	Course Name: Design Patterns/Thinking LAB	**EXPERIMENT NO. 5**	
	Course Code: 20CP210P **Faculty: Dr. Ketan Sabale**	**Branch: CSE**	**Semester: IV**
Submitted by: Rhythm Shah **Roll no: 22BCP071**			

Objective: To familiarize students with standard Creational design patterns.

Experiment: Explain the singleton design pattern and write a program using any object-oriented programming language to demonstrate the working of singleton design pattern.

Theory :

The Singleton design pattern is a creational pattern that ensures a class has only one instance and provides a global point of access to that instance. This pattern is useful when you want to control access to a resource that should have a single instance throughout the entire application, such as a database connection, configuration manager, or logger. The Singleton pattern typically involves defining a static method or property within the class to return the sole instance, and ensuring that no other instances can be created through private constructors or other means.

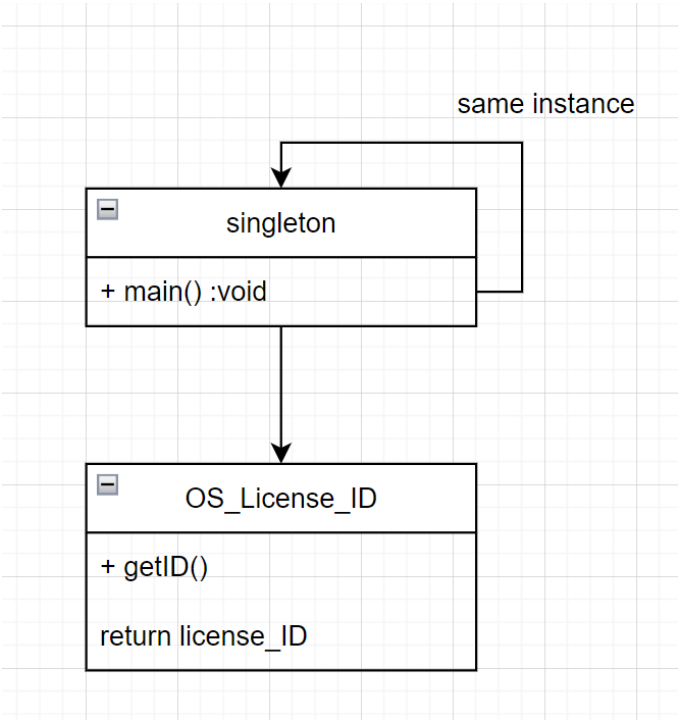
There are 5 methods through which this design pattern can be implemented.

Problem Statement Explanation:

I have taken a class OS_License_ID in which I am returning an object license_ID and having a main method called singleton. Similarly like these Singelton pattern can be implemented through five methods

a) Eagerly Instance
b) Lazy Installation
c) Synchronized
d) Double Checked Locking
e) enum

UML DIAGRAM:



Method 1 – Eagerly Instance

Code:

```
class OS_License_Id
{
   static OS_License_Id license_Id = new OS_License_Id();
   private OS_License_Id()
   {
      System.out.println("You created an object");
   }
   public static OS_License_Id getId()
   {
      return license_Id;
   }
}
public class singleton {
```

```
 public static void main(String[] args) {
      OS_License_Id License_Id1 = OS_License_Id.getId();
      OS_License_Id License_Id2 = OS_License_Id.getId();//if you allow one class than your class is singelton
   }
}
```

Output:

```
PS E:\Fourth sem\Design pattern lab> cd "e:\Fourth sem\Design pattern lab\singleton\" .java } ; if ($?) { java singleton }
You created an object
PS E:\Fourth sem\Design pattern lab\singleton>
```

Method 2: - Lazy Installation

Limitation of Method1 : Even if we are not using the object it will stay in the memory that means wastage of memory.

