Concurrency and Multi-threading

1) What is difference between start and run method in Java Thread?
This thread interview question is also ask as if start() method eventually call run() method than why do you need to call start() method, why not call run() method directly. well reason is that because start method creates a new thread and call the code written inside run method on new thread while calling run method executes that code on same thread. see start vs run method in Java for more details.

2) Write code to avoid deadlock in Java where n threads are accessing n shared resources?
This is a classic Java multithreading interview questions, which appears on almost every list of Java thread questions. This question is based on risk and issues faced by parallel programs without proper synchronization or incorrect synchronization. This question explores concept of looking and best practices on acquiring and releasing lock on shared resource. By the way it's been covered on many places as well and I suggest reading How to prevent deadlock in Java, not only for detail answer of this Java multithreading question but also to learn how to prevent deadlock in Java.

3) Which one is better to implement thread in Java ? extending Thread class or implementing Runnable?
Well this is another frequently asked questions on any Java thread interview. Essentially these are two way to implement Thread in Java and by extending class you are using your chance to extend one any only one class as Java does not support multiple inheritance, by implementing Runnable interface you can still extend another class. So extending Runnable or even Callable is better choice. see Runnable vs Thread class in Java for more answers on this questions. Given it's simplicity and fact based nature, this question mostly appear on either telephonic round or initial screening rounds. Key points to mention, while answering this question includes, multiple inheritance at class level and separation of defining a task and execution of task. Runnable only represent a task, while Thread represent both task and it's execution.

4) What is Busy Spinning? Why you will use Busy Spinning as wait strategy?
This is really an advanced concurrency interview questions in Java and only asked to experienced and senior Java developers, with lots of concurrent coding experience under belt. By the way concept of busy spinning is not new, but it's usage with multi core processor has risen recently. 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 micro second. 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. If you can answer this question, that rest assure of a good impression.

5) What is difference between CountDownLatch and CyclicBarrier in Java?
CountDownLatch and CyclicBarrier in Java are two important concurrency utility which is added on Java 5 Concurrency API. Both are used to implement scenario, where one thread has to wait for other thread before starting processing but there is difference between them. Key point to mention, while answering this question is that CountDownLatch is not reusable once count reaches to zero, while CyclicBarrier can be reused even after barrier is broken. You can also see my previous article difference between CyclicBarrier and CountDownLatch in Java for more detailed answer of this concurrency interview question and a real life example of where to use these concurrency utilities.

6) Difference between wait and sleep in Java?
One more classic Java multithreading question from telephonic round of interviews. Key point to mention while answering this question is to mention that wait will release lock and must be called from synchronized context, while sleep will only pause thread for some time and keep the lock. By the way, both throws IntrupptedException and can be interrupted, which can lead to some followup questions like, can we awake a sleeping or waiting thread in Java? You can also read detailed answer on my post of same title here.

7) How do you solve producer consumer problem in Java?
One of my favorite questions during any Java multithreading interview, Almost half of the concurrency problems can be categorized in producer consumer pattern. There are basically two ways to solve this problem in Java, One by using wait and notify method and other by using BlockingQueue in Java. later is easy to implement and good choice if you are coding in Java 5. Key points to mention, while answering this question is threadsafety and blocking nature of BlockingQueue and how that helps, while writing concurrent code. You can also expect lots of followup questions including, what happen if you have multiple producer or multiple consumer, what will happen if producer is faster than consumer thread or vice-versa. You can also see this link for example of how to code producer consumer design in Java using blocking queue

8) Why ConcurrentHashMap is faster than Hashtable in Java?
ConcurrentHashMap is introduced as alternative of Hashtable in Java 5, it is faster because of it's design. ConcurrentHashMap divides whole map into different segments and only lock a particular segment during update operation, instead of Hashtable, which locks whole Map. ConcurrentHashMap also provides lock free read, which is not possible in Hashtable, because of this and lock striping, ConcurrentHashMap is faster than Hashtable, especially when number of reader is more than number of writers. In

order to better answer this popular Java concurrency interview questions, I suggest reading my post about internal working of ConcurrentHashMap in Java.

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

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.

10) How do you share data between two threads in Java?

