What is Multitasking?

Answer:- Executing several tasks simultaneously is called Multitasking. There are two types of Multitasking Process based Multitasking and Thread based Multitasking.

What is Process based Multitasking (Multiprocessing)?

Answer:- Executing several tasks simultaneously where each task is separate independent process is called Process based Multitasking. It is Os level concepts.

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

What is Thread based Multitasking (Multithreading)?

Answer:- Executing several tasks simultaneously where each task is a separate independent part of the same program is called Thread based Multitasking.

- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.
- At least one process is required for each thread.

What is the difference between Processes and Threads?

Answer:-

Processes	Threads
A process has separate virtual address space. Two processes running on the same system at the same time do not overlap each other.	Threads are entities within a process. All threads of a process share its virtual address space and system resources but they have their own stack created.
Every process has its own data segment.	All threads created by the process share the same data segment. All threads belong to a process share common file descriptors, heap memory and other resource but each thread has its own exception handler and own stack in Java.
Processes use inter process communication (IPC) techniques to interact with other processes. , such as pipes and sockets. IPC	Threads do not need inter process communication techniques because they are not altogether separate address spaces.

is used not just for communication between processes on the same system, but processes on different systems.	They share the same address space; therefore, they can directly communicate with other threads of the process.
Process has no synchronization overhead in the same way a thread has.	Threads of a process share the same address space; therefore synchronizing the access to the shared data within the process's address space becomes very important.
Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a ProcessBuilder object.	

What is the Benefits of Multithreading ?

Answer:- There are following benefits of Multithreading:-

- Better resource utilization.
- Simpler program design in some situations.
- More responsive programs.

What is the cost of Multithreading ?

Answer:- There are following cost of Multithreading:-

- **More complex design:-**Though some parts of a multithreaded applications is simpler than a single threaded application, other parts are more complex. Code executed by multiple threads accessing shared data need special attention. Thread interaction is far from always simple. Errors arising from incorrect thread synchronization can be very hard to detect, reproduce and fix.
- **Context Switching Overhead:-**When a CPU switches from executing one thread to executing another, the CPU needs to save the local data, program pointer etc. of the current thread, and load the local data, program pointer etc. of the next thread to execute. This switch is called a "context switch". The CPU switches from executing in the context of one thread to executing in the context of another.Context switching isn't cheap. You don't want to switch between threads more than necessary.
- **Increased Resource Consumption:-** A thread needs some resources from the computer in order to run. Besides CPU time a thread needs some memory to keep its local stack. It may also take up some resources inside the operating system needed to manage the thread. Try creating a program that creates 100 threads that does nothing but wait, and see how much memory the application takes when running.

What is Thread ?

Answer:- The Thread is:-

- An independent sequential path of execution within a program.

- A lightweight sub process, a smallest unit of processing. It is a separate path of execution.

What do we understand by the term concurrency?

Answer:- Concurrency is the ability of a program to execute several computations simultaneously. This can be achieved by distributing the computations over the available CPU cores of a machine or even over different machines within the same network.

In Java, what is a process and a thread?

Answer:- In Java, processes correspond to a running Java Virtual Machine (JVM) whereas threads live within the JVM and can be created and stopped by the Java application dynamically at runtime.

What is a scheduler?

Answer:- A scheduler is the implementation of a scheduling algorithm that manages access of processes and threads to some limited resource like the processor or some I/O channel. The goal of most scheduling algorithms is to provide some kind of load balancing for the available processes/threads that guarantees that each process/thread gets an appropriate time frame to access the requested resource exclusively.

How many threads does a Java program have at least?

Answer:- Each Java program is executed within the main thread; hence each Java application has at least one thread.

How can a Java application access the current thread?

Answer:- The current thread can be accessed by calling the static method **currentThread()** of the JDK class java.lang.Thread:-

```
public class MainThread {
      public static void main(String[] args) {
         long id = Thread.currentThread().getId();
        String name = Thread.currentThread().getName();
        ...
     }
}
```

What properties does each Java thread have?

Answer:- Each Java thread has the following properties:-

- an identifier of type long that is unique within the JVM
- a name of type String
- a priority of type int
- a state of type java.lang.Thread.State

- a thread group the thread belongs to

What is the purpose of thread groups?

Answer:- Each thread belongs to a group of threads. The JDK class **java.lang.ThreadGroup** provides some methods to handle a whole group of Threads. With these methods we can, for example, interrupt all threads of a group or set their maximum priority.

What states can a thread have and what is the meaning of each state?

Answer:-

- **NEW:-** A thread that has not yet started is in this state.
- **RUNNABLE:-** A thread executing in the Java virtual machine is in this state.
- **BLOCKED:-** A thread that is blocked waiting for a monitor lock is in this state. A thread gets into this state after calling Object.wait method.
- **WAITING:-** A thread that is waiting indefinitely for another thread to perform a particular action is in this state. A thread can get into this state either by calling - ***Object.wait (without timeout), Thread.join (without timeout), or LockSupport.park methods.***
- **TIMED_WAITING:-** A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state. A thread can get into this state by calling either of these methods: ***Thread.sleep, Object.wait (with timeout specified), Thread.join (with timeout specified), LockSupport.parkNanos, LockSupport.parkUntil***
- **TERMINATED:-** A thread that has exited is in this state.

What is difference between BLOCKED and WAITING state?

Answer:- When a thread calls Object.wait method, it releases all the acquired monitors and is put into WAITING (or TIMED_WAITING if we call the timeout versions of the wait method) state.

Now when the thread is notified either by **notify() or by notifyAll()** call on the same object then the waiting state of the thread ends and the thread starts attempting to regain all the monitors which it had acquired at the time of wait call.
At one time there may be several threads trying to regain (or maybe gain for the first time) their monitors. If more than one threads attempt to acquire the monitor of a particular object then only one thread (selected by the JVM scheduler) is granted the monitor and all other threads are put into BLOCKED state.

What is difference between WAITING and TIMED_WAITING state?

Answer:- A thread in a TIMED_WAITING state will wait at max for the specified timeout period whereas a thread in the WAITING state keeps waiting for an indefinite period of time.

For example, if a thread has called Object.wait method to put itself into WAITING state then it'll keep waiting until the thread is interrupted either by notify() method (OR by notifyAll() method)

call on the same object by another thread. Similarly, if a thread has put itself into WAITING state by calling Thread.join method then it'll keep waiting until the specified thread terminates.

We can easily figure out that a thread in a WAITING state will always be dependent on an action performed by some other thread whereas a thread in TIMED_WAITING is not completely dependent on an action performed by some other thread as in this case the wait ends automatically after the completion of the timeout period.

How do we set the priority of a thread?

Answer:- The priority of a thread is set by using the method **setPriority(int)**.
To set the priority to the maximum value, we use the constant **Thread.MAX_PRIORITY** and to set it to the minimum value we use the constant **Thread.MIN_PRIORITY** because these values can differ between different JVM implementations.

How is a thread created in Java?

Answer:- Basically, there are two ways to create a thread in Java. The first one is to write a class that **extends the JDK class java.lang.Thread** and call its method **start()**:-

```
public class MyThread extends Thread {
public MyThread(String name) {
super(name);
}
@Override
public void run() {
System.out.println("Executing thread "+Thread.currentThread().getName());
}
public static void main(String[] args) throws InterruptedException {
MyThread myThread = new MyThread("myThread");
myThread.start();
}
}
```

The second way is to **implement the interface java.lang.Runnable** and pass this implementation as a parameter to the **constructor of java.lang.Thread**:-

```
public class MyRunnable implements Runnable {
public void run() {
System.out.println("Executing thread "+Thread.currentThread().getName());
}
public static void main(String[] args) throws InterruptedException {
Thread myThread = new Thread(new MyRunnable(), "myRunnable");
```

```
myThread.start();
}
}
```

How do we stop a thread in Java?

Answer:- To stop a thread one can use a volatile reference pointing to the current thread that can be set to null by other threads to indicate the current thread should stop its execution:-

```
private static class MyStopThread extends Thread {
        private volatile Thread stopIndicator;
        public void start() {
                stopIndicator = new Thread(this);
                stopIndicator.start();
       }

       public void stopThread() {
            stopIndicator = null;
       }

       @Override
       public void run() {
              Thread thisThread = Thread.currentThread();
              while(thisThread == stopIndicator) {
                    try {
                          Thread.sleep(1000);
                     } catch (InterruptedException e) { }
              }
      }
}
```

Why should a thread not be stopped by calling its method stop()?

Answer:- A thread should not be stopped by using the deprecated methods stop() of java.lang.Thread, as a call of this method causes the thread to unlock all monitors it has acquired. If any object protected by one of the released locks was in **an inconsistent state**, this state gets visible to all other threads. This can cause arbitrary behavior when other threads work this this inconsistent object.

Is it possible to start a thread twice?

Answer- No, after having started a thread by invoking its start() method, a second invocation of start() will throw an IllegalThreadStateException.

What is the output of the following code?

```
public class MultiThreading {
      private static class MyThread extends Thread {
               public MyThread(String name) {
                     super(name);
               }

               @Override
               public void run() {
                  System.out.println(Thread.currentThread().getName());
               }
      }

      public static void main(String[] args) {
            MyThread myThread = new MyThread("myThread");
             myThread.run();
     }
}
```

Answer:- The code above produces the output "main" and not "myThread". As can be seen in line two of the main() method, we invoke by mistake the method run() instead of start(). Hence, no new thread is started, but the method run() gets executed within the main thread.

What is a daemon thread?

Answer:- Daemon thread in Java are those thread which runs in background and mostly created by JVM for performing background task like Garbage collection.
Thread.setDaemon(true) makes a Thread daemon, If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

Difference between Daemon and Non Daemon:-

- JVM doesn't wait for any daemon thread to finish before exiting.
- Daemon Thread are treated differently than User Thread when JVM terminates, finally blocks are not called, Stacks are not unwounded and JVM just exits.

Why JVM terminates the daemon thread if there is no user thread?

Answer:- The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

A daemon thread is a thread whose execution state is not evaluated when the JVM decides if it should stop or not. The JVM stops when all user threads (in contrast to the daemon threads) are terminated. Hence daemon threads can be used to implement for example monitoring functionality as the thread is stopped by the JVM as soon as all user threads have stopped. The

example application below terminates even though the daemon thread is still running in its endless while loop.

```
public class Example {
private static class MyDaemonThread extends Thread {
public MyDaemonThread() {
setDaemon(true);
}
@Override
public void run() {
while (true) {
try {
Thread.sleep(1);
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}
}
public static void main(String[] args) throws InterruptedException {
Thread thread = new MyDaemonThread();
thread.start();
}
}
```

Is it possible to convert a normal user thread into a daemon thread after it has been started?

Answer:- A user thread cannot be converted into a daemon thread once it has been started. Invoking the method **thread.setDaemon(true)** on an already running thread instance causes a IllegalThreadStateException.

What do we understand by busy waiting?

Answer:- Busy waiting means implementations that wait for an event by performing some active computations that let the thread/process occupy the processor although it could be removed from it by the scheduler. An example for busy waiting would be to spend the waiting time within a loop that determines the current time again and again until a certain point in time is reached:

```
Thread thread = new Thread(new Runnable() {
@Override
public void run() {
long millisToStop = System.currentTimeMillis() + 5000;
long currentTimeMillis = System.currentTimeMillis();
while (millisToStop > currentTimeMillis) {
currentTimeMillis = System.currentTimeMillis();
}
```

```
}
});
```

How can we prevent busy waiting?

Answer:- One way to prevent busy waiting is to put the current thread to sleep for a given amount of time. This can be done by calling the **method java.lang.Thread.sleep(long)** by passing the number of milliseconds to sleep as an argument.

Can we use Thread.sleep() for real-time processing?

Answer:-The number of milliseconds passed to an invocation of Thread.sleep(long) is only an indication for the scheduler how long the current thread does not need to be executed. It may happen that the scheduler lets the thread execute again a few milliseconds earlier or later depending on the actual implementation. Hence an invocation of Thread.sleep() should not be used for real-time processing.

How can a thread be woken up that has been put to sleep before using Thread.sleep()?

Answer:- The method **interrupt() of java.lang.Thread** interrupts a sleeping thread. The interrupted thread that has been put to sleep by calling Thread.sleep() is woken up by an InterruptedException:-

```
public class InterruptExample implements Runnable {
public void run() {
try {
Thread.sleep(Long.MAX_VALUE);
} catch (InterruptedException e) {
System.out.println("["+Thread.currentThread().getName()+"]  -
Interrupted by exception!");
}
}
public static void main(String[] args) throws InterruptedException {
Thread myThread = new Thread(new InterruptExample(), "myThread");
myThread.start();
System.out.println("["+Thread.currentThread().getName()+"] Sleeping in main  -
thread for 5s...");
Thread.sleep(5000);
System.out.println("["+Thread.currentThread().getName()+"] Interrupting  -
myThread");
myThread.interrupt();
}
}
```

How can a thread query if it has been interrupted?

Answer:- If the thread is not within a method like Thread.sleep() that would throw an InterruptedException, the thread can query if it has been interrupted by calling either the **static method Thread.interrupted()** or the method **isInterrupted()** that it has inherited from java.lang.Thread.

How should an InterruptedException be handled?

Answer:- Methods like sleep() and join() throw an InterruptedException to tell the caller that another thread has interrupted this thread. In most cases this is done in order to tell the current thread to stop its current computations and to finish them unexpectedly. Hence ignoring the exception by catching it and only logging it to the console or some log file is often not the appropriate way to handle this kind of exception. The problem with this exception is, that ***the method run() of the Runnable interface does not allow that run() throws any exceptions***. So just rethrowing it does not help. This means the implementation of run() has to handle this checked exception itself and this often leads to the fact that it caught and ignored.

After having started a child thread, how do we wait in the parent thread for the termination of the child thread?

Answer:- Waiting for a thread's termination is done by invoking the method join() on the thread's instance variable:-

```
Thread thread = new Thread(new Runnable() {
@Override
public void run() {
}
});
thread.start();
thread.join();
```

What is the output of the following program?

```
public class MyThreads {
private static class MyDaemonThread extends Thread {
public MyDaemonThread() {
setDaemon(true);
}
@Override
public void run() {
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
}
}
```

```
}
public static void main(String[] args) throws InterruptedException {
Thread thread = new MyDaemonThread();
thread.start();
thread.join();
System.out.println(thread.isAlive());
}
}
```

Answer:- The output of the above code is "false". Although the instance of MyDaemonThread is a daemon thread, the invocation of join() causes the main thread to wait until the execution of the daemon thread has finished. Hence calling isAlive() on the thread instance reveals that the daemon thread is no longer running.

What happens when an uncaught exception leaves the run() method?

Answer:- It can happen that an unchecked exception escapes from the run() method. In this case the thread is stopped by the Java Virtual Machine. It is possible to catch this exception by registering an instance that implements the interface UncaughtException Handler as an exception handler.
This is either done by invoking the static method **Thread.setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler),** which tells the JVM to use the provided handler in case there was no specific handler registered on the thread itself, or by invoking **setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler)** on the thread instance itself.

What is a shutdown hook?

Answer:- A shutdown hook is a thread that gets executed when the JVM shuts down. It can be registered by invoking **addShutdownHook(Runnable)** on the Runtime instance:-

```
Runtime.getRuntime().addShutdownHook(new Thread() {
@Override
public void run() {
}
});
```

For what purposes is the keyword synchronized used?

Answer:-When you have to implement exclusive access to a resource, like some static value or some file reference, the code that works with the exclusive resource can be embraced with a synchronized block:-

```
synchronized (SynchronizedCounter.class) {
counter++;
}
```

What intrinsic lock does a synchronized method acquire?

Answer:-A synchronized method acquires the intrinsic lock for that method's object and releases it when the method returns. Even if the method throws an exception, the intrinsic lock is released. Hence a synchronized method is equal to the following code:-

```
public void method() {
synchronized(this) {
...
}
}
```

Can a constructor be synchronized?

Answer:-No, a constructor cannot be synchronized. In fact using the keyword "synchronized" with a constructor is actually a syntax error. Remember that the "synchronized" keyword is used to prevent 2 or more threads from accessing a group of methods before one thread finishes execution in those methods.

Why a constructor can not be synchronized?

Answer:- Synchronizing a constructor does not make sense simply because only one thread needs to have access to a constructor. Only the thread that creates an object needs to have access to it while it is being constructed. In other words, there is no need to switch out of the constructor to another thread. Also, when the constructor is called, the object does not even exist yet.

Can primitive values be used for intrinsic locks?

Answer-No, primitive values cannot be used for intrinsic locks.

Are intrinsic locks reentrant?

Answer:- Yes, intrinsic locks can be accessed by the same thread again and again. Otherwise code that acquires a lock would have to pay attention that it does not accidentally tries to acquire a lock it has already acquired.

What do we understand by an atomic operation?

Answer:-An atomic operation is an operation that is either executed completely or not at all.

Is the statement c++ atomic?

Answer:- No, the incrementation of an integer variable consist of more than one operation. First we have to load the current value of c, increment it and then finally store the new value back. The current thread performing this incrementation may be interrupted in-between any of these three steps, hence this operation is not atomic.

What operations are atomic in Java?

Answer:- The Java language provides some basic operations that are atomic and that therefore can be used to make sure that concurrent threads always see the same value:

- Read and write operations to reference variables and primitive variables (except long and double)
- Read and write operations for all variables declared as volatile.

Is the following implementation thread-safe?

```
public class DoubleCheckedSingleton {
private DoubleCheckedSingleton instance = null;
public DoubleCheckedSingleton getInstance() {
if(instance == null) {
synchronized (DoubleCheckedSingleton.class) {
if(instance == null) {
instance = new DoubleCheckedSingleton();
}
}
}
return instance;
}}
```

Answer:- The code above is not thread-safe. Although it checks the value of instance once again within the synchronized block (for performance reasons), the JIT compiler can rearrange the bytecode in a way that the reference to instance is set before the constructor has finished its execution. This means the method getInstance() returns an object that may not have been initialized completely. To make the code thread-safe, the keyword volatile can be used since Java 5 for the instance variable. Variables that are marked as volatile get only visible to other threads once the constructor of the object has finished its execution completely.

What do we understand by a deadlock?

