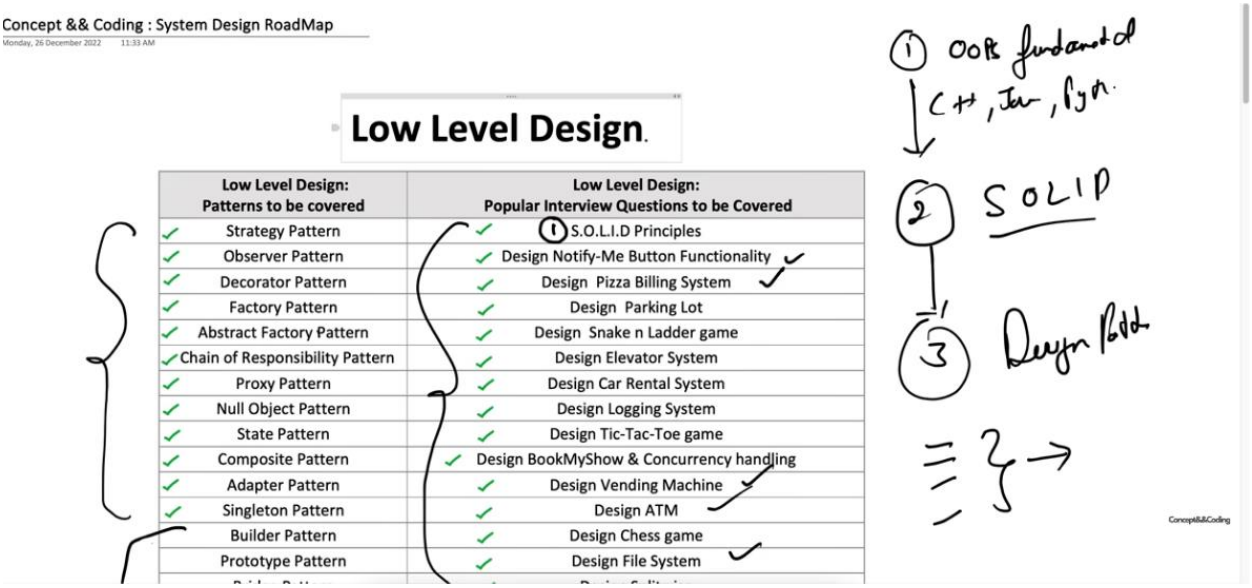
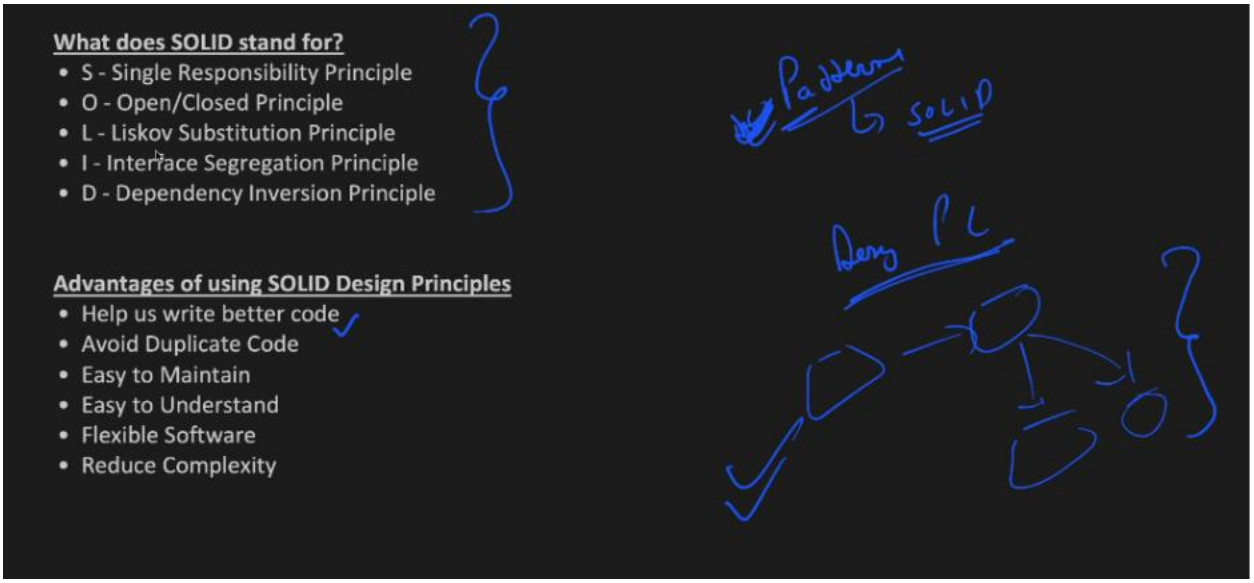
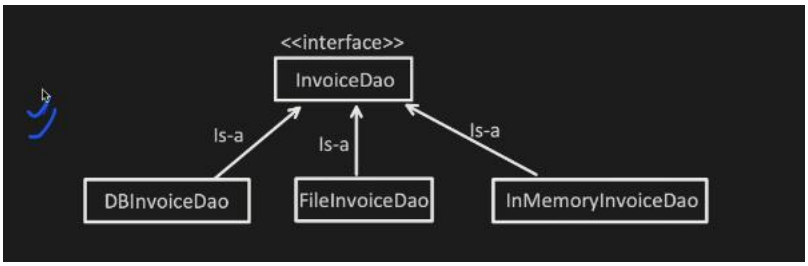
LLD :



