JAVA CONCURRENCY

→ Concurrency Evolution :-

JDK 1.x release, there were few classes present in the initial release

- java.lang.Thread
- java.lang.ThreadGroup
- java.lang.Runnable
- java.lang.Process
- java.lang.ThreadDeath
- and some exception classe.

e.g
- java.lang.IllegalMonitorStateException.
- java.lang.IllegalStateException
- java.lang.IllegalThreadStateException.

Some synchronised collection like java.util.Hashtable

JDK 1.5 was first big release after JDK 1.1 and it had include multiple concurrency utilities. Executor, semaphore, mutex, barrier, latches, concurrent collections and blocking queues. all were included in this release.

JDK 1.6 was more of platform bug fixes than API upgrade

JDK 1.7 added the support of forkJoinPool which implemented work-stealing technique. to maximize the throughput.

JDK 1.8 is largely known for lambda changes, but it also has some concurrency changes as well. Two new interfaces and four new classes were added in java.util.concurrent package. CompletableFuture and CompletionException.

#Object level lock Vs Class level Lock.

In Java, a synchronized block of code can only be executed by one thread at a time.

Sychronization is a process which keeps all the concurrent threads in execution to be in sync.

Sychronization avoids memory inconsitence error.

① OBJECT LEVEL LOCK IN JAVA :-

Object level lock is a mechanism when we want to synchronize a non-static method or a non-static code block such that only one thread will be able to execute the code block on given instance of the class. This should always be done to make instance level data thread safe.

Various ways for object level locking.

(a)
```
public class Democlass {
        public synchronized void demomethod(){}
}
```

(b)
```
public class Democlass {
    public void demomethod() {
        sychronized (this)
        {
            // other thread safe code
        }
    }
}
```

(c)
```
public class Democlass {
    private final Object lock = new Object();
    public void demomethod() {
        synchronized (lock) {
            //other sychronized code
        }
```

② CLASS LEVEL LOCK

Class level lock prevents multiple threads to enter in sychronized block in ony of all instances of the class on runtime. This means if thre are 100 instances of Democlass, then only on thread will be able to execute demoMethod() in any one instance at a time and all other instances will be locked for other threads.

class level locking should always be done to make static data thread safe.

Various ways to acheive class level locking.

→ (a)

```
public class Democlass {
        public sychronized static void demoMethod()
        {
            // Some code
        }
}
```

→ (b)

```
public class Democlass {
        public void demoMethod () {
            // Acquire lock on .class reference
        sychronized (Democlass.class) {
            // other code
        }
    }
}
```

→ (c)

```
public class Democlass {
    private final static Object lock = new Object();
    public void demoMethod () {
        sychronized (lock) {
            // Some code
        }
    }
}
```

↳ Lock object is static

Important Notes on ~~*~~ synchronized keyword

→ Sychronized keyword can be used only with method and block. These method or blocks can be static or non-static in nature.

→ Whenever a threads enters into a sychronized block, it aquaires the lock and whenever it leves the block, it releases the lock. Lock is released even if thread leaves sychronized method after completion or due to any Error or exception.

→ Java sychronized keyword is re-entrant in nature, it means if a sychronized method calls another sychronized method which require the same lock then current thread which is holding lock can enter into the method without aquring lock.

→ Java sychronization will throw null-pointer exception if object used in sychronization is null.

→ Sychronized keyword cannot be used with constructor.

→ Donot sychronize on non-final feild on sychronized block in java.

→ Donot used string literals becuse they might be reference elsewhere in application and can cause deaklock.

Java Compare And SWAP Algorithm :-

One of the best additions in Java 5 was Atomic operations supported in classes such as AtomicInteger and AtomicLong etc. Thes classes help in minimizing the need of complex multi-threading code for some basic operation. such as increment or decrement a value. which is shared accross multiple threads. These classes internally relied on an algorithm named CAS [Compare and SWAP] algorithm.

Optimistic and Pessimistic Locking.

Pessimistic ~~Optimistic~~ Locking :- This is the traditional Locking mechanism using sychronized keyword in java. It ask you to first guarantee that no other thread will interfare in between certain operation and then only allow the access to any instance/method. This method works but comes with a great performance penalty.

Optimistic Locking :- In this approach we proceed with the update, being hopefull that you can ~~proceed~~ complete it without any interferance. It uses the algorithm CAS → Compare And SWAP.

① → Read this value and make a copy of it.

② → It do some operation on A. Before doing any operation, it will make a copy.

③ → Before it go and update this new value in memory, it compare the memory value with its own copy value and update only if both are same. else return error.