Answer:- A deadlock is a situation in which two (or more) threads are each waiting on the other thread to free a resource that it has locked, while the thread itself has locked a resource the other thread is waiting on: Thread 1: locks resource A, waits for resource B Thread 2: locks resource B, waits for resource A.

What are the requirements for a deadlock situation?

Answer:-In general the following requirements for a deadlock can be identified:-

- **Mutual exclusion:** There is a resource which can be accessed only by one thread at any point in time.
- **Resource holding:** While having locked one resource, the thread tries to acquire another lock on some other exclusive resource.
- **No preemption:** There is no mechanism, which frees the resource if one thread holds the lock for a specific period of time.
- **Circular wait:** During runtime a constellation occurs in which two (or more) threads are each waiting on the other thread to free a resource that it has locked.

Is it possible to prevent deadlocks at all?

Answer:- In order to prevent deadlocks one (or more) of the requirements for a deadlock has to be eliminated:-

- **Mutual exclusion:** In some situation it is possible to prevent mutual exclusion by using *optimistic locking*.
- **Resource holding:** A thread may release all its exclusive locks, when it does not succeed in obtaining all exclusive locks.
- **No preemption:** Using a timeout for an exclusive lock frees the lock after a given amount of time.
- **Circular wait:** When all exclusive locks are obtained by all threads in the same sequence, no circular wait occurs.

Is it possible to implement a deadlock detection?

Answer:-When all exclusive locks are monitored and modelled as a directed graph, a deadlock detection system can search for two threads that are each waiting on the other thread to free a resource that it has locked. The waiting threads can then be forced by some kind of exception to release the lock the other thread is waiting on.

What is Optimistic Locking?

Answer:- Optimistic Locking is a strategy where we read a record, take note of a version number (other methods to do this involve dates, timestamps or checksums/hashes) and check that the version hasn't changed before you write the record back. When you write the record back you filter the update on the version to make sure it's atomic. (i.e. hasn't been updated between when you check the version and write the record to the disk) and update the version in one hit.f the record is dirty (i.e. different version to yours) you abort the transaction and the user can re-start it.

This strategy is most applicable to high-volume systems and three-tier architectures where you do not necessarily maintain a connection to the database for your session. In this situation the client cannot actually maintain database locks as the connections are taken from a pool and you may not be using the same connection from one access to the next.

What is Pessimistic Locking?

Answer:- Pessimistic Locking is when you lock the record for your exclusive use until you have finished with it. It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid Deadlocks. To use pessimistic locking you need either a direct connection to the database (as would typically be the case in a two tier client server application) or an externally available transaction ID that can be used independently of the connection.

In the latter case you open the transaction with the TxID and then reconnect using that ID. The DBMS maintains the locks and allows you to pick the session back up through the TxID. This is how distributed transactions using two-phase commit protocols (such as XA or COM+ Transactions) work.

What is a livelock?

Answer:- A livelock is a situation in which two or more threads block each other by responding to an action that is caused by another thread.
In contrast to a deadlock situation, where two or more threads wait in one specific state, the threads that participate in a livelock change their state in a way that prevents progress on their regular work.
An example would be a situation in which two threads try to acquire two locks, but release a lock they have acquired when they cannot acquire the second lock. It may now happen that both threads concurrently try to acquire the first thread. As only one thread succeeds, the second thread may succeed in acquiring the second lock. Now both threads hold two different locks, but as both want to have both locks, they release their lock and try again from the beginning. This situation may now happen again and again.

What do we understand by thread starvation?

Answer:-Threads with lower priority get less time for execution than threads with higher priority. When the threads with lower priority performs a long enduring computations, it may happen that these threads do not get enough time to finish their computations just in time. They seem to "starve" away as threads with higher priority steal them their computation time.

Can a synchronized block cause thread starvation?

Answer:- The order in which threads can enter a synchronized block is not defined. So in theory it may happen that in case many threads are waiting for the entrance to a synchronized block, some threads have to wait longer than other threads. Hence they do not get enough computation time to finish their work in time.

What do we understand by the term race condition?

Answer:-A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between

threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

A simple example for a race condition is the incrementation of an integer variable by two concurrent threads. As the operation consists of more than one single and atomic operation, it may happen that both threads read and increment the same value. After this concurrent incrementation the amount of the integer variable is not increased by two but only by one.

What do we understand by fair locks?

Answer:- A fair lock takes the waiting time of the threads into account when choosing the next thread that passes the barrier to some exclusive resource. An example implementation of a fair lock is provided by the Java SDK: **java.util.concurrent. locks.ReentrantLock**. If the constructor with the boolean flag set to true is used, the ReentrantLock grants access to the longest-waiting thread.

Which two methods that each object inherits from java.lang.Object can be used to implement a simple producer/consumer scenario?

Answer:-When a worker thread has finished its current task and the queue for new tasks is empty, it can free the processor by acquiring an intrinsic lock on the queue object and by calling the method **wait()**. The thread will be woken up by some producer thread that has put a new task into the queue and that again acquires the same intrinsic lock on the queue object and calls **notify()** on it.

What is the difference between notify() and notifyAll()?

Answer:- Both methods are used to wake up one or more threads that have put themselves to sleep by calling wait(). While notify() only wakes up one of the waiting threads, notifyAll() wakes up all waiting threads.

How it is determined which thread wakes up by calling notify()?

Answer:-It is not specified which threads will be woken up by calling notify() if more than one thread is waiting. Hence code should not rely on any concrete JVM implementation.

Is the following code that retrieves an integer value from some queue implementation correct?

```
public Integer getNextInt() {
Integer retVal = null;
synchronized (queue) {
try {
while (queue.isEmpty()) {
queue.wait();
}
```

```
} catch (InterruptedException e) {
e.printStackTrace();
}
}
synchronized (queue) {
retVal = queue.poll();
if (retVal == null) {
System.err.println("retVal is null");
throw new IllegalStateException();
}
}
return retVal;
}
```

Answer:-Although the code above uses the queue as object monitor, it does not behave correctly in a multithreaded environment. The reason for this is that it has two separate synchronized blocks. When two threads are woken up in line 6 by another thread that calls notifyAll(), both threads enter one after the other the second synchronized block. It this second block the queue has now only one new value, hence the second thread will poll on an empty queue and get null as return value.

Is it possible to check whether a thread holds a monitor lock on some given object?

Answer:- The class java.lang.Thread provides the static method **Thread.holdsLock(Object)** that returns true if and only if the current thread holds the lock on the object given as argument to the method invocation.

What does the method Thread.yield() do?

Answer:- An invocation of the static method Thread.yield() gives the scheduler a hint that the current thread is willing to free the processor. The scheduler is free to ignore this hint. As it is not defined which thread will get the processor after the invocation of Thread.yield(), it may even happen that the current thread becomes the "next" thread to be executed.

What do you have to consider when passing object instances from one thread to another?

Answer:- When passing objects between threads, we will have to pay attention that these objects are not manipulated by two threads at the same time. An example would be a Map implementation whose key/value pairs are modified by two concurrent threads. In order to avoid problems with concurrent modifications you can design an object to be immutable.

Which rules do you have to follow in order to implement an immutable class?

Answer:-

- All fields should be final and private.
- There should be not setter methods.
- The class itself should be declared final in order to prevent subclasses to violate the principle of immutability.
- If fields are not of a primitive type but a reference to another object:-
    - There should not be a getter method that exposes the reference directly to the caller.
    - Don't change the referenced objects (or at least changing these references is not visible to clients of the object).

What is the purpose of the class java.lang.ThreadLocal?

Answer:- As memory is shared between different threads, ThreadLocal provides a way to store and retrieve values for each thread separately. Implementations of ThreadLocal store and retrieve the values for each thread independently such that when thread A stores the value A1 and thread B stores the value B1 in the same instance of ThreadLocal, thread A later on retrieves value A1 from this ThreadLocal instance and thread B retrieves value B1.

What are possible use cases for java.lang.ThreadLocal?

Answer:- Instances of ThreadLocal can be used to transport information throughout the application without the need to pass this from method to method. Examples would be the transportation of security/login information within an instance of ThreadLocal such that it is accessible by each method. Another use case would be to transport transaction information or in general objects that should be accessible in all methods without passing them from method to method.

Is it possible to improve the performance of an application by the usage of multithreading? Name some examples.

Answer:- If we have more than one CPU core available, the performance of an application can be improved by multi-threading if it is possible to parallelize the computations over the available CPU cores. An example would be an application that should scale all images that are stored within a local directory structure. Instead of iterating over all images one after the other, a producer/consumer implementation can use a single thread to scan the directory structure and a bunch of worker threads that perform the actual scaling operation. Another example would be an application that mirrors some web page. Instead of loading one HTML page after the other, a producer thread can parse the first HTML page and issue the links it found into a queue. The worker threads monitor the queue and load the web pages found by the parser. While the worker threads wait for the page to get loaded completely, other threads can use the CPU to parse the already loaded pages and issue new requests.

What do we understand by the term scalability?

Answer:- Scalability means the ability of a program to improve the performance by adding further resources to it.

Is it possible to compute the theoretical maximum speed up for an application by using multiple processors?

Answer:- Amdahl's law provides a formula to compute the theoretical maximum speedup by providing multiple processors to an application. The theoretical speedup is computed by S(n) =1 /(B + (1-B)/n) where n denotes the number of processors and B the fraction of the program that cannot be executed in parallel. When n converges against infinity, the term (1-B)/n converges against zero. Hence the formula can be reduced in this special case to 1/B. As we can see, the theoretical maximum speedup behaves reciprocal to the fraction that has to be executed serially. This means the lower this fraction is, the more theoretical speedup can be achieved.

What do we understand by lock contention?

Answer:- Lock contention occurs, when two or more threads are competing in the acquisition of a lock. The scheduler has to decide whether it lets the thread, which has to wait sleeping and performs a context switch to let another thread occupy the CPU, or if letting the waiting thread busy-waiting is more efficient. Both ways introduce idle time to the inferior thread.

Which techniques help to reduce lock contention?

Answer:- In some cases lock contention can be reduced by applying one of the following techniques:

- The scope of the lock is reduced.
- The number of times a certain lock is acquired is reduced (lock splitting).
- Using hardware supported optimistic locking operations instead of synchronization.
- Avoid synchronization where possible.
- Avoid object pooling.

Which technique to reduce lock contention can be applied to the following code?

```
synchronized (map) {
UUID randomUUID = UUID.randomUUID();
Integer value = Integer.valueOf(42);
String key = randomUUID.toString();
map.put(key, value);
}
```

Answer:- The code above performs the computation of the random UUID and the conversion of the literal 42 into an Integer object within the synchronized block, although these two lines of code are local to the current thread and do not affect other threads. Hence they can be moved out of the synchronized block:

```
UUID randomUUID = UUID.randomUUID();
```

```
Integer value = Integer.valueOf(42);
String key = randomUUID.toString();
synchronized (map) {
map.put(key, value);
}
```

Explain by an example the technique lock splitting.

Answer:- Lock splitting may be a way to reduce lock contention when one lock is used to synchronize access to different aspects of the same application. Suppose we have a class that implements the computation of some statistical data of our application. A first version of this class uses the keyword synchronized in each method signature in order to guard the internal state before corruption by multiple concurrent threads. This also means that each method invocation may cause lock contention as other threads may try to acquire the same lock simultaneously. But it may be possible to split the lock on the object instance into a few smaller locks for each type of statistical data within each method. Hence thread T1 that tries to increment the statistical data D1 does not have to wait for the lock while thread T2 simultaneously updates the data D2.

What kind of technique for reducing lock contention is used by the SDK class ReadWriteLock?

Answer:- The SDK class **ReadWriteLock** uses the fact that concurrent threads do not have to acquire a lock when they want to read a value when no other thread tries to update the value. This is implemented by a pair of locks, one for read-only operations and one for writing operations. While the read-only lock may be obtained by more than one thread, the implementation guarantees that all read operation see an updated value once the write lock is released.

What do we understand by lock striping?

Answer:- In contrast to lock splitting, where we introduce different locks for different aspects of the application, lock striping uses multiple locks to guard different parts of the same data structure. An example for this technique is the class **ConcurrentHashMap** from JDK's **java.util.concurrent package**. The Map implementation uses internally different buckets to store its values. The bucket is chosen by the value's key. ConcurrentHashMap now uses different locks to guard different hash buckets. Hence one thread that tries to access the first hash bucket can acquire the lock for this bucket, while another thread can simultaneously access a second bucket. In contrast to a synchronized version of HashMap this technique can increase the performance when different threads work on different buckets.

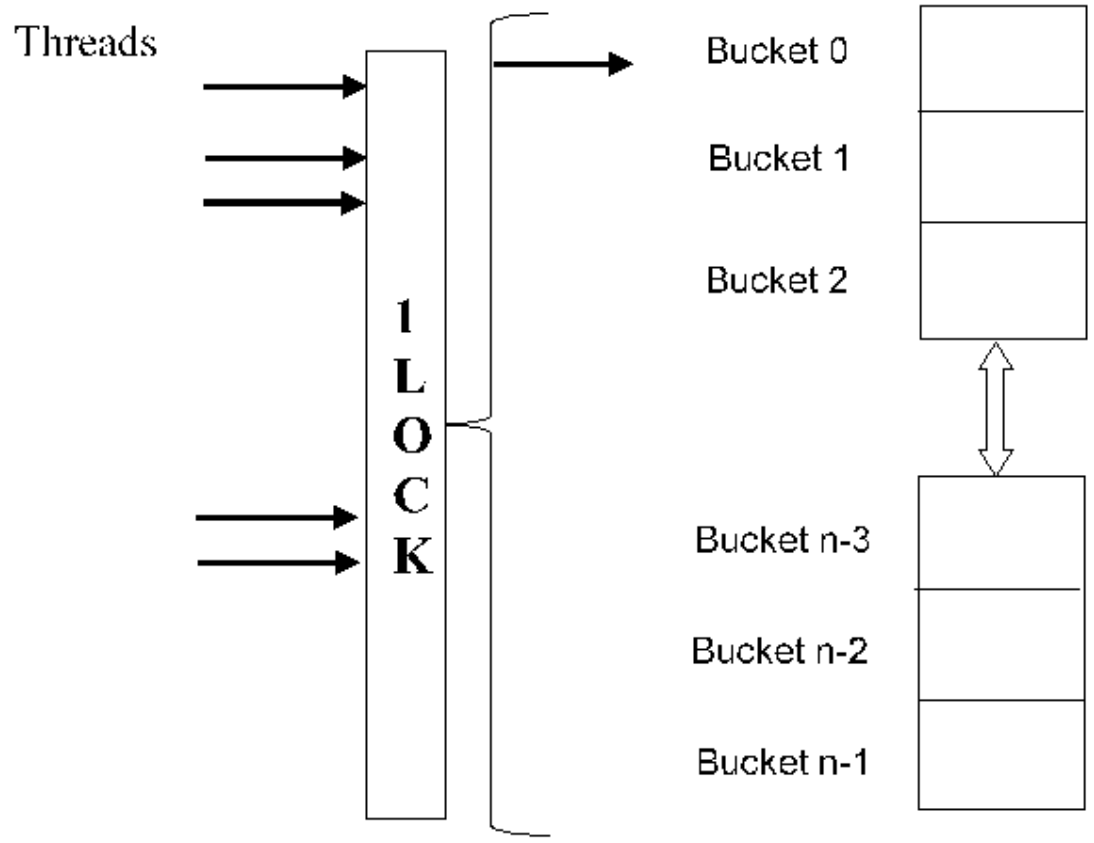
What is Lock Striping in Java Concurrency?

Answer:- In case you have only one lock for the whole data structure like Array or Map and you are synchronizing it and using it in a multi-threaded environment. Then, effectively any given

time only one thread can manipulate the map, as there is only a single lock. All the other threads would be waiting to get the monitor.

If you have to visualize a HashTable or a synchronized HashMap it can be depicted like the following image.

If there are 6 threads only one can get the lock and enter the synchronized collection. Even if the keys these threads want to get (or manipulate the values) are in different buckets as in the image if two threads want to access keys in bucket 0, two threads want to access keys in bucket 1 and two threads want to access keys in bucket n-3, any given time only one thread will get access.



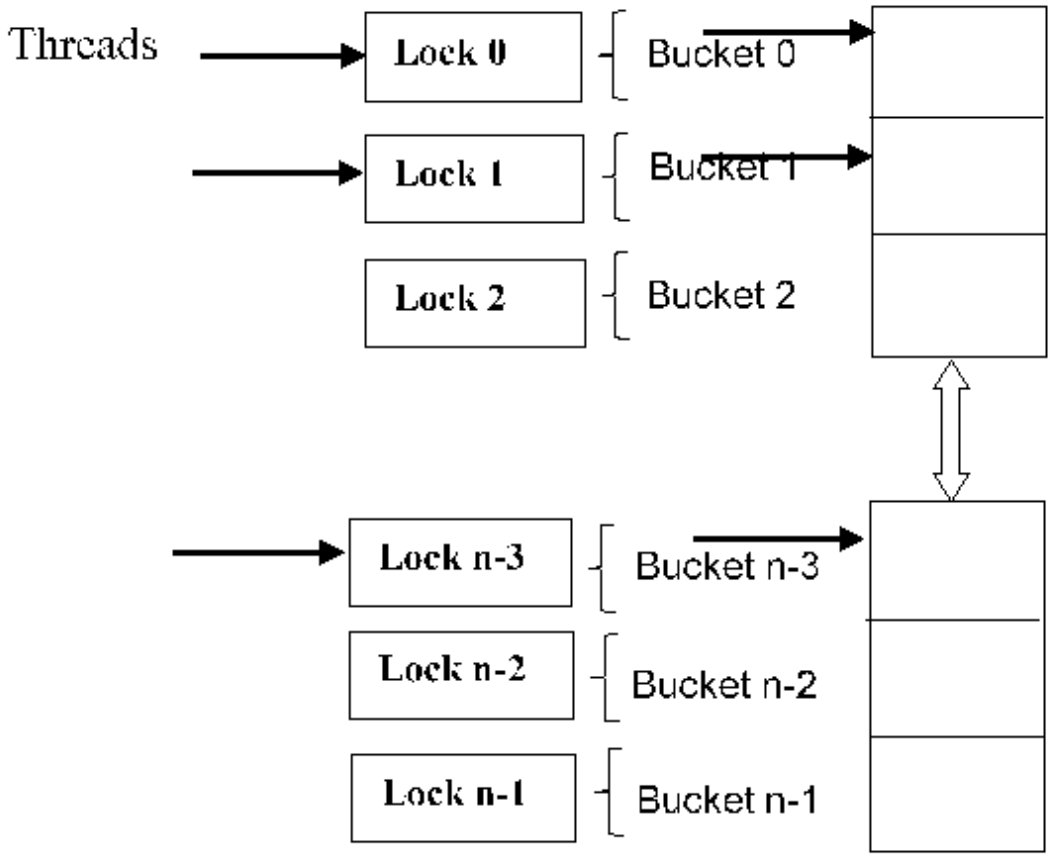
As expected this seriously degrades the performance, now think if there can be separate locks for separate buckets. Then the thread contention won't be at the whole data structure level but at the bucket level. That's the concept of lock striping. Having separate locks for a portion of a data structure where each lock is locking on a variable sized set of independent objects.