2) SOLID Principles

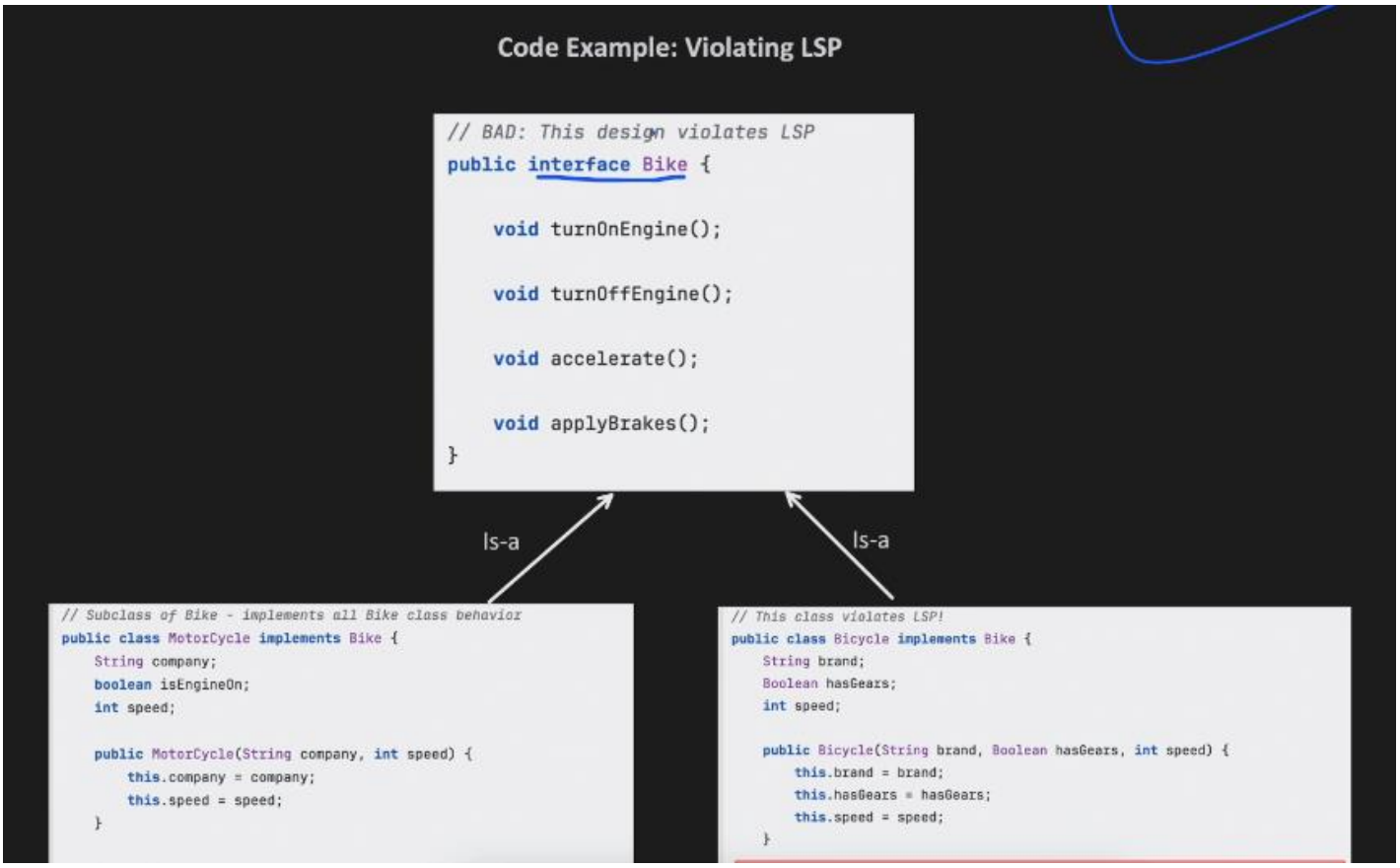


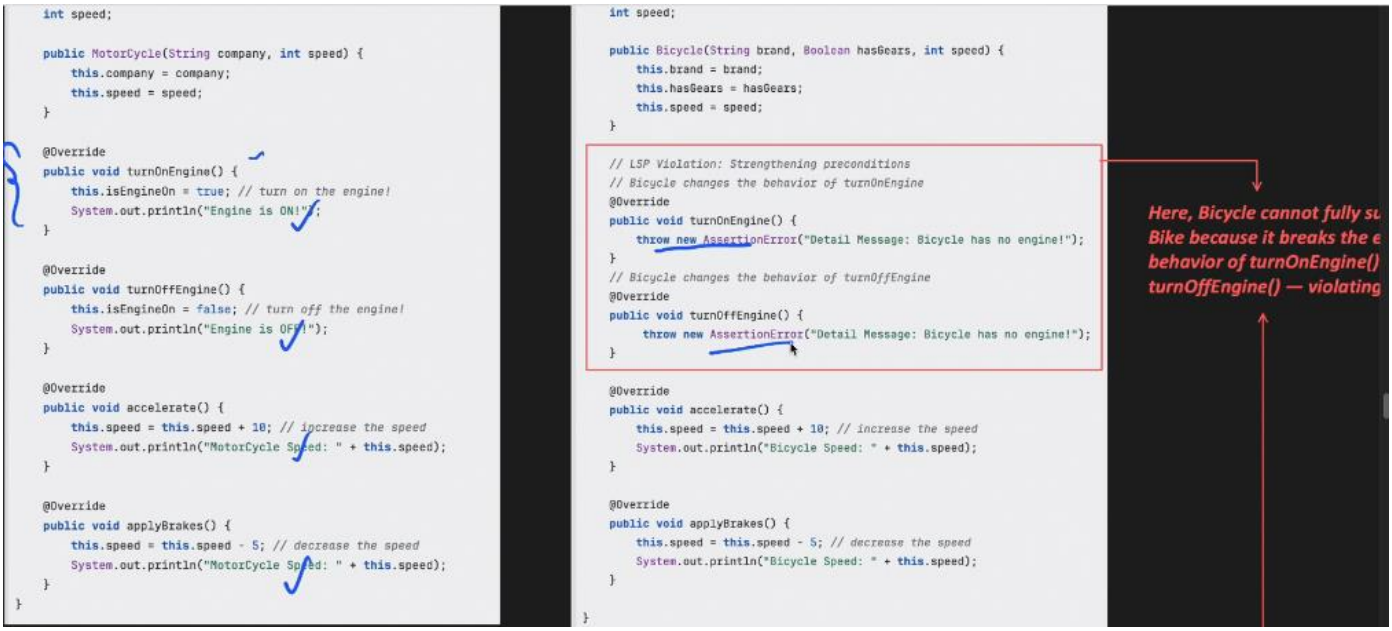
Open and Closed Principle

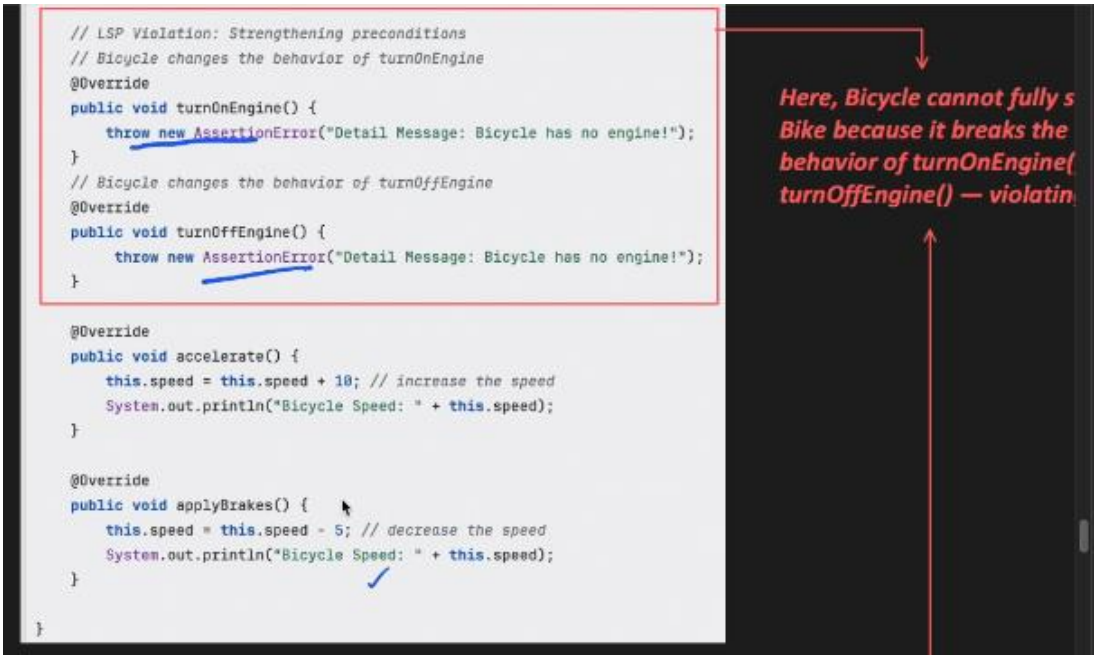


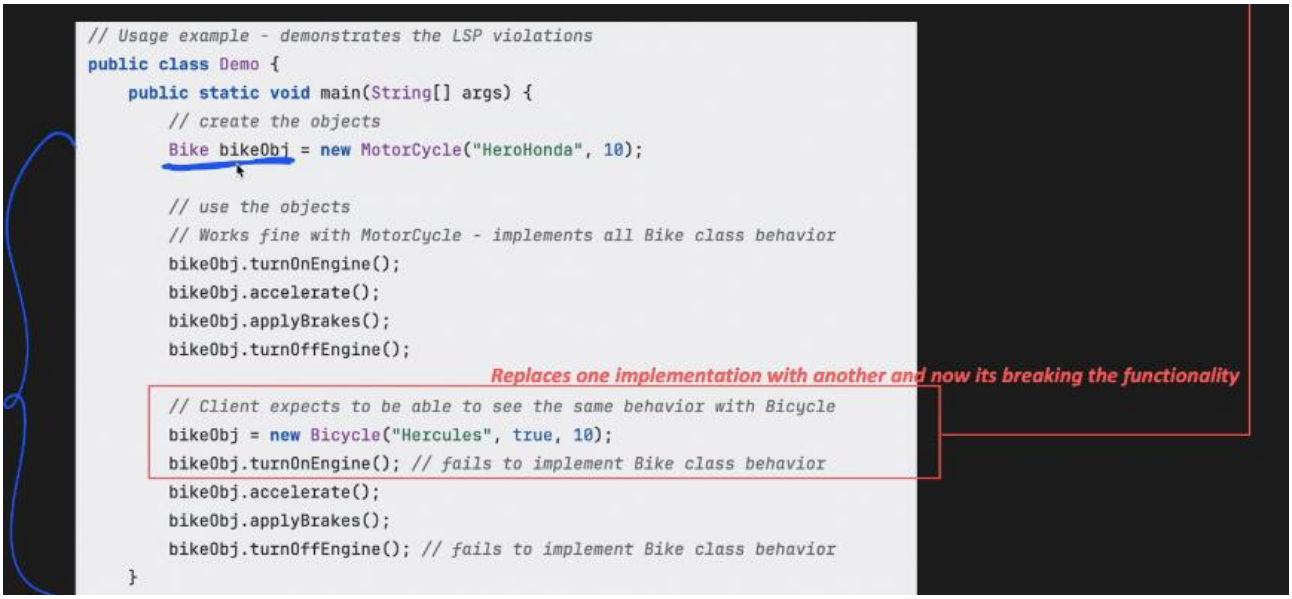
Liskov Substitution Principle

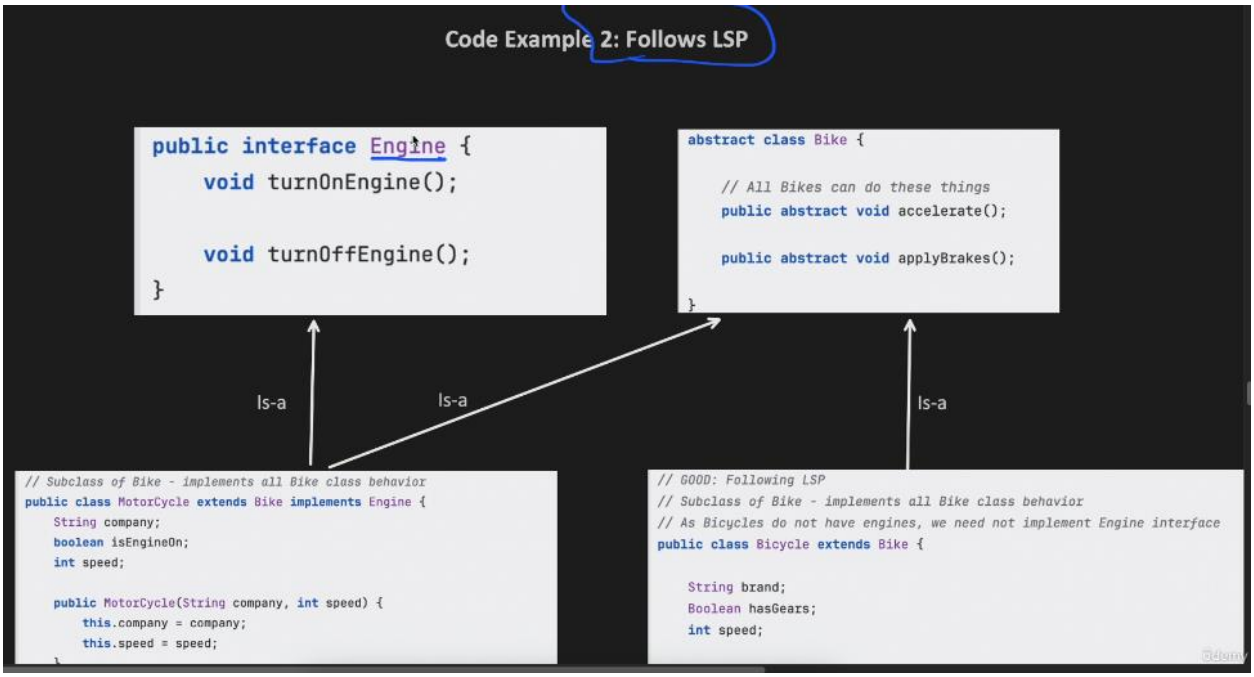
[AAR Proprietary]



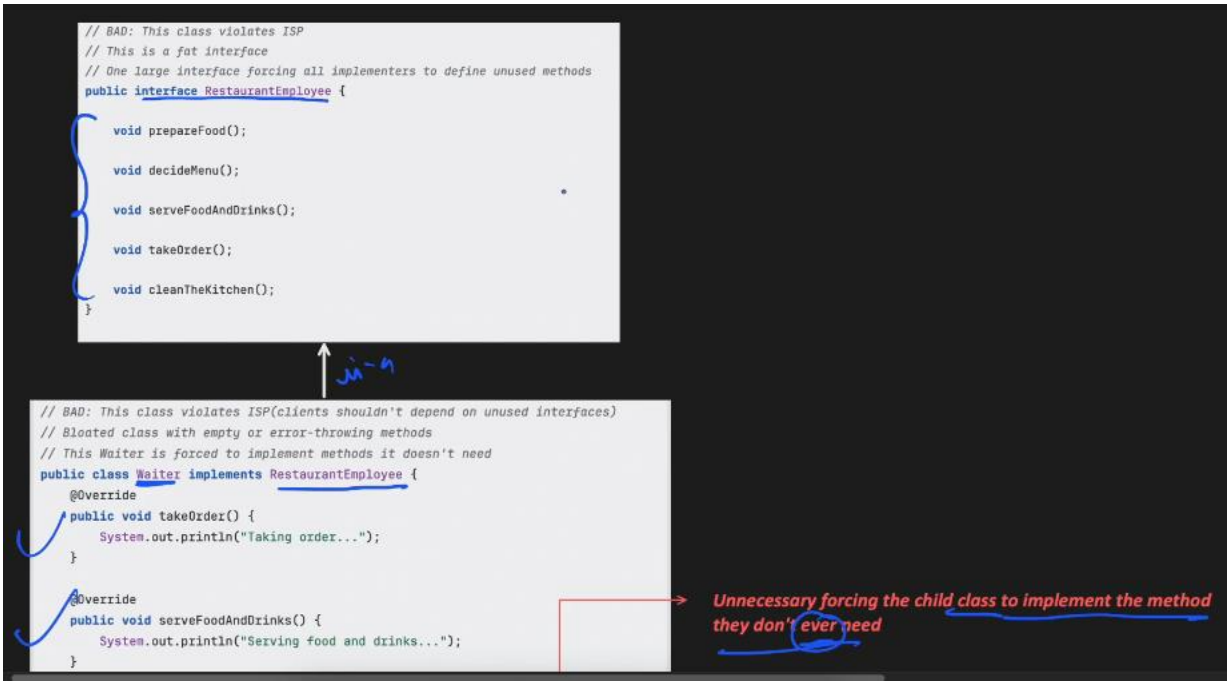


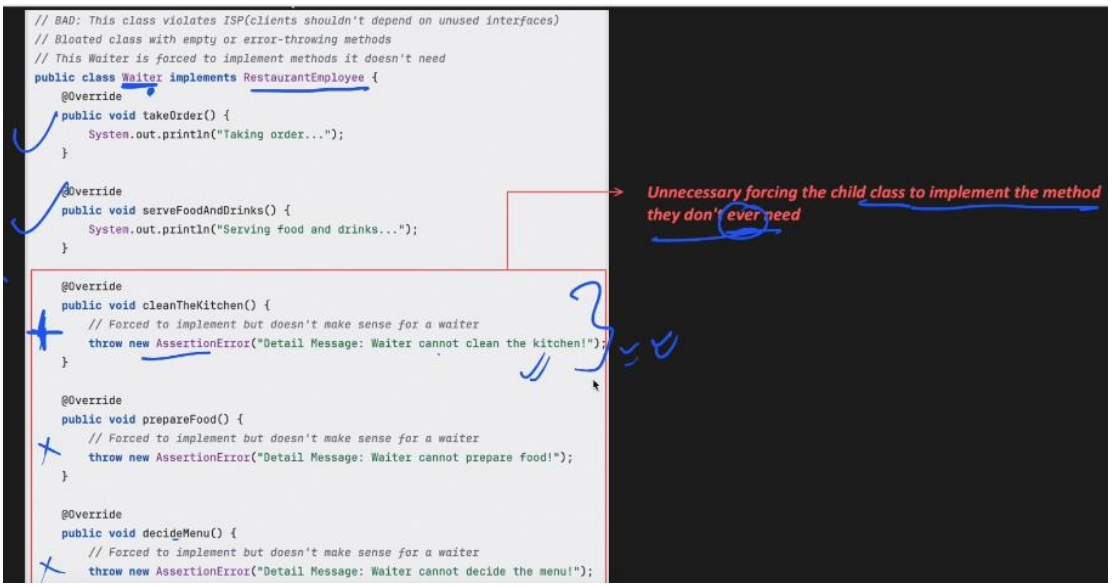

// LSP Violation: Strengthening preconditions
// Bicycle changes the behavior of turnOnEngine
@Override
public void turnOnEngine() {
throw new AssertionError("Detail Message: Bicycle has no engine!");
}
// Bicycle changes the behavior of turnOffEngine
@Override
public void turnOffEngine() {
throw new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

