicycle has no engine!");
}

// Bicycle changes the behavior of turnOffEngine
@Override
public void turnOffEngine() {
    throw new AssertionError("Detail Message: Bicycle has no engine!");
}

@Override
public void accelerate() {
    this.speed = this.speed + 10; // increase the speed
    System.out.println("Bicycle Speed: " + this.speed);
}

@Override
public void applyBrakes() {
    this.speed = this.speed - 5; // decrease the speed
    System.out.println("Bicycle Speed: " + this.speed);
}
```

**Here, Bicycle cannot fully support Bike because it breaks the established behavior of turnOnEngine() — turnOffEngine() — violating LSP!**

```

// LSP Violation: Strengthening preconditions
// Bicycle changes the behavior of turnOnEngine
@Override
public void turnOnEngine() {
    throw new AssertionError("Detail Message: Bicycle has no engine!");
}
// Bicycle changes the behavior of turnOffEngine
@Override
public void turnOffEngine() {
    throw new AssertionError("Detail Message: Bicycle has no engine!");
}

@Override
public void accelerate() {
    this.speed = this.speed + 10; // increase the speed
    System.out.println("Bicycle Speed: " + this.speed);
}

@Override
public void applyBrakes() {
    this.speed = this.speed - 5; // decrease the speed
    System.out.println("Bicycle Speed: " + this.speed);
}

```

Here, Bicycle cannot fully s  
Bike because it breaks the  
behavior of turnOnEngine(  
turnOffEngine()) — violatin

```

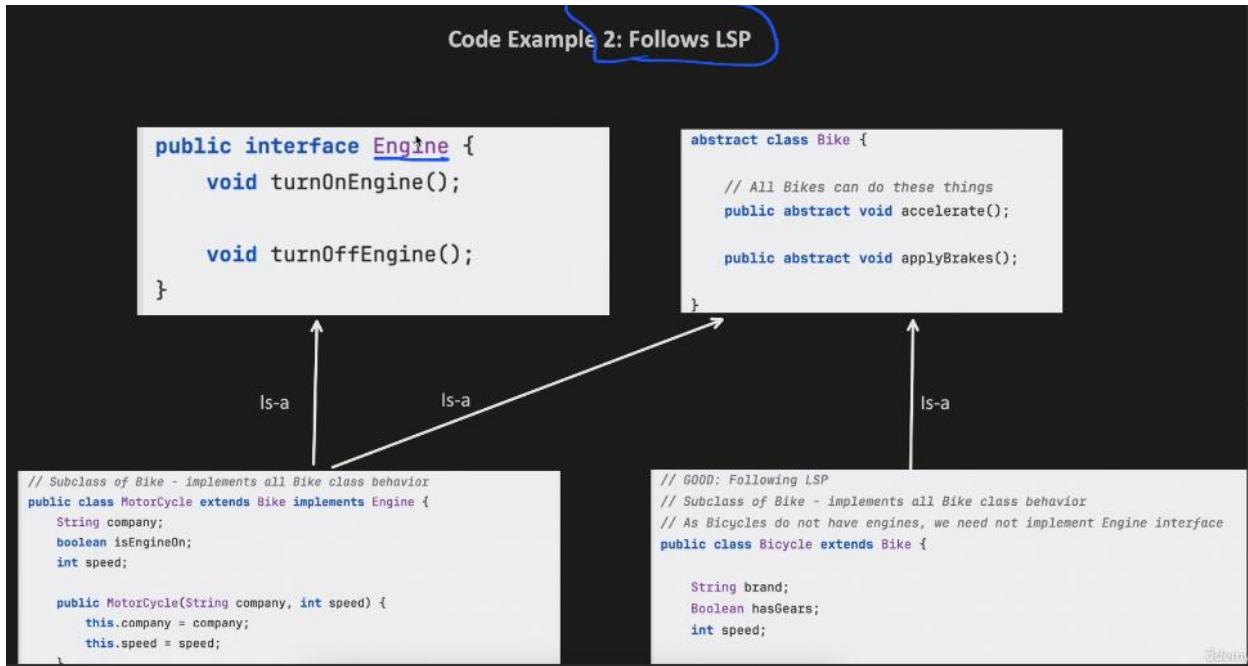
// Usage example - demonstrates the LSP violations
public class Demo {
    public static void main(String[] args) {
        // create the objects
        Bike bikeObj = new MotorCycle("HeroHonda", 10);

        // use the objects
        // Works fine with MotorCycle - implements all Bike class behavior
        bikeObj.turnOnEngine();
        bikeObj.accelerate();
        bikeObj.applyBrakes();
        bikeObj.turnOffEngine();

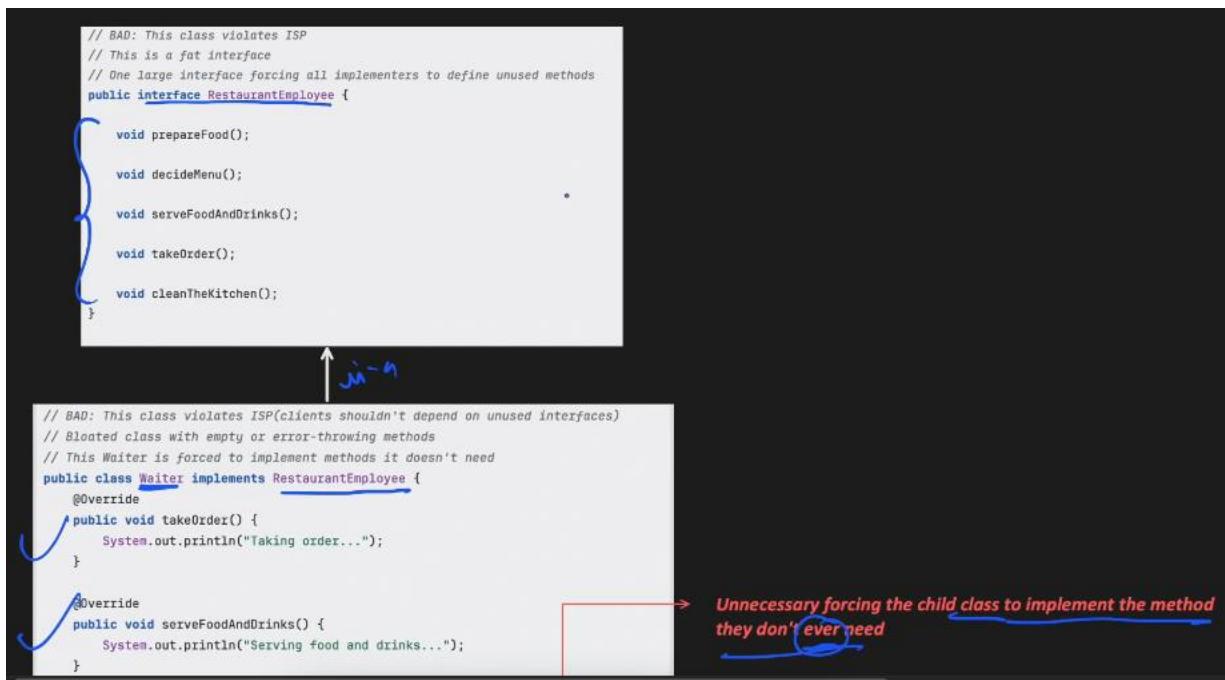
        // Client expects to be able to see the same behavior with Bicycle
        bikeObj = new Bicycle("Hercules", true, 10);
        bikeObj.turnOnEngine(); // fails to implement Bike class behavior
        bikeObj.accelerate();
        bikeObj.applyBrakes();
        bikeObj.turnOffEngine(); // fails to implement Bike class behavior
    }
}

```

*Replaces one implementation with another and now its breaking the functionality*



### Interface Segregation Principle :



