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

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.

Flyweight Design Pattern

The Flyweight Design Pattern is a structural design pattern that aims to optimize memory usage by sharing common parts of object state among multiple objects. It's particularly useful in scenarios where a large number of similar objects need to be created, thus reducing memory overhead.

Key Components:

1. **Intrinsic State**: This refers to the part of the object's state that can be shared among multiple objects. Intrinsic state is immutable and remains constant throughout the object's lifetime.
2. **Extrinsic State**: This represents the context-dependent state of the object, which cannot be shared among objects. Extrinsic state is typically passed as a parameter during object creation or method invocation.
3. **Flyweight Factory**: The Flyweight Factory is responsible for managing the creation and sharing of flyweight objects. It maintains a pool of already created flyweight objects and handles requests for creating new objects efficiently.
4. **Flyweight Objects**: These are the objects that encapsulate the intrinsic state shared among multiple objects. Flyweight objects are typically immutable and are stored separately from the instances of the objects that use them.

Benefits:

- **Memory Optimization**: By sharing common state, the Flyweight pattern reduces memory consumption, especially in scenarios with a large number of similar objects.

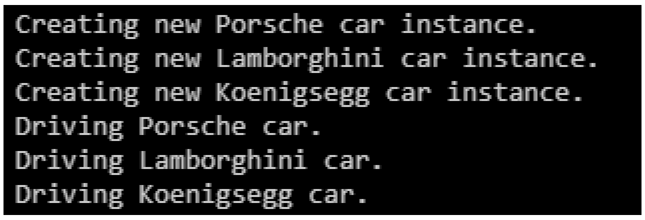
- **Improved Performance**: Reduced memory overhead leads to improved performance, particularly in memory-constrained environments or applications with high object instantiation rates.
- **Simplicity**: The Flyweight pattern simplifies the design by separating intrinsic and extrinsic state and providing a centralized mechanism for object sharing and creation.

Code:

```java
import java.util.HashMap;
import java.util.Map;

// Flyweight interface
interface Car {
    void drive();
}

// Concrete Flyweight
class ConcreteCar implements Car {
    private String brand;

    public ConcreteCar(String brand) {
        this.brand = brand;
    }

    @Override
    public void drive() {
        System.out.println("Driving " + brand + " car.");
    }
}

// Flyweight Factory
class CarFactory {
    private static Map<String, Car> cars = new HashMap<>();

    public static Car getCar(String brand) {
        Car car = cars.get(brand);
        if (car == null) {
            car = new ConcreteCar(brand);
            cars.put(brand, car);
            System.out.println("Creating new " + brand + " car instance.");
        }
        return car;
    }
}
```

```java
// Client
public class Pr9_Flyweight {
    public static void main(String[] args) {
        Car car1 = CarFactory.getCar("Porsche");
        Car car2 = CarFactory.getCar("Lamborghini");
        Car car3 = CarFactory.getCar("Koenigsegg");

        car1.drive();
        car2.drive();
        car3.drive();
    }
}
```

Output:



```
Creating new Porsche car instance.
Creating new Lamborghini car instance.
Creating new Koenigsegg car instance.
Driving Porsche car.
Driving Lamborghini car.
Driving Koenigsegg car.
```

Observation & Learning:

- Memory Saver: Sharing common parts of object state really cut down on memory usage. This was especially clear when dealing with lots of similar objects.
- Speed Boost: Less memory usage meant the program ran smoother and faster. It was a noticeable improvement, especially on devices with limited memory.
- Smart Design: Splitting state into intrinsic and extrinsic parts made the code cleaner and easier to understand. It felt like we were building with LEGO blocks – everything fit together neatly.
- Integration Challenges: While the pattern was great for memory, getting it to play nice with existing code needed careful handling. Making sure everything worked smoothly took some effort.

Conclusion:

In the end, the Flyweight Design Pattern proved itself as a handy tool for saving memory and boosting performance. By sharing common state among objects, it made our code sleeker and more efficient. D