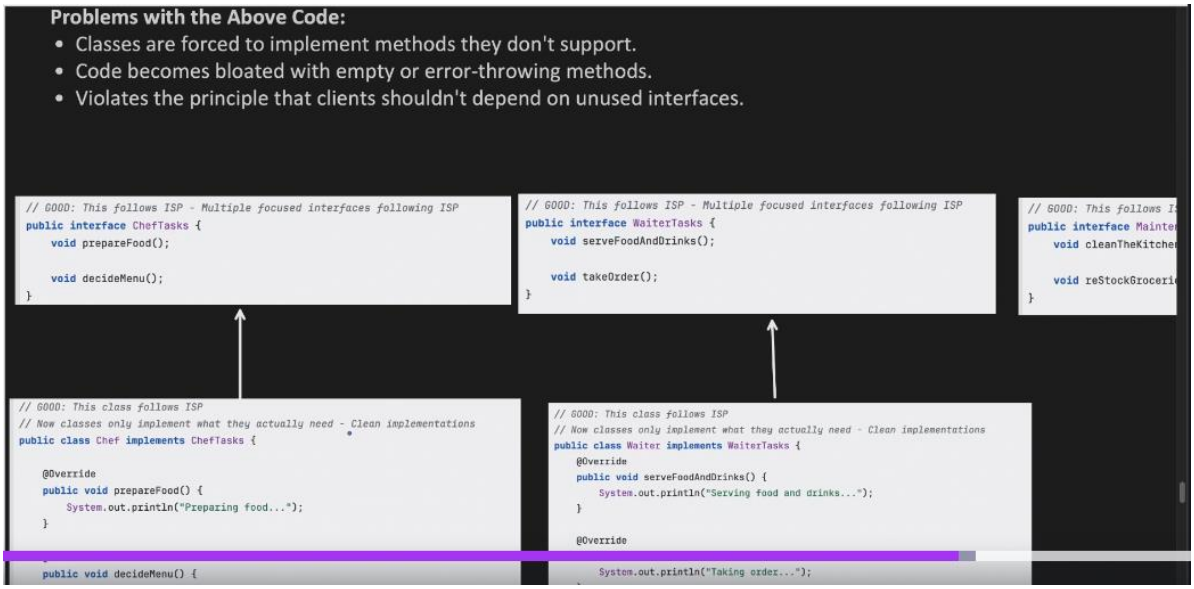

// Usage example - demonstrates the LSP violations
public class Demo {
public static void main(String[] args) {
// create the objects
Bike bikeObj = new MotorCycle("HeroHonda", 10);
// use the objects
// Works fine with MotorCycle - implements all Bike class behavior
bikeObj.turnOnEngine();
bikeObj.accelerate();
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 :

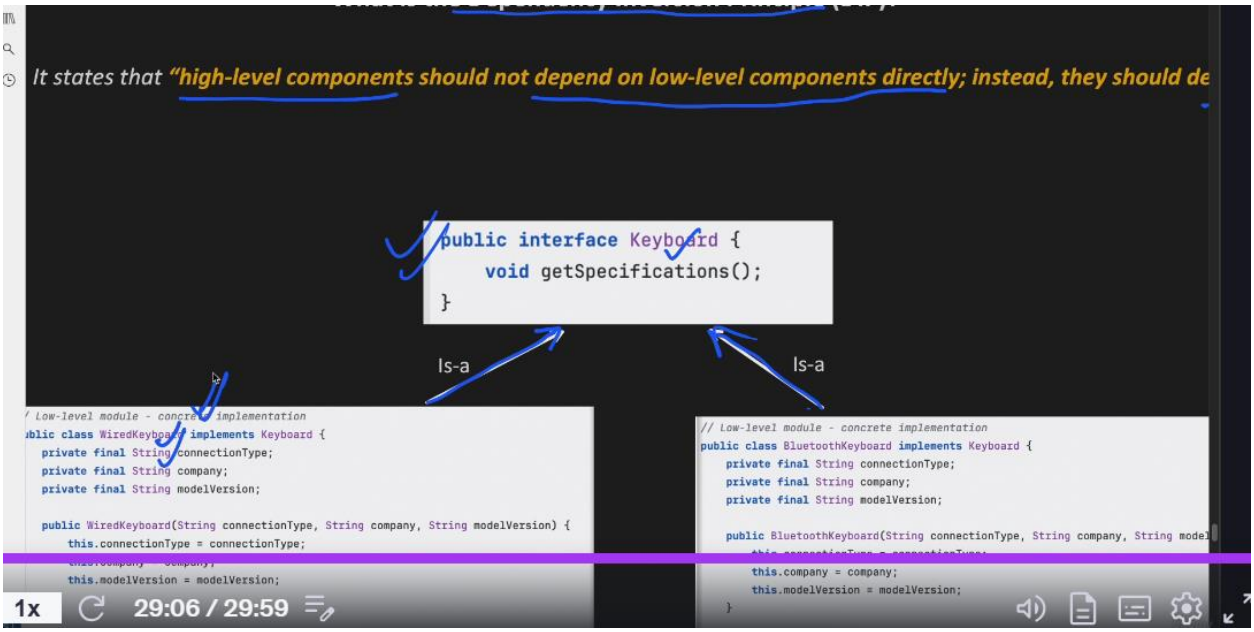






Interface	Defines a **strict contract** of methods with no implementation. Describes *what* a class can do.
Abstract Class	A **blueprint** that can provide partial implementation (shared code) along with unimplemented abstract methods. It cannot be instantiated.
Concrete Class	A **full implementation** of all methods (inherited or original). It can be instantiated into objects.

Dependency Inversion Principle :

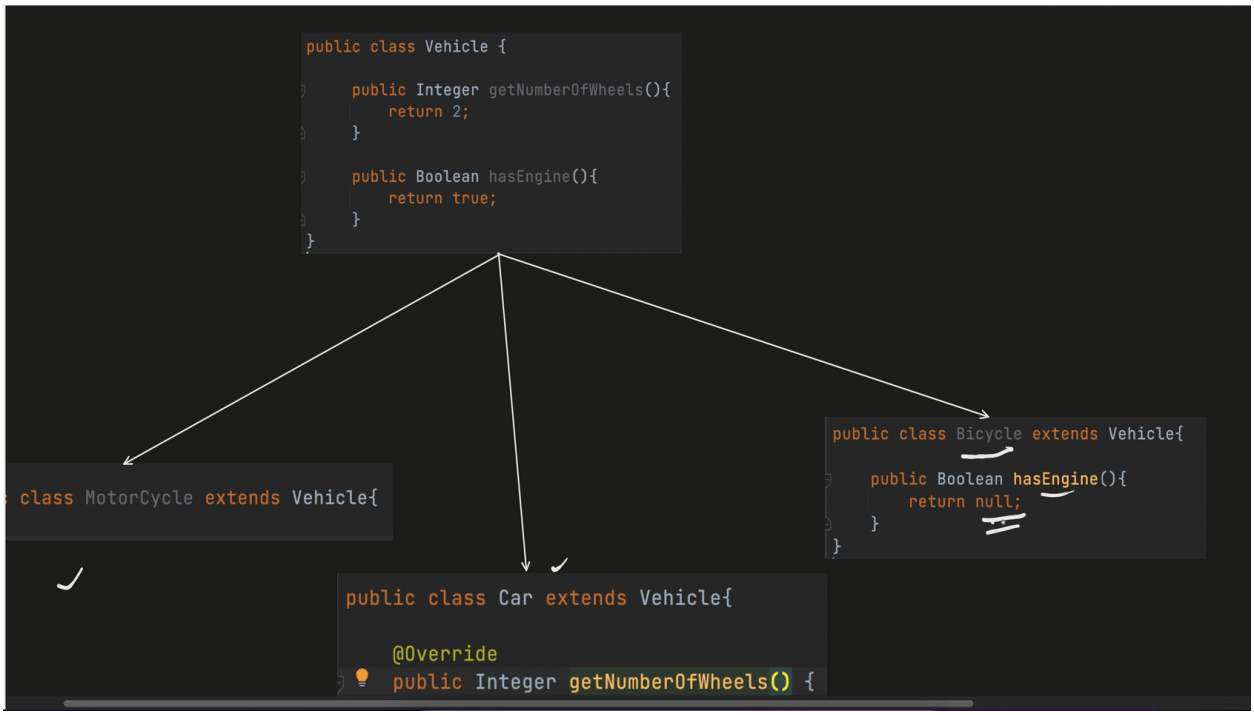


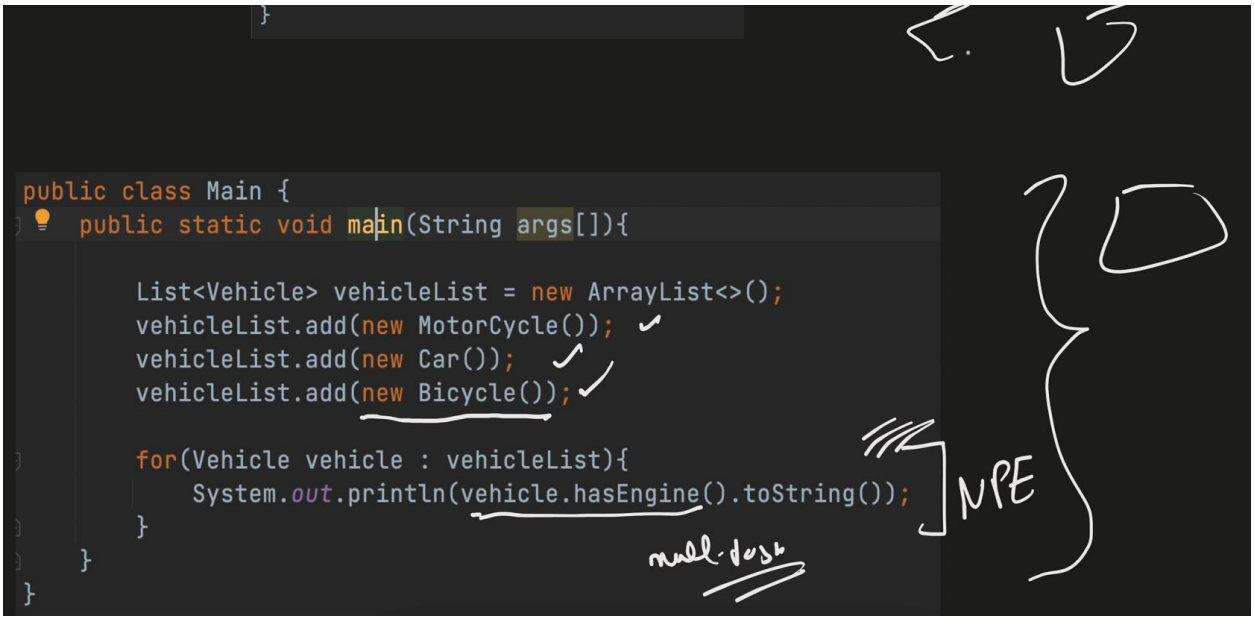
3. Liskov Substitution Principle (LSP) Solution