That's how ConcurrentHashMap in Java provides synchronization. By default ConcurrentHashMap has 16 buckets and each bucket has its own lock so there are 16 locks

too. So the threads which are accessing keys in separate buckets can access them simultaneously.

If you have to visualize it then following image would give you an idea how lock striping for a ConcurrentHashMap will look like.

Here two threads want to access keys in bucket 0 so one of them can enter, again two threads want to access keys in bucket 1 so one of them can enter. Same with bucket n-3. So, with lock striping out of 6 3 threads can work on the data structure.



Drawback of lock striping:- One of the downside of lock striping as mentioned in "Java Concurrency in practice" is that it is more difficult and costly, if you need to lock the collection for exclusive access than with a single lock.

What do we understand by a CAS operation?

Answer:- CAS stands for **compare-and-swap** and means that the processor provides a separate instruction that updates the value of a register only if the provided value is equal to the current value. CAS operations can be used to avoid synchronization as the thread can try to update a value by providing its current value and the new value to the CAS operation. If another

thread has meanwhile updated the value, the thread's value is not equal to the current value and the update operation fails. The thread then reads the new value and tries again. That way the necessary synchronization is interchanged by an optimistic spin waiting.

Which Java classes use the CAS operation?

Answer:- The SDK classes in the package **java.util.concurrent.atomic** like **AtomicInteger or AtomicBoolean** use internally the CAS operation to implement concurrent incrementation.

```
public class CounterAtomic {
private AtomicLong counter = new AtomicLong();
public void increment() {
counter.incrementAndGet();
}
public long get() {
return counter.get();
}
}
```

Provide an example why performance improvements for single-threaded applications can cause performance degradation for multi-threaded applications.

Answer:- A prominent example for such optimizations is a List implementation that holds the number of elements as a separate variable. This improves the performance for single-threaded applications as the size() operation does not have to iterate over all elements but can return the current number of elements directly. Within a multi-threaded application the additional counter has to be guarded by a lock as multiple concurrent threads may insert elements into the list. This additional lock can cost performance when there are more updates to the list than invocations of the size() operation.

Is object pooling always a performance improvement for multi-threaded applications?

Answer:- Object pools that try to avoid the construction of new objects by pooling them can improve the performance of single-threaded applications as the cost for object creation is interchanged by requesting a new object from the pool. In multi-threaded applications such an object pool has to have synchronized access to the pool and the additional costs of lock contention may outweigh the saved costs of the additional construction and garbage collection of the new objects. Hence object pooling may not always improve the overall performance of a multi-threaded application.

Describe Executors Framework?

Answer:- The java.util.concurrent package defines three executor interfaces:

1. **Executor**, a simple interface that supports launching new tasks.

2. **ExecutorService**, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
3. **ScheduledExecutorService**, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

Describe Executor Interface?
Answer:- An object that executes submitted Runnable tasks. This interface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc. An Executor is normally used instead of explicitly creating threads. For example, rather than invoking new Thread(new(RunnableTask())).start() for each of a set of tasks, you might use:

```
Executor executor = anExecutor;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
...
```

However, the Executor interface does not strictly require that execution be asynchronous. In the simplest case, an executor can run the submitted task immediately in the caller's thread:

```
class DirectExecutor implements Executor {
  public void execute(Runnable r) {
    r.run();
  }
}
```

More typically, tasks are executed in some thread other than the caller's thread.

Method of Executor interface:- void execute(Runnable command)
Executes the given command at some time in the future. The command may execute in a new thread, in a pooled thread, or in the calling thread, at the discretion of the Executor implementation.

Describe ExecutorService Interface?
Answer:- The ExecutorService interface supplements execute with a similar, but more versatile submit method. Like execute, submit accepts Runnable objects, but also accepts Callable objects, which allow the task to return a value. The submit method returns a Future object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.

ExecutorService also provides methods for submitting large collections of Callable objects. Finally, ExecutorService provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle interrupts correctly.

Describe ScheduledExecutorService Interface?
Answer;- The ScheduledExecutorService interface supplements the methods of its parent ExecutorService with schedule, which executes a Runnable or Callable task after a specified

delay. In addition, the interface defines scheduleAtFixedRate and scheduleWithFixedDelay, which executes specified tasks repeatedly, at defined intervals.

What is the relation between the two interfaces Executor and ExecutorService?

Answer:- The interface **Executor** only defines one method: **execute(Runnable)**. Implementations of this interface will have to execute the given Runnable instance at some time in the future.
The **ExecutorService** interface is an extension of the Executor interface and provides additional methods to shut down the underlying implementation, to await the termination of all submitted tasks and it allows submitting instances of Callable.

What happens when you submit() a new task to an ExecutorService instance whose queue is already full?

Answer:- As the method signature of submit() indicates, the ExecutorService implementation is supposed to throw a **RejectedExecutionException.**

What is a ScheduledExecutorService?

Answer:- The interface ScheduledExecutorService extends the interface ExecutorService and adds method that allow to submit new tasks to the underlying implementation that should be executed a given point in time. There are two methods to schedule one-shot tasks and two methods to create and execute periodic tasks.

Do you know an easy way to construct a thread pool with 5 threads that executes some tasks that return a value?

Answer:- The SDK provides a factory and utility class Executors whose static method **newFixedThreadPool(int nThreads)** allows the creation of a thread pool with a fixed number of threads (the implementation of MyCallable is omitted):-

```
public static void main(String[] args) throws InterruptedException, ExecutionException {
ExecutorService executorService = Executors.newFixedThreadPool(5);
Future<Integer>[] futures = new Future[5];
for (int i = 0; i < futures.length; i++) {
futures[i] = executorService.submit(new MyCallable());
}
for (int i = 0; i < futures.length; i++) {
Integer retVal = futures[i].get();
System.out.println(retVal);
}
```

```
executorService.shutdown();
}
```

Describe Lock Interface?

Answer:- A java.util.concurrent.locks.Lock is a thread synchronization mechanism just like synchronized blocks. A Lock is, however, more flexible and more sophisticated than a synchronized block. Since Lock is an interface, you need to use one of its implementations to use a Lock in your applications. ReentrantLock is one such implementation of Lock interface.

Difference between Lock Interface and synchronized keyword?

Answer:- The main differences between a Lock and a synchronized block are:-
Having a timeout trying to get access to a synchronized block is not possible. Using Lock.tryLock(long timeout, TimeUnit timeUnit), it is possible.
The synchronized block must be fully contained within a single method. A Lock can have it's calls to lock() and unlock() in separate methods.

Describe Object level locking?

Answer:- Object level locking is mechanism when you want to synchronize a non-static method or 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.

Describe Class level locking?

Answer:- Class level locking prevents multiple threads to enter in synchronized block in any of all available instances on runtime. This means if in runtime there are 100 instances of DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads. This should always be done to make static data thread safe.

Important notes on Synchronization :-

1. Synchronization in java guarantees that no two threads can execute a synchronized method which requires same lock simultaneously or concurrently.
2. synchronized keyword can be used only with methods and code blocks. These methods or blocks can be static or non-static both.
3. When ever a thread enters into java synchronized method or block it acquires a lock and whenever it leaves java synchronized method or block it releases the lock. Lock is released even if thread leaves synchronized method after completion or due to any Error or Exception.

4. java synchronized keyword is re-entrant in nature it means if a java synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.
5. Java Synchronization will throw NullPointerException if object used in java synchronized block is null. For example, in above code sample if lock is initialized as null, the synchronized (lock) will throw NullPointerException.
6. Synchronized methods in Java put a performance cost on your application. So use synchronization when it is absolutely required. Also, consider using synchronized code blocks for synchronizing only critical section of your code.
7. It's possible that both static synchronized and non static synchronized method can run simultaneously or concurrently because they lock on different object.
8. According to the Java language specification you can not use java synchronized keyword with constructor it's illegal and result in compilation error.
9. Do not synchronize on non final field on synchronized block in Java. because reference of non final field may change any time and then different thread might synchronizing on different objects i.e. no synchronization at all. Best is to use String class, which is already immutable and declared final.

What is the difference between the two interfaces Runnable and Callable?

Answer:- The interface Runnable defines the method run() without any return value whereas the interface Callable allows the method run() to return a value and to throw an exception.

Which are use cases for the class java.util.concurrent.Future?

Answer:- Instances of the class java.util.concurrent.Future are used to represent results of asynchronous computations whose result are not immediately available. Hence the class provides methods to check if the asynchronous computation has finished, canceling the task and to the retrieve the actual result. The latter can be done with the two get() methods provided. The first
get() methods takes no parameter and blocks until the result is available whereas the second get() method takes a timeout parameter that lets the method invocation return if the result does not get available within the given timeframe.

What is the difference between HashMap and Hashtable particularly with regard to thread-safety?

Answer:- The methods of Hashtable are all synchronized. This is not the case for the HashMap implementation. Hence Hashtable is thread-safe whereas HashMap is not thread-safe. For single-threaded applications it is therefore more efficient to use the "newer" HashMap implementation.

Is there a simple way to create a synchronized instance of an arbitrary implementation of Collection, List or Map?

Answer:- The utility class Collections provides the methods synchronizedCollection(Collection), synchronizedList(List) and synchronizedMap(Map) that return a thread-safe collection/list/map that is backed by the given instance.

What is a semaphore?

Answer:- A semaphore is a data structure that maintains a set of permits that have to be acquired by competing threads. Semaphores can therefore be used to control how many threads access a critical section or resource simultaneously. Hence the constructor of java.util.concurrent.Semaphore takes as first parameter the number of permits the threads compete about. Each invocation of its acquire() methods tries to obtain one of the available permits. The method acquire() without any parameter blocks until the next permit gets available. Later on, when the thread has finished its work on the critical resource, it can release the permit by invoking the method release() on an instance of Semaphore.

What is a CountDownLatch?

Answer:- The SDK class CountDownLatch provides a synchronization aid that can be used to implement scenarios in which threads have to wait until some other threads have reached the same state such that all thread can start. This is done by providing a synchronized counter that is decremented until it reaches the value zero. Having reached zero the CountDownLatch instance
lets all threads proceed. This can be either used to let all threads start at a given point in time by using the value 1 for the counter or to wait until a number of threads has finished. In the latter case the counter is initialized with the number of threads and each thread that has finished its work counts the latch down by one.

What is the difference between a CountDownLatch and a CyclicBarrier?

Answer:- Both SDK classes maintain internally a counter that is decremented by different threads. The threads wait until the internal counter reaches the value zero and proceed from there on. But in contrast to the CountDownLatch the class CyclicBarrier resets the internal value back to the initial value once the value reaches zero. As the name indicates instances of CyclicBarrier can therefore be used to implement use cases where threads have to wait on each other again and again.

What kind of tasks can be solved by using the Fork/Join framework?

Answer:- The base class of the Fork/Join Framework java.util.concurrent.ForkJoinPool is basically a thread pool that executes instances of java.util.concurrent.ForkJoinTask. The class ForkJoinTask provides the two methods fork() and join(). While fork() is used to start the asynchronous execution of the task, the method join() is used to await the result of the computation. Hence the Fork/Join framework can be used to implement divide-and-conquer algorithms where a more complex problem is divided into a number of smaller and easier to solve problems.

Is it possible to find the smallest number within an array of numbers using the Fork/Join-Framework?

Answer:- The problem of finding the smallest number within an array of numbers can be solved by using a divide-and-conquer algorithm. The smallest problem that can be solved very easily is an array of two numbers as we can determine the smaller of the two numbers directly by one comparison. Using a divide-and-conquer approach the initial array is divided into two parts of equal length and both parts are provided to two instances of RecursiveTask that extend the class ForkJoinTask. By forking the two tasks they get executed and either solve the problem directly, if their slice of the array has the length two, or they again recursively divide the array into two parts and fork two new **RecursiveTasks**. Finally each task instance returns its result (either by having it computed directly or by waiting for the two subtasks). The root tasks then returns the smallest number in the array.

What is the difference between the two classes RecursiveTask and RecursiveAction?

Answer:- In contrast to RecursiveTask the method compute() of RecursiveAction does not have to return a value. Hence RecursiveAction can be used when the action works directly on some data structure without having to return the computed value.

Is it possible to perform stream operations in Java 8 with a thread pool?

Answer:- Collections provide the method **parallelStream()** to create a stream that is processed by a thread pool. Alternatively you can call the intermediate method parallel() on a given stream to convert a sequential stream to a parallel counterpart.

How can we access the thread pool that is used by parallel stream operations?

Answer:- The thread pool used for parallel stream operations can be accessed by **ForkJoinPool.commonPool().**

This way we can query its level of parallelism with **commonPool.getParallelism().**

The level cannot be changed at runtime but it can be configured by providing the following JVM parameter:- **-Djava.util.concurrent.ForkJoinPool.common.parallelism=5.**

When to use Runnable vs Thread in Java? Or Difference between Thread vs Runnable interface in Java?

Answer:-

- Java doesn't support multiple inheritance, which means you can only extend one class in Java so once you extended Thread class you lost your chance and can not extend or inherit another class in Java.
- In Object oriented programming extending a class generally means adding new functionality, modifying or improving behaviors. If we are not making any modification on Thread then use Runnable interface instead.

- Runnable interface represent a Task which can be executed by either plain Thread or Executors or any other means. so logical separation of Task as Runnable than Thread is good design decision.
- Separating task as Runnable means we can reuse the task and also has liberty to execute it from different means. since you can not restart a Thread once it completes. again Runnable vs Thread for task, Runnable is winner.
- Java designer recognizes this and that's why Executors accept Runnable as Task and they have worker thread which executes those task.
- Inheriting all Thread methods are additional overhead just for representing a Task which can can be done easily with Runnable.
- When you extends Thread class, each of your thread creates unique object and associate with it.
- When you implements Runnable, it shares the same object to multiple threads.
- In "implements Runnable" , we are creating a different Runnable class for a specific behavior job (if the work you want to be done is job). It gives us the freedom to reuse the specific behavior job whenever required. "extends Thread" contains both thread and job specific behavior code. Hence once thread completes execution , it can not be restart again.
- Implementing Runnable should be preferred . It does not specializing or modifying the thread behavior . You are giving thread something to run. We conclude that Composition is the better way. Composition means two objects A and B satisfies has-a relationship. "extends Thread" is not a good Object Oriented practice.
- You usually extend a class to add or modify functionality. So, if you don't want to overwrite any Thread behavior, then use Runnable. In the same light, if you don't need to inherit thread methods, you can do without that overhead by using Runnable.
- "implements Runnable" makes the code loosely-coupled and easier to read . Because the code is split into two classes . Thread class for the thread specific code and your Runnable implementation class for your job that should be run by a thread code. "extends Thread" makes the code tightly coupled . Single class contains the thread code as well as the job that needs to be done by the thread.
- Functions overhead : "extends Thread" means inheriting all the functions of the Thread class which we may do not need . job can be done easily by Runnable without the Thread class functions overhead.

Recap : Difference between "implements Runnable" and "extends Thread"

	implements Runnable	extends Thread
Inheritance option	extends any java class	No
Reusability	Yes	No
Object Oriented Design	Good,allows composition	Bad

Loosely Coupled	Yes	No
Function Overhead	No	Yes

The following example helps you to understand more clearly.

ThreadVsRunnable.java

```
class ImplementsRunnable implements Runnable {

 private int counter = 0;

 public void run() {
 counter++;
 System.out.println("ImplementsRunnable : Counter : " + counter);
 }
 }

 class ExtendsThread extends Thread {

 private int counter = 0;

 public void run() {
 counter++;
 System.out.println("ExtendsThread : Counter : " + counter);
 }
 }

 public class ThreadVsRunnable {

 public static void main(String args[]) throws Exception {
 //Multiple threads share the same object.
 ImplementsRunnable rc = new ImplementsRunnable();
 Thread t1 = new Thread(rc);
 t1.start();
 Thread.sleep(1000); // Waiting for 1 second before starting next thread
 Thread t2 = new Thread(rc);
 t2.start();
 Thread.sleep(1000); // Waiting for 1 second before starting next thread
 Thread t3 = new Thread(rc);
 t3.start();

 //Creating new instance for every thread access.
 ExtendsThread tc1 = new ExtendsThread();
 tc1.start();
 Thread.sleep(1000); // Waiting for 1 second before starting next thread
 ExtendsThread tc2 = new ExtendsThread();
 tc2.start();
 Thread.sleep(1000); // Waiting for 1 second before starting next thread
 ExtendsThread tc3 = new ExtendsThread();
 tc3.start();
 }
 }
```

Output of the above program.

```
ImplementsRunnable : Counter : 1
```

```
ImplementsRunnable : Counter : 2
ImplementsRunnable : Counter : 3
ExtendsThread : Counter : 1
ExtendsThread : Counter : 1
ExtendsThread : Counter : 1
```

Descriptions:- In the Runnable interface approach, only one instance of a class is being created and it has been shared by different threads. So the value of counter is incremented for each and every thread access.
Whereas, Thread class approach, you must have to create separate instance for every thread access. Hence different memory is allocated for every class instances and each has separate counter, the value remains same, which means no increment will happen because none of the object reference is same.

When to use "extends Thread" over "implements Runnable"?

Answer:- The only time it make sense to use "extends Thread" is when you have a more specialized version of Thread class. In other words , because you have more specialized thread specific behavior. But the conditions are that the thread work you want is really just a job to be done by a thread. In that case you need to use "implements Runnable" which also leaves your class free to extend some other class.

When to use Runnable?

