Concurrency Questions

**Blocking Queue** : A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

The BlockingQueue is just like a regular Queue, except that it includes methods that will wait a specific amount of time to complete an operation.

offer(E e, long timeout,TimeUnit unit) Adds item to the queue waiting the specified time, returning false if time elapses before space is available

poll(long timeout, TimeUnit unit) Retrieves and removes an item from the queue, waiting the specified time, returning null if the time elapses before the item is available.

**try** {

BlockingQueue<Integer> blockingQueue = **new** LinkedBlockingQueue<>();

blockingQueue.offer(39);

blockingQueue.offer(3, 4, TimeUnit.SECONDS);

System.***out***.println(blockingQueue.poll());

System.***out***.println(blockingQueue.poll(10, TimeUnit.MILLISECONDS));

} **catch** (InterruptedException e) {

// Handle interruption

}

**Custom implementation**

**public** **class** BlockingQueue {

List<Integer> queue = **new** ArrayList<>();

**int** limit = 2;

**public** **synchronized** **boolean** enqueue(**final** **int** n) **throws** InterruptedException {

**while** (queue.size() == limit) {

wait();

}

**if** (queue.size() == 0) {

notifyAll();

}

**return** queue.add(n);

}

**public** **synchronized** **int** deque() **throws** InterruptedException {

**while** (queue.size() == 0) {

wait();

}

**if** (queue.size() == limit) {

notifyAll();

}

**return** queue.remove(0);

}

**public** **static** **void** main(**final** String[] args) {

BlockingQueue b = **new** BlockingQueue();

ExecutorService service = Executors.*newFixedThreadPool*(4);

**for** (**int** i = 0; i < 4; i++) {

service.submit(() -> b.enqueue(1));

}

}

}

**ShutdownHook**: Shutdown Hooks are a special construct that allow developers to plug in a piece of code to be executed when the JVM is shutting down. This comes in handy in cases where we need to do special clean up operations in case the VM is shutting down.

Runtime.getRuntime().addShutdownHook(new Thread(()-> System.out.println("Shutdown hook")));

**CyclicBarrier**: CyclicBarrier is used to make threads wait for each other. It is used when different threads process a part of computation and when all threads have completed the execution, the result needs to be combined in the parent thread. In other words, a CyclicBarrier is used when multiple thread carry out different sub tasks and the output of these sub tasks need to be combined to form the final output. After completing its execution, threads call await() method and wait for other threads to reach the barrier. Once all the threads have reached, the barriers then give the way for threads to proceed.

**public** **class** LionPenManager {

**public** **void** removeAnimals() {

System.***out***.println("Removing Lions");

}

**public** **void** cleanPen() {

System.***out***.println("Clean the pen");

}

**public** **void** addAnimals() {

System.***out***.println("Adding animals");

}

**public** **void** performTask(CyclicBarrier c1, CyclicBarrier c2) {

**try** {

removeAnimals();

c1.await();

cleanPen();

c2.await();

addAnimals();

}**catch**(InterruptedException | BrokenBarrierException e) {

e.printStackTrace();

}

}

**public** **static** **void** main(String[] args) {

ExecutorService service = Executors.*newFixedThreadPool*(4);

LionPenManager c = **new** LionPenManager();

CyclicBarrier c1 = **new** CyclicBarrier(4);

CyclicBarrier c2 = **new** CyclicBarrier(4,()-> {System.***out***.println("Pen cleaned");});

**try** {

**for**(**int** i=0;i<4;i++) {

service.submit(()->c.performTask(c1,c2));

}

}**finally** {

service.shutdown();

}

}

}

**Even Odd Print using 2 threads**

**public** **class** EvenOddPrint {

**int** count = 0;

**int** max = 10;

**boolean** isEven = **true**;

**public** **static** **void** main(**final** String[] args) **throws** InterruptedException {

EvenOddPrint obj = **new** EvenOddPrint();

Thread t1 = **new** Thread(() -> obj.printEven());

Thread t2 = **new** Thread(() -> obj.printOdd());

t1.start();

t2.start();

t1.join();

t2.join();

}

**public** **void** printEven() {

**try** {

**synchronized** (**this**) {

**while** (count < max) {

**while** (!isEven) {

wait();

}

System.***out***.println(count++);

isEven = **false**;

notify();

}

}

} **catch** (Exception e) {

}

}

**public** **void** printOdd() {

**try** {

**synchronized** (**this**) {

**while** (count < max) {

**while** (isEven) {

wait();

}

System.***out***.println(count++);

isEven = **true**;

notify();

}

}

} **catch** (Exception e) {

}

}

}

**Daemon Thread** : Daemon thread is a low priority thread that runs in background to perform tasks such as garbage collection. They can not prevent the JVM from exiting when all the user threads finish their execution.

