Singleton Design Pattern

Used when centralized Management of Resources is required

* Logging System
* Configuration Management
* DBConnection Pooling
* Caching
* Resource Management
* ThreadPoolManager

Why do we need these Logging Systems Single Instance?

* Logs to go to single File/console
* Easier Debugging
* No wastage of Resource
* if multiple instances are there we might end up Logging to the same file. Difficult to debug.

Eager Init()

```
class Logger{
    private static final instance = new Logger();
    private Logger(){
    }
    public static Logger getInstance(){
        return instance;
    }
}.
```

Init during Server Startup.

Constructor Private

Lazy init()

```
class Logger{
    private static instance;
    private Logger(){
    }
    private static Logger getInstance(){
        if(instance == null){
            instance = new Logger();
        }
        return instance;
    }
}.
```

init when method is called

```
class Logger{
    private static final instance;
    private Logger(){
    }
    private static Logger getInstance(){
        if(instance == null){
            synchronised(Logger.class){
                if(instance == null){
                    instance = new Logger();
                }
            }
        }
        return instance;
```

faster than method level Locking but code made Faster

(Singleton Design Pattern)

Bill Pugh's Holder Instance

This gives better performance than synchronized step.

```
class Logger {
    private Logger(){
    }
    static class InnerClass{
        private static final instance = new Logger();
    }
    public static Logger getInstance(){
        return InnerClass.instance;
    }
}.
```

This method works because how JVM internally handles the class creation and everything. So

Static Inner class is not initialized by JVM unless its static method or Static member type is called

Doing `Logger.InnerClass inner;`

⇓

This does not make JVM create the instance

`Logger.InnerClass.instance`

⇓

this is responsible for creating the instance.

New Learning

When we do

`ClassA.getClassFactory();`

⇓

Static method internally JVM creates an instance of ClassA. We just don't see it.

Next time `ClassA.getClassFactory()` is called we are using the same Instance created by JVM.