VALID Solution :


public class Vehicle {
public Integer getNumberOfWheels(){
return 2;
}
public Boolean hasEngine(){
return true;
}
}
public class MotorCycle extends Vehicle{
}
public class Car extends Vehicle{
@Override
public Integer getNumberOfWheels() {
return 4;
}
@Override
public Integer getNumberOfWheels() {
return 4;
}
}
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());
}
}
}

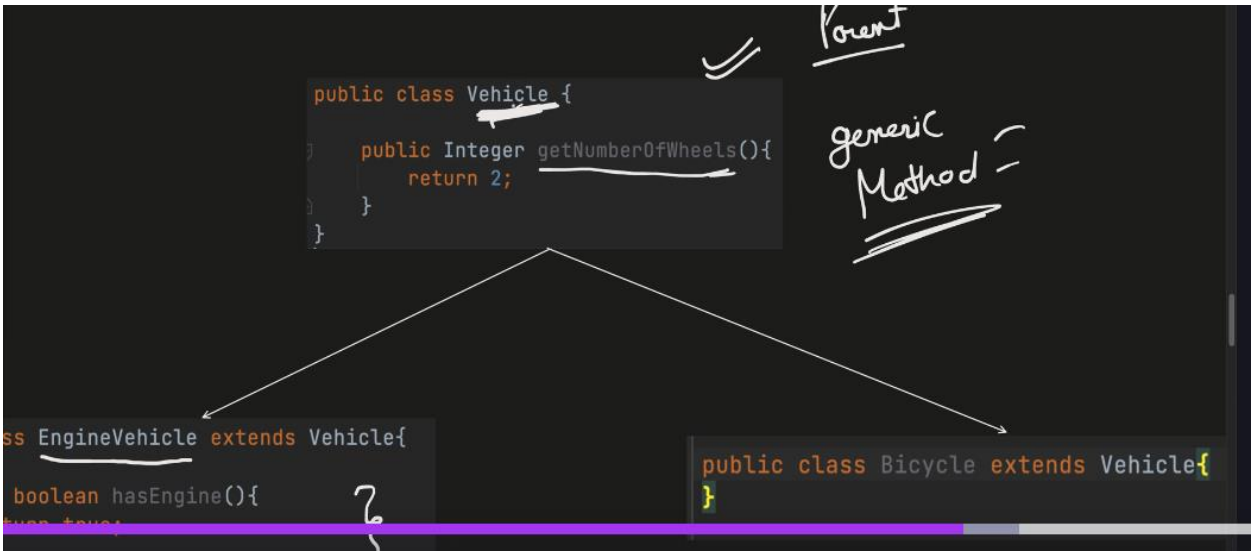
Problem :

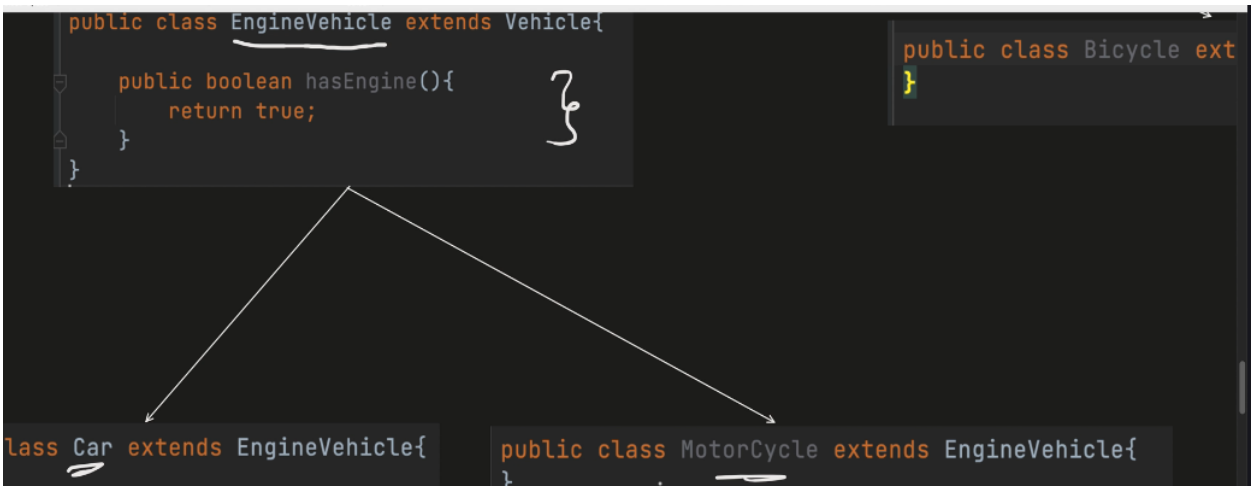

public class Vehicle {
public Integer getNumberOfWheels(){
return 2;
}
public Boolean hasEngine(){
return true;
}
}
class MotorCycle extends Vehicle{
public class Bicycle extends Vehicle{
public Boolean hasEngine(){
return null;
}
}
public class Car extends Vehicle{
@Override
public Integer getNumberOfWheels() {
}

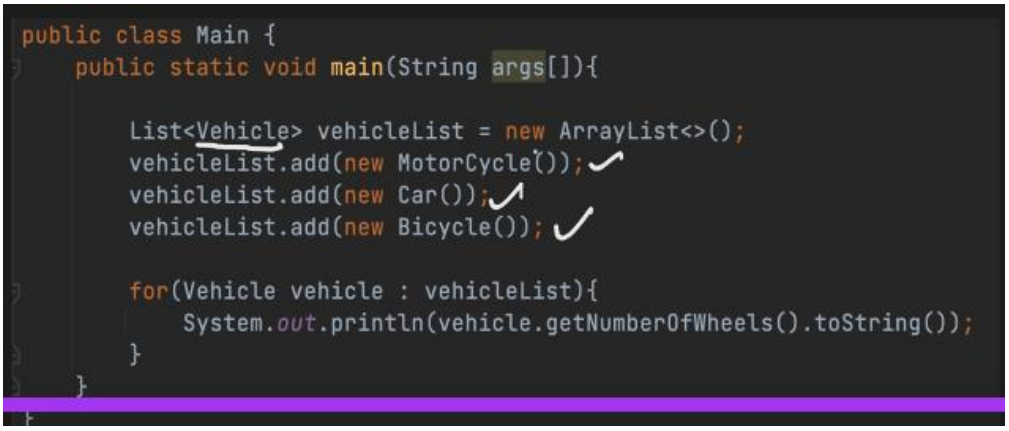

public class Main {
public static void main(String args[]){
List<Vehicle> vehicleList = new ArrayList<>();
vehicleList.add(new MotorCycle());
vehicleList.add(new Car());
vehicleList.add(new Bicycle());
for(Vehicle vehicle : vehicleList){
System.out.println(vehicle.hasEngine().toString());
}
}
}
NPE
null-dose


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

Solution :


Parent
public class Vehicle {
public Integer getNumberOfWheels(){
return 2;
}
}
generic Method
ss EngineVehicle extends Vehicle{
boolean hasEngine(){
public class Bicycle extends Vehicle{
}

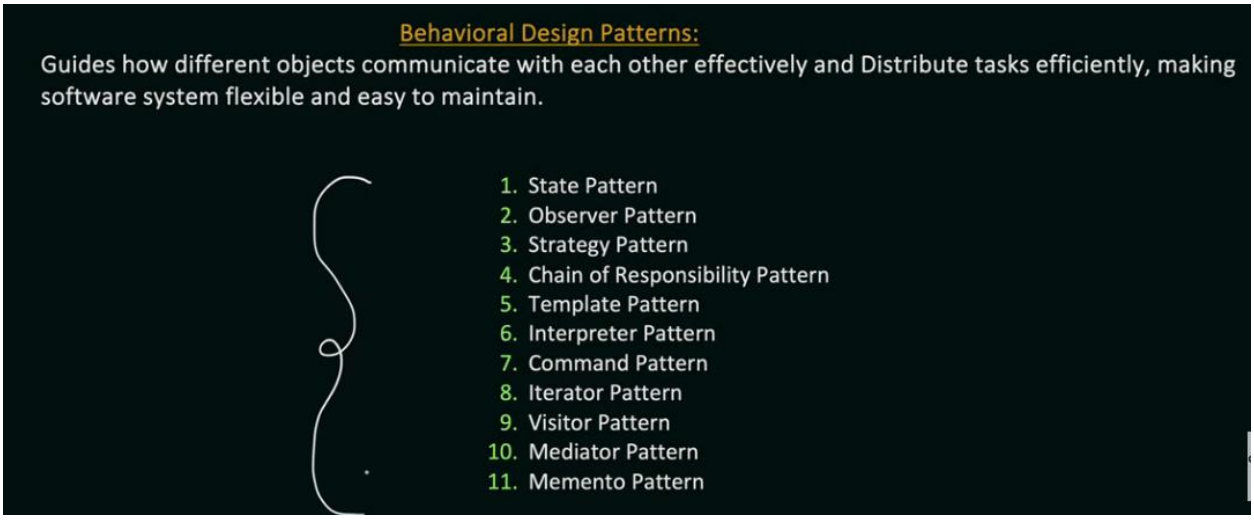

public class EngineVehicle extends Vehicle{
public boolean hasEngine(){
return true;
}
}
public class Bicycle ext
}
lass Car extends EngineVehicle{
public class MotorCycle extends EngineVehicle{


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.getNumberOfWheels().toString());
}
}

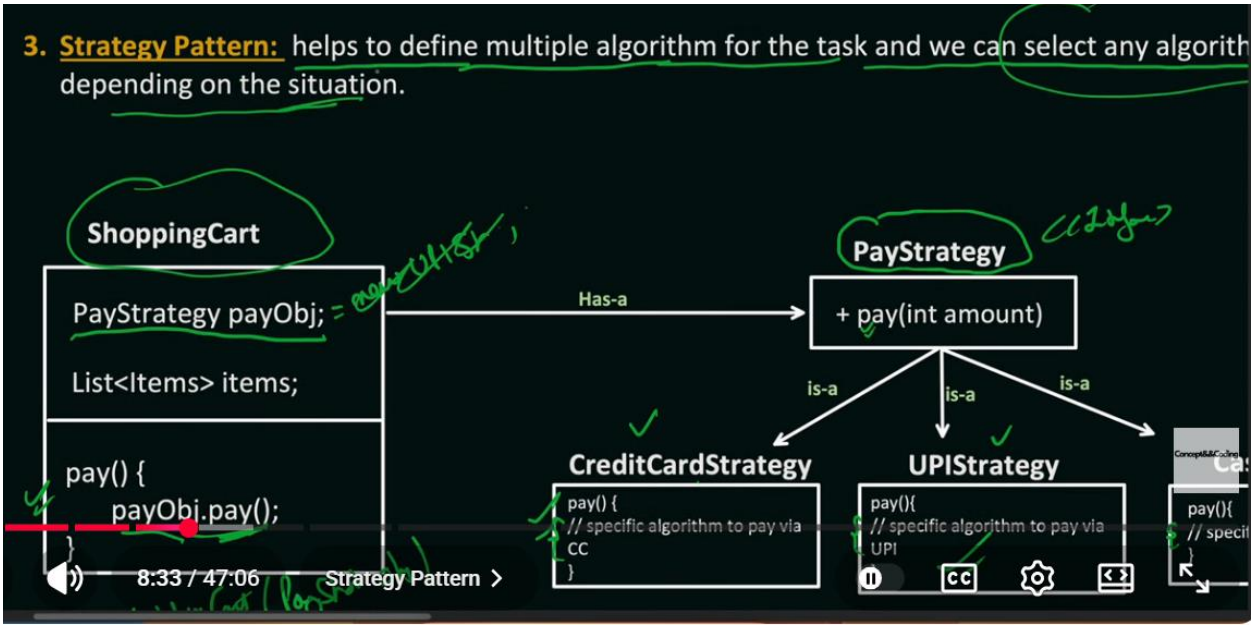
```java
public class Main {
    public static void main(String args[]){

        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());
        }
    }
```

```java
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());
        }
    }
}
```

