**Q1. What is the difference between a process and a thread?**

## Ans=Both processes and threads are units of concurrency, but they have a fundamental difference: processes do not share a common memory, while threads do.

## Thread vs Process

1) A program in execution is often referred as process. A thread is a subset(part) of the process.

2) A process consists of multiple threads. A thread is a smallest part of the process that can execute concurrently with other parts(threads) of the process.

3) A process is sometime referred as task. A thread is often referred as lightweight process.

4) A process has its own address space. A thread uses the process’s address space and share it with the other threads of that process.

5)

Per process items | Per thread items

------------------------------|-----------------

Address space | Program counter

Global variables | Registers

Open files | Stack

Child processes | State

Pending alarms |

Signals and signal handlers |

Accounting information |

6) A thread can communicate with other thread (of the same process) directly by using methods like wait(), notify(), notifyAll(). A process can communicate with other process by using [inter-process communication](https://en.wikipedia.org/wiki/Inter-process_communication).

7) New threads are easily created. However the creation of new processes require duplication of the parent process.

8) Threads have control over the other threads of the same process. A process does not have control over the sibling process, it has control over its child processes only.

### Q=Difference between Wait and Sleep in Java?

**Ans=**Here is the list of difference between wait and sleep in Java :  
  
1) wait is called from synchronized context only while sleep can be called without synchronized block. see [Why to wait and notify needs to call from synchronized method](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html) for more detail.  
  
2) waiting thread can be awake by calling notify and notifyAll while sleeping thread can not be awakened by calling notify method.  
  
3) wait is normally done on condition, Thread wait until a condition is true while sleep is just to put your thread on sleep.

4) wait for release lock on an object while waiting while sleep doesn’t release lock while waiting.  
  
5) The wait() method  is called on an Object on which the synchronized block is locked, while sleep is called on the Thread.

## Q: Difference between yield and sleep in Java

### Ans=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.  Question 10) Suppose you have a class which you serialized it and stored in persistence and later modified that class to add a new field. What will happen if you deserialize the object already serialized?

It depends on whether class has its own serialVersionUID or not. As we know from above question that if we don't provide serialVersionUID in our code java compiler will generate it and normally it’s [equal to hashCode of object](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html). by adding any new field there is chance that new serialVersionUID generated for that class version is not the same of already serialized object and in this case Java Serialization API will [throw](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html) java.io.InvalidClassException and this is the reason its recommended to have your own serialVersionUID in code and make sure to keep it same always for a single class.

## **Q=Main Differences Between Locks and Synchronized Blocks?**

Ans=The main differences between a Lock and a synchronized block are:

* A synchronized block makes no guarantees about the sequence in which threads waiting to entering it are granted access.
* You cannot pass any parameters to the entry of a synchronized block. Thus, having a timeout trying to get access to a synchronized block is not 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.

**Q=Why wait, notify and notifyAll is defined in Object Class and not on Thread class in Java**  
Ans=  
1) 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](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html) like race condition while wait and notify are communication mechanism between two thread.

2 )**Locks are made available on per Object basis**, which is another reason wait and notify is declared in Object class rather then Thread class.

3) 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.

**Q=Why wait notify and notifyAll called from synchronized block or method in Java?**

Ans=We use wait(), notify(), or notifyAll() method mostly for [inter-thread communication](http://javarevisited.blogspot.sg/2013/12/inter-thread-communication-in-java-wait-notify-example.html) in Java. One thread is waiting after checking a condition e.g. In the classic Producer-Consumer problem, the Producer thread waits if the buffer is full and Consumer thread notify Producer thread after it creates a space in the buffer by consuming an element.

Calling [notify() or notifyAll()](http://java67.blogspot.com/2013/03/difference-between-wait-vs-notify-vs-notifyAll-java-thread.html) methods issues a notification to a single or multiple thread that a condition has changed and once notification thread leaves synchronized block, all the threads which are waiting fight for object lock on which they are waiting and lucky **thread returns from wait() method after reacquiring the lock** and proceed further.  
  