Answer:- Use Runnable interface when you want to access the same resource from the group of threads. Avoid using Thread class here, because multiple objects creation consumes more memory and it becomes a big performance overhead.

Which one is best to use?

Answer:-Very simple, based on your application requirements you will use this appropriately. But I would suggest, try to use interface inheritance i.e., implements Runnable.

Note:- object oriented designs have some guidelines for better coding.-Coding to an interface rather than to implementation. This makes your software/application easier to extend. In other words, your code will work with all the interface's subclasses, even ones that have not been created yet. Interface inheritance (implements) is preferable – This makes your code is loosely coupling between classes/objects.(Note : Thread class internally implements the Runnable interface).

Difference between start() and run() method of Thread class?

Answer:- Main difference is that when program calls start() method a new Thread is created and code inside run() method is executed in new Thread while if you call run() method directly no new Thread is created and code inside run() will execute on current Thread.

Another difference between start vs run in Java thread is that you can not call start() method twice on thread object. once started, second call of start() will throw IllegalStateException in Java while you can call run() method twice.
In Summary only difference between start() and run() method in Thread is that start creates new thread while run doesn't create any thread and simply execute in current thread like a normal method call.

Why it's only proper to call the start() method to start the thread instead of calling the run() method directly? Or why not directly call run method?

Answer:- You should not invoke the run() method directly. If you call the run() method directly, it will simply execute in the caller's thread instead of as its own thread. Instead, you need to call the start() method, which schedules the thread with the JVM. The JVM will call the corresponding run() method when the resources and CPU is ready. The JVM is not guaranteed to call the run() method right way when the start() method is called, or in the order that the start() methods are called. Especially for the computers have a single processor, it is impossible to run all running threads at the same time. The JVM must implement a scheduling scheme that shares the processor among all running threads. This is why when you call the start() methods from more than one thread, the sequence of execution of the corresponding run() methods is random, controlled only by the JVM.

Can the run() method be called directly to start a thread?

Answer:- No.

Why do we need start() method in Thread class? In Java API description for Thread class is written : "Java Virtual Machine calls the run method of this thread..". Couldn't we call method run() ourselves, without doing double call: first we call start() method which calls run() method? What is a meaning to do things such complicate?

Answer:- There is some very small but important difference between using *start()* and *run()* methods. Look at two examples below:

Example one:-

```
Thread one = new Thread();
Thread two = new Thread();
one.run();
two.run();
```

Example two:-

```
Thread one = new Thread();
Thread two = new Thread();
one.start();
```

```
two.start();
```

The result of running examples will be different.

- In *Example one* the threads will run sequentially: first, thread number one runs, when it exits the thread number two starts.
- In *Example two* both threads start and run simultaneously.

Conclusion:- the *start()* method call *run()* method asynchronously (does not wait for any result, just fire up an action), while we run *run()* method synchronously - we wait when it quits and only then we can run the next line of our code.

What will happen if we don't override run method of thread?

Answer:- When we call start() method on thread, it internally calls run() method with newly created thread. So, if we don't override run() method newly created thread won't be called and nothing will happen.

What will happen if we override start method of thread?

Answer:- When we call start() method on thread, it internally calls run() method with newly created thread. So, if we override start() method, run() method will not be called until we write code for calling run() method.

Can we start Thread again in java?

Answer:- No, we cannot start Thread again, doing so will throw runtimeException **java.lang.IllegalThreadStateException**. The reason is once run() method is executed by Thread, it goes into dead state.

Difference between Runnable and Callable in Java?

Answer:- Both Runnable and Callable represent task which is intended to be executed in separate thread. Runnable is there from JDK 1.0, while Callable was added on JDK 1.5. Main difference between these two is that Callable's call() method can return value and throw Exception, which was not possible with Runnable's run() method. Callable return Future object, which can hold result of computation. Callable interface is a generic parameterized interface and Type of value is provided, when instanceof Callable implementation is created.

Difference between call() and run() method in Java.

Answer:- Runnable interface is older than Callable, there from JDK 1.0, while Callable is added on Java 5.0. Runnable interface has run() method to define task while Callable interface uses call() method for task definition. run() method does not return any value, it's return type is void while call method returns value. Callable interface is a generic parameterized interface and Type of value is provided, when instanceof Callable implementation is created. Another difference on run and call method is that run method can not throw checked exception, while call method can throw checked exception in Java.

Difference between CyclicBarrier and CountDownLatch in Java?

Answer:- Though both CyclicBarrier and CountDownLatch wait for number of threads on one or more events, main difference between them is that you can not re-use CountDownLatch once count reaches to zero, but you can reuse same CyclicBarrier by calling reset() method which resets Barrier to its initial State. What it implies that CountDownLatch is good for one time event like application startup time and CyclicBarrier can be used to in case of recurrent event e.g. concurrently calculating solution of big problem etc.

What is CyclicBarrier in Java

Answer:- CyclicBarrier in Java is a synchronizer introduced in JDK 5 on java.util.Concurrent package. CyclicBarrier allows multiple threads to wait for each other (barrier) before proceeding.CyclicBarrier is a natural requirement for concurrent program because it can be used to perform final part of task once individual tasks are completed. All threads which wait for each other to reach barrier are called parties, CyclicBarrier is initialized with number of parties to be wait and threads wait for each other by calling CyclicBarrier.await() method which is a blocking method in Java and blocks until all Thread or parties call await(). In general calling await() is shout out that Thread is waiting on barrier. await() is a blocking call but can be timed out or Interrupted by another thread.

Important point of CyclicBarrier in Java:-

- CyclicBarrier can perform a completion task once all thread reaches to barrier, This can be provided while creating CyclicBarrier.
- If CyclicBarrier is initialized with 3 parties means 3 thread needs to call await method to break the barrier.
- Thread will block on await() until all parties reaches to barrier, another thread interrupt or await timed out.

- If another thread interrupt the thread which is waiting on barrier it will throw BrokernBarrierException as shown below:

```
java.util.concurrent.BrokenBarrierException
        at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:172)
        at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:327)
```

- CyclicBarrier.reset() put Barrier on its initial state, other thread which is waiting or not yet reached barrier will terminate with java.util.concurrent.BrokenBarrierException.

When to use CyclicBarrier in Java

Answer:- Given the nature of CyclicBarrier it can be very handy to implement mapreduce kind of task similar to **fork-join framework of Java 7**, where a big task is broken down into smaller pieces and to complete the task you need output from individual small task e.g. to count population of India you can have 4 threads which counts population from North, South, East and West and once complete they can wait for each other, When last thread completed their task, Main thread or any other thread can add result from each zone and print total population.

You can use CyclicBarrier in Java :-

- To implement multi player game which can not begin until all player has joined.
- Perform lengthy calculation by breaking it into smaller individual tasks, In general to implement Map reduce technique.

What is CountDownLatch in Java

Answer:- CountDownLatch in Java is a kind of synchronizer which allows one Thread to wait for one or more Threads before starts processing. This is very crucial requirement and often needed in server side core Java application and having this functionality built-in as CountDownLatch greatly simplifies the development. CountDownLatch in Java is introduced on Java 5 in java.util.concurrent package.

When should we use CountDownLatch in Java :

Answer:- Use CountDownLatch when one of Thread like main thread, require to wait for one or more thread to complete, before its start doing processing. Classical example of using CountDownLatch in Java is any server side core Java application which uses services architecture, where multiple services is provided by multiple threads and application can not start processing until all services have started successfully.

Few points about Java CountDownLatch which is worth remembering:-

- You can not reuse CountDownLatch once count is reaches to zero, this is the main difference between CountDownLatch and CyclicBarrier, which is frequently asked in core Java interviews and multi-threading interviews.
- Main Thread wait on Latch by calling CountDownLatch.await() method while other thread calls CountDownLatch.countDown() to inform that they have completed.

What is Java Memory model?

Answer:- Java Memory model is set of rules and guidelines which allows Java programs to behave deterministically across multiple memory architecture, CPU, and operating system. It's particularly important in case of multi-threading. Java Memory Model provides some guarantee on which changes made by one thread should be visible to others, one of them is happens-before relationship. This relationship defines several rules which allows programmers to anticipate and reason behaviour of concurrent Java programs.
For example, happens-before relationship guarantees :-

- Each action in a thread happens-before every action in that thread that comes later in the program order, this is known as program order rule.
- An unlock on a monitor lock happens-before every subsequent lock on that same monitor lock, also known as Monitor lock rule.
- A write to a volatile field happens-before every subsequent read of that same field, known as Volatile variable rule.
- A call to Thread.start on a thread happens-before any other thread detects that thread has terminated, either by successfully return from Thread.join() or by Thread.isAlive() returning false, also known as Thread start rule.
- A thread calling interrupt on another thread happens-before the interrupted thread detects the interrupt(either by having InterruptedException thrown, or invoking isInterrupted or interrupted), popularly known as Thread Interruption rule.
- The end of a constructor for an object happens-before the start of the finalizer for that object, known as Finalizer rule.
- If A happens-before B, and B happens-before C, then A happens-before C, which means happens-before guarantees Transitivity.

What is volatile variable in Java?

Answer:- Volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from main memory. So if you want to share any variable in which read and write operation is atomic by implementation e.g. read and write in int or boolean variable you can declare them as volatile variable. Java volatile keyword cannot be used with method or class and it can only be used with variable. Java volatile keyword also guarantees visibility and ordering , after Java 5 write to any volatile variable happens before any read into volatile variable. By the way use of volatile keyword also prevents compiler or JVM from reordering of code or moving away them from synchronization barrier.

When to use Volatile variable in Java

Answer:- Here are couple of example to demonstrate when to use Volatile keyword in Java:-

- You can use Volatile variable if you want to read and write long and double variable atomically. long and double both are 64 bit data type and by default writing of long and double is not atomic and platform dependence. Many platform perform write in long and double variable 2 step, writing 32 bit in each step, due to this its possible for a Thread to see 32 bit from two different write. You can avoid this issue by making long and double variable volatile in Java.
- Volatile variable can be used as an alternative way of achieving synchronization in Java in some cases, like Visibility. with volatile variable its guaranteed that all reader thread will see updated value of volatile variable once write operation completed, without volatile keyword different reader thread may see different values.
- volatile variable can be used to inform compiler that a particular field is subject to be accessed by multiple threads, which will prevent compiler from doing any reordering or any kind of optimization which is not desirable in multi-threaded environment. Without volatile variable compiler can re-order code, free to cache value of volatile variable instead of always reading from main memory. like following example without volatile variable may result in infinite loop.

```
private boolean isActive = thread;
public void printMessage(){
  while(isActive){
     System.out.println("Thread is Active");
  }
```

```
}
```

Without volatile modifier it's not guaranteed that one Thread see the updated value of isActive from other thread. compiler is also free to cache value of isActive instead of reading it from main memory in every iteration. By making isActive a volatile variable you avoid these issue.

- Another place where volatile variable can be used is to fixing double checked locking in Singleton pattern.

Important points on Volatile keyword in Java

- volatile keyword in Java is only application to variable and using volatile keyword with class and method is illegal.
- volatile keyword in Java guarantees that value of volatile variable will always be read from main memory and not from Thread's local cache.
- In Java reads and writes are atomic for all variables declared using Java volatile keyword (including long and double variables).
- Using Volatile keyword in Java on variables reduces the risk of memory consistency errors, because any write to a volatile variable in Java establishes a happens-before relationship with subsequent reads of that same variable.
- From Java 5 changes to a volatile variable are always visible to other threads. What's more it also means that when a thread reads a volatile variable in Java, it sees not just the latest change to the volatile variable but also the side effects of the code that led up the change.
- Reads and writes are atomic for reference variables are for most primitive variables (all types except long and double) even without use of volatile keyword in Java.
- An access to a volatile variable in Java never has chance to block, since we are only doing a simple read or write, so unlike a synchronized block we will never hold on to any lock or wait for any lock.
- Java volatile variable that is an object reference may be null.
- Java volatile keyword doesn't means atomic, its common misconception that after declaring volatile ++ will be atomic, to make the operation atomic you still need to ensure exclusive access using synchronized method or block in Java.

- If a variable is not shared between multiple threads no need to use volatile keyword with that variable.

Difference between synchronized and volatile keyword in Java

Answer:- Remember volatile is not a replacement of synchronized keyword but can be used as an alternative in certain cases. Here are few differences between volatile and synchronized keyword in Java.

- Volatile keyword in Java is a field modifier, while synchronized modifies code blocks and methods.
- Synchronized obtains and releases lock on monitor's Java volatile keyword doesn't require that.
- Threads in Java can be blocked for waiting any monitor in case of synchronized, that is not the case with volatile keyword in Java.
- Synchronized method affects performance more than volatile keyword in Java.
- Since volatile keyword in Java only synchronizes the value of one variable between Thread memory and "main" memory while synchronized synchronizes the value of all variable between thread memory and "main" memory and locks and releases a monitor to boot. Due to this reason synchronized keyword in Java is likely to have more overhead than volatile.
- You can not synchronize on null object but our volatile variable in java could be null.
- From Java 5 Writing into a volatile field has the same memory effect as a monitor release, and reading from a volatile field has the same memory effect as a monitor acquire .

What is thread-safety? is Vector a thread-safe class?

Answer:- Thread-safety is a property of an object or code which guarantees that if executed or used by multiple thread in any manner e.g. read vs write it will behave as expected. For example, a thread-safe counter object will not miss any count if same instance of that counter is shared among multiple threads. Apparently, you can also divide collection classes in two category, thread-safe and non-thread-safe. Vector is indeed a thread-safe class and it achieves thread-safety by synchronizing methods which modifies state of Vector, on the other hand, its counterpart ArrayList is not thread-safe.

Here is some points worth remembering to write thread safe code in Java, these knowledge also helps you to avoid some serious concurrency issues in Java like race condition or deadlock in Java:-

- Immutable objects are by default thread-safe because there state can not be modified once created. Since String is immutable in Java, its inherently thread-safe.
- Read only or final variables in Java are also thread-safe in Java.
- Locking is one way of achieving thread-safety in Java.
- Static variables if not synchronized properly becomes major cause of thread-safety issues.
- Example of thread-safe class in Java: Vector, Hashtable, ConcurrentHashMap, String etc.
- Atomic operations in Java are thread-safe e.g. reading a 32 bit int from memory because its an atomic operation it can't interleave with other thread.
- local variables are also thread-safe because each thread has there own copy and using local variables is good way to writing thread-safe code in Java.
- In order to avoid thread-safety issue minimize sharing of objects between multiple thread.
- Volatile keyword in Java can also be used to instruct thread not to cache variables and read from main memory and can also instruct JVM not to reorder or optimize code from threading perspective.

What is race condition in Java? Given one example?

Answer:-In concurrent programming a Race Condition occurs when a second thread modifies the state of one (or more objects), making any assumptions, checks, made by the first threads invalid for example of Race Condition is shopping online. Say that you found an item you want to buy in an online store. The online store indicates that they only have one left. By the time you press buy, another user beats you to it and buys it after you browse the page (which page indicates one item left) but before you pressed buy. This is another example of Race Condition as the state changed while you were shopping without you knowing. Like before, for this condition to manifest itself, you need more than one users, purchasing the same item at the same time.

Race condition in Java occurs when two or more threads try to modify/update shared data at the same time.

A thread can prevent this from happening by locking an object. When an object is locked by one thread and another thread tries to call a synchronized method on the same object, the second thread will block until the object is unlocked.

How many types of race condition?

Answer:- There are two types of two types of race conditions:

1. Check-then-act
2. Read-modify-write.

Check-then-act race condition:- This race condition appears when you have a shared field and expect to serially execute the following steps:-

- Get a value from a field.
- Do something based on the result of the previous check.

The problem here is that when the first thread is going to act after the previous check, another thread may have interleaved and changed the value of the field. Now, the first thread will act based on a value that is no longer valid.

Read-modify-write race condition:- Here we have another type of race condition which appears when executing the following set of actions::-

- Fetch a value from a field.
- Modify the value.
- Store the new value to the field.

The problem here is that when the first thread is going to act after the previous check, another thread may have interleaved and changed the value of the field. Now, the first thread will act based on a value that is no longer valid.

How to avoid Race Condition in your Java Application?

Answer:-

- If the race condition is in updates to some shared in-memory data structures, you need to synchronize access and updates to the data structure appropriate way.
- If the race condition is in updates to the your database, you need to restructure your SQL to use transactions at the appropriate level of granularity.
- It’s not a bad idea to do Load testing before going live in production. More load may cause rare Race Condition. Better to fix it before rather fixing it later.

- Make sure you have no global variable that you write to.
- In Java, every object has one and only one monitor and mutex associated with it. The single monitor has several doors into it, however, each indicated by the synchronized keyword. When a thread passes over the synchronized keyword, it effectively locks all the doors.
- Of course, if a thread doesn't pass across the synchronized keyword, it hasn't locked the door, and some other thread is free barge in at any time.

What is Lock(), UnLock(), ReentrantLock(), TryLock() and How it's different from Synchronized Block in Java?

Answer:-

- Lock():- java.util.concurrent.locks. A lock is a thread synchronization mechanism like synchronized blocks except locks can be more sophisticated than Java's synchronized blocks. It is an interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors.
- UnLock():-UnLock() releases the lock on Object.
- ReentrantLock():-A ReentrantLock is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock.TryLock():-TryLock() acquires the lock only if it is free at the time of invocation.

What happens when an Exception occurs in a thread?

Answer:- If not caught thread will die, if an uncaught exception handler is registered then it will get a call back. Thread.UncaughtExceptionHandler is an interface, defined as nested interface for handlers invoked when a Thread abruptly terminates due to an uncaught exception. When a thread is about to terminate due to an uncaught exception the Java Virtual Machine will query the thread for its UncaughtExceptionHandler using Thread.getUncaughtExceptionHandler() and will invoke the handler's uncaughtException() method, passing the thread and the exception as arguments.

How do you share data between two thread in Java?

Answer:-You can share data between threads by using shared object, or concurrent data-structure like BlockingQueue.

How threads communicates?