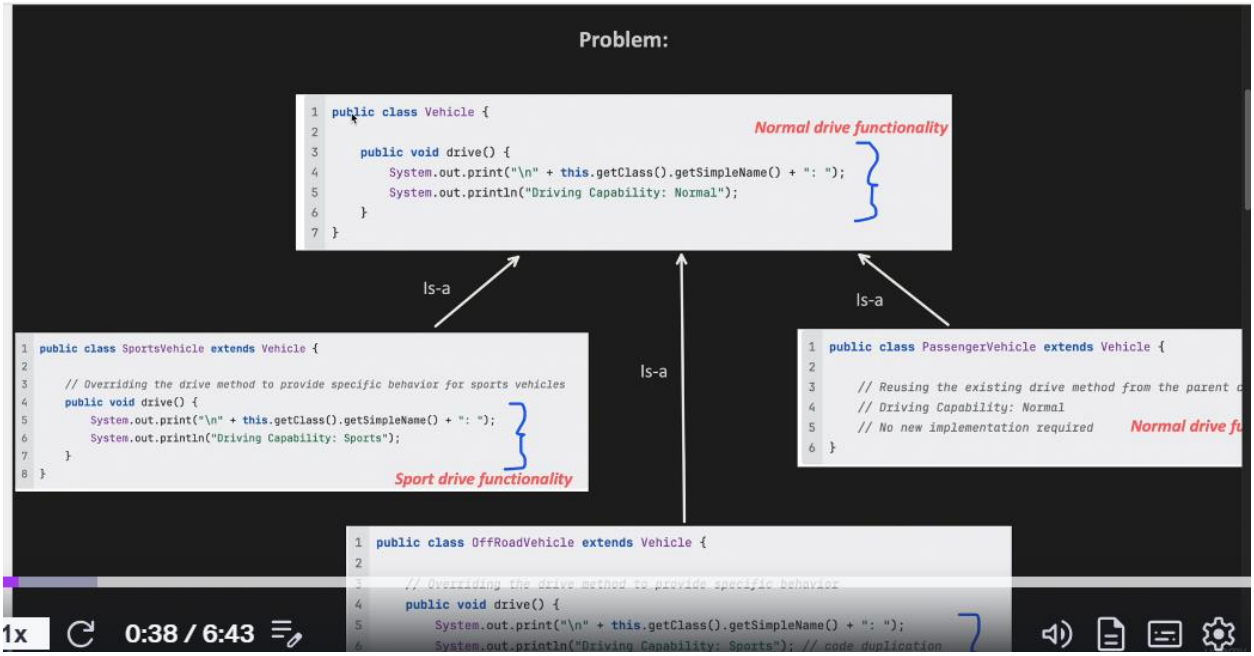
Behavior Design Pattern :



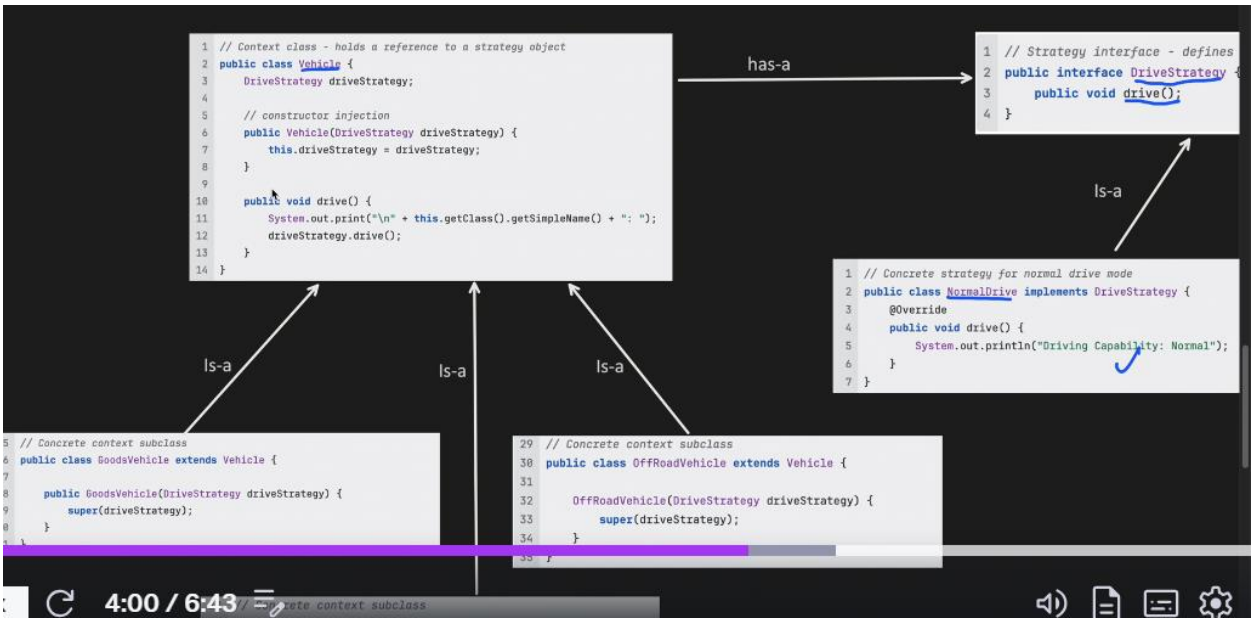
4. . Strategy Design Pattern (Behavioral Pattern)

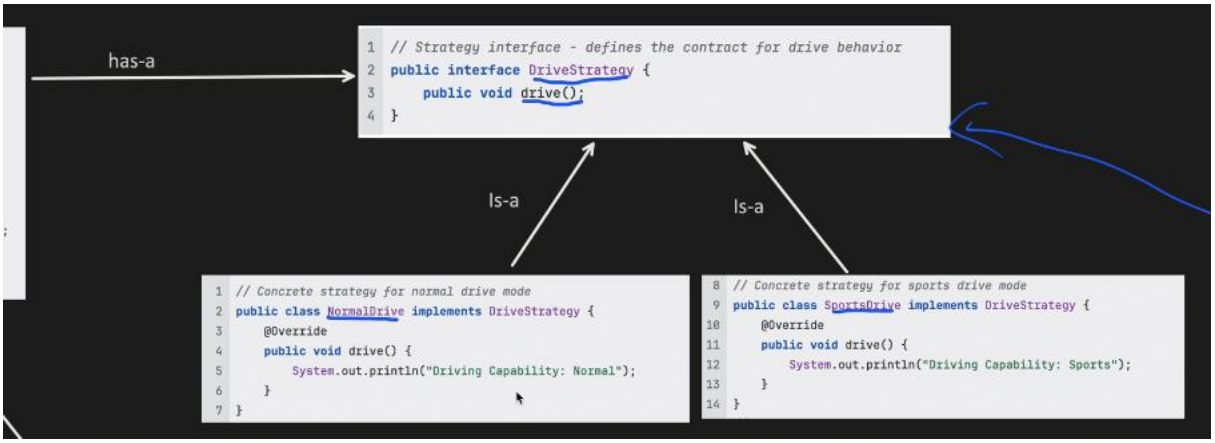


Violating Strategy design pattern :

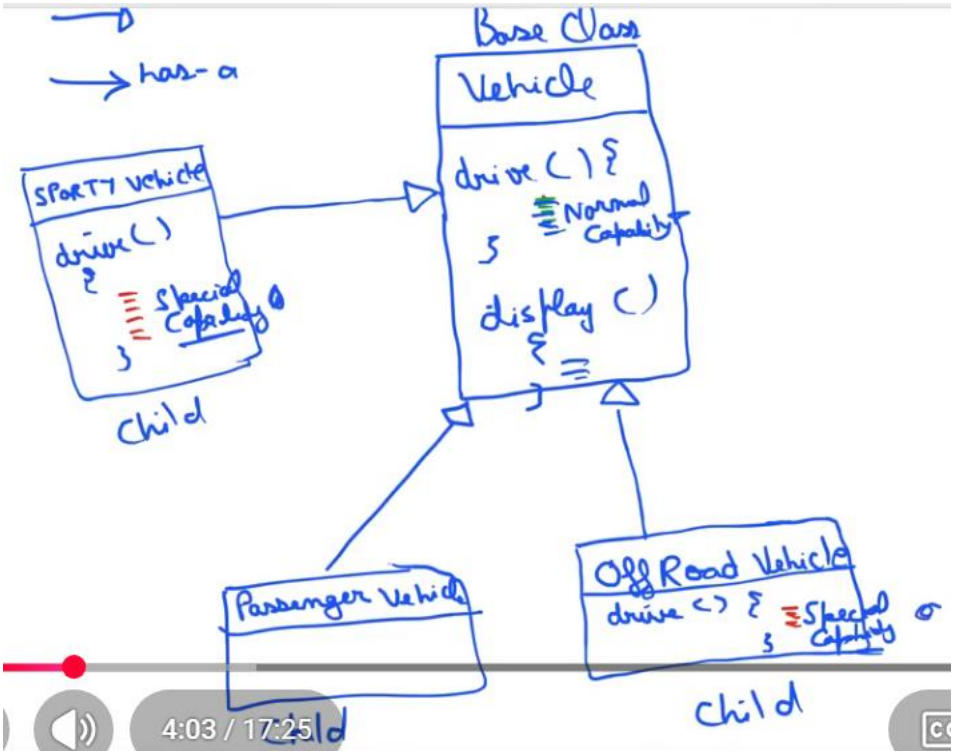


Following Strategy design pattern :

