**Assignment 5: Singleton Design Pattern**

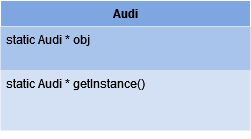
**What is Singleton Design Pattern?**

Singleton is a creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code.

**Intent**

It is a design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

**Structure (Class Diagram)**



**Implementation (Code)**

1. **Eager Singleton**

public class SingleTon {

public static void main(String[] args) {

Audi obj1 = Audi.getInstance();

Audi obj2 = Audi.getInstance();

}

}

class Audi {

public static Audi obj = new Audi(); **// Creating static object of class Audi**

private Audi(){ } **// Creating Constructor**

**// Creating Static object to achieve singleton pattern.**

public static Audi getInstance(){

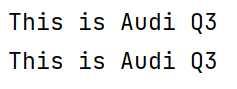
System.out.println("This is Audi Q3");

return obj;

}

}

**Output:**

****

1. **Lazy Singleton  
   Output:**

public class SingleTonLazyDemo {

public static void main(String[] args) {

BMW obj1 = BMW.getInstance();

BMW obj2 = BMW.getInstance();

}

}

class BMW {

public static BMW obj = new BMW();

private BMW(){

System.out.println("This is BMW I4");

}

public static BMW getInstance(){

if (obj == null){

obj = new BMW(); **// Creating the object here….lazy 😒**

}

return obj;

}

}

public class SingleTonLazyDemo {

public static void main(String[] args) {

BMW obj1 = BMW.getInstance();

BMW obj2 = BMW.getInstance();

}

}

class BMW {

public static BMW obj = new BMW();

private BMW(){

System.out.println("This is BMW I4");

}

public static BMW getInstance(){

if (obj == null){

obj = new BMW();

}

return obj;

}

}



1. **Double-checked Locking**

public class SynchronizedGetInstance {

public static void main(String[] args) {

Thread t1 = new Thread(new Runnable() {

public void run() {

Ferrari obj = Ferrari.getInstance();

}

});

Thread t2 = new Thread(new Runnable() {

public void run() {

Ferrari obj = Ferrari.getInstance();

}

});

t1.start();

t2.start();

}

}

class Ferrari {

public static Ferrari obj;

private Ferrari(){

System.out.println("Ferrari F8: 40200000");

}

public static Ferrari getInstance(){ **// Double checked Locking – removing synchronized**

if (obj == null){

synchronized (Ferrari.class) { **// Putting Synchronized here**

if (obj == null) {

obj = new Ferrari();

}

}

}

return obj;

}

}

**Output:**

****

1. **Enum Singleton**

public class SingleTonLazyDemo {

public static void main(String[] args) {

BMW obj1 = BMW.getInstance();

BMW obj2 = BMW.getInstance();

}

}

class BMW {

public static BMW obj = new BMW();

private BMW(){

System.out.println("This is BMW I4");

}

public static BMW getInstance(){

if (obj == null){

obj = new BMW();

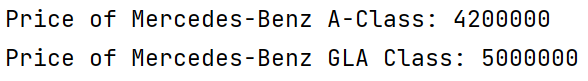
}

return obj;

}

}

**Output:**

****

**Applicability**

1. Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.
2. The Singleton pattern disables all other means of creating objects of a class except for the special creation method. This method either creates a new object or returns an existing one if it has already been created.
3. Use the Singleton pattern when you need stricter control over global variables.