Let’s divide this whole operation into steps to see a possibility of *race condition between wait() and notify() method in Java*, we will use [Produce Consumer thread example](http://javarevisited.blogspot.com/2015/06/java-lock-and-condition-example-producer-consumer.html) to understand the scenario better:

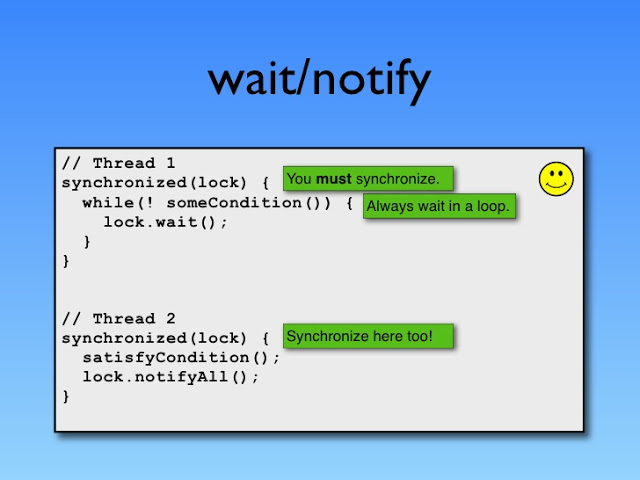
   1. The Producer thread tests the condition (buffer is full or not) and confirms that it must wait (after finding buffer is full).

   2. The Consumer thread sets the condition after consuming an element from a buffer.

   3. The Consumer thread calls the notify () method; this goes unheard since the Producer thread is not yet waiting.

   4. The Producer thread calls the wait () method and goes into waiting state.

So due to [race condition](http://javarevisited.blogspot.com/2012/02/what-is-race-condition-in.html) here we potential lost a notification and if we use buffer or just one element Produce thread will be waiting forever and your program will hang.

[](https://2.bp.blogspot.com/-7yTGxlOBfu4/VvUdKBO3RhI/AAAAAAAAFYE/tSNS9y3Gb2sGPE4nrnTvllraf0lYOt-SA/s1600/Wait+and+Notify+Java+Concurrency.jpg)

Now let's think how does this potential race condition get resolved? This race condition is resolved by using [synchronized keyword](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) and locking provided by Java. In order to call the **wait (), notify () or notifyAll () methods in Java**, we must have obtained the lock for the object on which we're calling the method.  
  
Since the wait() method in Java also releases the lock prior to waiting and reacquires the lock prior to returning from the [wait() method](http://javarevisited.blogspot.com/2011/12/difference-between-wait-sleep-yield.html), we must use this lock to ensure that checking the condition (buffer is full or not) and setting the condition (taking element from buffer) is atomic which can be achieved by using synchronized method or block in Java.

I am not sure if this is what interviewer was actually expecting but this what I thought would at least make sense, please correct me If I wrong and let us know if there is any other convincing reason of calling wait(), notify() or notifyAll method in Java.

Just to summarize we call **wait**(), **notify**() or **notifyAll**method in Java from synchronized method or synchronized block in Java to avoid:

1) **IllegalMonitorStateException in Java** which will occur if we don't call wait (), notify () or notifyAll () method from synchronized context.

2) **Any potential race condition between wait and notify method in Java**.

**Q4. What is the difference between the *Runnable* and *Callable* interfaces? How are they used?**

The *Runnable* interface has a single*run* method. It represents a unit of computation that has to be run in a separate thread. The *Runnable* interface does not allow this method to return value or to throw unchecked exceptions.

The *Callable* interface has a single *call* method and represents a task that has a value. That’s why the *call* method returns a value. It can also throw exceptions. *Callable* is generally used in *ExecutorService* instances to start an asynchronous task and then call the returned *Future* instance to get its value.

**Q=What is deadlock?**"

when two or more threads are waiting for each other to release lock and get stuck for infinite time, situation is called deadlock .

# Q= [What is CountDownLatch in Java](http://javarevisited.blogspot.in/2012/07/countdownlatch-example-in-java.html)

Ans= CountDownLatch in Java is a kind of synchronizer which allows one Thread  to wait for one or more Threads before starts processing.

**How CountDownLatch works in Java**

[CountDownLatch Example in Java 5 6 7](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html)Now 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](http://javarevisited.blogspot.sg/2012/02/why-wait-notify-and-notifyall-is.html) 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](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html) 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](http://javarevisited.blogspot.sg/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html) 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](http://javarevisited.blogspot.sg/2011/09/servlet-interview-questions-answers.html) 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

**Q=difference between Semaphore and Mutex?**

Ans=

1. Semaphores – Restrict the number of threads that can access a resource. Example, limit max 10 connections to access a file simultaneously.
2. Mutex – Only one thread to access a resource at once. Example, when a client is accessing a file, no one else should have access the same file at the same time.