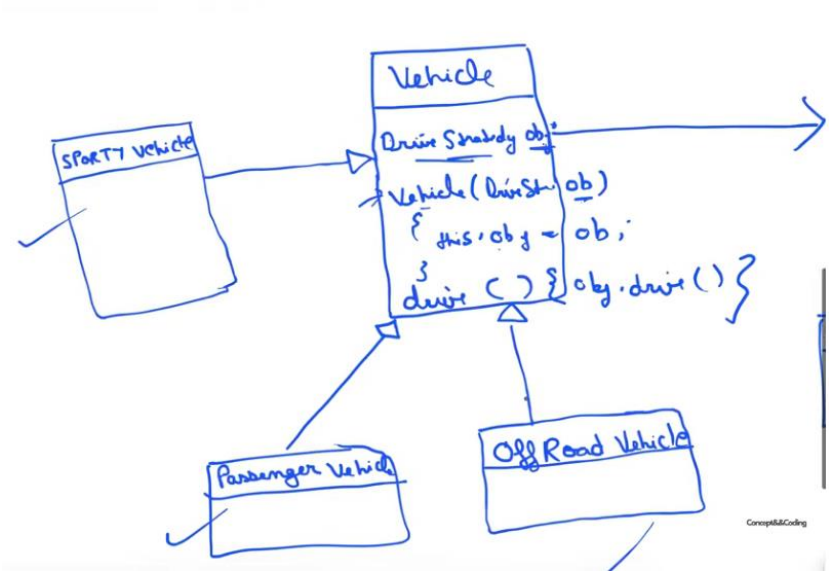


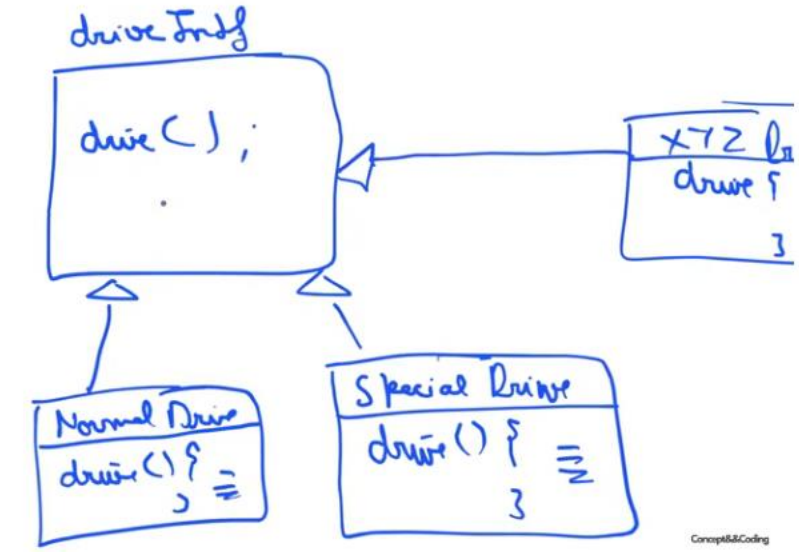


Without Strategy pattern :



With strategy pattern :

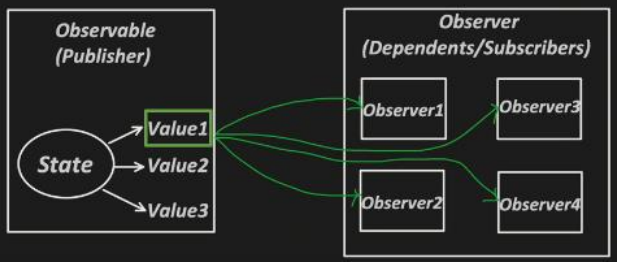

Vehicle
SPORTY Vehicle
Passenger Vehicle
Off Road Vehicle
drive ( ) { obj.drive ( ) }


driveInterface
drive ( ) ;
XYZ
drive {
}
Normal Drive
drive ( ) { }
Special Drive
drive ( ) { }

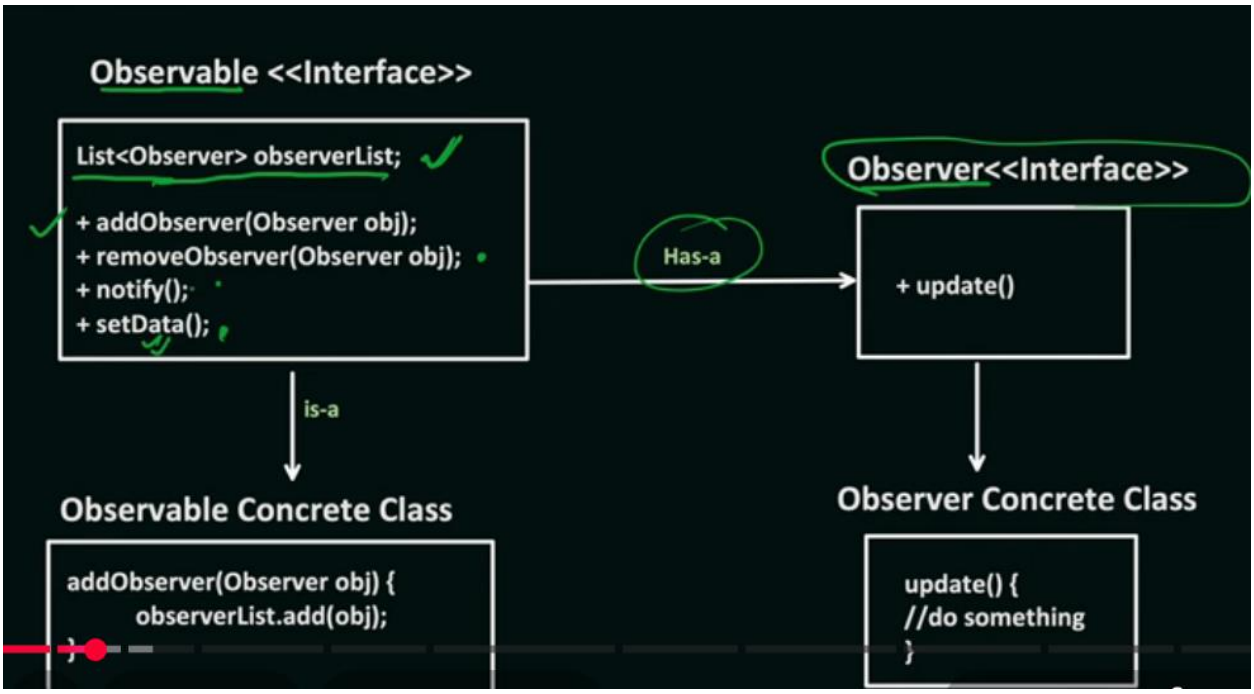
5. Observer Pattern :

Definition:

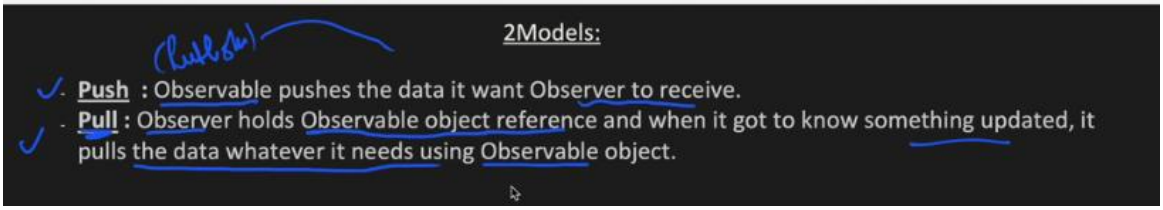
- *It's a design pattern where an object (aka "observable" or "publisher") maintains a list of dependents (called "observers").*
- *And automatically notifies dependents/observers whenever there is a change in its state.*

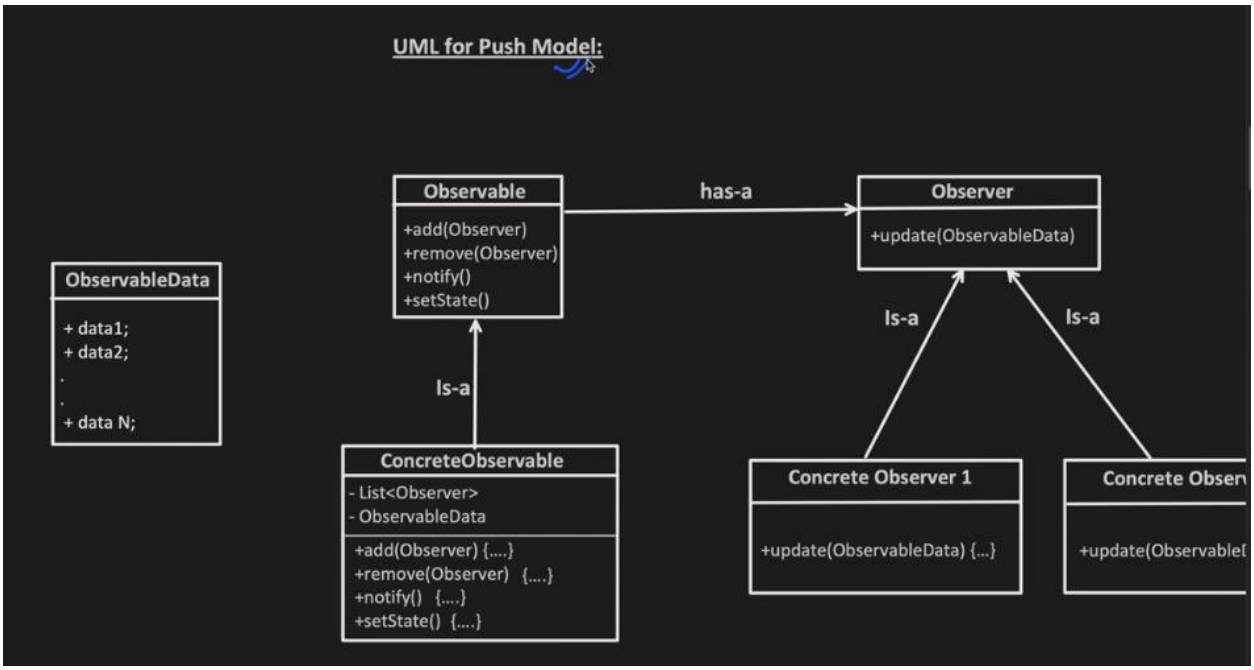


2. **Observer Pattern:** in this an object (Observable) maintains a list of its dependents (observers) a notifies them of any changes in its state.

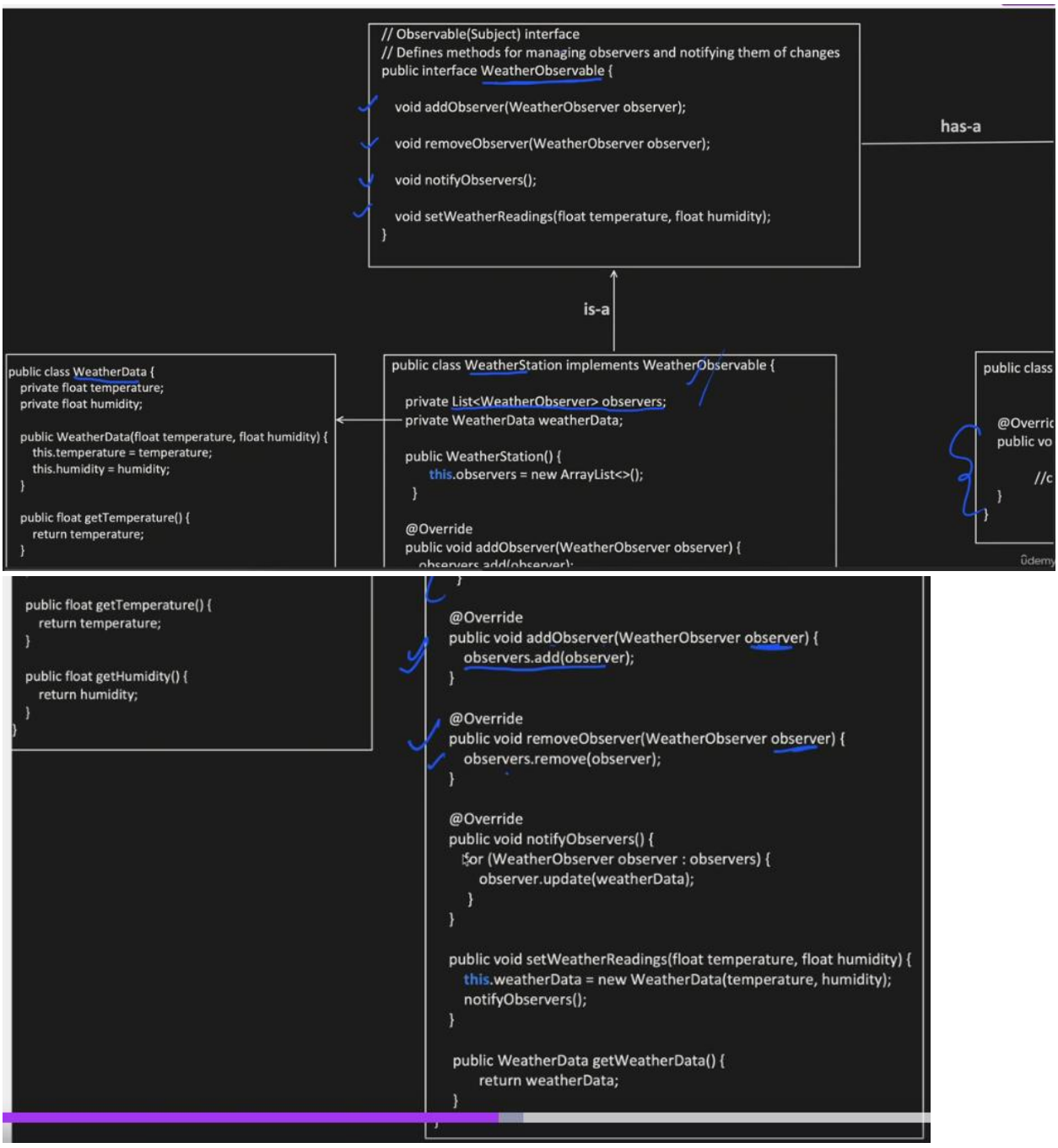

Observable <<Interface>>
List<Observer> observerList;
+ addObserver(Observer obj);
+ removeObserver(Observer obj);
+ notify();
+ setData();
Has-a
Observer<<Interface>>
+ update()
is-a
Observable Concrete Class
addObserver(Observer obj) {
observerList.add(obj);
Observer Concrete Class
update() {
//do something
}