```
// BAD: This class violates ISP(clients shouldn't depend on unused interfaces)
// Bloated class with empty or error-throwing methods
// This Waiter is forced to implement methods it doesn't need
public class Waiter implements RestaurantEmployee {
    @Override
    public void takeOrder() {
        System.out.println("Taking order...");
    }

    @Override
    public void serveFoodAndDrinks() {
        System.out.println("Serving food and drinks...");
    }

    @Override
    public void cleanTheKitchen() {
        // Forced to implement but doesn't make sense for a waiter
        throw new AssertionError("Detail Message: Waiter cannot clean the kitchen!");
    }

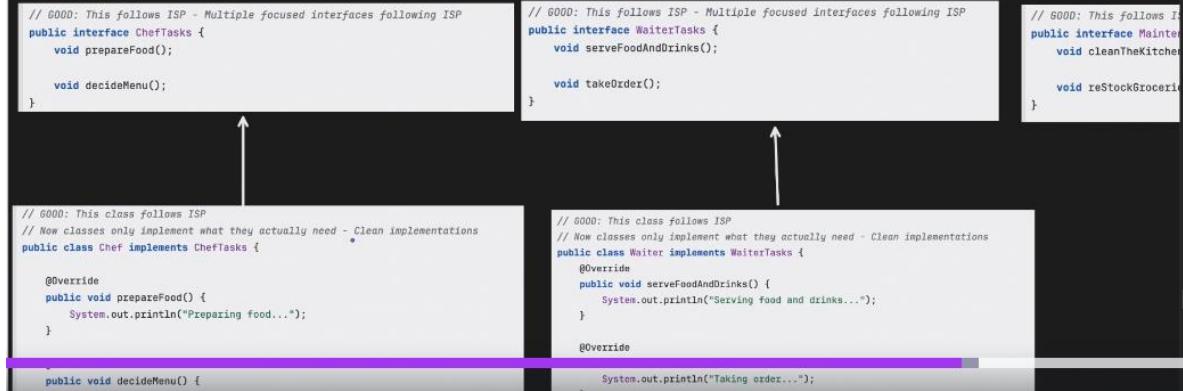
    @Override
    public void prepareFood() {
        // Forced to implement but doesn't make sense for a waiter
        throw new AssertionError("Detail Message: Waiter cannot prepare food!");
    }

    @Override
    public void decideMenu() {
        // Forced to implement but doesn't make sense for a waiter
        throw new AssertionError("Detail Message: Waiter cannot decide the menu!");
    }
}
```

*Unnecessary forcing the child class to implement the method they don't ever need*

**Problems with the Above Code:**

- Classes are forced to implement methods they don't support.
- Code becomes bloated with empty or error-throwing methods.
- Violates the principle that clients shouldn't depend on unused interfaces.



**Interface** Defines a **strict contract** of methods with no implementation. Describes *what* a class can do.

**Abstract Class** A **blueprint** that can provide partial implementation (shared code) along with unimplemented abstract methods. It cannot be instantiated.

**Concrete Class** A **full implementation** of all methods (inherited or original). It can be instantiated into objects.

## Dependency Inversion Principle :

It states that "**high-level components should not depend on low-level components directly; instead, they should depend on abstractions.**"

```
public interface Keyboard {
    void getSpecifications();
}

// Low-level module - concrete implementation
public class WiredKeyboard implements Keyboard {
    private final String connectionType;
    private final String company;
    private final String modelVersion;

    public WiredKeyboard(String connectionType, String company, String modelVersion) {
        this.connectionType = connectionType;
        this.company = company;
        this.modelVersion = modelVersion;
    }
}

// Low-level module - concrete implementation
public class BluetoothKeyboard implements Keyboard {
    private final String connectionType;
    private final String company;
    private final String modelVersion;

    public BluetoothKeyboard(String connectionType, String company, String modelVersion) {
        this.connectionType = connectionType;
        this.company = company;
        this.modelVersion = modelVersion;
    }
}
```

## 3. Liskov Substitution Principle (LSP) Solution

VALID Solution :

The diagram illustrates a class hierarchy and a main program. At the top is the `Vehicle` class:

```
public class Vehicle {  
    public Integer getNumberOfWheels(){  
        return 2;  
    }  
  
    public Boolean hasEngine(){  
        return true;  
    }  
}
```

Below it, two classes inherit from `Vehicle`: `MotorCycle` and `Car`. A checkmark icon is positioned above the `Vehicle` class, indicating it is correct.

```
public class MotorCycle extends Vehicle{  
}  
  
public class Car extends Vehicle{  
    @Override  
    public Integer getNumberOfWheels() {  
        return 4;  
    }  
  
    @Override  
    public Integer getNumberOfWheels() {  
        return 4;  
    }  
}
```

At the bottom is the `Main` class, which contains a list of `Vehicle` objects and prints the result of the `hasEngine()` method for each.

```
public class Main {  
    public static void main(String args[]){  
  
        List<Vehicle> vehicleList = new ArrayList<>();  
        vehicleList.add(new MotorCycle());  
        vehicleList.add(new Car());  
  
        for(Vehicle vehicle : vehicleList){  
            System.out.println(vehicle.hasEngine().toString());  
        }  
    }  
}
```

Problem :

```

public class Vehicle {
    public Integer getNumberOfWheels(){
        return 2;
    }
    public Boolean hasEngine(){
        return true;
    }
}

class MotorCycle extends Vehicle{
}

public class Car extends Vehicle{
    @Override
    public Integer getNumberOfWheels() {
}
}

public class Bicycle extends Vehicle{
    public Boolean hasEngine(){
        return null;
    }
}

```

Handwritten annotations:

- A checkmark is next to the MotorCycle class definition.
- A checkmark is next to the Car class definition.
- A checkmark is next to the hasEngine() method in the Car class.
- A checkmark is next to the hasEngine() method in the Bicycle class.
- A large curly brace groups the Vehicle, MotorCycle, and Car classes, indicating they are part of the same inheritance hierarchy.
- A curly brace groups the hasEngine() methods in the Car and Bicycle classes, with the word "NPE" written next to it, indicating a Null Pointer Exception.
- The word "null-dash" is written near the NPE annotation.
- A checkmark is next to the Main class definition.
- A checkmark is next to the main() method in the Main class.
- A checkmark is next to the add() calls in the vehicleList.add() statements.
- A checkmark is next to the System.out.println() statement.
- A checkmark is next to the closing brace of the for loop.
- A checkmark is next to the closing brace of the Main class.

