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

Singleton Design Pattern

The Singleton design pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. This pattern is useful when exactly one object is needed to coordinate actions across the system, such as a single database connection, a logging service, or a configuration manager.

Key features of the Singleton pattern include:

1. Single Instance: The Singleton class is responsible for creating and maintaining its own unique instance. This instance is usually created the first time it is requested and reused for subsequent requests.

2. Global Access Point: The Singleton provides a global point of access to its instance, allowing other classes and components to easily access it without the need to instantiate the object directly.

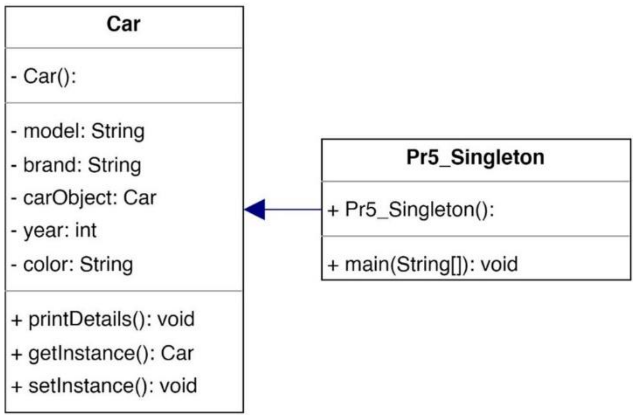
3. Private Constructor: The Singleton class typically has a private constructor to prevent external instantiation. This means that the class itself is responsible for controlling the instantiation process to ensure only one instance is created.

4. Lazy Initialization: The Singleton instance is often created lazily, meaning it is only created when it is first needed. This helps improve performance by avoiding unnecessary instantiation if the instance is not used during the application's lifecycle.

It's important to note that while the Singleton pattern provides a global point of access, it should be used judiciously, as it can introduce global state and dependencies, making the code less modular and harder to test.

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.

UML Class Diagram:

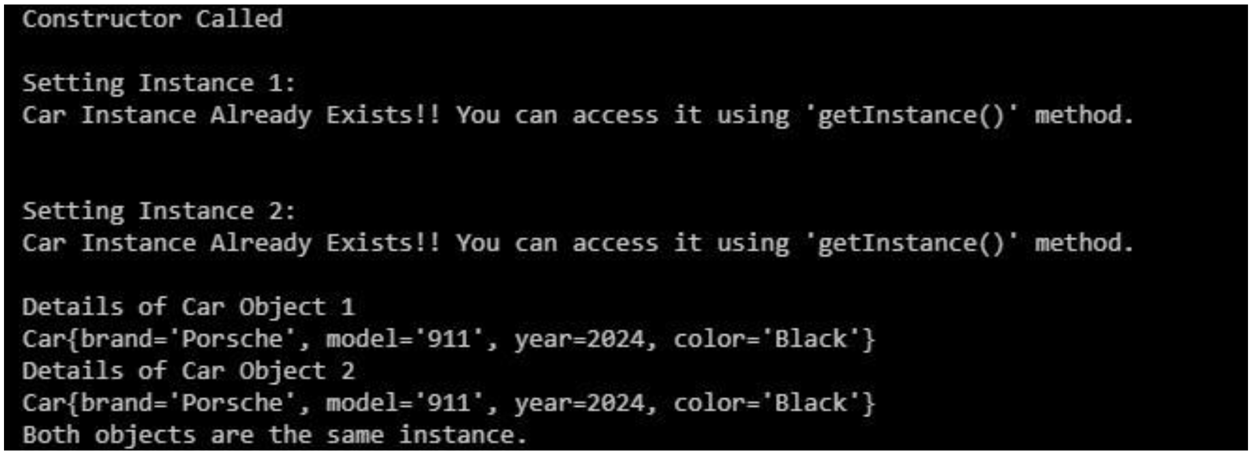


a) Eager Singleton

Implementation:

```
import java.util.Scanner;

class Car {
    private String model, color, brand;
    private int year;

    // Early, instance will be created at load time

    private static Car carObject = new Car();

    private Car() {
        this.brand = "Porsche";
        this.model = "911";
        this.year = 2024;
        this.color = "Black";
        System.out.println("Constructor Called");
    }
    public static Car getInstance() { return
            carObject;
    }



      public static void setInstance() { if (carObject
              == null) {
                      carObject = new Car();
                      System.out.println("Car Instance successfully set");
              } else
                      System.out.println("Car Instance Already Exists!! You can access it using
'getInstance()' method.\n");

      }

      public void printDetails() {
              System.out.println("Car{" +
                              "brand='" + brand + '\'' + ",
                              model='" + model + '\'' + ", year=" +
                              year +
                              ", color='" + color + '\'' +
                              "}");

      }

}

public class Pr5_Singleton_Eager extends Car {

      public static void main(String[] args) {

              System.out.println("\nSetting Instance 1: "); setInstance();

              System.out.println("\nSetting Instance 2: "); setInstance();

              Car carObject1 = getInstance(); System.out.println("Details of Car
              Object 1"); carObject1.printDetails();

              Car carObject2 = getInstance(); System.out.println("Details of Car
              Object 2"); carObject2.printDetails();

                     if (carObject1 == carObject2) {
                                  System.out.println("Both objects are the same instance.");
              }
```

Output:

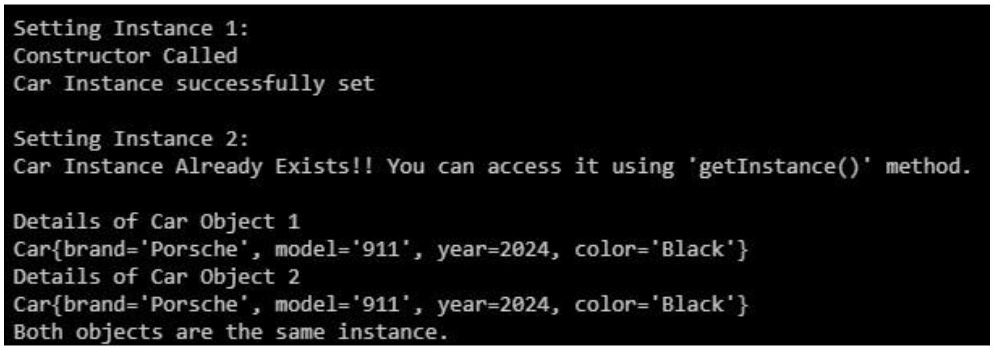

```
Constructor Called

Setting Instance 1:
Car Instance Already Exists!! You can access it using 'getInstance()' method.


Setting Instance 2:
Car Instance Already Exists!! You can access it using 'getInstance()' method.

Details of Car Object 1
Car{brand='Porsche', model='911', year=2024, color='Black'}
Details of Car Object 2
Car{brand='Porsche', model='911', year=2024, color='Black'}
Both objects are the same instance.
```