Answer:- Wait and notify methods in Java are used for inter-thread communication i.e. if one thread wants to tell something to another thread, it uses notify() and notifyAll() method of java.lang.Object. Classical example of wait and notify method is Producer Consumer design pattern, where One thread produce and put something on shared bucket, and then tell other thread that there is an item for your interest in shared object, consumer thread then pick the item and do his job, without wait() and notify(), consumer thread needs to be busy checking, even if there is no change in state of shared object. This brings an interesting point on using wait and notify mechanism, a call to notify() happens, when thread changed state of shared object i.e. in this case producer change bucket from empty to not empty, and consumer change state from non empty to empty. Also wait and notify method must be called from synchronized context. Another important thing to keep in mind while calling them is, using loop to check conditions instead of if block. In short, a waiting thread may woke up, without any change in it's waiting condition due to spurious wake up. For example, if a consumer thread, which is waiting because shared queue is empty, gets wake up due to a false alarm and try to get something from queue without further checking whether queue is empty or not than unexpected result is possible.

Why wait(), notify(), notifyAll() must be called inside a synchronized method/block?

Answer:- In Java, any object can act as a monitor - that's an entity with a single lock, an entry queue, and a waiting queue. An object's method without qualified by the keyword synchronized can be invoked by any number of threads at any time, the lock is ignored. The synchronized method of an object, one and only one thread, who owns the lock of that object, can be permitted to run that method at any time;i.e. a synchronized method is mutually exclusive . If, at the time of invocation, another thread owns the lock, then the calling thread will be put in the Blocked state and is added to the entry queue.

The wait(), notify(), and notifyAll() methods should be called for an object only when the current thread has already locked the object's lock. This point sometimes goes unnoticed because programmers are used to calling these methods from within synchronized methods or blocks. Otherwise, you will get **"java.lang.IllegalMonitorStateException:** current thread not owner" at runtime.

When a thread running in a synchronized method of an object is calling the wait() method of the same object, that thread releases the lock of the object and is added to that object's waiting queue. As long as it's there, it sits idle. Note also that wait() forces the thread to release its lock.

This means that it must own the lock of an object before calling the wait() method of that (same) object. Hence the thread must be in one of the object's synchronized methods or synchronized block before calling wait().

When a thread invokes an object's notify() or notifyAll() method, one (an arbitrary thread) or all of the threads in its waiting queue are removed from the waiting queue to the entry queue. They then actively contend for the object's lock, and the one that gets the lock goes on to execute. If no threads are waiting in the waiting queue, then notify() and notifyAll() have no effect. Before calling the notify() or notifyAll() method of an object, a thread must own the lock of the object. Hence it must be in one of the object's synchronized methods or synchronized block.
A thread in the waiting queue of an object can run again only when some other thread calls the notify() (or the notifyAll) method of the same object.
The reason to call wait() is that the thread does not want to execute a block of code until a particular state to be achieved. It wants to wait until a particular state to be achieved. The reason to call notify() or notifyAll() method is that the thread will signal others that "a particular state has been achieved". The state is a communication channel between threads and it must be shared mutable state.
For example, one thread read data from a buffer and one thread write data into buffer. The reading data thread needs to wait until the writing data thread completly write a block data into the buffer. The wirting data thread needs to wait until the reading data thread completly read the data from the buffer. If wait(), notify(), and notifyAll() methods can be called by a ordinary method , the reading thread calls wait() and the thread is being added to waiting queue . At just the same moment, the writing thread calls notify() to signal the condition changes. The reading thread misses the change and waits forever. Hence, they must be called inside a synchronized method or block which is mutually exclusive.

We all know that in order to invoke Object.wait(), this call must be placed in synchronized block, otherwise an IllegalMonitorStateException is thrown. But what's the reason for making this restriction? I know that wait() releases the monitor, but why do we need to explicitly acquire the monitor by making particular block synchronized and then release the monitor by calling wait()? What is the potential damage if it was possible to invoke wait() outside a synchronized block, retaining its semantics - suspending the caller thread?

Answer:- A wait() only makes sense when there is also a notify(), so it's always about communication between threads, and that needs synchronization to work correctly. One could argue that this should be implicit, but that would not really help, for the following reason: Semantically, you never just wait(). You need some condition to be satsified, and if it is not, you wait until it is. So what you really do is

```
if(!condition){
    wait();
}
```

But the condition is being set by a separate thread, so in order to have this work correctly you need synchronization.A couple more things wrong with it, where just because your thread quit waiting doesn't mean the condition you are looking for is true:-

- You can get spurious wakeups (meaning that a thread can wake up from waiting without ever having received a notification), or
- The condition can get set, but a third thread makes the condition false again by the time the waiting thread wakes up (and reacquires the monitor).

To deal with these cases what you really need is always some variation of this:-

```
synchronized(lock){
    while(!condition){
        lock.wait();
    }
}
```

Better yet, don't mess with the synchronization primitives at all and work with the abstractions offered in the java.util.concurrent packages.

What is the potential damage if it was possible to invoke wait() outside a synchronized block, retaining its semantics - suspending the caller thread?

Answer:- Let's illustrate what issues we would run into if wait() could be called outside of a synchronized block with a concrete example:-Suppose we were to implement a blocking queue.A first attempt (without synchronization) could look something along the lines below

```
class BlockingQueue {
    Queue<String> buffer = new LinkedList<String>();

    public void give(String data) {
        buffer.add(data);
        notify();                // Since someone may be waiting in take!
```

```
    }

    public String take() throws InterruptedException {
        while (buffer.isEmpty())    // don't use "if" due to spurious wakeups.
            wait();
        return buffer.remove();
    }
}
```

This is what could potentially happen:-

- A consumer thread calls take() and sees that the buffer.isEmpty().
- Before the consumer thread goes on to call wait(), a producer thread comes along and invokes a full give(), that is, buffer.add(data); notify();
- The consumer thread will now call wait() (and miss the notify() that was just called).
- If unlucky, the producer thread won't produce more give() as a result of the fact that the consumer thread never wakes up, and we have a dead-lock.

Once you understand the issue, the solution is obvious: Always perform give/notify and isEmpty/wait atomically.

Without going into details: This synchronization issue is universal. As Michael Borgwardt points out, wait/notify is all about communication between threads, so you'll always end up with a race condition similar to the one described above. This is why the "only wait inside synchronized" rule is enforced.

The predicate that the producer and consumer need to agree upon is in the above example buffer.isEmpty(). And the agreement is resolved by ensuring that the wait and notify are performed in synchronized blocks.

Difference between notify and notifyAll in Java?

Answer:- When you call notify only one of waiting thread will be woken and its not guaranteed which thread will be woken, it depends upon Thread scheduler. While if you call notifyAll method, all threads waiting on that lock will be woken up, but again all woken thread will fight for lock before executing remaining code and that's why wait is called on loop because if multiple threads are woken up, the thread which will get lock will first execute and it may reset waiting condition, which will force subsequent threads to wait. So key difference between notify and notifyAll is that notify() will cause only one thread to wake up while notifyAll method will make all thread to wake up.

Here are couple of main differences between notify and notifyAll method in Java :-

- First and main difference between notify() and notifyAll() method is that, if multiple thread is waiting on any lock in Java, notify send notification to only one of waiting thread while notifyAll informs all threads waiting on that lock.
- If you use notify method , It's not guaranteed that, which thread will be informed, but if you use notifyAll, since all thread will be notified, they will compete for lock and the lucky thread which gets lock will continue. In a way notifyAll method is more safe because it send notification to all threads, so if any thread misses the notification, there are other threads to do the job, while in case of notify method if the notified thread misses the notification then it could create subtle, hard to debug issues.
- Some people argue that using notifyAll can drain more CPU cycles than notify itself but if you really want to sure that your notification doesn't get wasted by any reason, use notifyAll. Since wait method is called from loop and they check condition even after waking up, calling notifyAll won't lead any side effect, instead it ensures that notification is not dropped.

Summary:- Having said that main difference between notify and notifyAll in Java comes down to fact how many threads will be wake up. Prefer notifyAll over notify whenever in doubt and if you can, avoid using notify and notifyAll altogether, instead use concurrency utility like CountDownLatch, CyclicBarrier, and Semaphore to write your concurrency code. It's not easy to get wait and notify method working correct in first attempt and concurrency bugs are often hard to figure out.

When to use notify and notifyAll in Java

Answer:-You can use notify over notifyAll if all thread are waiting for same condition and only one Thread at a time can benefit from condition becoming true. In this case notify is optimized call over notifyAll because waking up all of them because we know that only one thread will benefit and all other will wait again, so calling notifyAll method is just waste of cpu cycles. Though this looks quite reasonable there is still a caveat that unintended recipient swallowing critical notification. by using notifyAll we ensure that all recipient will get notify.

What is Busy Spinning? Why you will use Busy Spinning as wait strategy?

Answer:-It's a wait strategy, where one thread wait for a condition to become true, but instead of calling wait or sleep method and releasing CPU, it just spin. This is particularly useful if condition is going to be true quite quickly i.e. in millisecond or microsecond. Advantage of not releasing CPU is that, all cached data and instruction are remained unaffected, which may be lost, had this thread is suspended on one core and brought back to another thread.

A busy spin, or busy wait, is a programming technique where you repeatedly test for something and do nothing until the test passes. For example in Java:-

```
while (!someResource.IsAvailable())
    ;
// If you get here, you know that someResource.IsAvailable() returned true
```

A busy spin is a primitive form of synchronization; when you leave one you know some test has passed (in the above case, someResource.IsAvailable() was once true).
Usually busy spins are wasteful of CPU cycles and should be avoided, for example in Java you would typically use a synchronized block instead; your code will be “frozen” and not use CPU cycles until the situation you are waiting for is satisfied.
However, synchronized blocks take a bit of setup cycles and require runtime support. If the thing you are testing is almost always satisfied, and if not satisfied will always be satisfied very soon, a busy spin may use fewer cycles than proper synchronization.

Why wait, notify and notifyAll are not inside thread class?

Answer:- Wait and notify is not just normal methods or synchronization utility, more than that they are communication mechanism between two threads in Java. And Object class is correct place to make them available for every object if this mechanism is not available via any java keyword like synchronized. Remember synchronized and wait notify are two different area and don’t confuse that they are same or related. Synchronized is to provide mutual exclusion and ensuring thread safety of Java class like race condition while wait and notify are communication mechanism between two thread.
Locks are made available on per Object basis, which is another reason wait and notify is declared in Object class rather than Thread class.
In Java in order to enter critical section of code, Threads needs lock and they wait for lock, they don't know which threads holds lock instead they just know the lock is hold by some thread and they should wait for lock instead of knowing which thread is inside the synchronized block and

asking them to release lock. this analogy fits with wait and notify being on object class rather than thread in Java.
Java is based on Hoare's monitors idea. In Java all object has a monitor. Threads waits on monitors so, to perform a wait, we need 2 parameters:-

- a Thread
- a monitor (any object)

In the Java design, the thread can not be specified, it is always the current thread running the code. However, we can specify the monitor (which is the object we call wait on). This is a good design, because if we could make any other thread to wait on a desired monitor, this would lead to an "intrusion", posing difficulties on designing/programming concurrent programs. Remember that in Java all operations that are intrusive in another thread's execution are deprecated (e.g. stop()).

Which thread will be notified if I use notify()?

Answer:- No guaranteed, **ThreadScheduler** will pick a random thread from pool of waiting thread on that monitor. What is guaranteed is that only one Thread will be notified.

How do I know how many threads are waiting, so that I can use notifyAll() ?

Answer:- Its depend upon your application logic, while coding you need to think whether a piece of code can be run by multiple thread or not. A good example to understand inter-thread communication is implementing producer consumer pattern in Java.

How to call notify()?

Answer:-Wait() and notify() method can only be called from synchronized method or block, you need to call notify method on object on which other threads are waiting.

Why are these thread waiting for being notified etc.

Answer:-Thread wait on some condition e.g. in producer consumer problem, producer thread wait if shared queue is full and consumer thread wait if shared queue is empty. Since multiple thread are working with a shared resource they communicate with each other using wait and notify method.

What is ThreadLocal variable in Java?

Answer:- ThreadLocal variables are special kind of variable available to Java programmer. Just like instance variable is per instance, ThreadLocal variable is per thread. It's a nice way to achieve thread-safety of expensive-to-create objects, for example you can make SimpleDateFormat thread-safe using ThreadLocal. Since that class is expensive, its not good to use it in local scope, which requires separate instance on each invocation. By providing each thread their own copy, you shoot two birds in one arrow. First, you reduce number of instance of expensive object by reusing fixed number of instances, and Second, you achieve thread-safety without paying cost of synchronization or immutability. Another good example of thread local variable is ThreadLocalRandom class, which reduces number of instances of expensive-to-create Random object in multi-threading environment. See this answer to learn more about thread local variables in Java.
ThreadLocal in Java is a different way to achieve thread-safety, it doesn't address synchronization requirement, instead it eliminates sharing by providing explicitly copy of Object to each thread. Since Object is no more shared there is no requirement of Synchronization which can improve scalability and performance of application.

When to use ThreadLocal in Java?

Answer:- Below are some well know usage of ThreadLocal class in Java:-

- ThreadLocal are fantastic to implement Per Thread Singleton classes or per thread context information like transaction id.
- You can wrap any non Thread Safe object in ThreadLocal and suddenly its uses becomes Thread-safe, as its only being used by Thread Safe. One of the classic example of ThreadLocal is sharing SimpleDateForamt. Since SimpleDateFormat is not thread safe, having a global formatter may not work but having per Thread formatter will certainly work.
- ThreadLocal provides another way to extend Thread. If you want to preserve or carry information from one method call to another you can carry it by using ThreadLocal. This can provide immense flexibility as you don't need to modify any method.
- On basic level ThreadLocal provides Thread Confinement which is extension of local variable. while local variable only accessible on block they are declared, ThreadLocal are visible only in Single Thread. No two Thread can see each others ThreadLocal variable. Real Life example of ThreadLocal are in J2EE application servers which uses java ThreadLocal variable to keep track of transaction and security Context. It makes lot

of sense to share heavy object like Database Connection as ThreadLocal in order to avoid excessive creation and cost of locking in case of sharing global instance.

Important points on Java ThreadLocal Class:-

- ThreadLocal in Java is introduced in JDK 1.2 but it later generified in JDK 1.4 to introduce type safety on ThreadLocal variable.
- ThreadLocal can be associated with Thread scope, all the code which is executed by Thread has access to ThreadLocal variables but two thread can not see each others ThreadLocal variable.
- Each thread holds an exclusive copy of ThreadLocal variable which becomes eligible to Garbage collection after thread finished or died, normally or due to any Exception, Given those ThreadLocal variable doesn't have any other live references.
- ThreadLocal variables in Java are generally private static fields in Classes and maintain its state inside Thread.
- 5Think about ThreadLocal variable while designing concurrency in your application. Don't misunderstood that ThreadLocal is alternative of Synchronization, it all depends upon design. If design allows each thread to have there own copy of object than ThreadLocal is there to use.

What is FutureTask in Java?

Answer:- FutureTask represents a cancellable asynchronous computation in concurrent Java application. This class provides a base implementation of Future, with methods to start and cancel a computation, query to see if the computation is complete, and retrieve the result of the computation. The result can only be retrieved when the computation has completed; the get methods will block if the computation has not yet completed. A FutureTask object can be used to wrap a Callable or Runnable object. Since FutureTask also implements Runnable, it can be submitted to an Executor for execution.

Difference between interrupted and isInterrupted method in Java?

Answer:-Main difference between interrupted() and isInterrupted() is that former clears the interrupt status while later does not. The interrupt mechanism in Java multithreading is implemented using an internal flag known as the interrupt status. Interrupting a thread by calling Thread.interrupt() sets this flag. When interrupted thread checks for an interrupt by invoking the static method Thread.interrupted(), interrupt status is cleared. The non-static isInterrupted()

method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag. By convention, any method that exits by throwing an InterruptedException clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.

Why you should check condition for waiting in a loop?

Answer:- It's possible for a waiting thread to receive false alerts and spurious wake up calls, if it doesn't check the waiting condition in loop, it will simply exit even if condition is not met. As such, when a waiting thread wakes up, it cannot assume that the state it was waiting for is still valid. It may have been valid in the past, but the state may have been changed after the notify() method was called and before the waiting thread woke up. That's why it always better to call wait() method from loop, you can even create template for calling wait and notify in Eclipse. To learn more about this question, I would recommend you to read Effective Java items on thread and synchronization.

Difference between synchronized and concurrent collection in Java?

Answer:- Though both Synchronized and Concurrent Collection class provide thread-safety, main difference between them is in performance, scalability and how they achieve thread-safety. Synchronized collections like synchronized HashMap, Hashtable, HashSet, Vector and synchronized ArrayList are much slower than their concurrent counterparts ConcurrentHashMap, CopyOnWriteArrayList and CopyOnWriteHashSet. Main reason of this slowness is locking, synchronized collection locks whole object, while concurrent collection achieves thread-safety smartly e.g. ConcurrentHashMap divides the whole map into several segments and locks only on segments, which allows multiple threads to access other collections without locking. CopyOnWriteArrayList allows multiple reader thread to read without synchronization and when their is write happens it copies the ArrayList and swaps. So if you use concurrent collection classes in their favorable conditions e.g. for more reading and less updates, they are much more scalable than synchronized collections.

Synchronized Collections vs Concurrent Collections in Java

The synchronized collections classes, Hashtable and Vector, and the synchronized wrapper classes, Collections.synchronizedMap() and Collections.synchronizedList(), provides a basic conditionally thread-safe implementation of Map and List.

However, several factors make them unsuitable for use in highly concurrent applications -- their single collection-wide lock is an impediment to scalability and it often becomes necessary to lock a collection for a considerable time during iteration to prevent ConcurrentModificationException.
The ConcurrentHashMap and CopyOnWriteArrayList implementations provide much higher concurrency while preserving thread safety, with some minor compromises in their promises to callers. ConcurrentHashMap and CopyOnWriteArrayList are not necessarily useful everywhere you might use HashMap or ArrayList, but are designed to optimize specific common situations. Many concurrent applications will benefit from their use.
So what is the difference between Hashtable and ConcurrentHashMap , both can be used in multi-threaded environment but once the size of Hashtable becomes considerable large performance degrade because for iteration it has to be locked for longer duration.
Since ConcurrentHashMap introduced concept of segmentation, It doesn't mater whether how large it becomes because only certain part of it get locked to provide thread safety so many other readers can still access map without waiting for iteration to complete.
In Summary ConcurrentHashMap only locks certain portion of Map while Hashtable lock full map while doing iteration or performing any write operation.