**void setDaemon(boolean status)**

**boolean isDaemon():**

**We can prevent the execution of a thread by using one of the following methods of Thread class. Yield,sleep,join**

**Yield:**

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. yeild() helps us in doing so.  
**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.

**sleep():** This method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

[**join()**:](https://www.geeksforgeeks.org/joining-threads-in-java/) **java.lang.Thread** class provides the join() method which allows one thread to wait until another thread completes its execution. If **t** is a Thread object whose thread is currently executing, then **t.join()** will make sure that **t** is terminated before the next instruction is executed by the program.

**public** **class** Yield {

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Thread t1 = **new** Thread(**new** Runnable() {

**public** **void** run() {

**for**(**int** i=0;i<10;i++) {

System.***out***.println(i);

}

Thread.*yield*();

System.***out***.println("In thread");

}

});

t1.start();

t1.join();

System.***out***.println("In main");

}

}

**Prints In Thread then In main**, so it waits for the t1 to finish before main if we use join.

Wait and notify ( producer consumer )

**public** **class** ProducerConsumerProblem {

**static** **int** *count* = 0;

**final** **static** **int** ***MAX*** = 5;

**public** **void** consume() {

**try** {

**while** (**true**) {

**synchronized** (**this**) {

**if** (*count* == 0) {

wait();

}

System.***out***.println("Consumed : " + *count*--);

notify();

}

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**public** **void** produce() {

**try** {

**while** (**true**) {

**synchronized** (**this**) {

**if** (*count* == ***MAX***) {

wait();

}

System.***out***.println("Produced : " + ++*count*);

notify();

}

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**public** **static** **void** main(String[] args) {

ProducerConsumerProblem p = **new** ProducerConsumerProblem();

Thread t1 = **new** Thread(() -> {

p.produce();

});

Thread t2 = **new** Thread(() -> {

p.consume();

});

t1.start();

t2.start();

}

}

**Closeable and AutoCloseable**

Both are interfaces used to close resources. Closeable was introduced with JDK5 and now it looks as follows:

public interface Closeable extends AutoCloseable {

public void close() throws IOException;

}

AutoCloseable was introduced with JDK7 and looks as follows:

public interface AutoCloseable {

void close() throws Exception;

}

Closeable has some limitations, as it can only throw IOException, and it couldn't be changed without breaking legacy code. So AutoCloseable was introduced, which can throw Exception. If you use JDK7+, you are supposed to use AutoCloseable. As JDK7+ libraries use AutoCloseable and legacy code that implements Closeable still need to work with JDK7+, the solution was having Closeable extend AutoCloseable.

**Runnable and Callable**

public interface Runnable {

    public void run();

}

public interface Callable<V> {

    V call() throws Exception;

}

*Runnable* tasks can be run using the *Thread* class or *ExecutorService* whereas*Callables* can be run only using the latter.

A Runnable, however, does not return a result and cannot throw a checked exception.



**The purpose of start() is to create a separate call stack for the thread. A separate call stack is created by it, and then run() is called by JVM.**

A thread will be in WAITING state if it is waiting for notification from other threads. A thread will be in BLOCKED state if it is waiting for other thread to release the lock it wants.

**ReentrantLock**

ReentrantLock allow threads to enter into lock on a resource more than once. When the thread first enters into lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlock request, hold count is decremented by one and when hold count is 0, the resource is unlocked.

Reentrant Locks also offer a fairness parameter, by which the lock would abide by the order of the lock request i.e. after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.

**public** **class** ReentrantLockExample {

ReentrantLock lock = **new** ReentrantLock();

**public** **static** **void** main(**final** String[] args) {

ReentrantLockExample r = **new** ReentrantLockExample();

Thread t1 = **new** Thread(() -> r.displayMessage());

Thread t2 = **new** Thread(() -> r.displayMessage());

t1.start();

t2.start();

}

**public** **void** displayMessage() {

lock.lock();

System.***out***.println("Display By Thread" + Thread.*currentThread*().getName());

lock.unlock();

}

}

**DeadLock**

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order.

**public** **class** DeadLock {

**static** Object *lock1* = **new** Object();

**static** Object *lock2* = **new** Object();

**public** **static** **void** main(**final** String[] args) {

Thread t1 = **new** Thread(() -> {

**synchronized** (*lock1*) {

System.***out***.println("Thread 1 accquired lock1");

System.***out***.println("Thread 1 waiting for lock2");

**synchronized** (*lock2*) {

System.***out***.println("Thread 1 accquired lock2");

}

}

});

Thread t2 = **new** Thread(() -> {

**synchronized** (*lock2*) {

System.***out***.println("Thread 2 accquired lock2");

System.***out***.println("Thread 2 waiting for lock1");

**synchronized** (*lock1*) {

System.***out***.println("Thread 2 accquired lock1");

}

}

});

t1.start();

t2.start();

}

}