DESIGN PATTERNS

Design patterns are typical solution to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

- Creational patterns provides object creation mechanism that increases flexibility and reuse of existing code.
- Structural patterns explain how to assemble objects and class into larger structure, while keeping these structures flexible and efficient.
- Behavioral patterns take care of effective communication and the assignment of responsibility b/w objects.

• Creational Design Pattern

1. Singleton

Ensures a class has only one instance and provide global access point to it.

When to use singleton?
- → Shared config manager.
- → Logging system.
- → cache manager.
- → Threadpool.
- → Hardware Resource manager.

- private constructor
- static instance reference.
- public static methods to access instance.

} Together achieve singelton.

- Problems with Singleton :-
  - → Global state
  - → Tight coupling.
  - → Hard to test code.
  - → Hidden dependencies
  - → Violates DIP in many cases.

A) Eager Initilization

```
Class Singleton{
  Private Static Final Singleton INSTANCE =new Singleton();
  Private Singleton(){ }
  Public Static Singleton getInstance(){
    return INSTANCE;
  }
}
```

✓ Thread Safe
✓ Simple.
× Instance created even if not used.

B) Lazy Initilization

```
Class Singleton{
  Private static Singleton instance;
  Private Singleton (){}
  Public Static Singleton getInstance (){
    if (instance ==null){
      instance= new Singleton();
    }
    return instance;
  }
}
```

✓ Lazy loaded
× not-thread Safe.

C) Thread-Safe (Synchronized).

```
Public Static synchronized Singleton getInstance()
```

✓ Thread Safe
× Slower due to locking.

D) Double-checked Locking

```
Class Singleton {
  Private static volatile Singleton instance;
  private Singleton(){}
  Public static Singleton getInstance(){
    if (instance ==null){
      Synchronized (Singleton.class){
        if(instance ==null) {instance= new Singleton();}
      }
    } return instance;
}
```

✓ Efficient.
✓ Thread Safe
⚠ Requires volatile.

E) Bill Pugh / Holder Pattern (Best Pattern)

```
Class Singleton{
  private Singleton(){}
  private static class Holder{
     private static final Singleton INSTANCE = new Singleton();
  }
  Public static Singleton getInstance(){
   return Holder.Instance;
  }
}
```

- Lazy.
- Thread-safe.
- No synchronization overhead
- clean.