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

Iterator Design Pattern

The Iterator design pattern is a design pattern that provides a way to access the elements of an aggregate object (such as a collection) sequentially without exposing its underlying representation. It separates the traversal logic from the collection, allowing different traversal algorithms to be used interchangeably.

Key Components:

1. **Iterator:** This is an interface that defines methods for accessing elements of the aggregate object sequentially. It typically includes methods like hasNext() to check if there are more elements, and next() to retrieve the next element.

2. **Concrete Iterator:** This is a concrete implementation of the Iterator interface, providing specific traversal logic for a particular type of aggregate object.

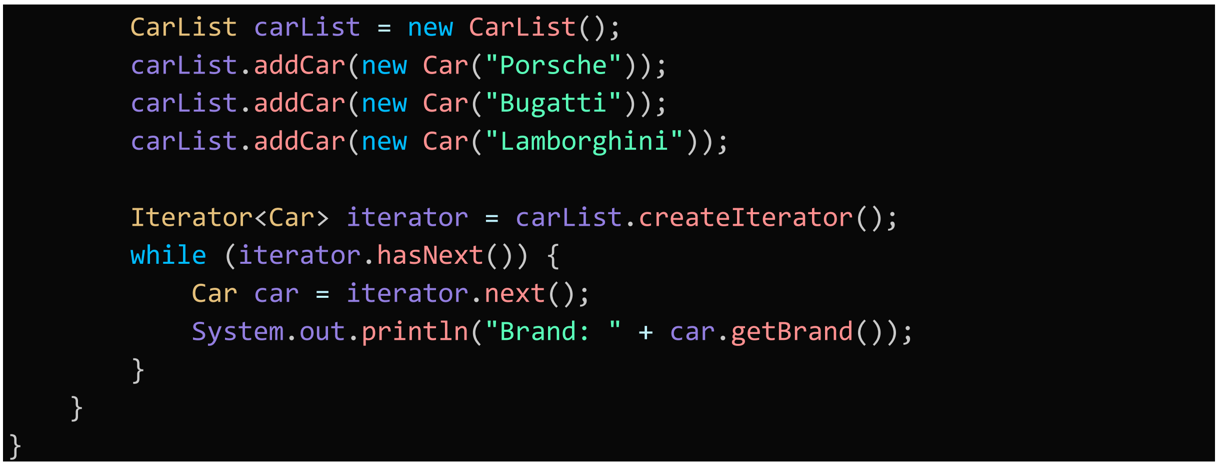
3. **Aggregate:** This is an interface that defines a method for creating an iterator object. It represents the collection of objects that the iterator will traverse.

4. **Concrete Aggregate:** This is a concrete implementation of the Aggregate interface, representing a specific collection of objects. It provides the implementation for creating an iterator object that can traverse its elements.

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.

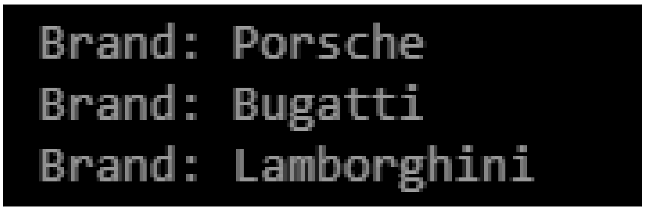
Code:

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

// Aggregate interface
interface CarCollection {
    Iterator<Car> createIterator();
}

// Concrete Aggregate
class CarList implements CarCollection {
    private List<Car> cars = new ArrayList<>();

    public void addCar(Car car) {
        cars.add(car);
    }

    @Override
    public Iterator<Car> createIterator() {
        return cars.iterator();
    }
}

// Concrete Item
class Car {
    private String brand;

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

    public String getBrand() {
        return brand;
    }
}

// Client code
public class Pr14_Iterator {
    public static void main(String[] args) {
```



```
        CarList carList = new CarList();
        carList.addCar(new Car("Porsche"));
        carList.addCar(new Car("Bugatti"));
        carList.addCar(new Car("Lamborghini"));

        Iterator<Car> iterator = carList.createIterator();
        while (iterator.hasNext()) {
            Car car = iterator.next();
            System.out.println("Brand: " + car.getBrand());
        }
    }
}
```

Output:



```
Brand: Porsche
Brand: Bugatti
Brand: Lamborghini
```

Observation & Learning:

- Simplifying the Iterator pattern's implementation improved my understanding of its purpose and usage.
- The pattern separates the logic for traversing a collection from the collection itself, promoting code modularity and reusability.
- By providing a uniform way to access elements of different collections, the Iterator pattern enhances code flexibility and maintainability**.**

Conclusion:

- Experimenting with the Iterator pattern in a simplified context deepened my comprehension of its principles.
- Clear explanations and concise implementations facilitated learning and understanding.
- The Iterator pattern proves valuable for managing collections and traversing their elements in a systematic and flexible manner.