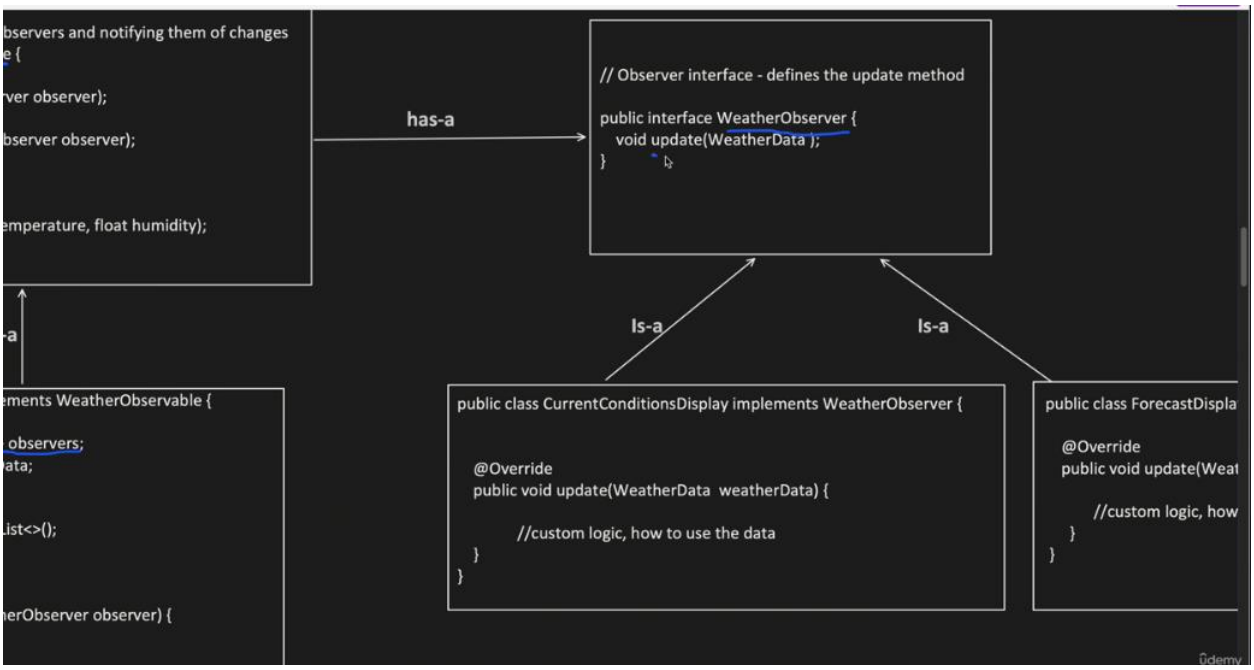

Observable Concrete Class
addObserver(Observer obj) {
observerList.add(obj);
}
removeObserver(Observer obj) {
observerList.remove(obj);
}
notify() {
for(Observer obj: observerList) {
obj.update();
}
}
setData() {
notify();
}

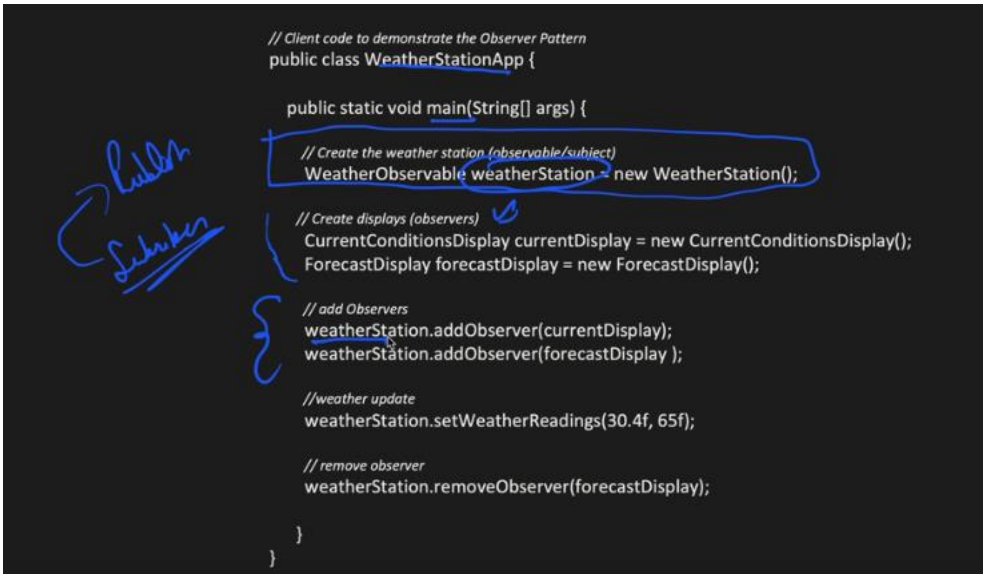


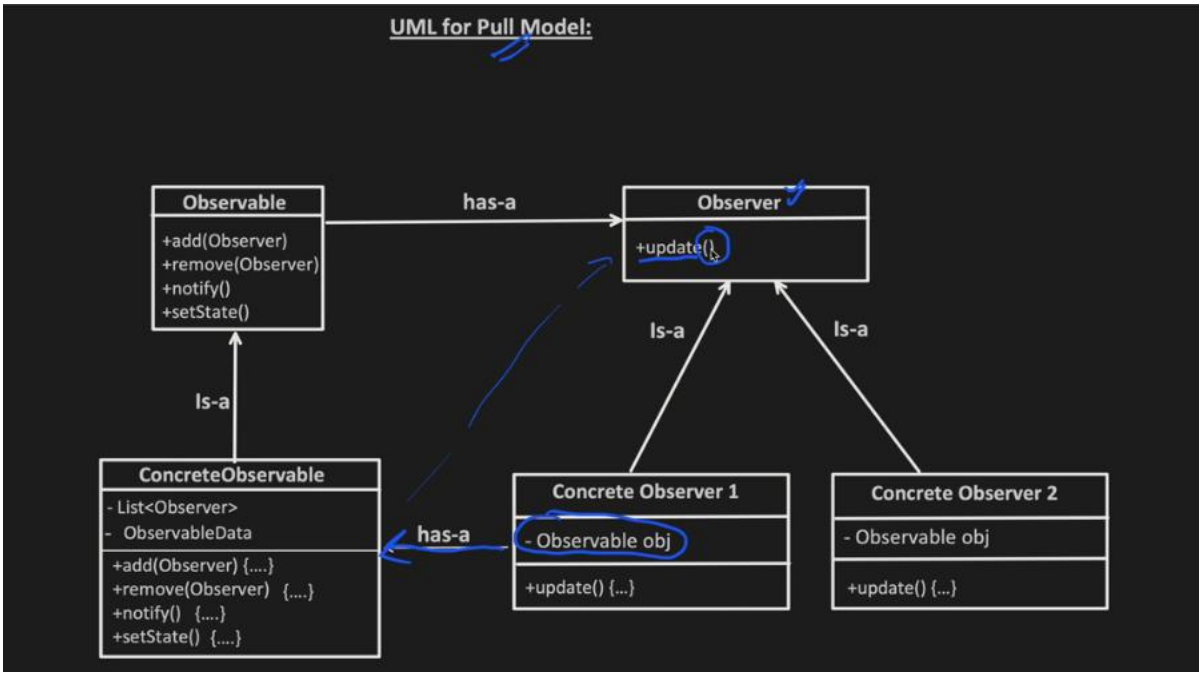


Push Model Example :

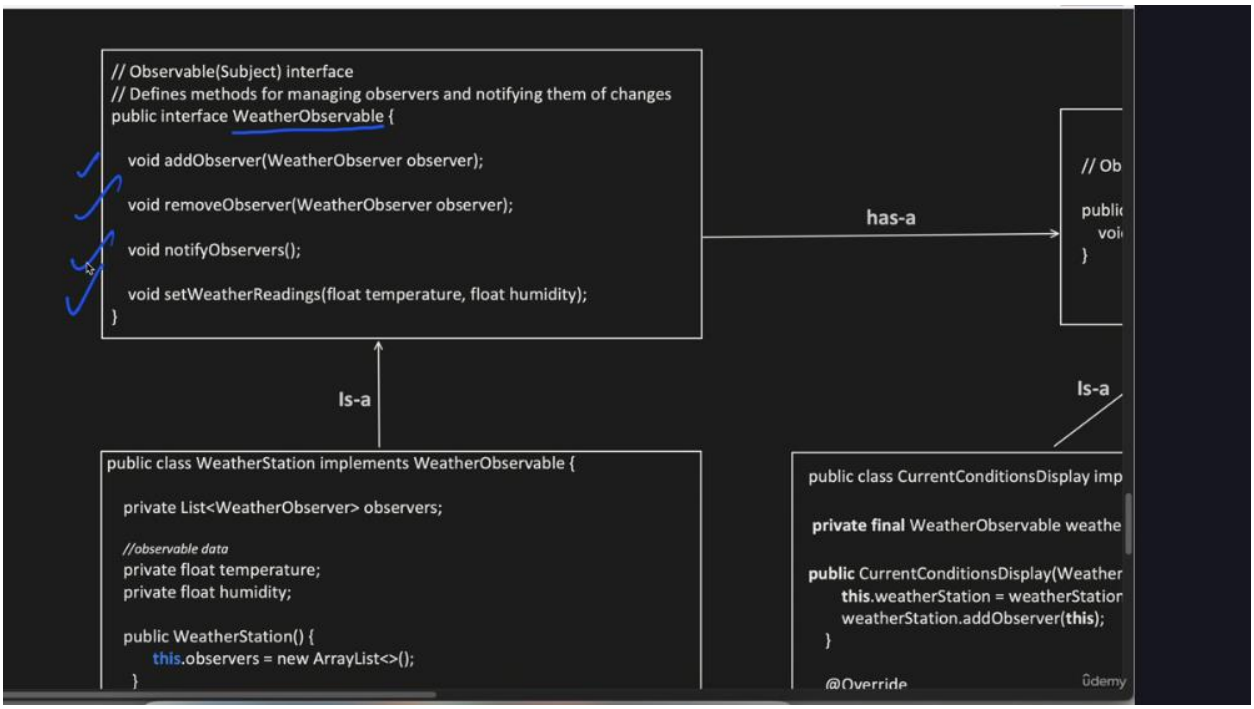

// Observable(Subject) interface
// Defines methods for managing observers and notifying them of changes
public interface WeatherObservable {
void addObserver(WeatherObserver observer);
void removeObserver(WeatherObserver observer);
void notifyObservers();
void setWeatherReadings(float temperature, float humidity);
}
has-a
is-a
public class WeatherData {
private float temperature;
private float humidity;
public WeatherData(float temperature, float humidity) {
this.temperature = temperature;
this.humidity = humidity;
}
public float getTemperature() {
return temperature;
}
public float getHumidity() {
return humidity;
}
}
public class WeatherStation implements WeatherObservable {
private List<WeatherObserver> observers;
private WeatherData weatherData;
public WeatherStation() {
this.observers = new ArrayList<>();
}
@Override
public void addObserver(WeatherObserver observer) {
observers.add(observer);
}
@Override
public void removeObserver(WeatherObserver observer) {
observers.remove(observer);
}
@Override
public void notifyObservers() {
for (WeatherObserver observer : observers) {
observer.update(weatherData);
}
}
public void setWeatherReadings(float temperature, float humidity) {
this.weatherData = new WeatherData(temperature, humidity);
notifyObservers();
}
public WeatherData getWeatherData() {
return weatherData;
}
public class
@Overric
public vo
//c
}
}

