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

Memento Design Pattern

The Memento Design Pattern is a behavioral pattern that enables capturing and externalizing an object's internal state without violating encapsulation, and subsequently restoring that state when needed. It's particularly useful when you need to implement undo mechanisms, snapshots, or checkpoints in an application.

Key Components:

1. **Originator**: This is the object whose state needs to be saved and restored. The Originator creates a Memento containing a snapshot of its current state and can also restore its state from a Memento.
2. **Memento**: This is an immutable object that stores the state of the Originator. It provides methods for retrieving the state but does not allow direct modification of the state.
3. **Caretaker**: This is responsible for holding and managing Mementos. It requests Mementos from the Originator, stores them, and can later ask the Originator to restore its state from a specific Memento.

Benefits:

- **Encapsulation**: The Memento pattern promotes encapsulation by keeping the state of an object private and accessible only through well-defined interfaces.
- **Undo/Redo**: It facilitates implementing undo and redo functionalities by storing and restoring previous states of objects.
- **Snapshotting**: The pattern allows creating snapshots of an object's state, which can be useful for implementing checkpoints or versioning systems.

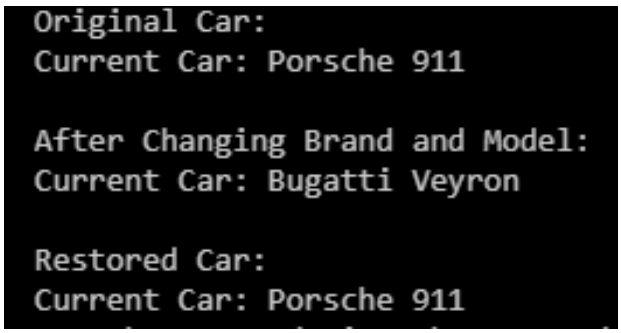
Part B
Steps: Write procedure/code here.
Output: Paste Output here.
Observation & Learning: Give your observation and learning about the experiment.
Conclusion: Give your conclusion for the experiment.

Code:

```
// Snapshot class
class CarSnapshot {
    private String state;

    public CarSnapshot(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }
}

// Car class
class Car {
    private String brand;
    private String model;

    public Car(String brand, String model) {
        this.brand = brand;
        this.model = model;
    }

    public void changeBrandAndModel(String brand, String model) {
        this.brand = brand;
        this.model = model;
    }

    public CarSnapshot saveSnapshot() {
        return new CarSnapshot(brand + " " + model);
    }

    public void restoreFromSnapshot(CarSnapshot snapshot) {
        String[] parts = snapshot.getState().split(" ");
        this.brand = parts[0];
        this.model = parts[1];
    }
    public void display() {
        System.out.println("Current Car: " + brand + " " + model);
```

```
    }
}

// Caretaker class
class CarSnapshotCaretaker {
    private CarSnapshot snapshot;

    public void saveSnapshot(CarSnapshot snapshot) {
        this.snapshot = snapshot;
    }

    public CarSnapshot retrieveSnapshot() {
        return snapshot;
    }
}

// Client code
public class Pr11_Memento {
    public static void main(String[] args) {
        Car car = new Car("Porsche", "911");
        CarSnapshotCaretaker caretaker = new CarSnapshotCaretaker();

        // Save initial state
        caretaker.saveSnapshot(car.saveSnapshot());
        System.out.println("Original Car:");
        car.display();

        // Change state
        car.changeBrandAndModel("Bugatti", "Veyron");
        System.out.println("\nAfter Changing Brand and Model:");
        car.display();

        // Restore saved state
        car.restoreFromSnapshot(caretaker.retrieveSnapshot());
        System.out.println("\nRestored Car:");
        car.display();
    }
}
```

Output:



Observation & Learning:

1. **State Management:** The Memento pattern helped us manage the state of a car object effectively. By saving snapshots of the car's state and restoring them when needed, we could track and revert changes easily.
2. **Encapsulation:** We observed how the Memento pattern promotes encapsulation by encapsulating the car's state within the snapshot objects. This ensures that the car's internal state remains private and accessible only through well-defined interfaces.
3. **Flexibility:** The pattern provided flexibility in managing the car's state. We could save multiple snapshots at different points in time and restore any of them as required, enabling features like undo/redo functionality.

Conclusion:

The Memento Design Pattern proved to be a valuable tool for managing the state of objects, such as cars, in a flexible and encapsulated manner. By providing mechanisms for saving and restoring snapshots of the object's state, the pattern enables features like undo, redo, and checkpointing in software applications. We'll keep this pattern in mind for future projects where state management and undo functionality are crucial.