# Q=**Semaphore in Java?**

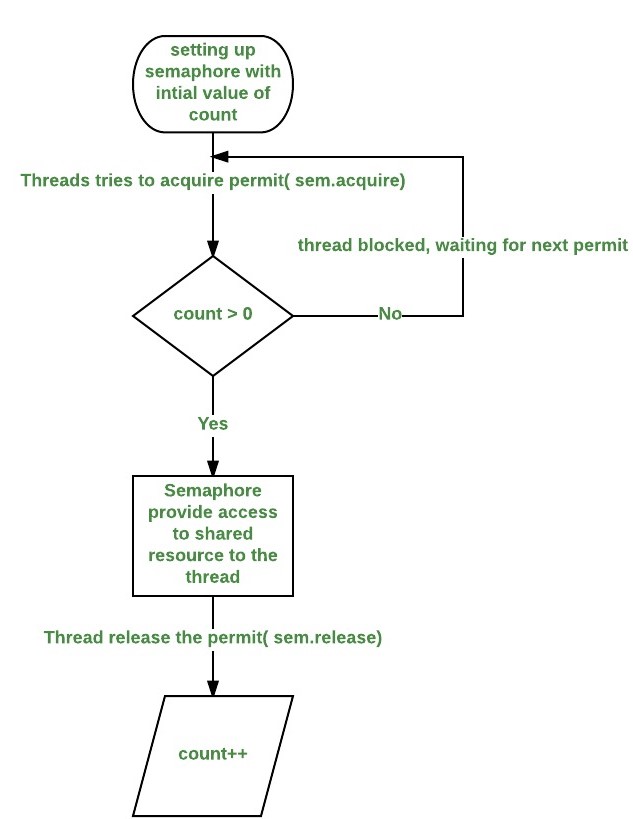
Ans=A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied.

**Working of semaphore**

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit.

* If the semaphore’s count is greater than zero, then the thread acquires a permit, which causes the semaphore’s count to be decremented.
* Otherwise, the thread will be blocked until a permit can be acquired.
* When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore’s count to be incremented.
* If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

Java provide **Semaphore** class in *java.util.concurrent* package that implements this mechanism, so you don’t have to implement your own semaphores.

**Flow Diagram :**  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/d2.jpeg)

**Constructors in Semaphore class :** There are two constructors in Semaphore class.

Semaphore(int num)

Semaphore(int num, boolean how)

Here, num specifies the initial permit count. Thus, it specifies the number of threads that can access a shared resource at any one time. If it is one, then only one thread can access the resource at any one time. By default, all waiting threads are granted a permit in an undefined order. By setting how to true, you can ensure that waiting threads are granted a permit in the order in which they requested access.

**Using Semaphores as Locks(**[**preventing race condition**](http://practice.geeksforgeeks.org/problems/what-is-race-condition)**)**

We can use a semaphore to lock access to a resource, each thread that wants to use that resource must first call acquire( ) before accessing the resource to acquire the lock. When the thread is done with the resource, it must call release( ) to release lock. Here is an example that demonstrate this:

|  |
| --- |
| // java program to demonstrate  // use of semaphores Locks  import java.util.concurrent.\*;   //A shared resource/class.  class Shared  {      static int count = 0;  }    class MyThread extends Thread  {      Semaphore sem;      String threadName;      public MyThread(Semaphore sem, String threadName)      {          super(threadName);          this.sem = sem;          this.threadName = threadName;      }        @Override      public void run() {            // run by thread A          if(this.getName().equals("A"))          {              System.out.println("Starting " + threadName);              try              {                  // First, get a permit.                  System.out.println(threadName + " is waiting for a permit.");                    // acquiring the lock                  sem.acquire();                    System.out.println(threadName + " gets a permit.");                    // Now, accessing the shared resource.                  // other waiting threads will wait, until this                  // thread release the lock                  for(int i=0; i < 5; i++)                  {                      Shared.count++;                      System.out.println(threadName + ": " + Shared.count);                        // Now, allowing a context switch -- if possible.                      // for thread B to execute                      Thread.sleep(10);                  }              } catch (InterruptedException exc) {                      System.out.println(exc);                  }                    // Release the permit.                  System.out.println(threadName + " releases the permit.");                  sem.release();          }            // run by thread B          else          {              System.out.println("Starting " + threadName);              try              {                  // First, get a permit.                  System.out.println(threadName + " is waiting for a permit.");                    // acquiring the lock                  sem.acquire();                    System.out.println(threadName + " gets a permit.");                    // Now, accessing the shared resource.                  // other waiting threads will wait, until this                  // thread release the lock                  for(int i=0; i < 5; i++)                  {                      Shared.count--;                      System.out.println(threadName + ": " + Shared.count);                        // Now, allowing a context switch -- if possible.                      // for thread A to execute                      Thread.sleep(10);                  }              } catch (InterruptedException exc) {                      System.out.println(exc);                  }                  // Release the permit.                  System.out.println(threadName + " releases the permit.");                  sem.release();          }      }  }    // Driver class  public class SemaphoreDemo  {      public static void main(String args[]) throws InterruptedException      {          // creating a Semaphore object          // with number of permits 1          Semaphore sem = new Semaphore(1);            // creating two threads with name A and B          // Note that thread A will increment the count          // and thread B will decrement the count          MyThread mt1 = new MyThread(sem, "A");          MyThread mt2 = new MyThread(sem, "B");            // stating threads A and B          mt1.start();          mt2.start();            // waiting for threads A and B          mt1.join();          mt2.join();            // count will always remain 0 after          // both threads will complete their execution          System.out.println("count: " + Shared.count);      }  } |

Run on IDE

Output:

Starting A

Starting B

B is waiting for a permit.

B gets a permit.

A is waiting for a permit.

B: -1

B: -2

B: -3

B: -4

B: -5

B releases the permit.

A gets a permit.

A: -4

A: -3

A: -2

A: -1

A: 0

A releases the permit.

count: 0

**Note :**The output can be different in different executions of above program, but final value of countvariable will always remain 0.

**Explanation of above program :**

* The program uses a semaphore to control access to the count variable, which is a static variable within the Shared class. Shared.count is incremented five times by thread A and decremented five times by thread B.To prevent these two threads from accessing Shared.count at the same time, access is allowed only after a permit is acquired from the controlling semaphore. After access is complete, the permit is released. In this way, only one thread at a time will access Shared.count, as the output shows.
* Notice the call to sleep( ) within run( ) method inside MyThread class. It is used to “prove” that accesses to Shared.count are synchronized by the semaphore. In run( ), the call to sleep( ) causes the invoking thread to pause between each access to Shared.count. This would normally enable the second thread to run. However, because of the semaphore, the second thread must wait until the first has released the permit, which happens only after all accesses by the first thread are complete. Thus, Shared.count is first incremented five times by thread A and then decremented five times by thread B. The increments and decrements are not intermixed at assembly code.
* Without the use of the semaphore, accesses to Shared.count by both threads would have occurred simultaneously, and the increments and decrements would be intermixed.To confirm this, try commenting out the calls to acquire( ) and release( ). When you run the program, you will see that access to Shared.count is no longer synchronized, thus you will not always get count value 0.

## **Counting Semaphore**

The Semaphore implementation in the previous section does not count the number of signals sent to it by take() method calls. We can change the Semaphore to do so. This is called a counting semaphore. Here is a simple implementation of a counting semaphore:

public class CountingSemaphore {

private int signals = 0;

public synchronized void take() {

this.signals++;

this.notify();

}

public synchronized void release() throws InterruptedException{

while(this.signals == 0) wait();

this.signals--;

}

}

## Bounded Semaphore

The CoutingSemaphore has no upper bound on how many signals it can store. We can change the semaphore implementation to have an upper bound, like this:

public class BoundedSemaphore {

private int signals = 0;

private int bound = 0;

public BoundedSemaphore(int upperBound){

this.bound = upperBound;

}

public synchronized void take() throws InterruptedException{

while(this.signals == bound) wait();

this.signals++;

this.notify();

}

public synchronized void release() throws InterruptedException{

while(this.signals == 0) wait();

this.signals--;

this.notify();

}

}

Notice how the take() method now blocks if the number of signals is equal to the upper bound. Not until a thread has called release() will the thread calling take() be allowed to deliver its signal, if the BoundedSemaphore has reached its upper signal limit.