b) Lazy Singleton

Implementation:

```
// Car class representing a car with specific details

class Car {
        private String model, color, brand; private int year;

        // Lazy-initialized singleton instance of Car

        private static Car carObject = null;

        // Private constructor to prevent direct instantiation

        private Car() {
                // Setting default values for brand, model, year, and color

                this.brand = "Porsche";
                this.model = "911";
                this.year = 2024; this.color =
                "Black";
                System.out.println("Constructor Called");
        }


        // Getter method to retrieve the singleton instance of Car

        public static Car getInstance() { return
                carObject;
        }


        // Setter method to create a new Car instance if it doesn't already exist

        public static void setInstance() { if (carObject
                == null) {
                        carObject = new Car();
                        System.out.println("Car Instance successfully set");
                } else
                        System.out.println("Car Instance Already Exists!! You can access it using
'getInstance()' method.\n");
        }


        // Method to print the details of the car

        public void printDetails() {
                System.out.println("Car{" +
                        "brand='" + brand + '\'' + ",
                        model='" + model + '\'' +
                        ", year=" + year +
                        ", color='" + color + '\'' +
                        "}");

         }

}


// Main class to demonstrate the lazy singleton pattern

public class Pr5_Singleton_Lazy {

      public static void main(String[] args) {

            // Setting and displaying the first instance of the Car
            System.out.println("\nSetting Instance 1: "); Car.setInstance();


            // Setting and displaying the second instance of the Car
            System.out.println("\nSetting Instance 2: "); Car.setInstance();


            // Retrieving the first instance and printing its details Car carObject1 =
            Car.getInstance(); System.out.println("Details of Car Object 1");
            carObject1.printDetails();


            // Retrieving the second instance and printing its details Car carObject2 =
            Car.getInstance(); System.out.println("Details of Car Object 2");
            carObject2.printDetails();


            // Checking if both objects refer to the same instance

            if (carObject1 == carObject2) {
                  System.out.println("Both objects are the same instance.");
            }

      }

}
```

Output:



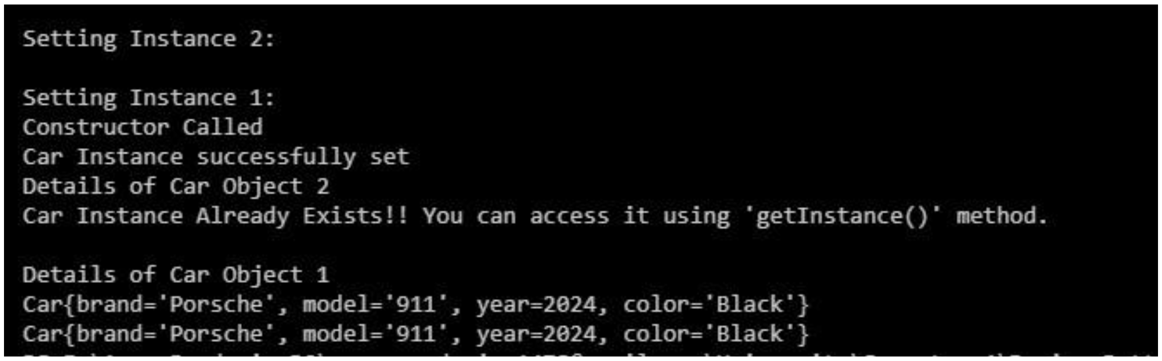
```
Setting Instance 1:
Constructor Called
Car Instance successfully set

Setting Instance 2:
Car Instance Already Exists!! You can access it using 'getInstance()' method.

Details of Car Object 1
Car{brand='Porsche', model='911', year=2024, color='Black'}
Details of Car Object 2
Car{brand='Porsche', model='911', year=2024, color='Black'}
Both objects are the same instance.
```

c) Synchronized Singleton

Implementation:

```
// Car class representing a car with specific details

class Car {
        private String model, color, brand; private int year;

        // Singleton instance of Car, initialized to null

        private static Car carObject = null;

        // Private constructor to prevent direct instantiation

        private Car() {
                // Setting default values for brand, model, year, and color

                this.brand = "Porsche";
                this.model = "911";
                this.year = 2024; this.color =
                "Black";
                System.out.println("Constructor Called");
        }


        // Getter method to retrieve the singleton instance of Car

        public static Car getInstance() { return
                carObject;
        }


        // Synchronized setter method to create a new Car instance if it doesn't already

        // exist

        public static synchronized void setInstance() { if (carObject
                == null) {
                        carObject = new Car();
                        System.out.println("Car Instance successfully set");
                } else
                        System.out.println("Car Instance Already Exists!! You can access it using
'getInstance()' method.\n");
        }


        // Method to print the details of the car

        public void printDetails() {
            System.out.println("Car{" +
                    "brand='" + brand + '\'' + ",
                    model='" + model + '\'' + ",
                    year=" + year +
                   ", color='" + color + '\'' +
                   "}");

    }

}


// Main class to demonstrate the synchronized singleton pattern

public class Pr5_Singleton_Synchronized { public static

    void main(String[] args) {

        // Creating and starting Thread t1

        Thread t1 = new Thread(new Runnable() { public void
            run() {
                System.out.println("\nSetting Instance 1: "); Car.setInstance();
                Car carObject1 = Car.getInstance(); System.out.println("Details
                of Car Object 1"); carObject1.printDetails();
            }

        });


        // Creating and starting Thread t2

        Thread t2 = new Thread(new Runnable() { public void
            run() {
                System.out.println("\nSetting Instance 2: "); Car.setInstance();
                Car carObject2 = Car.getInstance(); System.out.println("Details
                of Car Object 2"); carObject2.printDetails();
            }

        });
```