One more Java multithreading question from telephonic round of interview. You can share data between thread by using shared object or shared data structures like Queue. Depending upon, what you are using, you need to provide thread-safety guarantee, and one way of providing thread-safety is using synchronized keyword. If you use concurrent collection classes from Java 5 e.g. BlockingQueue, you can easily share data without being bothered about thread safety and inter thread communication. I like this thread question, because of it's simplicity and effectiveness. This also leads further follow-up questions on issues which arises due to sharing data between threads e.g. race conditions.

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

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.

12) What is ReadWriteLock in Java? What is benefit of using ReadWriteLock in Java?

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.

Difference between start and run in Java Thread

start vs run method in Java with ExampleSo what is difference between start and run method? 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. Most of the time calling run() is bug or programming mistake because caller has intention of calling start() to create new thread and this error can be detect by many static code coverage tools like findbugs. If you want to perform time consuming task than always call start() method otherwise your main thread will stuck while performing time consuming task if you call run() method directly. 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.

Code Example of start vs run method

Here is a simple code example which prints name of Thread which executes run() method of Runnable task. Its clear that if you call start() method a new Thread executes Runnable task while if you directly call run() method task, current thread which is main in this case will execute the task.

```
public class StartVsRunCall{
    public static void main(String args[]) {
        //creating two threads for start and run method call
        Thread startThread = new Thread(new Task("start"));
        Thread runThread = new Thread(new Task("run"));
        startThread.start(); //calling start method of Thread - will execute in new Thread
        runThread.run();  //calling run method of Thread - will execute in current Thread
    }
    /*
     * Simple Runnable implementation
     */
    private static class Task implements Runnable{
        private String caller;
        public Task(String caller){
            this.caller = caller;
        }
        @Override
```

```
        public void run() {
            System.out.println("Caller: "+ caller + " and code on this Thread is executed by :
" + Thread.currentThread().getName());
        }
    }
}
```

Output:
Caller: start and code on this Thread is executed by : Thread-0
Caller: run and code on this Thread is executed by : main

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.

How to avoid deadlock in Java Threads

How to avoid deadlock in Java is one of the question which is flavor of the season for multithreading , asked more at a senior level and with lots of follow up questions , though question looks very basic but most of developer get stuck once you start going deep.
Questions starts with "What is deadlock ?"
Answer is simple , when two or more threads waiting for each other to release lock and get stuck for infinite time , situation is called deadlock . it will only happen in case of multitasking.

How do you detect deadlock in Java ?

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 , jconsole will show you exactly which threads are get locked and on which object.
Once you answer this , they may ask you to write code which will result in deadlock ? here is one of my version

```
public void method1(){
	synchronized(String.class){
		System.out.println("Aquired lock on String.class object");
		synchronized (Integer.class) {
			System.out.println("Aquired lock on Integer.class object");
		}
	}
}
public void method2(){
	synchronized(Integer.class){
		System.out.println("Aquired lock on Integer.class object");
		synchronized (String.class) {
			System.out.println("Aquired lock on String.class object");
		}
	}
}
```

If method1() and method2() both will be called by two or many threads , there is a good chance of deadlock because if thead 1 aquires lock on Sting object while executing method1() and thread 2 acquires lock on Integer object while executing method2() both will be waiting for each other to release lock on Integer and String to proceed further which will never happen.

Now interviewer comes to final part , one of the most important in my view ,

How to fix deadlock ? or How to avoid deadlock in Java ?

If you have looked above code carefully you may have figured out that real reason for deadlock is not multiple threads but the way they access lock , if you provide an ordered access then problem will be resolved , here is the fixed version.

```
public void method1(){
	synchronized(Integer.class){
		System.out.println("Aquired lock on Integer.class object");
		synchronized (String.class) {
			System.out.println("Aquired lock on String.class object");
		}
	}
}
public void method2(){
	synchronized(Integer.class){
		System.out.println("Aquired lock on Integer.class object");
		synchronized (String.class) {
			System.out.println("Aquired lock on String.class object");
		}
	}
```

}
Now there would not be any deadlock because both method is accessing lock on Integer and String object in same order . so if thread A acquires lock on Integer object , thread B will not proceed until thread A releases Integer lock , same way thread A will not be blocked even if thread B holds String lock because now thread B will not expect thread A to release Integer lock to proceed further.