## Using Semaphores as Locks

It is possible to use a bounded semaphore as a lock. To do so, set the upper bound to 1, and have the call to take() and release() guard the critical section. Here is an example:

BoundedSemaphore semaphore = new BoundedSemaphore(1);

...

semaphore.take();

try{

//critical section

} finally {

semaphore.release();

}

In contrast to the signaling use case the methods take() and release() are now called by the same thread. Since only one thread is allowed to take the semaphore, all other threads calling take() will be blocked until release() is called. The call to release() will never block since there has always been a call to take() first.

# Q=Starvation and Livelock

Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

## Starvation

*Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Or

Starvation is a situation when a thread is in waiting state from long period because it not getting access of shared resources or because higher priority threads are coming.

## Livelock

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

# Q=**CountDownLatch in Java**

Ans=CountDownLatch is used to make sure that a task waits for other threads before it starts. To understand its application, let us consider a server where the main task can only start when all the required services have started.

**Working of CountDownLatch:**  
When we create an object of CountDownLatch, we specify the number if threads it should wait for, all such 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, the waiting task starts running.

**Example of CountDownLatch in JAVA:**

|  |
| --- |
| /\* Java Program to demonstrate how to use CountDownLatch,     Its used when a thread needs to wait for other threads     before starting its work. \*/  import java.util.concurrent.CountDownLatch;    public class CountDownLatchDemo  {      public static void main(String args[]) throws InterruptedException      {          // Let us create task that is going to wait for four          // threads before it starts          CountDownLatch latch = new CountDownLatch(4);            // Let us create four worker threads and start them.          Worker first = new Worker(1000, latch, "WORKER-1");          Worker second = new Worker(2000, latch, "WORKER-2");          Worker third = new Worker(3000, latch, "WORKER-3");          Worker fourth = new Worker(4000, latch, "WORKER-4");          first.start();          second.start();          third.start();          fourth.start();            // The main task waits for four threads          latch.await();            // Main thread has started          System.out.println(Thread.currentThread().getName() +                             " has finished");      }  }    // A class to represent threads for which the main thread  // waits.  class Worker extends Thread  {      private int delay;      private CountDownLatch latch;        public Worker(int delay, CountDownLatch latch,                                       String name)      {          super(name);          this.delay = delay;          this.latch = latch;      }        @Override      public void run()      {          try          {              Thread.sleep(delay);              latch.countDown();              System.out.println(Thread.currentThread().getName()                                 + " finished");          }          catch (InterruptedException e)          {              e.printStackTrace();          }      }  } |

Run on IDE

**Output:**

WORKER-1 finished

WORKER-2 finished

WORKER-3 finished

WORKER-4 finished

main has finished

**Facts about CountDownLatch:**

1. Creating an object of CountDownLatch by passing an int to its constructor (the count), is actually number of invited parties (threads) for an event.
2. The thread, which is dependent on other threads to start processing, waits on until every other thread has called count down. All threads, which are waiting on await() proceed together once count down reaches to zero.
3. countDown() method decrements the count and await() method blocks until count == 0

## **Some Important notes for 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.

# Q=Java Fork and Join using ForkJoinPool

Ans=The fork/join framework was presented in Java 7. It provides tools to help speed up parallel processing by attempting to use all available processor cores – which is accomplished **through a divide and conquer approach**.

**the framework first “forks”**, recursively breaking the task into smaller independent subtasks until they are simple enough to be executed asynchronously.

After that, **the “join” part begins**, in which results of all subtasks are recursively joined into a single result, or in the case of a task which returns void, the program simply waits until every subtask is executed.

To provide effective parallel execution, the fork/join framework uses a pool of threads called the *ForkJoinPool*,which manages worker threads of type *ForkJoinWorkerThread*.

## **2.** ForkJoinPool

The ForkJoinPool is the heart of the framework. It is an implementation of the [*ExecutorService*](http://www.baeldung.com/java-executor-service-tutorial) that manages worker threads and provides us with tools to get information about the thread pool state and performance.

Worker threads can execute only one task at the time, but the ForkJoinPool doesn’t create a separate thread for every single subtask. Instead, each thread in the pool has its own double-ended queue (or [deque](https://en.wikipedia.org/wiki/Double-ended_queue), pronounced deck) which stores tasks.

This architecture is vital for balancing the thread’s workload with the help of the**work-stealing algorithm.**

### ****2.1. Work Stealing Algorithm****

**Simply put – free threads try to “steal” work from deques of busy threads.**

By default, a worker thread gets tasks from the head of its own deque. When it is empty, the thread takes a task from the tail of the deque of another busy thread or from the global entry queue, since this is where the biggest pieces of work are likely to be located.