Code:

```
class OS_License_Id
{
   static OS_License_Id license_Id;
   private OS_License_Id()
   {
      System.out.println("You created an object");
   }
   public static OS_License_Id getId()
   {
      if(license_Id == null)
      {
         license_Id = new OS_License_Id();
      }
```

```
        return license_Id;
    }
}
public class lazy_installation {
    public static void main(String[] args) {
        OS_License_Id License_Id1 = OS_License_Id.getId();
        OS_License_Id License_Id2 = OS_License_Id.getId();//if you allow one class than your class is singelton
    }
}
```

Output:

```
PS E:\Fourth sem\Design pattern lab> cd "e:\Fourth sem\Design pattern lab\singleton\"
allation.java } ; if ($?) { java lazy_installation }
You created an object
PS E:\Fourth sem\Design pattern lab\singleton>
```

Method 3: - Synchronized

Limitation of Method2 : If we create multiple thread's which access the method parallelly and due to that multiple objects can be created.

Code:

```
class OS_License_Id
{
    static OS_License_Id license_Id;
    private OS_License_Id()
    {
        System.out.println("You created an object");
    }
```

```
    public static synchronized OS_License_Id getId()
    {
        if(license_Id == null)
        {
            license_Id = new OS_License_Id();
        }

        return license_Id;
    }
}
public class synchronize {
    public static void main(String[] args) {
        Thread t1 = new Thread(new Runnable()
        {
            public void run()
            {
                OS_License_Id License_Id1 = OS_License_Id.getId();
            }
        });

        Thread t2 = new Thread(new Runnable()
        {
            public void run()
            {
                OS_License_Id License_Id2 = OS_License_Id.getId();
            }
        });

    t1.start();
    t2.start();  //OS_License_Id License_Id2 = OS_License_Id.getId() if you allow one class than your class is singelton
```

```
    }
}
```

Output:

```
PS E:\Fourth sem\Design pattern lab> cd "e:\Fourth sem\Design pattern lab\singleton\"
ze.java } ; if ($?) { java synchronize }
You created an object
PS E:\Fourth sem\Design pattern lab\singleton>
```

Method 4: - Double Checked Locking

Limitation of Method 3 :

If we mark whole method as synchronized the task other than creating object also have to wait for one thread to complete so don't make the entire method synchronize.

Code:

```
class OS_License_Id
{
    static OS_License_Id license_Id;
    private OS_License_Id()
    {
        System.out.println("You created an object");
    }
    public static OS_License_Id getId()
    {
        if(license_Id == null)
        {
            synchronized(OS_License_Id.class)
            {
            if(license_Id == null)
            license_Id = new OS_License_Id();
```

```
            }
        }
        return license_Id;
    }
}
public class doublecheckedlocking {
    public static void main(String[] args) {
        Thread t1 = new Thread(new Runnable()
        {
            public void run()
            {
                OS_License_Id License_Id1 = OS_License_Id.getId();
            }
        });

        Thread t2 = new Thread(new Runnable()
        {
            public void run()
            {
                OS_License_Id License_Id2 = OS_License_Id.getId();
            }
        });

    t1.start();
    t2.start();  //OS_License_Id License_Id2 = OS_License_Id.getId() if you allow one class than your class is singelton
    }

}
```

Output:

```
PS E:\Fourth sem\Design pattern lab> cd "e:\Fourth sem\Design pattern lab\singleton\"
ze.java } ; if ($?) { java synchronize }
You created an object
PS E:\Fourth sem\Design pattern lab\singleton>
```

Method 5: - enum

There is no limitation of Double Checked Locking. Just another way to implement more efficiently

Code:

```
enum OS_License_Id
{
   INSTANCE;
   int i;
   public void show()
   {
      System.out.println(i);
   }
}
public class enum_pattern{
   public static void main(String[] args) {
      OS_License_Id License_Id1 = OS_License_Id.INSTANCE;
      License_Id1.i = 4;
      OS_License_Id License_Id2 = OS_License_Id.INSTANCE;
      License_Id2.i = 5;
      License_Id2.show();
      License_Id2.show();
```

```
    }
}
```

Output:

```
PS E:\Fourth sem\Design pattern lab> cd "e:\Fourth sem\Design pattern lab\singleton\"
ern.java } ; if ($?) { java enum_pattern }
5
5
PS E:\Fourth sem\Design pattern lab\singleton>
```