What is CountDownLatch in Java

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 along with other concurrent utilities like CyclicBarrier, Semaphore, ConcurrentHashMap and BlockingQueue in java.util.concurrent package. In this Java concurrency tutorial we will what is CountDownLatch in Java, How CountDownLatch works in Java, an example of CountDownLatch in Java and finally some worth noting points about this concurrent utility. You can also implement same functionality using wait and notify mechanism in Java but it requires lot of code and getting it write in first attempt is tricky, With CountDownLatch it can be done in just few lines. CountDownLatch also allows flexibility on number of thread for which main thread should wait, It can wait for one thread or n number of thread, there is not much change on code. Key point is that you need to figure out where to use CountDownLatch in Java application which is not difficult if you understand What is CountDownLatch in Java, What does CountDownLatch do and How CountDownLatch works in Java.

How CountDownLatch works in Java

CountDownLatch Example in Java 5 6 7Now we know What is CountDownLatch in Java, its time to find out How CountDownLatch works in Java. CountDownLatch works in latch principle, main thread will wait until Gate is open. One thread waits for n number of threads specified while creating CountDownLatch in Java. Any thread, usually main thread of application, which calls CountDownLatch.await() will wait until count reaches zero or its interrupted by another Thread. All other thread are required to do count down by calling CountDownLatch.countDown() once they are completed or ready to the job. as soon as count reaches zero, Thread awaiting starts running. One of the disadvantage of CountDownLatch is that its not reusable once count reaches to zero you can not use CountDownLatch any more, but don't worry Java concurrency API has another concurrent utility called CyclicBarrier for such requirements.

CountDownLatch Exmaple in Java

In this section we will see a full featured real world example of using CountDownLatch in Java. In following CountDownLatch example, Java program requires 3 services namely CacheService, AlertService and ValidationService to be started and ready before application can handle any request and this is achieved by using CountDownLatch in Java.

```
import java.util.Date;
import java.util.concurrent.CountDownLatch;
import java.util.logging.Level;
import java.util.logging.Logger;
/**
 * Java program to demonstrate How to use CountDownLatch in Java. CountDownLatch
is
 * useful if you want to start main processing thread once its dependency is completed
 * as illustrated in this CountDownLatch Example
 * @author Javin Paul
 */
public class CountDownLatchDemo {
    public static void main(String args[]) {
        final CountDownLatch latch = new CountDownLatch(3);
        Thread cacheService = new Thread(new Service("CacheService", 1000, latch));
        Thread alertService = new Thread(new Service("AlertService", 1000, latch));
        Thread validationService = new Thread(new Service("ValidationService", 1000,
latch));
        cacheService.start(); //separate thread will initialize CacheService
        alertService.start(); //another thread for AlertService initialization
        validationService.start();
        // application should not start processing any thread until all service is up
        // and ready to do there job.
        // Countdown latch is idle choice here, main thread will start with count 3
        // and wait until count reaches zero. each thread once up and read will do
        // a count down. this will ensure that main thread is not started processing
        // until all services is up.
        //count is 3 since we have 3 Threads (Services)
        try{
            latch.await();  //main thread is waiting on CountDownLatch to finish
            System.out.println("All services are up, Application is starting now");
        }catch(InterruptedException ie){
            ie.printStackTrace();
        }
    }
```

```
}
/**
 * Service class which will be executed by Thread using CountDownLatch synchronizer.
 */
class Service implements Runnable{
    private final String name;
    private final int timeToStart;
    private final CountDownLatch latch;
    public Service(String name, int timeToStart, CountDownLatch latch){
        this.name = name;
        this.timeToStart = timeToStart;
        this.latch = latch;
    }
    @Override
    public void run() {
        try {
            Thread.sleep(timeToStart);
        } catch (InterruptedException ex) {
            Logger.getLogger(Service.class.getName()).log(Level.SEVERE, null, ex);
        }
        System.out.println( name + " is Up");
        latch.countDown(); //reduce count of CountDownLatch by 1
    }
}
```

Output:

```
ValidationService is Up
AlertService is Up
CacheService is Up
All services are up, Application is starting now
```

By looking at output of this CountDownLatch example in Java, you can see that Application is not started until all services started by individual Threads are completed.

When should we use CountDownLatch in Java :

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 as shown in our CountDownLatch example.

CountDownLatch in Java – Things to remember

Few points about Java CountDownLatch which is worth remembering:
1) 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.
2) Main Thread wait on Latch by calling CountDownLatch.await() method while other thread calls CountDownLatch.countDown() to inform that they have completed.

That's all on What is CountDownLatch in Java, What does CountDownLatch do in Java, How CountDownLatch works in Java along with a real life CountDownLatch example in Java. This is a very useful concurrency utility and if you master when to use CountDownLatch and how to use CountDownLatch you will be able to reduce good amount of complex concurrency control code written using wait and notify in Java.