### ****2.2.****ForkJoinPool ****Instantiation****

In Java 8, the most convenient way to get access to the instance of the ForkJoinPool is to use its static method [*commonPool*](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html#commonPool--)(). As its name suggests, this will provide a reference to the common pool, which is a default thread pool for every ForkJoinTask.

|  |  |
| --- | --- |
| 1 | ForkJoinPool commonPool = ForkJoinPool.commonPool(); |

The same behavior can be achieved in Java 7 by creating a *ForkJoinPool* and assigning it to a *public static* field of a utility class:

|  |  |
| --- | --- |
| 1 | public static ForkJoinPool forkJoinPool = new ForkJoinPool(2); |

## **3.** ForkJoinTask<V>

ForkJoinTask is the base type for tasks executed inside ForkJoinPool. In practice, one of its two subclasses should be extended: the RecursiveAction for void tasks and the RecursiveTask<V> for tasks that return a value.They both have an abstract method compute() in which the task’s logic is defined.

## RecursiveAction

A RecursiveAction is a task which does not return any value. It just does some work, e.g. writing data to disk, and then exits.

A RecursiveAction may still need to break up its work into smaller chunks which can be executed by independent threads or CPUs.

## RecursiveTask

A RecursiveTask is a task that returns a result. It may split its work up into smaller tasks, and merge the result of these smaller tasks into a collective result. The splitting and merging may take place on several levels. Here is a RecursiveTask example:

## **4. Submitting Tasks to the**ForkJoinPool

To submit tasks to the thread pool, few approaches can be used.

The **submit()**or ***execute()*** method (their use cases are the same):

|  |  |
| --- | --- |
| 1  2 | forkJoinPool.execute(customRecursiveTask);  int result = customRecursiveTask.join(); |

The***invoke()*** method forks the task and waits for the result, and doesn’t need any manual joining:

|  |  |
| --- | --- |
| 1 | int result = forkJoinPool.invoke(customRecursiveTask); |

The **invokeAll()** method is the most convenient way to submit a sequence of ForkJoinTasks to the ForkJoinPool. It takes tasks as parameters (two tasks, var args, or a collection), forks them returns a collection of Future objects in the order in which they were produced.

Alternatively, you can use separate **fork() and join()** methods. The fork() method submits a task to a pool, but it doesn’t trigger its execution. The join() method is be used for this purpose. In the case of RecursiveAction, the join() returns nothing butnull; for RecursiveTask<V>, it returns the result of the task’s execution:

|  |  |
| --- | --- |
| 1  2 | customRecursiveTaskFirst.fork();  result = customRecursiveTaskLast.join(); |

## Q=Java ThreadLocal

Ans=The ThreadLocal class in Java enables you to create variables that can only be read and written by the same thread. Thus, even if two threads are executing the same code, and the code has a reference to aThreadLocal variable, then the two threads cannot see each other's ThreadLocal variables.

## Creating a ThreadLocal

Here is a code example that shows how to create a ThreadLocal variable:

private ThreadLocal myThreadLocal = new ThreadLocal();

myThreadLocal.set("A thread local value");

String threadLocalValue = (String) myThreadLocal.get();

Generic ThreadLocal

private ThreadLocal<String> myThreadLocal = new ThreadLocal<String>();

myThreadLocal.set("Hello ThreadLocal");

String threadLocalValue = myThreadLocal.get();

## Initial ThreadLocal Value

private ThreadLocal myThreadLocal = new ThreadLocal<String>() {

@Override protected String initialValue() {

return "This is the initial value";