Output:



```
Setting Instance 2:

Setting Instance 1:
Constructor Called
Car Instance successfully set
Details of Car Object 2
Car Instance Already Exists!! You can access it using 'getInstance()' method.

Details of Car Object 1
Car{brand='Porsche', model='911', year=2024, color='Black'}
Car{brand='Porsche', model='911', year=2024, color='Black'}
```

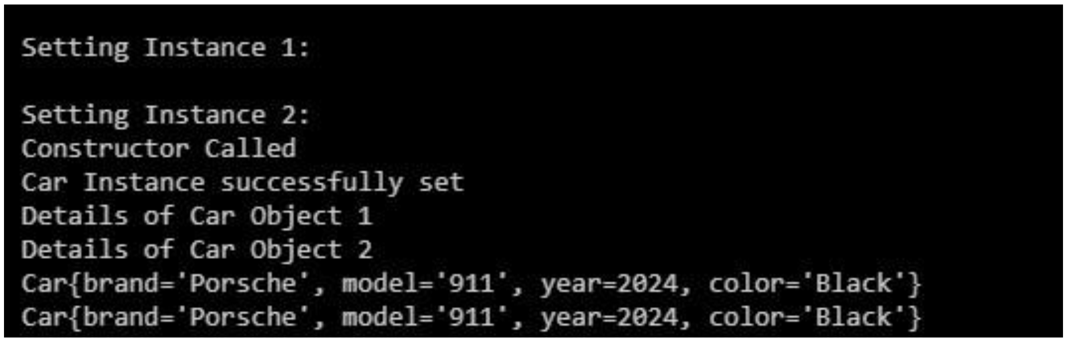
d) Double Checking Singleton

Implementation:

```
// Car class representing a car with specific details

class Car {
        private String model, color, brand; private int year;

        // Singleton instance of Car, initialized to null

        private static Car carObject = null;

        // Private constructor to prevent direct instantiation

        private Car() {
                // Setting default values for brand, model, year, and color

                this.brand = "Porsche";
                this.model = "911";
                this.year = 2024; this.color =
                "Black";
                System.out.println("Constructor Called");
        }


        // Getter method to retrieve the singleton instance of Car

        public static Car getInstance() { return
                carObject;
        }


        // Setter method to create a new Car instance using double-checking for thread

        // safety

        public static void setInstance() { if (carObject
                == null) {
                        synchronized (Car.class) { if
                                (carObject == null) {
                                        carObject = new Car();
                                        System.out.println("Car Instance successfully set");
                                }

                        }

                } else
                        System.out.println("Car Instance Already Exists!! You can access it using
'getInstance()' method.\n");
        }
```

```
    // Method to print the details of the car

    public void printDetails() {
        System.out.println("Car{" + "brand='" +
                brand + '\'' + ", model='" + model +
                '\'' + ", year=" + year +
                ", color='" + color + '\'' +
                "}");

    }}


// Main class to demonstrate the double-checking singleton pattern

public class Pr5_Singleton_DoubleChecking {

    public static void main(String[] args) {

        // Creating and starting Thread t1

        Thread t1 = new Thread(new Runnable() { public void
            run() {
                System.out.println("\nSetting Instance 1: "); Car.setInstance();
                Car carObject1 = Car.getInstance();
                System.out.println("Details of Car Object 1");
                carObject1.printDetails();
            }

        });

        // Creating and starting Thread t2

        Thread t2 = new Thread(new Runnable() { public void
            run() {
                System.out.println("\nSetting Instance 2: "); Car.setInstance();
                Car carObject2 = Car.getInstance();
                System.out.println("Details of Car Object 2");
                carObject2.printDetails();
            }

        });


        t1.start(); // Start Thread t1
        t2.start(); // Start Thread t2

    }
```