```

public class Main {
    public static void main(String args[]){
        List<Vehicle> vehicleList = new ArrayList<>();
        vehicleList.add(new MotorCycle()); ✓
        vehicleList.add(new Car()); ✓
        vehicleList.add(new Bicycle()); ✓

        for(Vehicle vehicle : vehicleList){
            System.out.println(vehicle.hasEngine().toString());
        }
    }
}

```

```

Exception in thread "main" java.lang.NullPointerException Create breakpoint
at LiskovPrinciple.Main.main(Main.java:14)

```

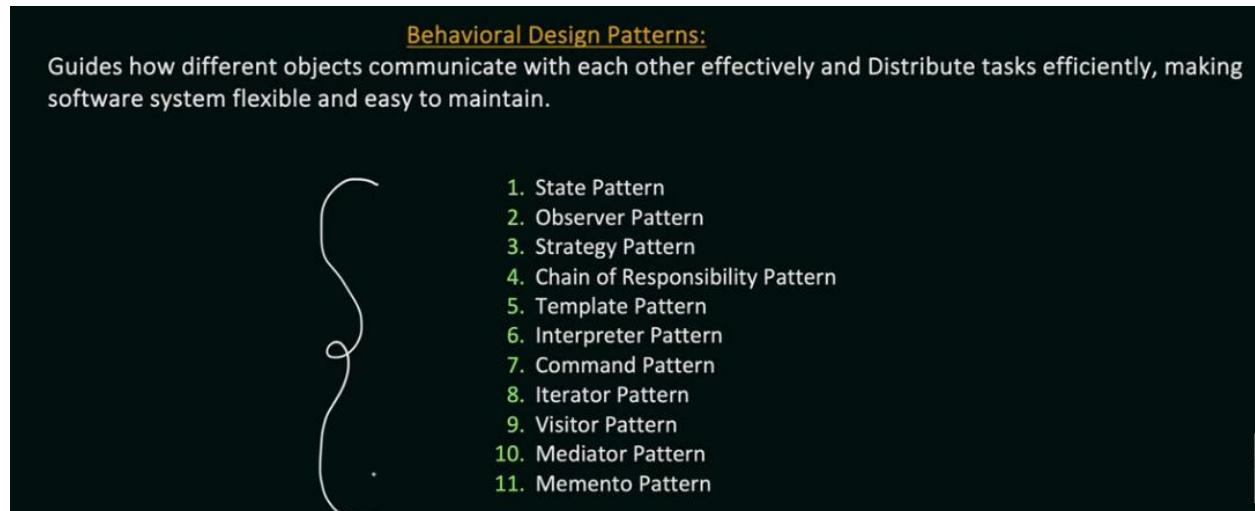
Solution :



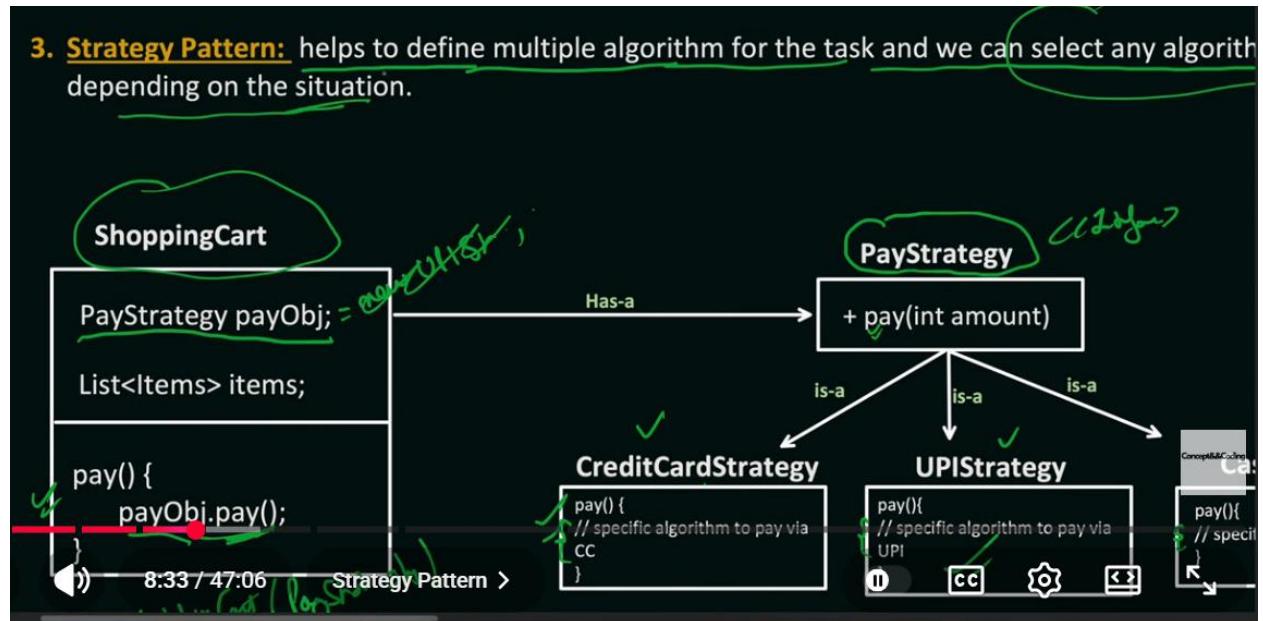
```
public class Main {  
    public static void main(String args[]){  
  
        List<Vehicle> vehicleList = new ArrayList<>();  
        vehicleList.add(new MotorCycle());  
        vehicleList.add(new Car());  
        vehicleList.add(new Bicycle());  
  
        for(Vehicle vehicle : vehicleList){  
            System.out.println(vehicle.hasEngine());  
        }  
    }  
}
```

```
public class Main {  
    public static void main(String args[]){  
  
        List<EngineVehicle> vehicleList = new ArrayList<>();  
        vehicleList.add(new MotorCycle());✓  
        vehicleList.add(new Car());✓  
        vehicleList.add(new Bicycle());✗  
  
        for(EngineVehicle vehicle : vehicleList){  
            System.out.println(vehicle.hasEngine());  
        }  
    }  
}
```

## Behavior Design Pattern :



## 4. . Strategy Design Pattern (Behavioral Pattern)



## Violating Strategy design pattern :

**Problem:**

```

1 public class Vehicle {
2     public void drive() {
3         System.out.print("\n" + this.getClass().getSimpleName() + ": ");
4         System.out.println("Driving Capability: Normal");
5     }
6 }
7 }

1 public class SportsVehicle extends Vehicle {
2
3     // Overriding the drive method to provide specific behavior for sports vehicles
4     public void drive() {
5         System.out.print("\n" + this.getClass().getSimpleName() + ": ");
6         System.out.println("Driving Capability: Sports");
7     }
8 }

```

*Sport drive functionality*

