Part A
Class B Tech CSE 2nd Year Sub: Design Pattern Lab
Aim: Write a program to implement the Observer Design Pattern
Prerequisite: Basics of Object Oriented Programming
Outcome: To impart knowledge of Design Pattern techniques
Theory: Explain about the Observer Design Pattern.

Observer Design Pattern:

The Observer Design Pattern is a behavioral pattern where an object, known as the subject, maintains a list of its dependents, called observers, and notifies them of any changes in its state. It allows for a one-to-many dependency between objects, ensuring that all dependent objects are updated automatically when the state of the subject changes.

Key Components:

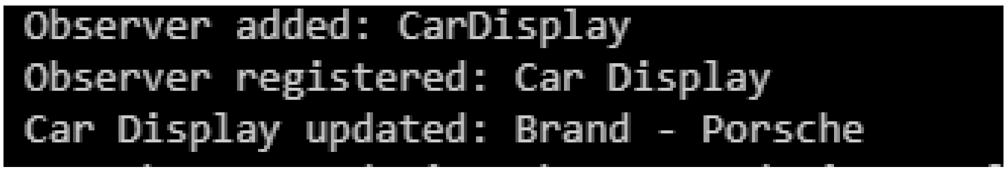
1. **Subject**: This is the object that is being observed. It maintains a list of observers and provides methods to add, remove, and notify observers of any changes in its state.
2. **Observer**: This is the interface or abstract class implemented by the observers. It defines the method(s) that the subject will call to notify observers of state changes.
3. **Concrete Subject**: This is a subclass of the subject that implements the specific business logic and state changes. It notifies observers when its state changes.
4. **Concrete Observer**: This is a subclass of the observer that implements the behavior to be performed in response to notifications from the subject.

Part B
Steps: Write procedure/code here.
Output: Paste Output here.
Observation & Learning: Give your observation and learning about the experiment.
Conclusion: Give your conclusion for the experiment.

Code:

```java
import java.util.ArrayList;

import java.util.List;

// Subject interface
interface Car {
    void addObserver(Observer observer);

    void removeObserver(Observer observer);

    void notifyObservers();
}

// Concrete Subject
class CarModel implements Car {
    private String brand;
    private List<Observer> observers = new ArrayList<>();

    public void setBrand(String brand) {
        this.brand = brand;
        notifyObservers();
    }

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

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

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(brand);
        }
    }
}

// Observer interface
interface Observer {
    void update(String brand);
}

// Concrete Observer
class CarDisplay implements Observer {
    @Override
    public void update(String brand) {
        System.out.println("Car Display updated: Brand - " + brand);
    }
}

// Client code
public class Pr12_Observer {
    public static void main(String[] args) {
        CarModel carModel = new CarModel();
        CarDisplay carDisplay = new CarDisplay();

        carModel.addObserver(carDisplay);

        System.out.println("Observer registered: Car Display");

        // Simulate brand change
        carModel.setBrand("Porsche");
    }
}
```

Output:



```
Observer added: CarDisplay
Observer registered: Car Display
Car Display updated: Brand - Porsche
```

Observation & Learning:

- **Flexibility**: The pattern provided a flexible way to establish relationships between objects, allowing them to communicate without knowing the details of each other's implementation.
- **Decoupling**: Subjects and observers were loosely coupled, promoting better maintainability and extensibility of the codebase.
- **Dynamic Behavior**: We observed how the pattern facilitated dynamic behavior, allowing observers to be added or removed from the subject at runtime.

Conclusion:

The Observer Design Pattern proved to be a valuable tool for implementing communication between objects in a flexible and decoupled manner. By allowing objects to subscribe and react to changes in the state of other objects, the pattern enables the creation of dynamic and responsive systems. We'll consider using this pattern in future projects where objects need to communicate and respond to changes effectively.