Difference between Stack and Heap in Java?

Answer:- Here are few differences between stack and heap memory in Java:-

- Main difference between heap and stack is that stack memory is used to store local variables and function call, while heap memory is used to store objects in Java. No matter, where object is created in code e.g. as member variable, local variable or class variable, they are always created inside heap space in Java.
- Each Thread in Java has there own stack which can be specified using -Xss JVM parameter, similarly you can also specify heap size of Java program using JVM option -Xms and -Xmx where -Xms is starting size of heap and -Xmx is maximum size of java heap. to learn more about JVM options see my post 10 JVM option Java programmer should know.
- If there is no memory left in stack for storing function call or local variable, JVM will throw java.lang.StackOverFlowError, while if there is no more heap space for creating object, JVM will throw java.lang.OutOfMemoryError: Java Heap Space. Read more about how to deal with java.lang.OutOfMemoryError in my post 2 ways to solve OutOfMemoryError in Java.

- If you are using Recursion, on which method calls itself, You can quickly fill up stack memory. Another difference between stack and heap is that size of stack memory is lot lesser than size of heap memory in Java.
- Variables stored in stacks are only visible to the owner Thread, while objects created in heap are visible to all thread. In other words stack memory is kind of private memory of Java Threads, while heap memory is shared among all threads.

What is Thread Pool in Java and why we need it?

Answer:- Creating thread is expensive in terms of time and resource. If you create thread at time of request processing it will slow down your response time, also there is only a limited number of threads a process can create. To avoid both of these issue, a pool of thread is created when application starts-up and threads are reused for request processing. This pool of thread is known as "thread pool" and threads are known as worker thread. From JDK 1.5 release, Java API provides Executor framework, which allows you to create different types of thread pools e.g. single thread pool, which process one task at a time, fixed thread pool (a pool of fixed number of thread) or cached thread pool (an expandable thread pool suitable for applications with many short lived tasks).

Benefits of Thread Pool in Java:- Thread Pool offers several benefit to Java application, biggest of them is separating submission of task to execution of task ,which result if more loose coupled and flexible design than tightly coupled create and execute pattern. Here are some more benefits of using Thread pool in Java:

- Use of Thread Pool reduces response time by avoiding thread creation during request or task processing.
- Use of Thread Pool allows you to change your execution policy as you need. you can go from single thread to multiple thread by just replacing ExecutorService implementation.
- Thread Pool in Java application increases stability of system by creating a configured number of threads decided based on system load and available resource.
- Thread Pool frees application developer from thread management stuff and allows to focus on business logic.

Java Thread Pool - Executor Framework in Java 5:-Java 5 introduced several useful features like Enum, Generics, Variable arguments and several concurrency collections and utilities like ConcurrentHashMap and BlockingQueue etc, It also introduced a full feature built-in

Thread Pool framework commonly known as Executor framework. Core of this thread pool framework is Executor interface which defines abstraction of task execution with method execute(Runnable task) and ExecutorService which extends Executor to add various life-cycle and thread pool management facilities like shutting down thread pool. Executor framework also provides an static utility class called Executors (similar to Collections) which provides several static factory method to create various type of Thread Pool implementation in Java e.g. fixed size thread pool, cached thread pool and scheduled thread pool. Runnable and Callable interface are used to represent task executed by worker thread managed in these Thread pools. Interesting point about Executor framework is that, it is based on Producer consumer design pattern, where application thread produces task and worker thread consumers or execute those task, So it also suffers with limitation of Producer consumer task like if production speed is substantially higher than consumption than you may run OutOfMemory because of queued task, of course only if your queue is unbounded.

How to create fixed size thread pool using Executor framework in Java?

Answer:-Creating fixed size thread pool using Java 5 Executor framework is pretty easy because of static factory methods provided by Executors class. All you need to do is define your task which you want to execute concurrently and then submit that task to ExecutorService. from them Thread pool will take care of how to execute that task, it can be executed by any free worker thread and if you are interested in result you can query Future object returned by submit() method. Executor framework also provides different kind of Thread Pool e.g. SingleThreadExecutor which creates just one worker thread or CachedThreadPool which creates worker threads as and when necessary. You can also check Java documentation of Executor Framework for complete details of services provided by this API. Java concurrency in Practice also has couple of chapters dedicated to effective use of Java 5 Executor framework, which is worth reading for any senior Java developer.

Example of Thread Pool in Java

Here is an example of Thread pool in Java, which uses Executor framework of Java 5 to create a fixed thread pool with number of worker thread as 10. It will then create task and submit that to Thread pool for execution:

```
public class ThreadPoolExample {
```

```
    public static void main(String args[]) {
      ExecutorService service = Executors.newFixedThreadPool(10);
      for (int i =0; i<100; i++){
          service.submit(new Task(i));
      }
    }

}

final class Task implements Runnable{
    private int taskId;

    public Task(int id){
       this.taskId = id;
    }

    @Override
    public void run() {
       System.out.println("Task ID : " + this.taskId +" performed by "
                    + Thread.currentThread().getName());
    }

}
```

Output:

```
Task ID : 0 performed by pool-1-thread-1
Task ID : 3 performed by pool-1-thread-4
Task ID : 2 performed by pool-1-thread-3
Task ID : 1 performed by pool-1-thread-2
Task ID : 5 performed by pool-1-thread-6
Task ID : 4 performed by pool-1-thread-5
```

If you look at output of this Java example you will find different threads from thread pool are executing tasks.

How do you avoid deadlock in Java? Write Code?

Answer:-Deadlock is a condition in which two threads wait for each other to take action which allows them to move further. It's a serious issue because when it happen your program hangs and doesn't do the task it is intended for. In order for deadlock to happen, following four condition must be true :-

- **Mutual Exclusion :** At least one resource must be held in a non-sharable mode. Only one process can use the resource at any given instant of time.
- **Hold and Wait :** A process is currently holding at least one resource and requesting additional resources which are being held by other processes.
- **No Pre-emption :** The operating system must not de-allocate resources once they have been allocated; they must be released by the holding process voluntarily.
- **Circular Wait :** A process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource.

Easiest way to avoid deadlock is to prevent Circular wait, and this can be done by acquiring locks in a particular order and releasing them in reverse order, so that a thread can only proceed to acquire a lock if it held the other one.

How do you detect deadlock in Java ?

Answer:-Though this could have many answers , my version is first I would look the code if I see nested synchronized block or calling one synchronized method from other or trying to get lock on different object then there is good chance of deadlock if developer is not very careful.Other way is to find it when you actually get locked while running the application , try to take thread dump , in Linux you can do this by command "kill -3" , this will print status of all the thread in application log file and you can see which thread is locked on which object.Other way is to use jconsole, it will show you exactly which threads are get locked and on which object.

Detect deadlocks programmatically using ThreadMXBean class

Java 5 introduced ThreadMXBean - an interface that provides various monitoring methods for threads. I recommend you to check all of the methods as there are many useful operations for monitoring the performance of your application in case you are not using an external tool. The method of our interest is findMonitorDeadlockedThreads, or, if you are using Java 6,findDeadlockedThreads. The difference is that findDeadlockedThreads can also detect deadlocks caused by owner locks (java.util.concurrent), while findMonitorDeadlockedThreads

can only detect monitor locks (i.e. synchronized blocks). Since the old version is kept for compatibility purposes only, I am going to use the second version. The idea is to encapsulate periodical checking for deadlocks into a reusable component so we can just fire and forget about it.
https://dzone.com/articles/how-detect-java-deadlocks

Difference between livelock and deadlock in Java?

Answer:- This question is extension of previous interview question. A livelock is similar to a deadlock, except that the states of the threads or processes involved in the livelock constantly change with regard to one another, without anyone progressing further. Livelock is a special case of resource starvation. A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time. In short, main difference between livelock and deadlock is that in former state of process change but no progress is made.

How do you check if a Thread holds a lock or not?

Answer:- By method called holdsLock() on java.lang.Thread, it returns true if and only if the current thread holds the monitor lock on the specified object. Or thought about IllegalMonitorStateException which wait() and notify() methods throw when they get called from non-synchronized context so I said I would call newspaper.wait() and if this call throws exception it means thread in java is not holding lock, otherwise thread holds lock.

How do you take thread dump in Java?

Answer:- There are multiple ways to take thread dump of Java process depending upon operating system. When you take thread dump, JVM dumps state of all threads in log files or standard error console. In windows you can use Ctrl + Break key combination to take thread dump, on Linux you can use kill -3 command for same. You can also use a tool called jstack for taking thread dump, it operate on process id, which can be found using another tool called jps.

Which JVM parameter is used to control stack size of thread?

Answer:- This is the simple one, -Xss parameter is used to control stack size of Thread in Java.

What is ReentrantLock in Java

Answer:- On class level, ReentrantLock is a concrete implementation of Lock interface provided in Java concurrency package from Java 1.5 onwards. As per Javadoc, ReentrantLock is mutual exclusive lock, similar to implicit locking provided by synchronized keyword in Java, with extended feature like fairness, which can be used to provide lock to longest waiting thread. Lock is acquired by lock() method and held by Thread until a call to unlock() method. Fairness parameter is provided while creating instance of ReentrantLock in constructor. ReentrantLock provides same visibility and ordering guarantee, provided by implicitly locking, which means, unlock() happens before another thread get lock().

Benefits of ReentrantLock in Java:-

- Ability to lockinterruptibly.
- Ability to timeout while waiting for lock.
- Power to create fair lock.
- API to get list of waiting thread for lock.
- Flexibility to try for lock without blocking.

Disadvantages of ReentrantLock in Java:-Major drawback of using ReentrantLock in Java is wrapping method body inside try-finally block, which makes code unreadable and hides business logic. It's really cluttered and I hate it most, though IDE like Eclipse and Netbeans can add those try catch block for you. Another disadvantage is that, now programmer is responsible for acquiring and releasing lock, which is a power but also opens gate for new subtle bugs, when programmer forget to release the lock in finally block.

Difference between ReentrantLock and synchronized keyword in Java

Answer:- Though ReentrantLock provides same visibility and orderings guaranteed as implicit lock, acquired by synchronized keyword in Java, it provides more functionality and differ in certain aspect. As stated earlier, main difference between synchronized and ReentrantLock is ability to trying for lock interruptibly, and with timeout. Thread doesn't need to block infinitely, which was the case with synchronized. Let's see few more differences between synchronized and Lock in Java.

- Another significant difference between ReentrantLock and synchronized keyword is fairness. synchronized keyword doesn't support fairness. Any thread can acquire lock once released, no preference can be specified, on the other hand you can make

ReentrantLock fair by specifying fairness property, while creating instance of ReentrantLock. Fairness property provides lock to longest waiting thread, in case of contention.

- Second difference between synchronized and Reentrant lock is tryLock() method. ReentrantLock provides convenient tryLock() method, which acquires lock only if its available or not held by any other thread. This reduce blocking of thread waiting for lock in Java application.
- One more worth noting difference between ReentrantLock and synchronized keyword in Java is, ability to interrupt Thread while waiting for Lock. In case of synchronized keyword, a thread can be blocked waiting for lock, for an indefinite period of time and there was no way to control that. ReentrantLock provides a method called lockInterruptibly(), which can be used to interrupt thread when it is waiting for lock. Similarly tryLock() with timeout can be used to timeout if lock is not available in certain time period.
- ReentrantLock also provides convenient method to get List of all threads waiting for lock.

In short, Lock interface adds lot of power and flexibility and allows some control over lock acquisition process, which can be leveraged to write highly scalable systems in Java..

There are three threads T1, T2 and T3? How do you ensure sequence T1, T2, T3 in Java?

Answer:-Sequencing in multi-threading can be achieved by different means but you can simply use join() method of thread class to start a thread when another one is finished its execution. To ensure three threads execute you need to start the last one first e.g. T3 and then call join methods in reverse order e.g. T3 calls T2. join, and T2 calls T1.join, this ways T1 will finish first and T3 will finish last.

How to Join Multiple Threads in Java?

Answer:- Join method from Thread class is an important method and used to impose order on execution of multiple Threads.

When to use join method in Java?

Answer:- Similar to wait method, by using join method, we can make one Thread to wait for another. Primary use of Thread.join() is to wait for another thread and start execution once that

Thread has completed execution or died. Join is also a blocking method, which blocks until thread on which join has called die or specified waiting time is over.

Important point on Thread.join method:-

- Join is a final method in java.lang.Thread class and you cannot override it.
- Join method throw IntrupptedException if another thread interrupted waiting thread as a result of join() call.
- Join is also an overloaded method in Java, three version of join() available, check javadoc for details.

What does yield method of Thread class do?

Answer:-Yield method is one way to request current thread to relinquish CPU so that other thread can get chance to execute. Yield is a static method and only guarantees that current thread will relinquish the CPU but doesn't say anything about which other thread will get CPU. It's possible for same thread to get CPU back and start its execution again.

Difference between wait and yield in Java

Answer:-

- First difference between wait vs yield method is that, wait() is declared in java.lang.Object class while Yield is declared on java.lang.Thread class.
- Second difference between wait and yield in Java is that wait is overloaded method and has two version of wait, normal and timed wait while yield is not overloaded.
- Third difference between wait and yield is that wait is an instance method while yield is an static method and work on current thread.
- Another difference on wait and yield is that When a Thread call wait it releases the monitor.
- Fifth difference between yield vs wait which is quite important as well is that wait() method must be called from either synchronized block or synchronized method, There is no such requirement for Yield method.
- Another Java best practice which differentiate wait and yield is that, it's advised to call wait method inside loop but yield is better to be called outside of loop.

What is difference between submit() and execute() method of Executor and ExecutorService in Java?

Answer:- Main difference between submit and execute method from ExecutorService interface is that former return a result in form of Future object, while later doesn't return result. By the way both are used to submit task to thread pool in Java but one is defined in Executor interface,while other is added into ExecutorService interface. This multithreading interview question is also asked at first round of Java interviews.

Difference between submit() and execute() method thread pool in Java?
Answer:- Both method are ways to submit task to thread pools but there is slight difference between them. execute(Runnable command) is defined in Executor interface and executes given task in future, but more importantly it does not return anything. It's return type is void. On other hand submit() is overloaded method, it can take either Runnable or Callable task and can return Future object which can hold pending result of computation. This method is defined on ExecutorService interface, which extends Executor interface, and every other thread pool class e.g. ThreadPoolExecutor or ScheduledThreadPoolExecutor gets these methods.

What is ReentrantLock in Java? Have you used it before?

Answer:- ReentrantLock is an alternative of synchronized keyword in Java, it is introduced to handle some of the limitations of synchronized keywords. Many concurrency utility classes and concurrent collection classes from Java 5, including ConcurrentHashMap uses ReentrantLock, to leverage optimization. ReentrantLock mostly uses atomic variable and faster CAS operation to provides better performance. Key points to mention is difference between ReentrantLock and synchronized keyword in Java, which includes ability to acquire lock interruptibly, timeout feature while waiting for lock etc. ReentrantLock also gives option to create fair lock in Java.Once again a very good Java concurrency interview question for experienced Java programmers.

What is ReadWriteLock in Java? What is benefit of using ReadWriteLock in Java?
Answer:- This is usually a followup question of previous Java concurrency questions. ReadWriteLock is again based upon lock striping by providing separate lock for reading and writing operations. If you have noticed before, reading operation can be done without locking if there is no writer and that can hugely improve performance of any application. ReadWriteLock leverage this idea and provide policies to allow maximum concurrency level. Java Concurrency API also provides an implementation of this concept as ReentrantReadWriteLock. Depending upon Interviewer and experience of candidate, you can even expect to provide your own implementation of ReadWriteLock, so be prepare for that as well.

What will happen if you put return statement or System.exit () on try or catch block ? Will finally block execute?

Answer:-This is a very popular tricky Java question and its tricky because many programmer think that no matter what, but finally block will always execute. This question challenge that misconcept by putting return statement in try or catch block or calling System.exit from try or catch block. Answer of this tricky question in Java is that finally block will execute even if you put return statement in try block or catch block but finally block won't run if you call System.exit form try or catch.

What is concurrence level of ConcurrentHashMap in Java?

Answer:- ConcurrentHashMap achieves it's scalability and thread-safety by partitioning actual map into number of sections. This partitioning is achieved using concurrency level. It's optional parameter of ConcurrentHashMap constructor and it's default value is 16. The table is internally partitioned to try to permit the indicated number of concurrent updates without contention.

How to use ConcurrentHashMap in Java - Example Tutorial and Working?

Answer:- ConcurrentHashMap in Java is introduced as an alternative of Hashtable in Java 1.5 as part of Java concurrency package. Prior to Java 1.5 if you need a Map implementation, which can be safely used in a concurrent and multi-threaded Java program, than, you only have Hashtable or synchronized Map because HashMap is not thread-safe. With ConcurrentHashMap, now you have better choice; because, not only it can be safely used in concurrent multi-threaded environment but also provides better performance over Hashtable and synchronizedMap. ConcurrentHashMap performs better than earlier two because it only locks a portion of Map, instead of whole Map, which is the case with Hashtable and synchronized Map. CHM allows concurred read operations and same time, maintains integrity by synchronizing write operations.

How ConcurrentHashMap is implemented in Java

Answer:- ConcurrentHashMap is introduced as an alternative of Hashtable and provided all functions supported by Hashtable with additional feature called "concurrency level", which allows ConcurrentHashMap to partition Map. ConcurrentHashMap allows multiple readers to read concurrently without any blocking. This is achieved by partitioning Map into different parts based on concurrency level and locking only a portion of Map during updates. Default

concurrency level is 16, and accordingly Map is divided into 16 part and each part is governed with different lock. This means, 16 thread can operate on Map simultaneously, until they are operating on different part of Map. This makes ConcurrentHashMap high performance despite keeping thread-safety intact. Though, it comes with caveat. Since update operations like put(), remove(), putAll() or clear() is not synchronized, concurrent retrieval may not reflect most recent change on Map.

In case of putAll() or clear(), which operates on whole Map, concurrent read may reflect insertion and removal of only some entries. Another important point to remember is iteration over CHM, Iterator returned by keySet of ConcurrentHashMap are weekly consistent and they only reflect state of ConcurrentHashMap and certain point and may not reflect any recent change. Iterator of ConcurrentHashMap's keySet area also fail-safe and doesn't throw ConcurrentModificationExceptoin..