```

1 public class PassengerVehicle extends Vehicle {
2
3     // Reusing the existing drive method from the parent class
4     // Driving Capability: Normal
5     // No new implementation required
6 }

```

*Normal drive functionality*

1x C 0:38 / 6:43 ⏴

### Following Strategy design pattern :

```

1 // Context class - holds a reference to a strategy object
2 public class Vehicle {
3     DriveStrategy driveStrategy;
4
5     // constructor injection
6     public Vehicle(DriveStrategy driveStrategy) {
7         this.driveStrategy = driveStrategy;
8     }
9
10    public void drive() {
11        System.out.print("\n" + this.getClass().getSimpleName() + ": ");
12        driveStrategy.drive();
13    }
14 }

```

```

1 // Strategy interface - defines the drive behavior
2 public interface DriveStrategy {
3     public void drive();
4 }

```

```

1 // Concrete context subclass
2 public class GoodsVehicle extends Vehicle {
3
4     public GoodsVehicle(DriveStrategy driveStrategy) {
5         super(driveStrategy);
6     }
7 }

```

```

1 // Concrete strategy for normal drive mode
2 public class NormalDrive implements DriveStrategy {
3     @Override
4     public void drive() {
5         System.out.println("Driving Capability: Normal");
6     }
7 }

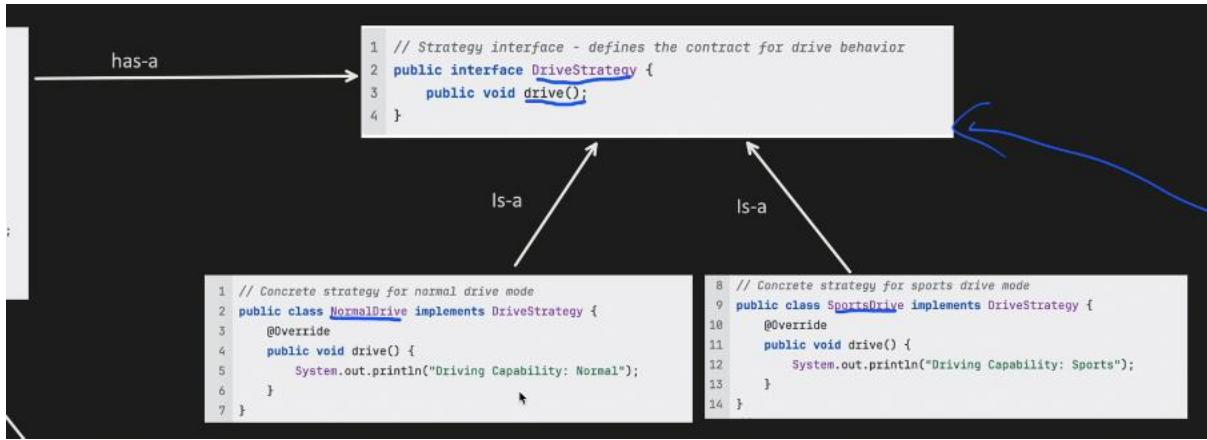
```

```

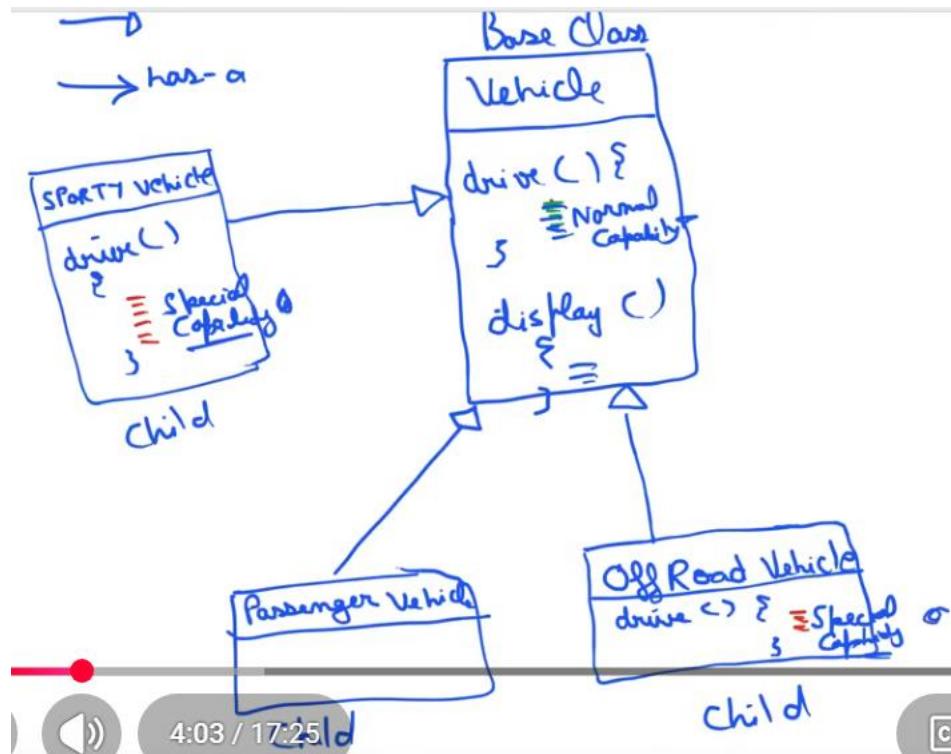
1 // Concrete context subclass
2 public class OffRoadVehicle extends Vehicle {
3
4     OffRoadVehicle(DriveStrategy driveStrategy) {
5         super(driveStrategy);
6     }
7 }

```

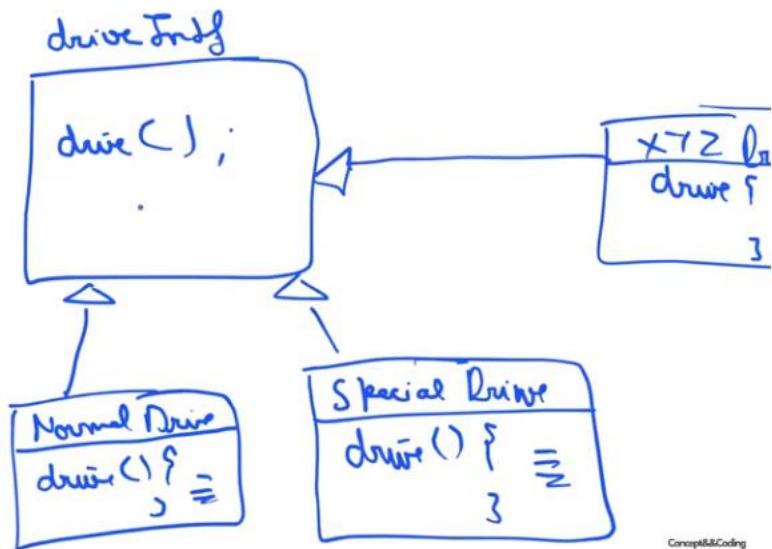
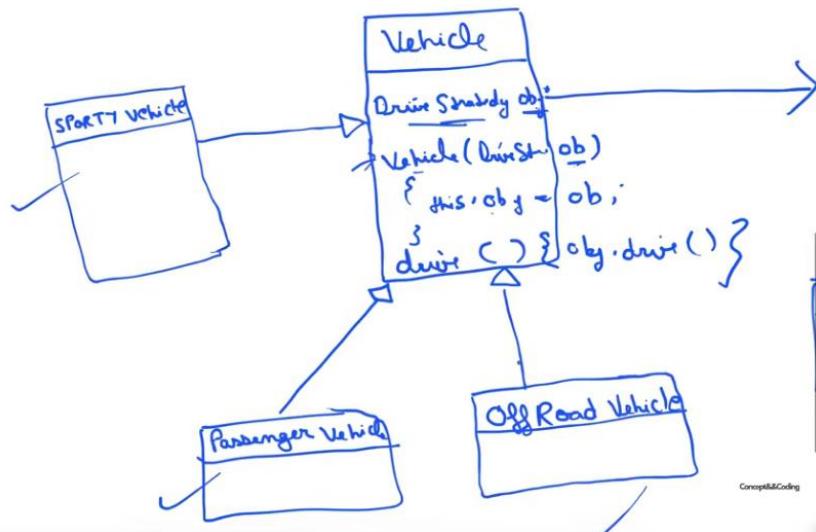