What is CyclicBarrier in Java

CyclicBarrier in Java is a synchronizer introduced in JDK 5 on java.util.Concurrent package along with other concurrent utility like Counting Semaphore, BlockingQueue, ConcurrentHashMap etc. CyclicBarrier is similar to CountDownLatch which we have seen in last article What is CountDownLatch in Java and allows multiple threads to wait for each other (barrier) before proceeding. Difference between CountDownLatch and CyclicBarrier is a also very popular multi-threading interview question in Java. 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 other thread. In this Java concurrency tutorial we will see What is CyclicBarrier in Java and an example of CyclicBarrier on which three Threads will wait for each other before proceeding further.

Difference between CountDownLatch and CyclicBarrier in Java

In our last article we have see how CountDownLatch can be used to implement multiple threads waiting for each other. If you look at CyclicBarrier it also the does the same thing but there is a different you can not reuse CountDownLatch once count reaches zero while you can reuse 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 start-up time and CyclicBarrier can be used to in case of recurrent event e.g. concurrently calculating solution of big problem etc. If you like to learn more about threading and concurrency in Java you can also check my post on When to use Volatile

variable in Java and How Synchronization works in Java.

CyclicBarrier in Java – Example

Java CyclicBarrier Example and Tutorial programNow we know what is CyclicBarrier in Java and it's time to see example of CyclicBarrier in Java. Here is a simple example of CyclicBarrier in Java on which we initialize CyclicBarrier with 3 parties, means in order to cross barrier, 3 thread needs to call await() method. each thread calls await method in short duration but they don't proceed until all 3 threads reached barrier, once all thread reach barrier, barrier gets broker and each thread started there execution from that point. Its much clear with the output of following example of CyclicBarrier in Java:

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.logging.Level;
import java.util.logging.Logger;
/**
 * Java program to demonstrate how to use CyclicBarrier in Java. CyclicBarrier is a
 * new Concurrency Utility added in Java 5 Concurrent package.
 *
 * @author Javin Paul
 */
public class CyclicBarrierExample {
    //Runnable task for each thread
    private static class Task implements Runnable {
        private CyclicBarrier barrier;
        public Task(CyclicBarrier barrier) {
            this.barrier = barrier;
        }
        @Override
        public void run() {
            try {
                System.out.println(Thread.currentThread().getName() + " is waiting on
barrier");
                barrier.await();
                System.out.println(Thread.currentThread().getName() + " has crossed the
barrier");
            } catch (InterruptedException ex) {

Logger.getLogger(CyclicBarrierExample.class.getName()).log(Level.SEVERE, null,
ex);
            } catch (BrokenBarrierException ex) {
```

```
Logger.getLogger(CyclicBarrierExample.class.getName()).log(Level.SEVERE, null,
ex);
            }
        }
    }

    public static void main(String args[]) {
        //creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call await()
        final CyclicBarrier cb = new CyclicBarrier(3, new Runnable(){
            @Override
            public void run(){
                //This task will be executed once all thread reaches barrier
                System.out.println("All parties are arrived at barrier, lets play");
            }
        });
        //starting each of thread
        Thread t1 = new Thread(new Task(cb), "Thread 1");
        Thread t2 = new Thread(new Task(cb), "Thread 2");
        Thread t3 = new Thread(new Task(cb), "Thread 3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output:

```
Thread 1 is waiting on barrier
Thread 3 is waiting on barrier
Thread 2 is waiting on barrier

All parties are arrived at barrier, lets play

Thread 3 has crossed the barrier
Thread 1 has crossed the barrier
Thread 2 has crossed the barrier
```

When to use CyclicBarrier in Java

Given the nature of CyclicBarrier it can be very handy to implement map reduce kind of task similar to fork-join framework of Java 7, where a big task is broker 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 there task, Main thread or any other thread can add result from each

zone and print total population. You can use CyclicBarrier in Java :

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

Important point of CyclicBarrier in Java

1. CyclicBarrier can perform a completion task once all thread reaches to barrier, This can be provided while creating CyclicBarrier.
2. If CyclicBarrier is initialized with 3 parties means 3 thread needs to call await method to break the barrier.
3. Thread will block on await() until all parties reaches to barrier, another thread interrupt or await timed out.
4. 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)
```

5.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.

That's all on What is CyclicBarrier in Java , When to use CyclicBarrier in Java and a Simple Example of How to use CyclicBarrier in Java . We have also seen difference between CountDownLatch and CyclicBarrier in Java and got some idea where we can use CyclicBarrier in Java Concurrent code.

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)

Java 5 Semaphore Example codea 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
```

Some Scenario where Semaphore can be used:

1) 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.