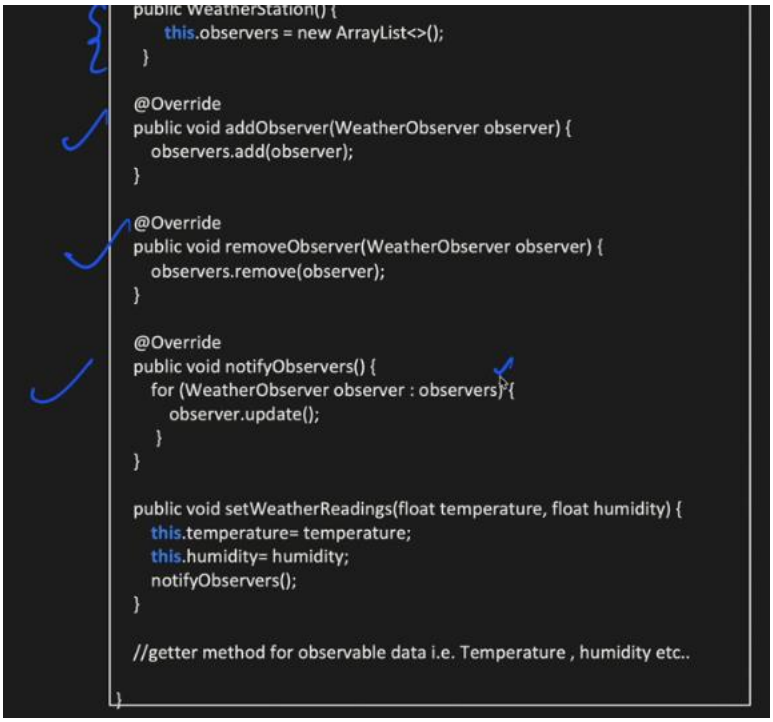

bservers and notifying them of changes
e {
ver observer);
bserver observer);
emperature, float humidity);
has-a
// Observer interface - defines the update method
public interface WeatherObserver {
void update(WeatherData);
}
Is-a
Is-a
ements WeatherObservable {
observers;
ata;
ist<>();
erObserver observer) {
public class CurrentConditionsDisplay implements WeatherObserver {
@Override
public void update(WeatherData weatherData) {
//custom logic, how to use the data
}
}
public class ForecastDispla
@Override
public void update(Weat
//custom logic, how
}
}
Udemy

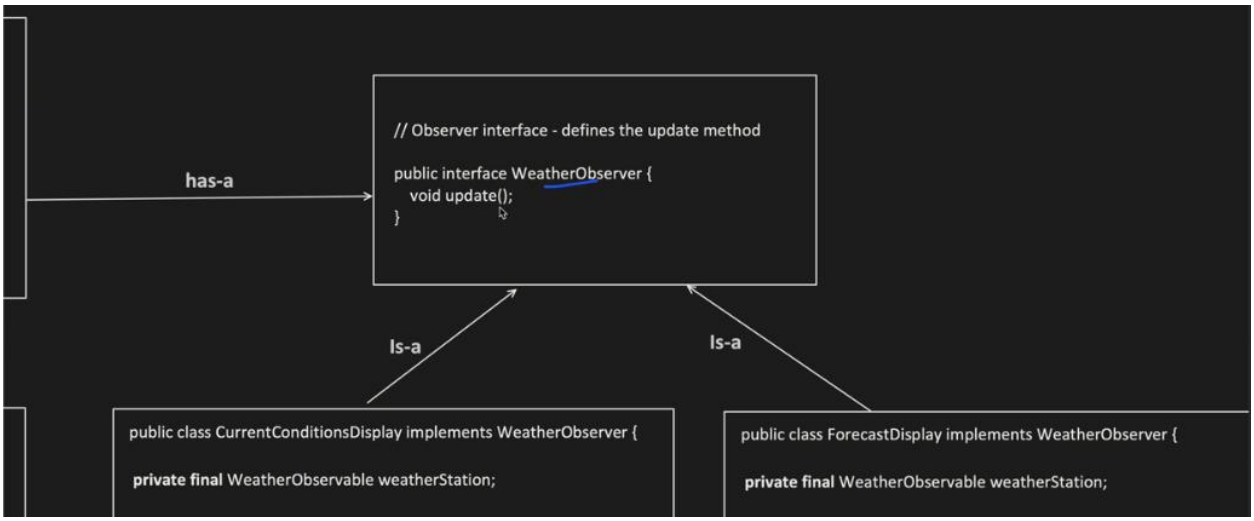

// Client code to demonstrate the Observer Pattern
public class WeatherStationApp {
public static void main(String[] args) {
// Create the weather station (observable/subject)
WeatherObservable weatherStation = new WeatherStation();
// Create displays (observers)
CurrentConditionsDisplay currentDisplay = 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);
}
}

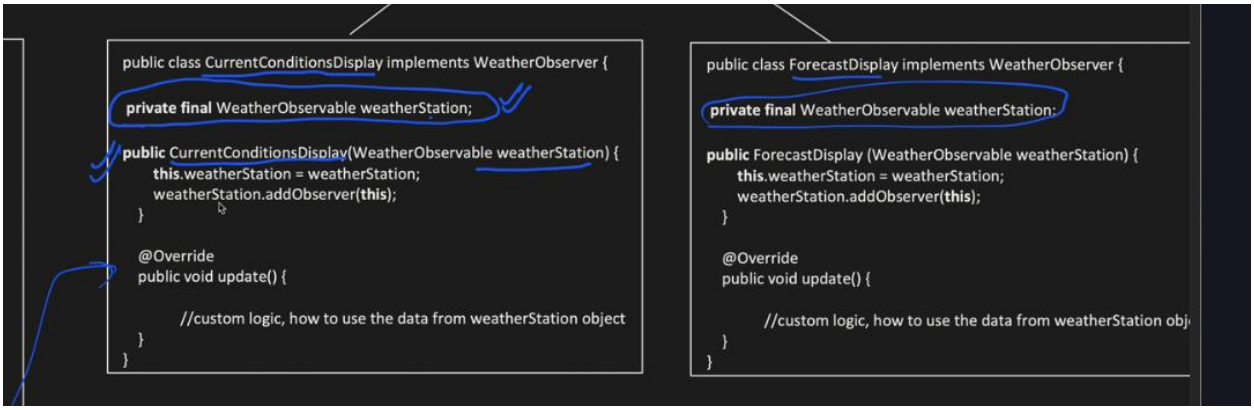


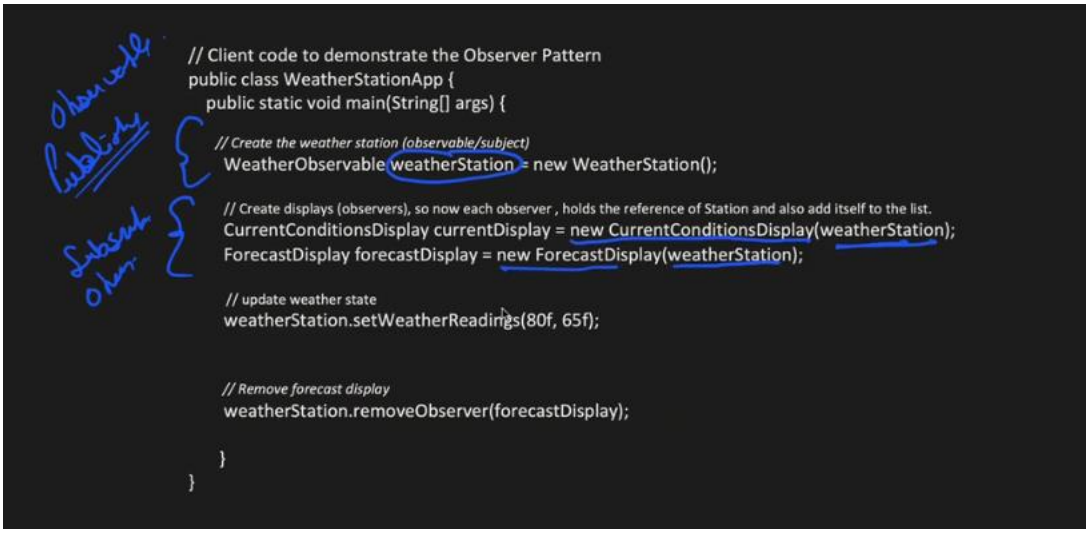
Pull Model Example :


// 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);
}
has-a
// Ob
publi
voi
}
Is-a
Is-a
public class WeatherStation implements WeatherObservable {
private List<WeatherObserver> observers;
//observable data
private float temperature;
private float humidity;
public WeatherStation() {
this.observers = new ArrayList<>();
}
public class CurrentConditionsDisplay imp
private final WeatherObservable weathe
public CurrentConditionsDisplay(Weather
this.weatherStation = weatherStatior
weatherStation.addObserver(this);
}
@Override
Udemy


public WeatherStation() {
this.observers = new ArrayList<>();
}
@Override
public void addObserver(WeatherObserver observer) {
observers.add(observer);
}
@Override
public void removeObserver(WeatherObserver observer) {
observers.remove(observer);
}
@Override
public void notifyObservers() {
for (WeatherObserver observer : observers){
observer.update();
}
}
public void setWeatherReadings(float temperature, float humidity) {
this.temperature= temperature;
this.humidity= humidity;
notifyObservers();
}
//getter method for observable data i.e. Temperature , humidity etc..
}


has-a
// Observer interface - defines the update method
public interface WeatherObserver {
void update();
}
Is-a
Is-a
public class CurrentConditionsDisplay implements WeatherObserver {
private final WeatherObservable weatherStation;
public class ForecastDisplay implements WeatherObserver {
private final WeatherObservable weatherStation;


public class CurrentConditionsDisplay implements WeatherObserver {
private final WeatherObservable weatherStation;
public CurrentConditionsDisplay(WeatherObservable weatherStation) {
this.weatherStation = weatherStation;
weatherStation.addObserver(this);
}
@Override
public void update() {
//custom logic, how to use the data from weatherStation object
}
}
public class ForecastDisplay implements WeatherObserver {
private final WeatherObservable weatherStation;
public ForecastDisplay (WeatherObservable weatherStation) {
this.weatherStation = weatherStation;
weatherStation.addObserver(this);
}
@Override
public void update() {
//custom logic, how to use the data from weatherStation obj
}
}


// 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), 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);
}
}