1x C 4:00 / 6:43 ⏴



Without Strategy pattern :



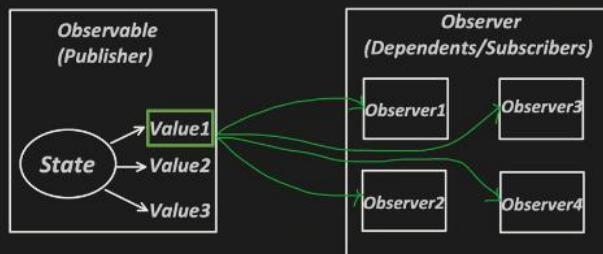
With strategy pattern :



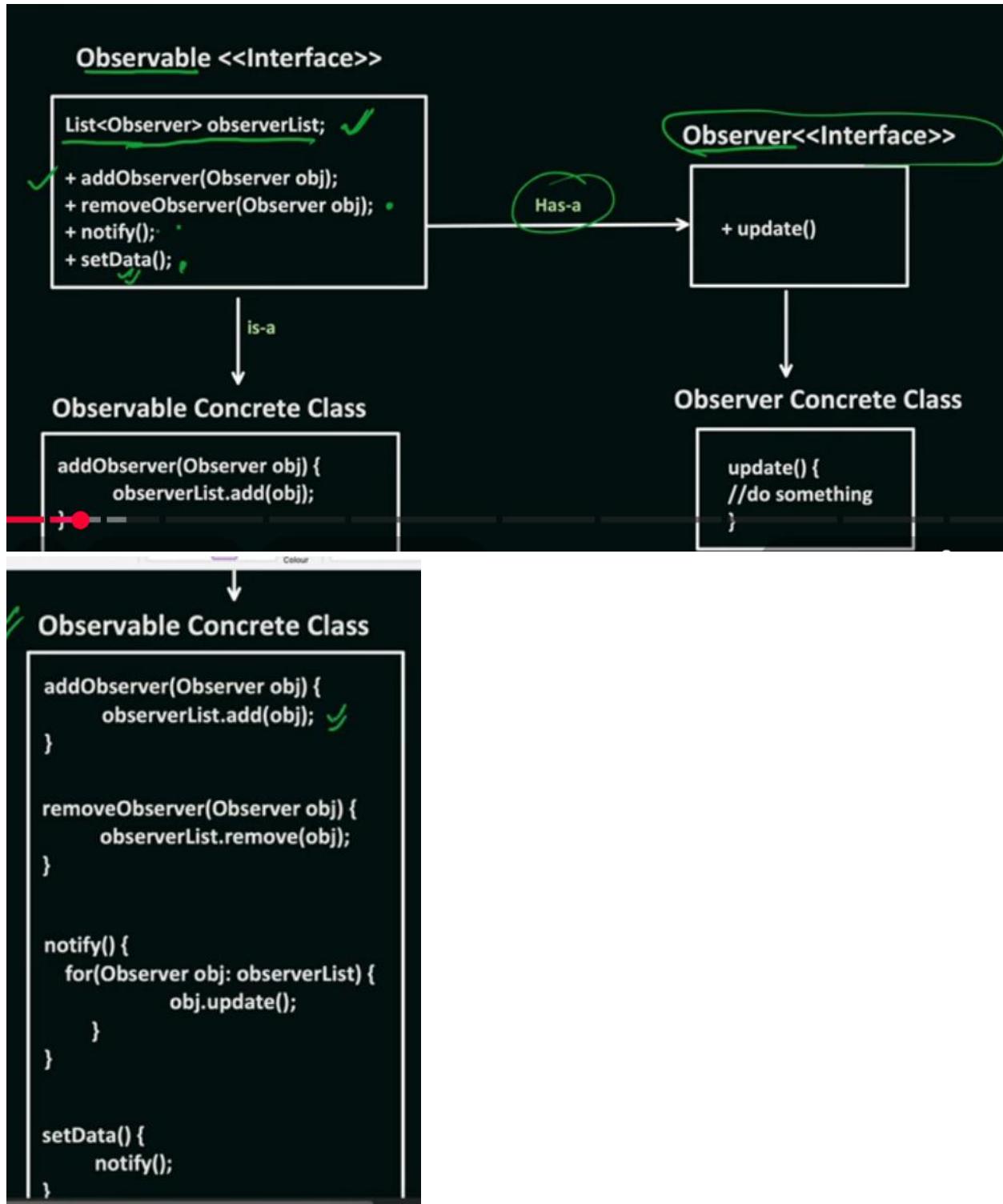
## 5. Observer Pattern :

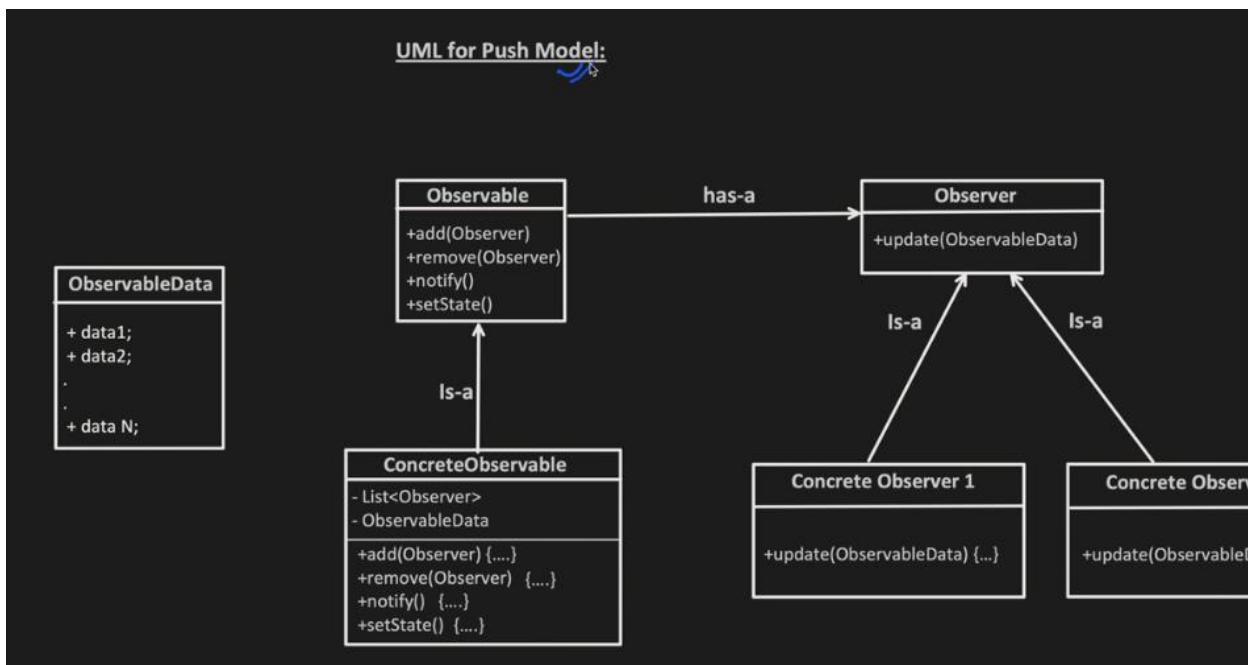
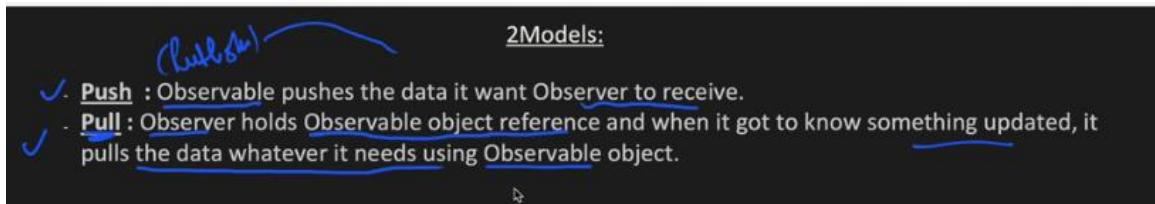
### Definition:

- It's a design pattern where an object (aka "observable" or "publisher") maintains a list of dependents (called "observers").
- And automatically notifies dependents/observers whenever there is a change in its state.

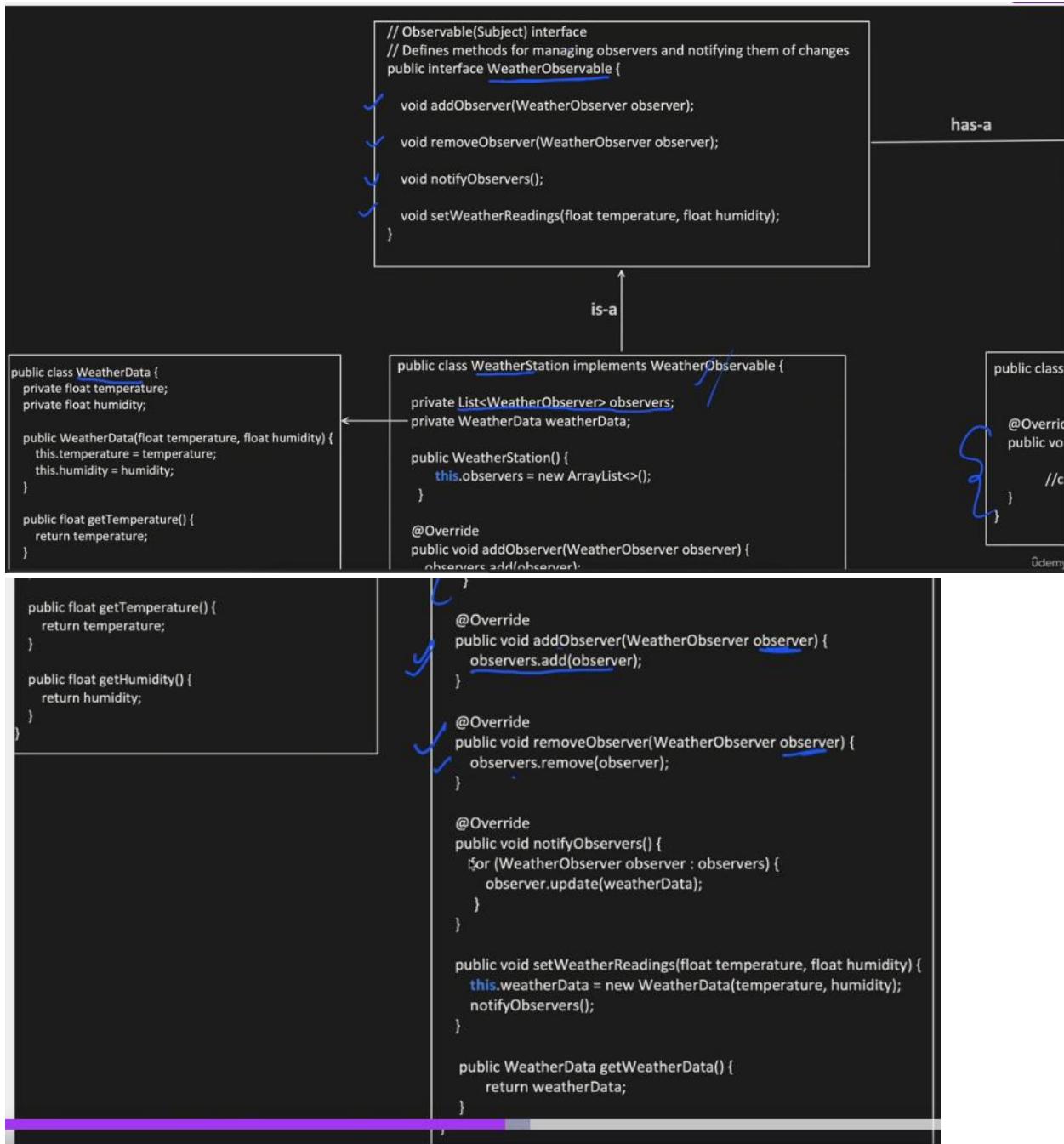


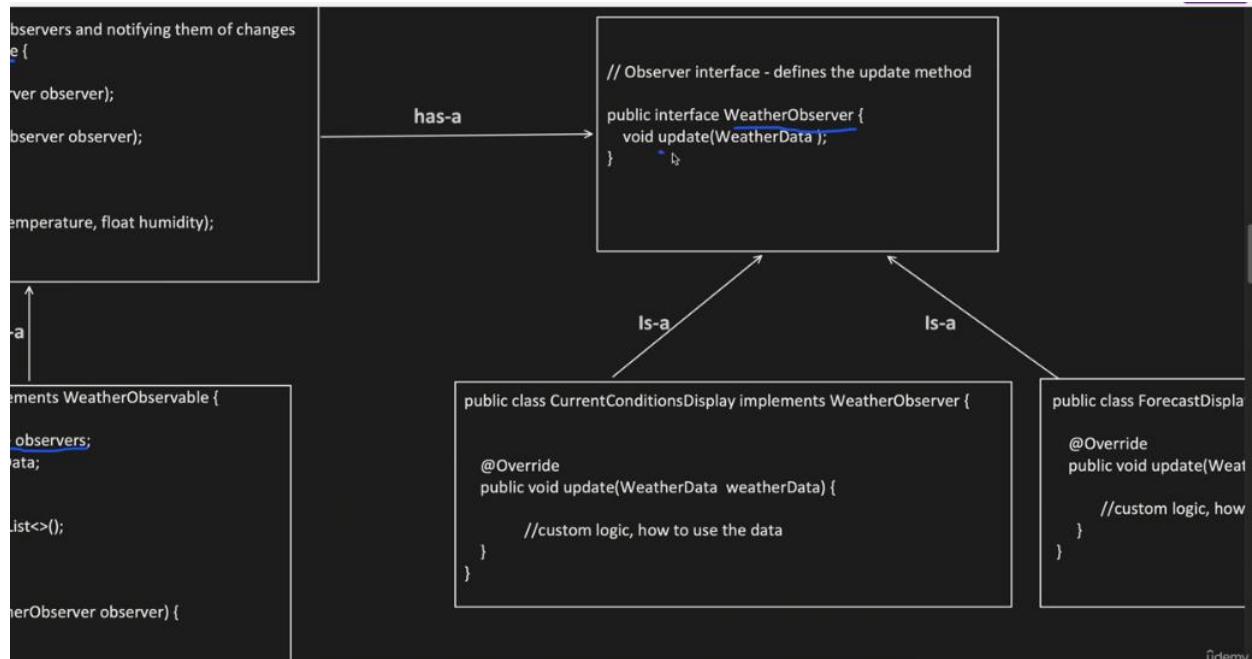
2. **Observer Pattern:** in this an object (Observable) maintains a list of its dependents (observers) and notifies them of any changes in its state.





Push Model Example :





*// Client code to demonstrate the Observer Pattern*