2) To put a bound on collection classes. by using semaphore you can implement bounded collection whose bound is specified by counting semaphore.

That's all on Counting semaphore example in Java. Semaphore is real nice concurrent utility which can greatly simply design and implementation of bounded resource pool. Java 5 has added several useful concurrent utility and deserve a better attention than casual look.

Important points of Counting Semaphore in Java

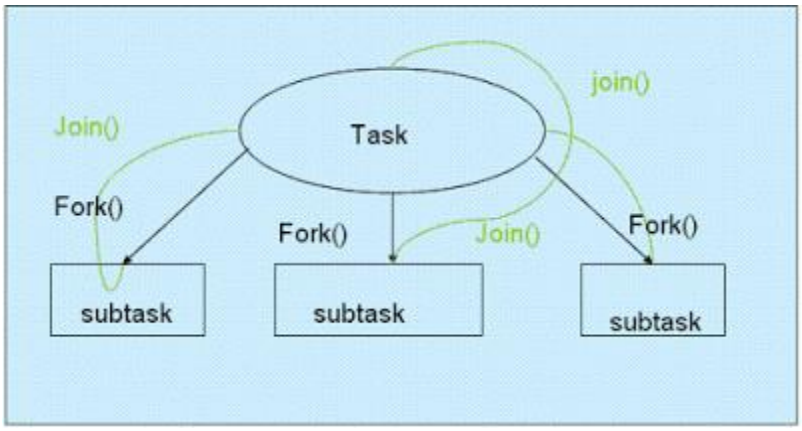
1. Semaphore class supports various overloaded version of tryAquire() method which acquires permit from semaphore only if its available during time of call.
2. Another worth noting method from Semaphore is acquireUninterruptibly() which is a blocking call and wait until a permit is available.

What is fork Join framework in Java:

Already popular project coin of JDK7 release has presented lot of good feature e.g automatic resource management, string in switch case, better exception handling in JDK7 etc. On of other important feature to note is fork join as name implies it divide one task into several small task as a new fork means child and join all the fork when all the sub-tasks complete. Fork/join tasks is “pure” in-memory algorithms in which no I/O operations come into picture.it is based on a work-stealing algorithm. Concept of fork join would be much clear by following diagram.

How fork join comes in existence:

Java's most attractive part is it makes things easier and easier.for doing things faster java has given us concurrency concept but dealing with concurrency is not easy because we have to deal with thread synchronization and shared data. When we have to work with small piece of code it is easy to handle synchronization and atomicity, but it becomes for complex when code base and number of threads increased, its really challenging where several threads are working together to accomplish a large task so again java has tried to make things easy and simplifies this concurrency using Executors and Thread Queue.



When we compare Executors with old Thread it has made management of concurrent task very easy and it work on divide and conquer algorithm and create sub-tasks and communicate with each other to complete. But the problem with the executors framework is that a Callable is free to submit a new sub-task to its executor and wait for its result in a synchronous or asynchronous fashion. The issue is that of parallelism: When a Callable waits for the result of another Callable, it is put in a waiting state, and thus wasting an opportunity to handle another Callable queued for execution.

To solve this issue java 7 has given the concept of parallelism.New fork-join framework has been added in java.util.concurrent package.New fork-join executor framework has been created which is responsible for creating one new task object which is again responsible for creating new sub-task object and waiting for sub-task to be completed.internally it maintains a thread pool and executor assign pending task to this thread pool to complete when one task is waiting for another task to complete. whole Idea of fork-join framework is to leverage multiple processors of advanced machine.

Fork Join framework in JDK7

How to code using fork-join framework:
Fork-join functionality is achieved by ForkjoinTask object, it has two method fork() and join () Method.

The fork() method allows a new ForkJoinTask to be launched from an existing one.
The join() method allows a ForkJoinTask to wait for the completion of another one.

Again ForkjoinTask object has been of two types: RecursiveAction and RecursiveTask which is more specialized form of this instance. While RecursiveAction represent executions that do not yield a return value, Instances of RecursiveTask yield return values.