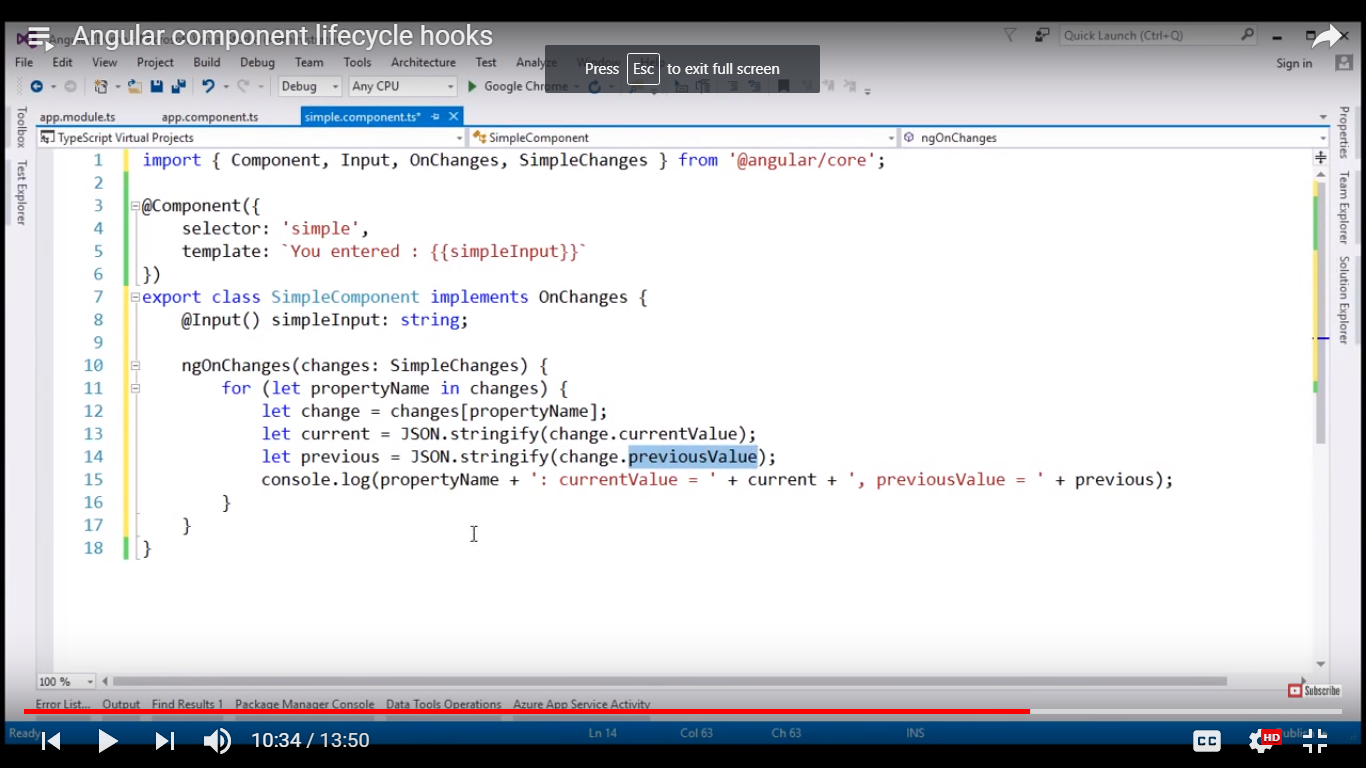
}

};

## **5. Do not use**ThreadLocal**with**ExecutorService

If we want to use an ExecutorService and submit a Runnable to it, using ThreadLocal will yield non-deterministic results – because we do not have a guarantee that every Runnable action for a given userId will be handled by the same thread every time it is executed.

Because of that, our ThreadLocal will be shared among different userIds. That’s why we should not use a TheadLocal together with ExecutorService. It should only be used when we have full control over which thread will pick which runnable action to execute.



# Q=[How to catch an Exception from a thread](https://stackoverflow.com/questions/6546193/how-to-catch-an-exception-from-a-thread)?

Ans=

|  |  |
| --- | --- |
| cepted | Use a Thread.UncaughtExceptionHandler.  Thread.UncaughtExceptionHandler h = new Thread.UncaughtExceptionHandler() {  public void uncaughtException(Thread th, Throwable ex) {  System.out.println("Uncaught exception: " + ex);  }  };  Thread t = new Thread() {  public void run() {  System.out.println("Sleeping ...");  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  System.out.println("Interrupted.");  }  System.out.println("Throwing exception ...");  throw new RuntimeException();  }  };  t.setUncaughtExceptionHandler(h);  t.start(); |

Q=How to create a class Thread safe?