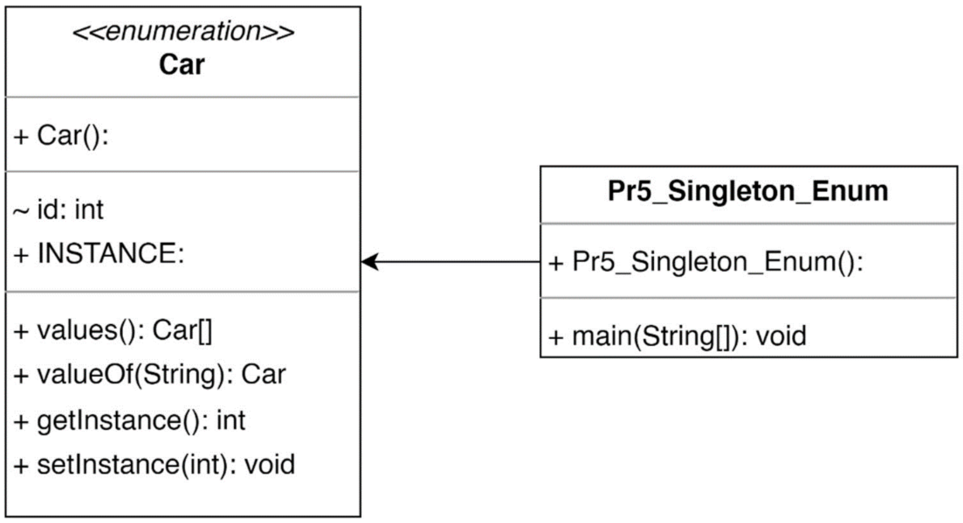
Output:



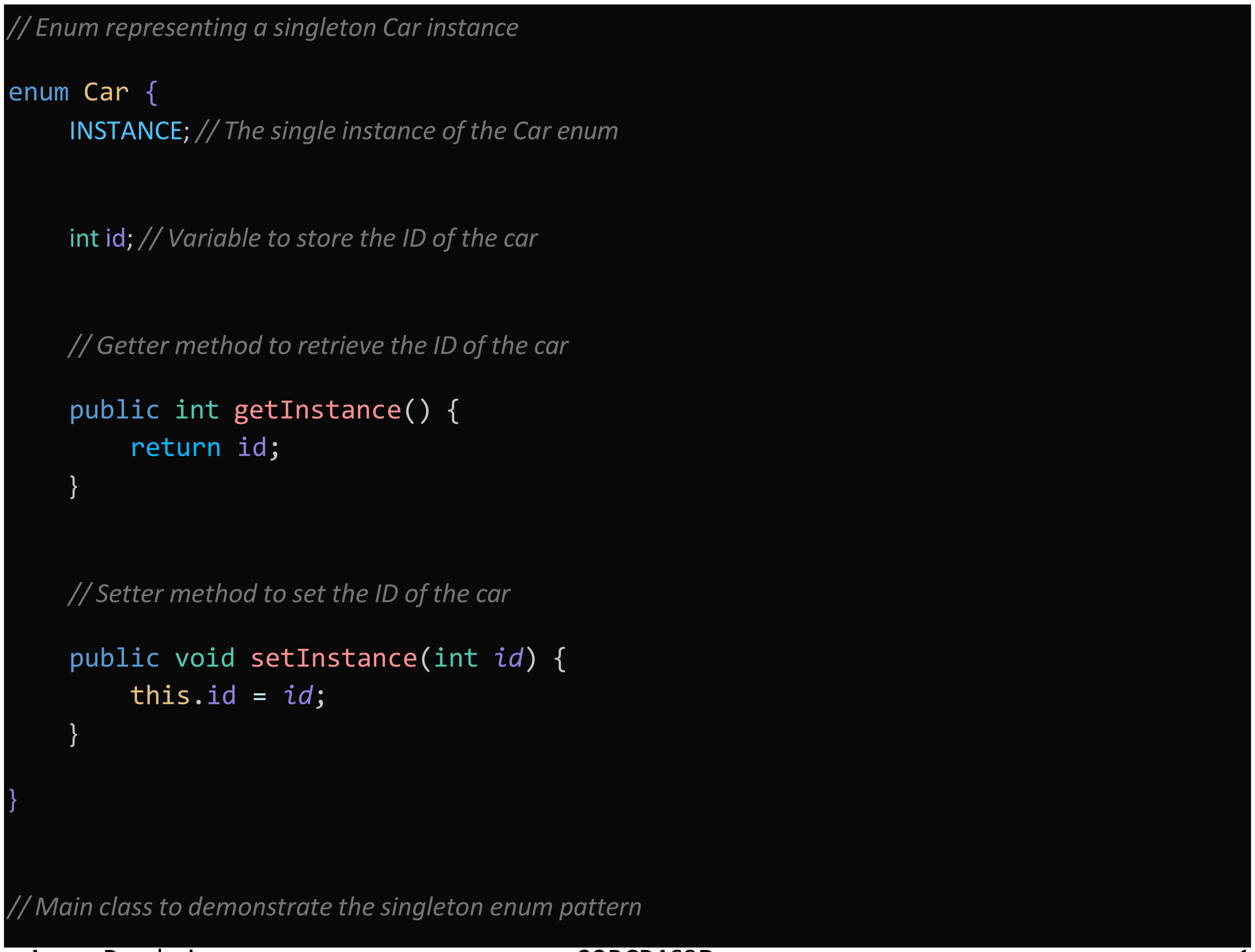
```
Setting Instance 1:

Setting Instance 2:
Constructor Called
Car Instance successfully set
Details of Car Object 1
Details of Car Object 2
Car{brand='Porsche', model='911', year=2024, color='Black'}
Car{brand='Porsche', model='911', year=2024, color='Black'}
```