Difference between CountDownLatch and CyclicBarrier in Java

Both CyclicBarrier and CountDownLatch are used to implement a scenario where one Thread waits for one or more Thread to complete there job before starts processing but there is one Difference between CountDownLatch and CyclicBarrier in Java which separates them apart and that is, you can not reuse same CountDownLatch instance once count reaches to zero and latch is open, on the other hand CyclicBarrier can be reused by resetting Barrier, Once barrier is broken.

Difference between CountDownLatch and CyclicBarrier in JavaA useful property of a CountDownLatch is that it doesn't require that threads calling countDown wait for the count to reach zero before proceeding, it simply prevents any thread from proceeding past an await until all threads could pass.

A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This barrier action is useful for updating shared-state before any of the parties continue.

The CyclicBarrier uses a fast-fail all-or-none breakage model for failed synchronization attempts: If a thread leaves a barrier point prematurely because of interruption, failure, or timeout, all other threads, even those that have not yet resumed from a previous await(), will also leave abnormally via BrokenBarrierException (or InterruptedException if they too were interrupted at about the same time).

Wait vs sleep in Java

Differences between wait and sleep method in Java Thread is one of the very old question asked in Java interviews. Though both wait and sleep puts thread on waiting state, they are completely different in terms of behaviour and use cases. Sleep is meant for introducing pause, releasing CPU and giving another thread opportunity to execute while wait is used for inter thread communication, by using wait() and notify() method two threads can communicate with each other which is key to solve many Concurrent problems like Produce consumer issue, Dining philosopher issue and to implement several Concurrency designs. In this tutorial we will see

What is wait method in Java
What is Sleep method in Java
Difference between wait and sleep in Java
Where to use wait and sleep in Java

What is wait and sleep method in Java

Difference between wait and sleep in Java
Wait method is defined in Object class and it available to all object, wait() method is always discussed along with its counterpart notify() and notifyAll() method and used in inter thread communication in Java. wait method puts a thread on wait by checking some condition like in Producer Consumer problem, producer thread should wait if Queue is full or Consumer thread should wait if Queue is empty. notify() method is used to wake up waiting thread by communicating that waiting condition is over now for example once producer thread puts an item on empty queue it can notify Consumer thread that Queue is not empty any more. On the other hand Sleep() method is used to introduce pause on Java application. You can put a Thread on sleep, where it does not do anything and relinquish the CPU for specified duration. When a Thread goes to Sleep it can be either wake up normally after sleep duration elapsed or it can be woken up abnormally by interrupting it.

Difference between Wait and Sleep method in Java Thread

In last section we saw what is wait and sleep method and in this section we will see what are differences between wait and sleep method in Java. As I told before apart from waiting they are completely different to each other:
1) First and most important difference between Wait and sleep method is that wait method must be called from synchronized context i.e. from synchronized method or block in Java. If you call wait method without synchronization, it will throw IllegalMonitorStateException in Java. On the other hand there is no requirement of synchronization for calling sleep method , you can call it normally.
2) Second worth noting difference between wait and sleep method is that, wait operates on Object and defined in Object class while sleep operates on current Thread and defined in java.lang.Thread class.

3) Third and another significant difference between wait and sleep in Java is that, wait() method releases the lock of object on which it has called, it does release other locks if it holds any while sleep method of Thread class does not release any lock at all.
4) wait method needs to be called from a loop in order to deal with false alarm i.e. waking even though waiting condition still holds true, while there is no such thing for sleep method in Java. its better not to call Sleep method from loop.
Here is code snippet for calling wait and sleep method in Java

```
synchronized(monitor)
while(condition == true){ monitor.wait())  //releases monitor lock
        Thread.sleep(100); //puts current thread on Sleep
```

5) One more difference between wait and sleep method which is not as significant as previous ones is that wait() is a non static method while sleep() is static method in Java.

Where to use wait and sleep method in Java

By reading properties and behavior of wait and sleep method it's clear that wait() method should be used in conjunction with notify() or notifyAll() method and intended for communication between two threads in Java while Thread.sleep() method is a utility method to introduce short pauses during program or thread execution. Given the requirement of synchronization for wait, it should not be used just to introduce pause or sleep in Java.

In summary wait and sleep method are completely different to each other and have different use cases. Use wait and notify method for intended thread communication while use sleep method for introducing small pause during thread execution.