Design Patterns:

- Design patterns are well established solutions to the common software design problems.
- Types of design patterns
  - Creational
  - Structural
  - Behavioural

Creational Design Patterns:

- Creational design pattern deals with how will be an object created?, how many objects will be created for a class?

Singleton Design Pattern:

- Singleton design pattern allows to create a class for which only object can be created across the system.
- Singletons are always immutable, if we create a singleton from mutable class/object, then making changes in 1 place will affect in all the places where it's used.
- Singleton objects permits lazy loading, In case of static object (eager loading), the singleton object will be created at the class load time & it will take up unnecessary memory even though it was not used.

Why Singleton design pattern needed?

- When shared resources are used
  - In case of the DatabaseConnection object, internally it contains list of TCP connection pool, wherein each TCP connection can be used to handle separate sql work.
  - In this case, we don't need to create multiple objects for DatabaseConnection.
- Creation of the object is expensive
  - In case of the DatabaseConnection object, its not a simple object creation.
- Logging is a common real-world use case for singletons, because all objects that wish to write messages to a single source.

Singleton Different Versions:

- v1 - https://github.com/PraveenS1997/DesignPatterns/blob/main/Singleton/v1/DatabaseConnection.java
- v2 - Eager Loading - https://github.com/PraveenS1997/DesignPatterns/blob/main/Singleton/v2_Eager_Loading/DatabaseConnection.java
- v3 - Mutex Locking - https://github.com/PraveenS1997/DesignPatterns/blob/main/Singleton/v3_mutex/DatabaseConnection.java

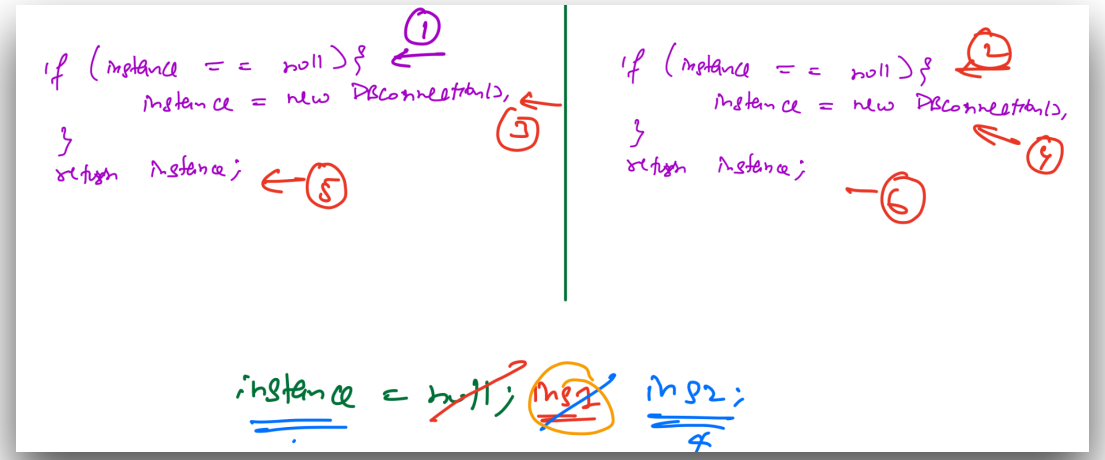
- v4 - Double check locking - https://github.com/PraveenS1997/DesignPatterns/blob/main/Singleton/v4_double_check_locking/DatabaseConnection.java

Steps to make a class Singleton (Version 1)

- Make the constructor private
- Create a static method to return the DatabaseConnection instance
- Create a static variable of type DatabaseConnection & initialise with null
- Have checks in the getInstance() method,
  - If the object is created -> return
  - else -> create new object & return

Concurrency Issues

- Let's say the instance is initially null and 2 different threads enters the below if condition in the getInstance() method at the same time if(instance == null) { instance = new DatabaseConnection(); }
- Since the instance is null, both the threads will get pass the if condition & both will create 2 different DatabaseConnection object, which violates the Singleton pattern
- If thread 1 created the instance, it will assign the object to instance variable, then the thread 2 will create another instance & overwrite the instance variable. One of the DatabaseConnection object will be garbage collected



Eager Loading/Execution: (Version 2)

- We can create the object of DatabaseConnection at the time of static variable initialisation
- So the actual object will be created during the class load or when the application starts

Cons of Eager loading

- The issues with creating the object at the compile time slow down the application start time, if we follow this approach for all singleton objects present in the application
- If any singleton object requires certain arguments to be passed (any configuration details) to create the object, we can't pass those parameters at the class load/ application startup time (as the application is not completely started) and the configuration details are not available, for example we want to pass the environment parameter (prod, dev) & based on that, the log level will be decided.
- Due to lazy lading all the singleton objects are created & available even though it was not used by any code unit, which will consume lots of unwanted memory unnecessarily.

Lock the getInstance() method: (Version 3)

- We can lock the getInstance() method using the mutex or mark the method synchronised, so that only one thread can access the method at any given time.

Cons of locking or marking the method synchronised

- In case of Singleton objects, we will face the concurrency only for the first time if the object is not initialised.
- Once the Singleton object is initialised, there won't be any concurrency issues, but by locking the entire method or marking it as synchronised will give access to only thread at any given point in time.
- Even though the Singleton object is created & set, still multiple threads can't access the getInstance() method due to locking
- In the version 3 changes, the actual critical section which can create concurrency issue is the below line

```
public static DatabaseConnection getInstance_1(){
    synchronized (DatabaseConnection.class){
        if(instance == null){
            // the below line is the critical section
            instance = new DatabaseConnection();
        }
    }
    return instance;
}
```

Very important observation

- In the above code, even though the critical section is just one line, but we did the locking above the if condition.
- The reason is, if the instance is null & 2 different threads execute the if condition at the same time & get inside the if block and
    - One of the thread will acquire the lock & enter the critical section & the other thread will wait for the lock to be freed (still the other thread is inside the if block)
    - Thread 1 will create the DatabaseConnection object & assign it to the instance variable & release the lock.
        - Now Thread 2 will acquire the lock & since its already inside the if block, so it will again create another new instance of DatabaseConnection & overwrite the instance variable.
        - So, here if we lock the critical section alone, still multiple objects can be created for the Singleton
- The above issue can be resolved using "Double Check Locking" technique.

Double check locking (Version 4)

- If 2 different threads execute the if condition & enter the if block of if(instance == null) { } at the same time, then one thread will acquire the lock & create the DatabaseConnection object & assign it to instance variable.
- Then the 2nd thread acquires the lock & it will enter the critical section, so here instead of creating new DatabaseConnection object directly & assign it to instance variable, we will have one more if condition of if(instance == null) to check if any other prior threads which acquired the lock had created the object.
- This is called Double check locking. Using this approach, we do the locking only if the object is not initialised at the first time, once its initialised we don't block multiple threads to execute the getInstance() method at the same time