Default concurrency level is 16 and can be changed, by providing a number which make sense and work for you while creating ConcurrentHashMap. Since concurrency level is used for internal sizing and indicate number of concurrent update without contention, so, if you just have few writers or thread to update Map keeping it low is much better. ConcurrentHashMap also uses ReentrantLock to internally lock its segments.

When to use ConcurrentHashMap in Java?

Answer:- ConcurrentHashMap is best suited when you have multiple readers and few writers. If writers outnumber reader, or writer is equal to reader, than performance of ConcurrentHashMap effectively reduces to synchronized map or Hashtable. Performance of CHM drops, because you got to lock all portion of Map, and effectively each reader will wait for another writer, operating on that portion of Map. ConcurrentHashMap is a good choice for caches, which can be initialized during application start up and later accessed my many request processing threads. As javadoc states, CHM is also a good replacement of Hashtable and should be used whenever possible, keeping in mind, that CHM provides slightly weeker form of synchronization than Hashtable.

Summary:-Now we know What is ConcurrentHashMap in Java and when to use ConcurrentHashMap, it's time to know and revise some important points about CHM in Java.

- ConcurrentHashMap allows concurrent read and thread-safe update operation.
- During update operation, ConcurrentHashMap only lock a portion of Map instead of whole Map.

- Concurrent update is achieved by internally dividing Map into small portion which is defined by concurrency level.
- Choose concurrency level carefully as a significant higher number can be waste of time and space and lower number may introduce thread contention in case writers over number concurrency level.
- All operations of ConcurrentHashMap are thread-safe.
- ConcurrentHashMap implementation doesn't lock whole Map, there is chance of read overlapping with update operations like put() and remove(). In that case result returned by get() method will reflect most recently completed operation from there start.
- Iterator returned by ConcurrentHashMap is weekly consistent, fail safe and never throw ConcurrentModificationException.
- In Java ConcurrentHashMap doesn't allow null as key or value.
- You can use ConcurrentHashMap in place of Hashtable but with caution as CHM doesn't lock whole Map.
- During putAll() and clear() operations, concurrent read may only reflect insertion or deletion of some entries.

ConcurrentHashMap putifAbsent example in Java:- ConcurrentHashMap examples are similar to Hashtable examples, we have seen earlier, but worth knowing is use of putIfAbsent() method. Many times we need to insert entry into Map, if its not present already, and we wrote following kind of code:-

```
synchronized(map){
  if (map.get(key) == null){
      return map.put(key, value);
  } else{
      return map.get(key);
  }
}
```

hough this code will work fine in HashMap and Hashtable, This won't work in ConcurrentHashMap; because, during put operation whole map is not locked, and while one thread is putting value, other thread's get() call can still return null which result in one thread overriding value inserted by other thread. Ofcourse, you can wrap whole code in synchronized block and make it thread-safe but that will only make your code single threaded.

ConcurrentHashMap provides putIfAbsent(key, value) which does same thing but atomically and thus eliminates above race condition.

What is Semaphore in Java?

Answer:- Semaphore in Java is a new kind of synchronizer. It's a counting semaphore. Conceptually, a semaphore maintains a set of permits. Each acquire() blocks if necessary until a permit is available, and then takes it. Each release() adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly. Semaphore is used to protect expensive resource which is available in fixed number e.g. database connection in pool.
Some Scenario where Semaphore can be used:-

- To implement better Database connection pool which will block if no more connection is available instead of failing and handover Connection as soon as its available.
- To put a bound on collection classes. by using semaphore you can implement bounded collection whose bound is specified by counting semaphore.

Important points of Counting Semaphore in Java:-

- Semaphore class supports various overloaded version of tryAquire() method which acquires permit from semaphore only if its available during time of call.
- Another worth noting method from Semaphore is acquireUninterruptibly() which is a blocking call and wait until a permit is available.

Counting Semaphore Example in Java 5:-

Counting Semaphore in Java is a synchronizer which allows to impose a bound on resource is added in Java 5 along with other popular concurrent utilities like CountDownLatch, CyclicBarrier and Exchanger etc. Counting Semaphore in Java maintains specified number of pass or permits, In order to access a shared resource, Current Thread must acquire a permit. If permit is already exhausted by other thread than it can wait until a permit is available due to release of permit from different thread. This concurrency utility can be very useful to implement producer consumer design pattern or implement bounded pool or resources like Thread Pool, DB Connection pool etc. java.util.Semaphore class represent a Counting semaphore which is initialized with number of permits. Semaphore provides two main method acquire() and release() for getting permits and releasing permits. acquire() method blocks until permit is available. Semaphore provides both blocking method as well as unblocking method to acquire permits.

This Java concurrency tutorial focus on a very simple example of Binary Semaphore and demonstrate how mutual exclusion can be achieved using Semaphore in Java.

Counting Semaphore Example in Java (Binary Semaphore):- a Counting semaphore with one permit is known as binary semaphore because it has only two state permit available or permit unavailable. Binary semaphore can be used to implement mutual exclusion or critical section where only one thread is allowed to execute. Thread will wait on acquire() until Thread inside critical section release permit by calling release() on semaphore. Here is a simple example of counting semaphore in Java where we are using binary semaphore to provide mutual exclusive access on critical section of code in java:-

```
import java.util.concurrent.Semaphore;

public class SemaphoreTest {

   Semaphore binary = new Semaphore(1);

   public static void main(String args[]) {
      final SemaphoreTest test = new SemaphoreTest();
      new Thread(){
         @Override
         public void run(){
           test.mutualExclusion();
         }
      }.start();

      new Thread(){
         @Override
         public void run(){
           test.mutualExclusion();
         }
      }.start();

   }

   private void mutualExclusion() {
      try {
         binary.acquire();

         //mutual exclusive region
         System.out.println(Thread.currentThread().getName() + " inside mutual exclusive
region");
         Thread.sleep(1000);
```

```
        } catch (InterruptedException i.e.) {
            ie.printStackTrace();
        } finally {
            binary.release();
            System.out.println(Thread.currentThread().getName() + " outside of mutual exclusive
region");
        }
    }

}

Output:
Thread-0 inside mutual exclusive region
Thread-0 outside of mutual exclusive region
Thread-1 inside mutual exclusive region
Thread-1 outside of mutual exclusive region
```

What happens if you submit task, when queue of thread pool is already fill?

Answer:- This is another tricky question in my list. Many programmer will think that it will block until a task is cleared but it's true. ThreadPoolExecutor's submit() method throws RejectedExecutionException if the task cannot be scheduled for execution.

What is blocking method in Java?

Answer:- A blocking method is a method which blocks until task is done, for example accept() method of ServerSocket blocks until a client is connected. here blocking means control will not return to caller until task is finished. On the other hand there are asynchronous or non-blocking method which returns even before task is finished.

Examples of blocking methods in Java:-There are lots of blocking methods in Java API and good thing is that javadoc clearly informs about it and always mention whether a method call is blocking or not. In General methods related to reading or writing file, opening network connection, reading from Socket, updating GUI synchronously uses blocking call. here are some of most common methods in Java which are blocking in nature:-

- InputStream.read() which blocks until input data is available, an exception is thrown or end of Stream is detected.
- ServerSocket.accept() which listens for incoming socket connection in Java and blocks until a connection is made.
- InvokeAndWait() wait until code is executed from Event Dispatcher thread.

Disadvantages of blocking method:-Blocking methods poses significant threat to scalability of System. Imagine you are writing a client server application and you can only serve one client at a time because your code blocks. there is no way you can make use of that System and its not even using resources properly because you might have high speed CPU sitting idle waiting for something. Yes there are ways to mitigate blocking and using multiple threads for serving multiple clients is a classical solution of blocking call. Though most important aspect is design because a poorly designed system even if its multi-threaded can not scale beyond a point, if you are relying solely of number of Threads for scalability means it can not be more than few hundred or thousands since there is limit on number of thread JVM can support. Java5 addresses this issue by adding non blocking and asynchronous alternative of blocking IO calls and those utility can be used to write high performance servers application in core Java.

Best practices while calling blocking method in Java:-Blocking methods are for a purpose or may be due to limitation of API but there are guidelines available in terms of common and best practices to deal with them. here I am listing some standard ways which I employ while using blocking method in Java

- If you are writing GUI application may be in Swing never call blocking method in Event dispatcher thread or in the event handler. for example if you are reading a file or opening a network connection when a button is clicked don't do that on actionPerformed() method, instead just create another worker thread to do that job and return from actionPerformed(). this will keep your GUI responsive, but again it depends upon design if the operation is something which requires user to wait than consider using invokeAndWait() for synchronous update.
- Always let separate worker thread handles time consuming operations e.g. reading and writing to file, database or socket.
- Use timeout while calling blocking method. so if your blocking call doesn't return in specified time period, consider aborting it and returning back but again this depends upon scenario. if you are using Executor Framework for managing your worker threads, which is by the way recommended way than you can use Future object whose get() methods support timeout, but ensure that you properly terminate a blocking call.
- Extension of first practices, don't call blocking methods on keyPressed() or paint() method which are supposed to return as quickly as possible.
- Use call-back functions to process result of a blocking call.

Important points:-

- If a Thread is blocked in a blocking method it remain in any of blocking state e.g. WAITING, BLOCKED or TIMED_WAITING.
- Some of the blocking method throws checked InterrupptedException which indicates that they may allow cancel the task and return before completion like Thread.sleep() or BlockingQueue.put() or take() throws InterruptedException.
- interupt() method of Thread class can be used to interuupt a thread blocked inside blocking operation, but this is mere a request not guarantee and works most of the time.

Is Swing thread-safe? What do you mean by Swing thread-safe?

Answer:- You can simply this question as No, Swing is not thread-safe, but you have to explain what you mean by that even if interviewer doesn't ask about it. When we say swing is not thread-safe we usually refer its component, which can not be modified in multiple threads. All update to GUI components has to be done on AWT thread, and Swing provides synchronous and asynchronous callback methods to schedule such updates.

Difference between invokeAndWait and invokeLater in Java?

Answer:- These are two methods Swing API provides Java developers to update GUI components from threads other than Event dispatcher thread. InvokeAndWait() synchronously update GUI component, for example a progress bar, once progress is made, bar should also be updated to reflect that change. If progress is tracked in a different thread, it has to call invokeAndWait() to schedule an update of that component by Event dispatcher thread. On other hand, invokeLater() is asynchronous call to update components.

Which method of Swing API are thread-safe in Java?

Answer:- This question is again related to swing and thread-safety, though components are not thread-safe there are certain method which can be safely call from multiple threads. I know about repaint(), and revalidate() being thread-safe but there are other methods on different swing components e.g. setText() method of JTextComponent, insert() and append() method of JTextArea class.

What is ReadWriteLock in Java?

Answer:- In general, read write lock is result of lock stripping technique to improve performance of concurrent applications. In Java, ReadWriteLock is an interface which was added in Java 5 release. A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive. If you want you can implement this interface with your own set of rules, otherwise you can use ReentrantReadWriteLock, which comes along with JDK and supports a maximum of 65535 recursive write locks and 65535 read locks.

Difference between volatile and atomic variable in Java?

Answer:- This is an interesting question for Java programmer, at first, volatile and atomic variable look very similar, but they are different. Volatile variable provides you happens-before guarantee that a write will happen before any subsequent write, it doesn't guarantee atomicity. For example count++ operation will not become atomic just by declaring count variable as volatile. On the other hand AtomicInteger class provides atomic method to perform such compound operation atomically e.g. getAndIncrement() is atomic replacement of increment operator. It can be used to atomically increment current value by one. Similarly you have atomic version for other data type and reference variable as well.

What happens if a thread throws an Exception inside synchronized block?

Answer:- This is one more tricky question for average Java programmer, if he can bring the fact about whether lock is released or not is key indicator of his understanding. To answer this question, no matter how you exist synchronized block, either normally by finishing execution or abruptly by throwing exception, thread releases the lock it acquired while entering that synchronized block. This is actually one of the reason I like synchronized block over lock interface, which requires explicit attention to release lock, generally this is achieved by releasing lock in finally block.

What is double checked locking of Singleton?

Answer:- Double checked locking of Singleton is a way to ensure only one instance of Singleton class is created through application life cycle. As name suggests, in double checked locking, code checks for an existing instance of Singleton class twice with and without locking to double ensure that no more than one instance of singleton gets created. By the way, it was broken before Java fixed its memory models issues in JDK 1.5.

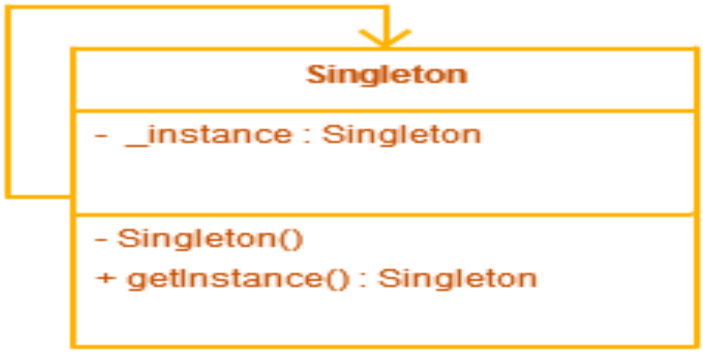
Why you need Double checked Locking of Singleton Class?

Answer:- One of the common scenario, where a Singleton class breaks its contracts is multi-threading. If you ask a beginner to write code for Singleton design pattern, there is good chance that he will come up with something like below :-

```
private static Singleton _instance;
public static Singleton getInstance(){
        if (_instance == null){
                _instance = new Singleton();
        }
        return _instance;
}
```

and when you point out that this code will create multiple instances of Singleton class if called by more than one thread parallel, he would probably make this whole getInstance() method synchronized, as shown in our 2nd code example getInstanceTS() method. Though it's a thread-safe and solves issue of multiple instance, it's not very efficient. You need to bear cost of synchronization all the time you call this method, while synchronization is only needed on first class, when Singleton instance is created. This will bring us to double checked locking pattern, where only critical section of code is locked. Programmer call it double checked locking because there are two checks for _instance == null, one without locking and other with locking (inside synchronized) block. Here is how double checked locking looks like in Java :-

```
public static Singleton getInstanceDC(){
    if (_instance == null){
        // Single Checked synchronized (Singleton.class) {
                if (_instance == null) {
                    // Double checked _instance = new Singleton();
                }
        }
    }
    return _instance;
}
```



On surface this method looks perfect, as you only need to pay price for synchronized block one time, but it still broken, until you make _instance variable volatile. Without volatile modifier it's possible for another thread in Java to see half initialized state of _instance variable, but with volatile variable guaranteeing happens-before relationship, all the write will happen on volatile _instance before any read of _instance variable. This was not the case prior to Java 5, and that's why double checked locking was broken before. Now, with happens-before guarantee, you can safely assume that this will work. By the way this is not the best way to create thread-safe Singleton, you can use Enum as Singleton, which provides inbuilt thread-safety during instance creation. Another way is to use static holder pattern.

```
/* * A journey to write double checked locking of Singleton class in Java. */
class Singleton{ private volatile static Singleton _instance; private Singleton() { //
preventing Singleton object instantiation from outside } /* * 1st version: creates multiple
instance if two thread access * this method simultaneously */ public static Singleton
getInstance() { if (_instance == null) { _instance = new Singleton(); } return _instance; } /* *
2nd version : this definitely thread-safe and only * creates one instance of Singleton on
concurrent environment * but unnecessarily expensive due to cost of synchronization * at
every call. */ public static synchronized Singleton getInstanceTS() { if (_instance == null) {
_instance = new Singleton(); } return _instance; } /* * 3rd version : An implementation of
double checkedlocking of Singleton. * Intention is to minimize cost of synchronization and
improve performance, * by only locking critical section of code, the code which creates
instance of Singleton class. * By the way this is still broken, if we don't make _instance
volatile, as another thread can * see a half initialized instance of Singleton. */ public static
Singleton getInstanceDC() { if (_instance == null) {synchronized (Singleton.class) { if
(_instance == null) { _instance = new Singleton(); } } } return _instance; } }
```

That's all about double checked locking of Singleton class in Java. This is one of the controversial way to create thread-safe Singleton in Java, with simpler alternatives available in

terms of using Enum as Singleton class. I don't suggest you to implement your Singleton like that as there are many better way to implement Singleton pattern in Java. Though, this question has historical significance and also teaches how concurrency can introduce subtle bugs. As I said before, this is very important from interview point of view. Practice writing double checked locking of Singleton class by hand before going for any Java interview. This will develop your insight on coding mistakes made by Java programmers. On related note, In modern day of Test driven development, Singleton is regarded as anti pattern because of difficulty it present to mock its behaviour, so if you are TDD practitioner better avoid using Singleton pattern.

List down 3 multi-threading best practice you follow?

Answer:- Following are three best practices I follow :-

- **Always give meaningful name to your thread** This goes a long way to find a bug or trace an execution in concurrent code. OrderProcessor, QuoteProcessor or TradeProcessor is much better than Thread-1. Thread-2 and Thread-3. Name should say about task done by that thread. All major framework and even JDK follow this best practice.
- **Avoid locking or Reduce scope of Synchronization** Locking is costly and context switching is even more costlier. Try to avoid synchronization and locking as much as possible and at bare minimum, you should reduce critical section. That's why I prefer synchronized block over synchronized method, because it gives you absolute control on scope of locking.
- **Prefer Synchronizers over wait and notify** Synchronizers like CountDownLatch, Semaphore, CyclicBarrier or Exchanger simplifies coding. It's very difficult to implement complex control flow right using wait and notify. Secondly, these classes are written and maintained by best in business and there is good chance that they are optimized or replaced by better performance code in subsequent JDK releases. By using higher level synchronization utilities, you automatically get all these benefits.
- **Prefer Concurrent Collection over Synchronized Collection** This is another simple best practice which is easy to follow but reap good benefits. Concurrent collection are more scalable than their synchronized counterpart, that's why its better to use them while writing concurrent code. So next time if you need map, think about ConcurrentHashMap before thinking Hashtable.

How do you force start a Thread in Java?

Answer:- This question is like how do you force garbage collection in Java, their is no way, though you can make request using System.gc() but its not guaranteed. On Java multi-threading their is absolute no way to force start a thread, this is controlled by thread scheduler and Java exposes no API to control thread schedule. This is still a random bit in Java.