e) Enum Singleton

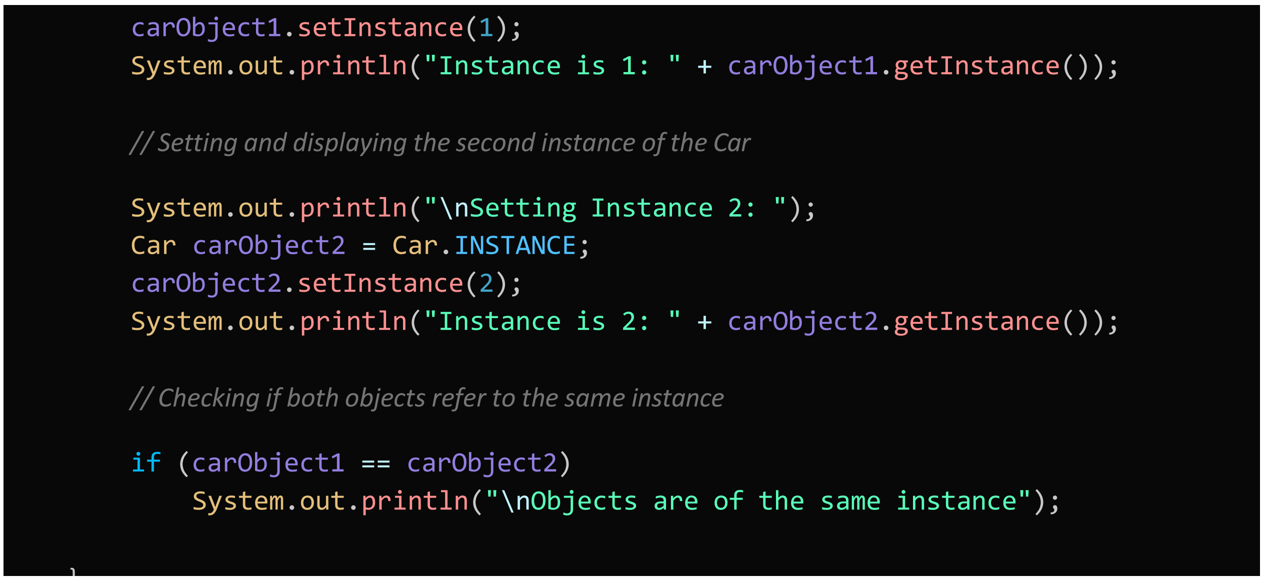
UML Class Diagram:



Implementation:

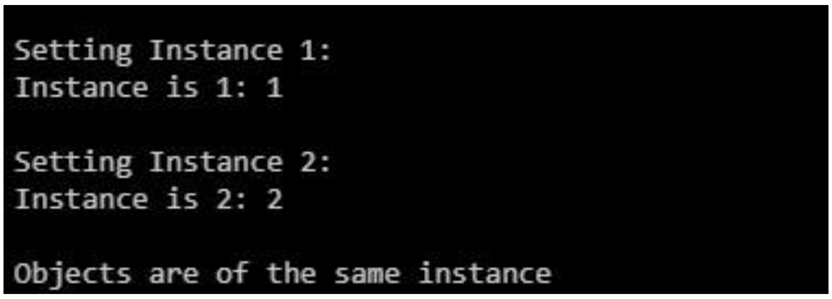


```java
// Enum representing a singleton Car instance

enum Car {
    INSTANCE; // The single instance of the Car enum


    int id; // Variable to store the ID of the car


    // Getter method to retrieve the ID of the car

    public int getInstance() {
        return id;
    }


    // Setter method to set the ID of the car

    public void setInstance(int id) {
        this.id = id;
    }

}


// Main class to demonstrate the singleton enum pattern
```



```java
        carObject1.setInstance(1);
        System.out.println("Instance is 1: " + carObject1.getInstance());

        // Setting and displaying the second instance of the Car

        System.out.println("\nSetting Instance 2: ");
        Car carObject2 = Car.INSTANCE;
        carObject2.setInstance(2);
        System.out.println("Instance is 2: " + carObject2.getInstance());

        // Checking if both objects refer to the same instance

        if (carObject1 == carObject2)
            System.out.println("\nObjects are of the same instance");
    }
```

Output:



```
Setting Instance 1:
Instance is 1: 1

Setting Instance 2:
Instance is 2: 2

Objects are of the same instance
```

Observation & Learning:

The singleton design pattern ensures that only one instance of a class exists and provides a global point of access to it.

Observation:

1. Singleton restricts class instantiation to one object.

2. It often uses a static member variable to hold the single instance.

3. Access to this instance is typically provided through a static method.

Learning:

1. Singleton maintains a single instance throughout the program.

2. It's useful for coordinating actions across the system.

3. Singleton can be implemented with lazy or eager initialization.

4. Thread safety is crucial in multithreaded environments.

5. Overuse can lead to tight coupling and hinder testability.

Conclusion:

Singleton is valuable for managing single instances and ensuring consistent application state. However, it should be used wisely to avoid unnecessary complexity and tight coupling. Thread safety is essential in implementation.