Ans=Let's say we have very simple Java class MyClass.

public class MyClass {

private int number;

public MyClass(int number) {

this.number = number;

}

public int getNumber() {

return number;

}

public void setNumber(int number) {

this.number = number;

}

}

There are three ways to construct thread-safe Java class which has some state:

1. Make it truly immutable
2. public class MyClass {
3. private final int number;
4. public MyClass(int number) {
5. this.number = number;
6. }
7. public int getNumber() {
8. return number;
9. }

}

1. Make field number volatile.
2. public class MyClass {
3. private volatile int number;
4. public MyClass(int number) {
5. this.number = number;
6. }
7. public int getNumber() {
8. return number;
9. }
10. public void setNumber(int number) {
11. this.number = number;
12. }

}

1. Use a synchronized block. Classic version of this approach described in Chapter 4.3.5 of Java Concurrency in practice. And the funny thing about that it has an error in the example which is mentioned in a errata for this book.
2. public class MyClass {
3. private int number;
4. public MyClass(int number) {
5. setNumber(number);
6. }
7. public synchronized int getNumber() {
8. return number;
9. }
10. public synchronized void setNumber(int number) {
11. this.number = number;
12. }

}

<https://stackoverflow.com/questions/9622061/thread-safe-class-in-java-by-means-of-synchronized-blocks>

**Q=Race Condition?**

**Ans=** 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 *race condition* is a special condition that may occur inside a critical section. A *critical section* is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.

When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, the critical section is said to contain a race condition. The term race condition stems from the metaphor that the threads are racing through the critical section, and that the result of that race impacts the result of executing the critical section.

**Q=how to create thread?**

**Ans=**

Threads can be created mainly in 3 different ways

1. **Extend** the java.lang.**Thread** class'

class SampleThread extends Thread {

//method where the thread execution will start

public void run(){

//logic to execute in a thread

}

//let’s see how to start the threads

public static void main(String[] args){

Thread t1 = new SampleThread();

Thread t2 = new SampleThread();

t1.start(); //start the first thread. This calls the run() method.

t2.start(); //this starts the 2nd thread. This calls the run() method.

}

}

1. **Implement** the java.lang.**Runnable** interface

class A implements Runnable{

@Override

public void run() {

// implement run method here

}

public static void main() {

final A obj = new A();

Thread t1 = new Thread(new A());

t1.start();

}

}

1. **Implement** the java.util.concurrent.**Callable** interface

class Counter implements Callable {

private static final int THREAD\_POOL\_SIZE = 2;

// method where the thread execution takes place

public String call() {

return Thread.currentThread().getName() + " executing ...";

}

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

ExecutionException {

// create a pool of 2 threads

ExecutorService executor = Executors

.newFixedThreadPool(THREAD\_POOL\_SIZE);

Future future1 = executor.submit(new Counter());

Future future2 = executor.submit(new Counter());

System.out.println(Thread.currentThread().getName() + " executing ...");

//asynchronously get from the worker threads

System.out.println(future1.get());

System.out.println(future2.get());

}

}

Favor Callable interface with the Executor framework for thread pooling.

**Q=difference btween Thread.sleep and intrupt() method in Thread?**

**Ans=** When you call interrupt() it triggers a boolean flag which tells the run function it should stop. That is why I usually write my run functions like this.

void run()

{

while(!Thread.interrupted())

{

//Do something

}

}

Just because you call interrupt doesn't mean the thread will stop immediately or at all. It depends on what is in the run function.

**public** **class** ThreadIntruption **extends** Thread{

@Override

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

**int** i =0;

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

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

**if**(isInterrupted()){

System.***out***.println("The Thread is interrupted.");

**return**;

}

}

}

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

Thread thread = **new** ThreadIntruption();

thread.start();

Thread.*sleep*(1000);

thread.interrupt();

}

}

# Q= [**Thread.sleep vs. TimeUnit.SECONDS.sleep**](https://stackoverflow.com/questions/9587673/thread-sleep-vs-timeunit-seconds-sleep)?

Ans= [TimeUnit.SECONDS.sleep(x)](https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/TimeUnit.html#sleep-long-) will call [Thread.sleep](https://docs.oracle.com/javase/9/docs/api/java/lang/Thread.html). The only difference is readability and using [TimeUnit](https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/TimeUnit.html) is probably easier to understand for non obvious durations (for example: Thread.sleep(180000) vs. TimeUnit.MINUTES.sleep(3)).

For reference, see below the code of sleep() in TimeUnit:

public void sleep(long timeout) throws InterruptedException {

if (timeout > 0) {

long ms = toMillis(timeout);

int ns = excessNanos(timeout, ms);

Thread.sleep(ms, ns);

}

}

**Q= difference between intrinsic lock and Explicit lock in java?**

Ans=

Q=Design pattern used in Executer framework?

Q=what is phaser?

Q=what is exchanger?

Q=what is atomic variables in java?

Ans=