What is fork join framework in Java?

Answer:- The fork join framework, introduced in JDK 7 is a powerful tool available to Java developer to take advantage of multiple processors of modern day servers. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application. One significant advantage of The fork/join framework is that it uses a work-stealing algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

What is difference between sleep() and yield() methods?

Answer:-

sleep()	yield()
sleep() causes the thread to **definitely stop executing for a given amount of time**; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power saving mode).	yield () basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should. Otherwise, the current thread will continue to run.
sleep() method **throws the Interrupted exception** if another thread interrupts the sleeping thread.	yield() method **does not throws the Interrupted exception**
sleep() method does not cause currently executing thread to give up any monitors	yield() method give up the monitors
Stops for the specified time	Pauses the current thread

What are the Similarities between Yield and Sleep method

Answer:-

1. **Static method :** Both yield and sleep method are static method . Hence , they always change the state of currently executing thread .

2. **java.lang.Thread :** Both yield and sleep method belongs to the java.lang.Thread class.
3. **Currently executing thread :** Both methods affect the state of currently executing thread .yield method may pause it while sleep method will stop it for a specified time .

What is yield() method?

Answer:- Suppose there are three threads t1, t2, and t3. Thread t1 gets the processor and starts its execution and thread t2 and t3 are in Ready/Runnable state. Completion time for thread t1 is 5 hour and completion time for t2 is 5 minutes. Since t1 will complete its execution after 5 hours, t2 has to wait for 5 hours to just finish 5 minutes job. In such scenarios where one thread is taking too much time to complete its execution, we need a way to prevent execution of a thread in between if something important is pending. yield () helps us in doing so.

1. **yield()** basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.
2. **yield()** method pauses the currently executing thread temporarily for giving a chance to the remaining waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution. The yielded thread when it will get the chance for execution is decided by the thread scheduler whose behavior is vendor dependent.
3. **According to Oracle docs,** Yield method temporarily pauses the currently executing thread to give a chance to the remaining waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads of low priority then the current thread will continue its execution.
4. **The major difference between yield and sleep** in Java is that yield() method pauses the currently executing thread temporarily for giving a chance to the remaining waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution. The yielded thread when it will get the chance for execution is decided by the thread scheduler whose behavior is vendor dependent. Yield method doesn't guarantee that current thread will pause or stop but it guarantee that CPU will be relinquished by current Thread as a result of a call to Thread.yield() method in java.
5. Once a thread has executed yield() method and there are many threads with same priority is waiting for processor, then we can't specify which thread will get execution chance first.

6. The thread which executes the yield() method will enter in the Runnable state from Running state.
7. Once a thread pauses its execution, we can't specify when it will get chance again it depends on thread scheduler.
8. Underlying platform must provide support for preemptive scheduling if we are using yield method.
9. There is no guarantee that Yield will make the currently executing thread to runnable state immediately. It can only make a thread from Running State to Runnable State, not in wait or blocked state.

What are the uses of yield() methods?

Answer:- There are following uses of yield() methods:-

- Whenever a thread calls java.lang.Thread.yield method, it gives hint to the thread scheduler that it is ready to pause its execution. Thread scheduler is free to ignore this hint.
- If any thread executes yield method, thread scheduler checks if there is any thread with same or high priority than this thread. If processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give processor to other thread and if not – current thread will keep executing.

What is difference between calling wait() and sleep() method in Java multi-threading?

Answer:- Though both wait and sleep introduce some form of pause in Java application, they are tool for different needs. Wait method is used for inter thread communication, it relinquish lock if waiting condition is true and wait for notification when due to action of another thread waiting condition becomes false. On the other hand sleep() method is just to relinquish CPU or stop execution of current thread for specified time duration. Calling sleep method doesn't release the lock held by current thread. wait is called from synchronized context only while sleep can be called without synchronized block. wait is called on Object while sleep is called on Thread. waiting thread can be awake by calling notify and notifyAll while sleeping thread can not be awaken by calling notify method. wait is normally done on condition, Thread wait until a condition is true while sleep is just to put your thread on sleep. wait release lock on object while waiting while sleep doesn’t release lock while waiting.

Difference between yield and sleep in java

Major difference between yield and sleep in Java is that yield() method pauses the currently executing thread temporarily for giving a chance to the remaining waiting threads of the same

priority to execute. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution. The yielded thread when it will get the chance for execution is decided by the thread scheduler whose behavior is vendor dependent. Yield method doesn't guarantee that current thread will pause or stop but it guarantee that CPU will be relinquish by current Thread as a result of call to Thread.yield() method in java.Sleep method in Java has two variants one which takes millisecond as sleeping time while other which takes both mill and nano second for sleeping duration.
sleep(long millis) or sleep(long millis,int nanos)
Cause the currently executing thread to sleep for the specified number of milliseconds plus the specified number of nanoseconds.

Points about Thread sleep() method in Java:-

- Thread.sleep() method is used to pause the execution, relinquish the CPU and return it to thread scheduler.
- Thread.sleep() method is a static method and always puts current thread on sleep.
- Java has two variants of sleep method in Thread class one with one argument which takes milliseconds as duration for sleep and other method with two arguments one is millisecond and other is nanosecond.
- Unlike wait() method in Java, sleep() method of Thread class doesn't relinquish the lock it has acquired.
- sleep() method throws Interrupted Exception if another thread interrupt a sleeping thread in java.
- With sleep() in Java its not guaranteed that when sleeping thread woke up it will definitely get CPU, instead it will go to Runnable state and fight for CPU with other thread.
- There is a misconception about sleep method in Java that calling t.sleep() will put Thread "t" into sleeping state, that's not true because Thread.sleep method is a static method it always put current thread into Sleeping state and not thread "t". just keep in mind that both sleep() and yield() operate on current thread.

What is difference between wait() and sleep?

Answer:- There are following difference between wait() and sleep() methods:-

1. **Class belongs :** The wait() method belongs to **java.lang.Object** class, thus can be called on any Object. The sleep() method belongs to **java.lang.Thread** class, thus can be called on Threads.
2. **Context :** The wait() method can only be **called from Synchronized context i.e. using synchronized block or synchronized method** If you call wait method without synchronization, it will **throw IllegalMonitorStateException** in Java**.** The sleep() method can be called from any context, there is no requirement of synchronization for calling sleep method , you can call it normally..
3. **Locking :** The wait() method **releases the lock on an object** and gives others chance to execute. The sleep() method **does not releases the lock of an object** for specified time or until interrupt.
4. **Wake up condition :** A waiting thread can be awake by notify() or notifyAll() method. A sleeping can be awaked by interrupt or time expires.
5. **Execution :** Each object has each wait() method for inter-communication between threads. The sleep() method is static method belonging to Thread class. There is a common mistake to write t.sleep(1000) because sleep() is a class method and will pause the current running thread not t.
6. **Use case :** We should use sleep() **for controlling execution time** of one thread and wait() **for multi-thread-synchronization**.

What are the Similarities between wait() and sleep()?

Answer:- There are following Similarities between wait() and sleep():-

1. **Thread state:** Both the method wait() and sleep() makes the running thread into Not Runnable state.
2. **Running time:** Both the method wait() and sleep() takes total execution time in milliseconds as an argument, after that it will be expired.

When to use wait()?

Answer:- The wait() is used for multi threaded synchronization, where single resource is shared among multiple thread. For example, file resources over network. wait() method should be used in conjunction with notify() or notifyAll() method and intended for communication between two threads in Java.

When to use sleep()?

Answer:- The wait() is used for time synchronization, where the thread actually needs a delay in background. For example, process something on specific interval. Thread.sleep() method is a utility method to introduce short pauses during program or thread execution.

What are the difference between wait() and yield()?

Answer:- wait() and yield() are completely different and there for different purpose. There are following difference between wait() and yield() methods:-

1. wait() is declared in **java.lang.Object** class while yield() is declared on **java.lang.Thread** class.
2. wait() is overloaded method and has two version of wait, normal and timed wait while yield() is not overloaded.
3. wait() is an instance method while yield() is an static method and work on current thread.
4. When a Thread call waits it releases the lock.
5. wait() method must be called from either synchronized block or synchronized method, There is no such requirement for yield() method.
6. its advised to call wait method inside the loop but the yield is better to be called outside of the loop.
7. Use wait() for inter thread communication while yield() is not just reliable enough even for the mentioned task. prefer Thread.sleep(1) instead of yield.

What are the difference between notify() and notifyAll()?

Answer:- There are following difference between notify() and notifyAll() methods:-

1. if multiple threads is waiting on any lock in Java, notify() method send notification to only one of waiting thread while notifyAll() informs all threads waiting on that lock.
2. If you use notify method , It's not guaranteed that, which thread will be informed, but if you use notifyAll since all thread will be notified, they will compete for lock and the lucky thread which gets lock will continue. In a way, notifyAll method is safer because it sends notification to all threads, so if any thread misses the notification, there are other threads to do the job, while in the case of notify() method if the notified thread misses the notification then it could create subtle, hard to debug issues.
3. if you really want to sure that your notification doesn't get wasted by any reason, use notifyAll. Since wait method is called from the loop and they check condition even after waking up, calling notifyAll won't lead any side effect, instead it ensures that notification is not dropped.

4. main difference between notify and notifyAll in Java comes down to the fact how many threads will be wake up. Prefer notifyAll() over notify() whenever in doubt and if you can, avoid using notify() and notifyAll() altogether, instead use concurrency utility like CountDownLatch, CyclicBarrier, and Semaphore to write your concurrency code. It's not easy to get wait and notify method working correctly in first attempt and concurrency bugs are often hard to figure out.

What are the difference between yield() and join()?

Answer:- There are following difference between yield() and join() methods:-

1. If a thread wants to pass its execution to give chance to remaining threads of same priority then we should go for yield(). If a thread wants to wait until completing of some other thread then we should go for join().
2. yield() method is not overloaded. While join() method is overloaded.
3. yield() method is not final While join() method is final.
4. yield() method doesnot throw exception While join() method throw exception.
5. yield() method is static While join() method is not.

What are the difference between Object level lock vs Class level lock?

Answer:- There are following difference between Object level lock and Class level lock:-

1. Object level lock is mechanism when we want to synchronize a non-static method or 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.
2. Class level lock prevents multiple threads to enter in synchronized block in any of all available instances of the class on runtime. This means if in runtime there are 100 instances of DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads.
3. Class level locking should always be done to make static data thread safe. As we know that static keyword associate data of methods to class level, so use locking at static fields or methods to make it on class level.

What are the difference between lock and monitor ?

Answer:- locks provide necessary support for implementing monitors.

Lock:- A lock is kind of data which is logically part of an object's header on the heap memory. Each object in a JVM has this lock (or mutex) that any program can use to coordinate multi-threaded access to the object. If any thread want to access instance variables of that object; then thread must "own" the object's lock (set some flag in lock memory area). All other threads that attempt to access the object's variables have to wait until the owning thread releases the object's lock (unset the flag).
Once a thread owns a lock, it can request the same lock again multiple times, but then has to release the lock the same number of times before it is made available to other threads. If a thread requests a lock three times, for example, that thread will continue to own the lock until it has "released" it three times.
Please note that lock is acquired by a thread, when it explicitly ask for it. In Java, this is done with the synchronized keyword, or with wait and notify.

Monitor:- Monitor is a synchronization construct that allows threads to have both mutual exclusion (using locks) and cooperation i.e. the ability to make threads wait for certain condition to be true (using wait-set).
In other words, along with data that implements a lock, every Java object is logically associated with data that implements a wait-set. Whereas locks help threads to work independently on shared data without interfering with one another, wait-sets help threads to cooperate with one another to work together towards a common goal e.g. all waiting threads will be moved to this wait-set and all will be notified once lock is released. This wait-set helps in building monitors with additional help of lock (mutex).

Mutual exclusion

Putting in very simple words, a monitor is like a building that contains one special room (object instance) that can be occupied by only one thread at a time. The room usually contains some data which needs to be protected from concurrent access. From the time a thread enters this room to the time it leaves, it has exclusive access to any data in the room. Entering the monitor building is called "entering the monitor." Entering the special room inside the building is called "acquiring the monitor." Occupying the room is called "owning the monitor," and leaving the room is called "releasing the monitor." Leaving the entire building is called "exiting the monitor."

When a thread arrives to access protected data (enter the special room), it is first put in queue in building reception (entry-set). If no other thread is waiting (own the monitor), the thread

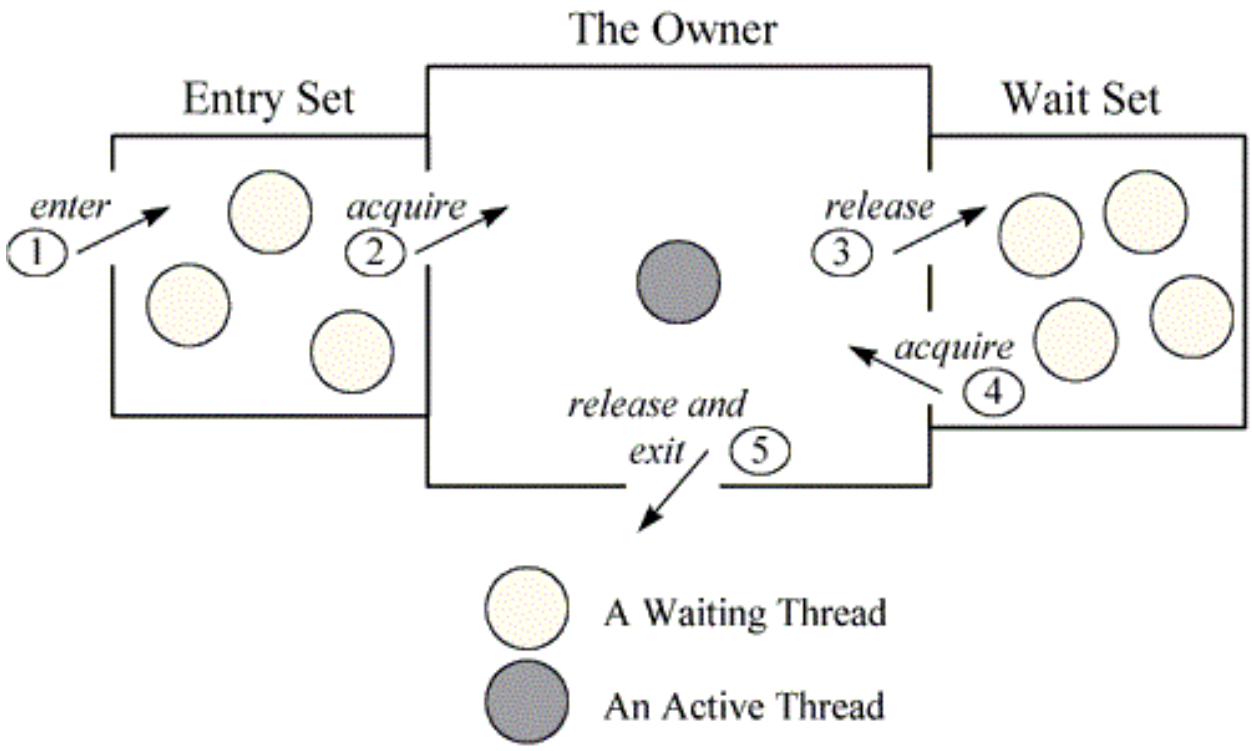
acquires the lock and continues executing the protected code. When the thread finishes execution, it release the lock and exits the building (exiting the monitor).

If when a thread arrives and another thread is already owning the monitor, it must wait in reception queue (entry-set). When the current owner exits the monitor, the newly arrived thread must compete with any other threads also waiting in the entry-set. Only one thread will win the competition and own the lock. There is no role of wait-set feature.

Cooperation

In general, mutual exclusion is important only when multiple threads are sharing data or some other resource. If two threads are not working with any common data or resource, they usually can't interfere with each other and needn't execute in a mutually exclusive way. Whereas mutual exclusion helps keep threads from interfering with one another while sharing data, cooperation helps threads to work together towards some common goal.

This cooperation requires both i.e. entry-set and wait-set. Below given diagram will help you in understand this cooperation.



Above figure shows the monitor as three rectangles. In the center, a large rectangle contains a single thread, the monitor's owner. On the left, a small rectangle contains the entry set. On the right, another small rectangle contains the wait set.

Cooperation is important when one thread needs some data to be in a particular state and another thread is responsible for getting the data into that state e.g. producer/consumer problem where read thread needs the buffer to be in a "not empty" state before it can read any data out of the buffer. If the read thread discovers that the buffer is empty, it must wait. The write thread is responsible for filling the buffer with data. Once the write thread has done some more writing, the read thread can do some more reading. It is also sometimes called a "Wait and Notify" OR "Signal and Continue" monitor because it retains ownership of the monitor and continues executing the monitor region (the continue) if needed. At some later time, the notifying thread releases the monitor and a waiting thread is resurrected to own the lock.

Describe Synchronization?

Answer:-

1. Synchronization in Java guarantees that no two threads can execute a synchronized method, which requires same lock, simultaneously or concurrently.
2. synchronized keyword can be used only with methods and code blocks. These methods or blocks can be static or non-static both.
3. When ever a thread enters into Java synchronized method or block it acquires a lock and whenever it leaves synchronized method or block it releases the lock. Lock is released even if thread leaves synchronized method after completion or due to any Error or Exception.
4. Java synchronized keyword is re-entrant in nature it means if a synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.
5. Java synchronization will throw NullPointerException if object used in synchronized block is null. For example, in above code sample if lock is initialized as null, the "synchronized (lock)" will throw NullPointerException.
6. Synchronized methods in Java put a performance cost on your application. So use synchronization when it is absolutely required. Also, consider using synchronized code blocks for synchronizing only critical section of your code.

7. It’s possible that both static synchronized and non static synchronized method can run simultaneously or concurrently because they lock on different object.
8. According to the Java language specification you can not use synchronized keyword with constructor. It is illegal and result in compilation error.
9. Do not synchronize on non final field on synchronized block in Java. because reference of non final field may change any time and then different thread might synchronizing on different objects i.e. no synchronization at all.
10. Do not use String literals because they might be referenced else where in the application and can cause deadlock. String objects created with new keyword can be used safely. But as a best practice, create a new private scoped Object instance OR lock on the shared variable itself which we want to protect. [Thanks to Anu to point this out in comments.]