```

public class WeatherStationApp {

    public static void main(String[] args) {
        // Create the weather station (observable/subject)
        WeatherObservable weatherStation = new WeatherStation();

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

        // add Observers
        weatherStation.addObserver(currentDisplay);
        weatherStation.addObserver(forecastDisplay);

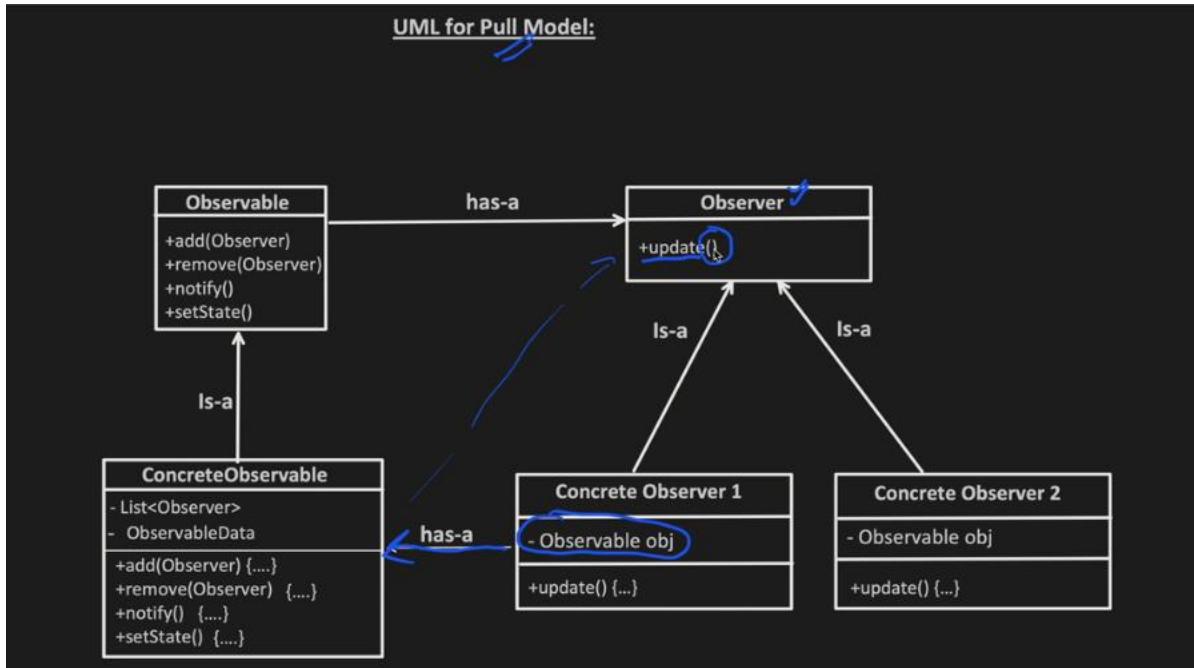
        //weather update
        weatherStation.setWeatherReadings(30.4f, 65f);

        //remove observer
        weatherStation.removeObserver(forecastDisplay);
    }
}

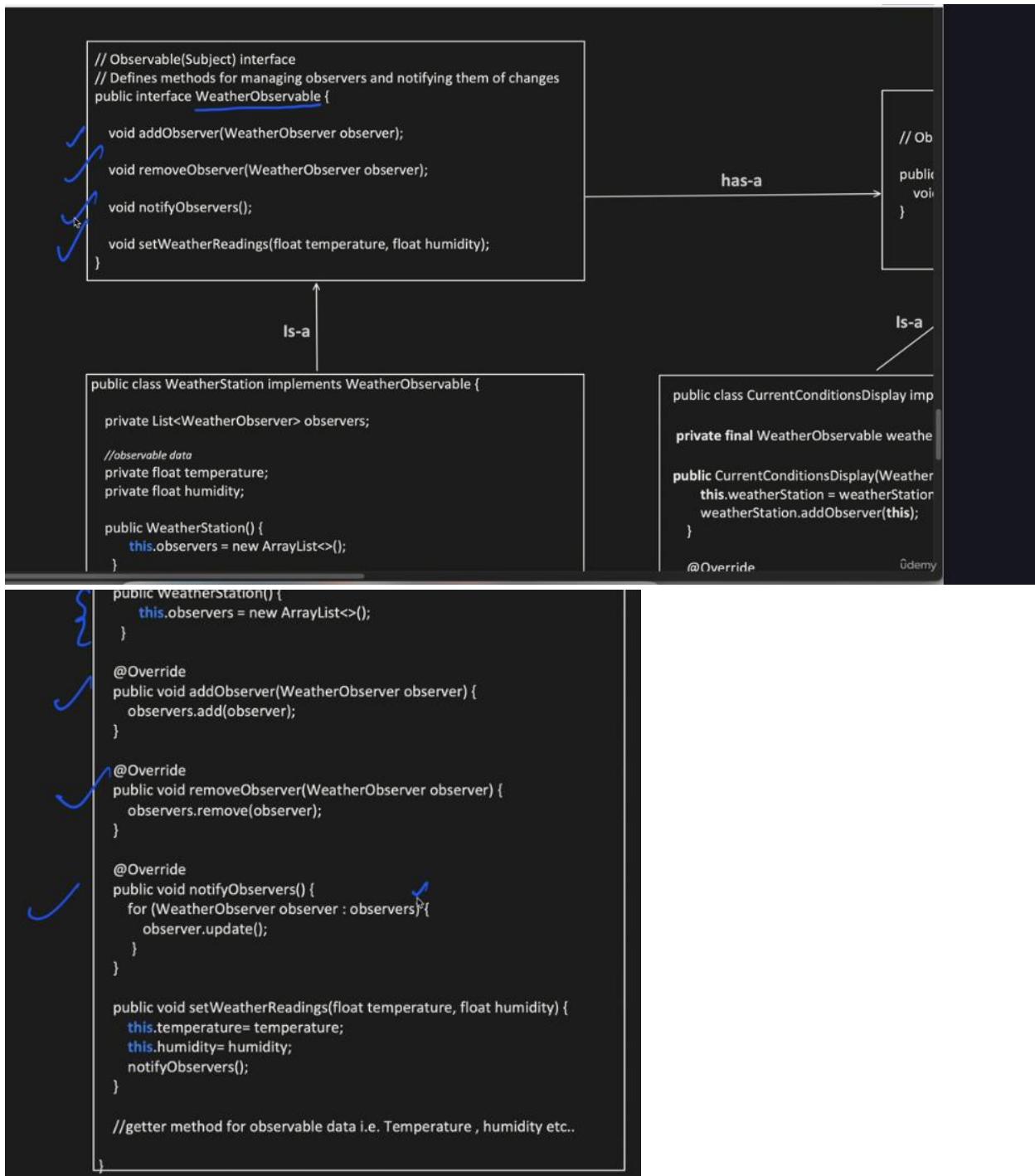
```

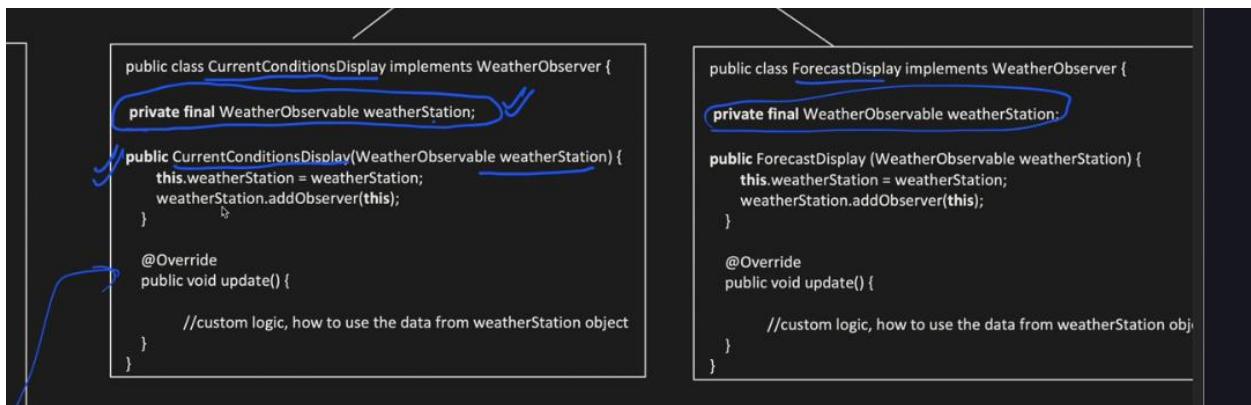
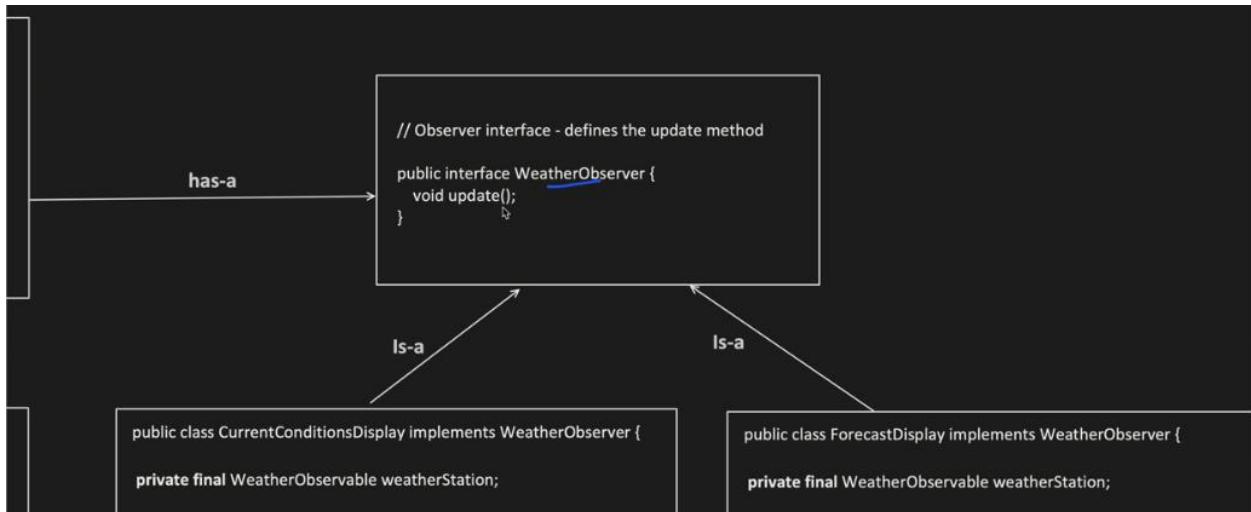
*Subject*

This block contains the client-side code demonstrating the Observer pattern. It shows the creation of a **WeatherStation** object (the subject), the creation of two display objects (the observers), and the addition of these observers to the subject. It also demonstrates triggering an update and removing an observer. Handwritten annotations include "Subject" with a circled arrow pointing to the **WeatherStation** class and "Observer" with a circled arrow pointing to the **CurrentConditionsDisplay** and **ForecastDisplay** classes.



Pull Model Example :





*Observes*

*Published*

*Subscribes*

*Other*

```

public class WeatherStationApp {
    public static void main(String[] args) {
        // Create the weather station (observable/subject)
        WeatherObservable weatherStation = new WeatherStation();

        // Create displays (observers), so now each observer , holds the reference of Station and also add itself to the list.
        CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay(weatherStation);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherStation);

        // update weather state
        weatherStation.setWeatherReadings(80f, 65f);

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

```

This block contains client code demonstrating the Observer pattern. It shows the creation of a `WeatherStation` object (the subject) and two `Display` objects (`CurrentConditionsDisplay` and `ForecastDisplay`) that observe the station. The `WeatherStation` updates its observers by calling `removeObserver` on the `forecastDisplay`.