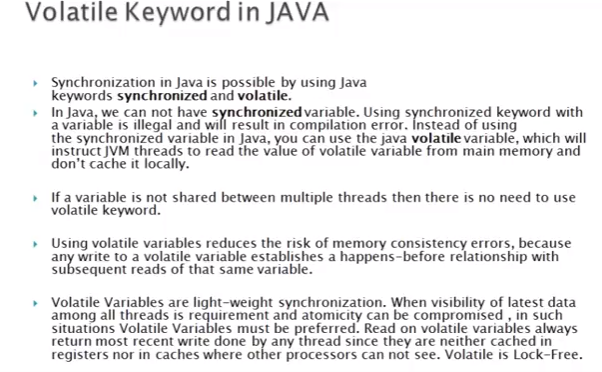
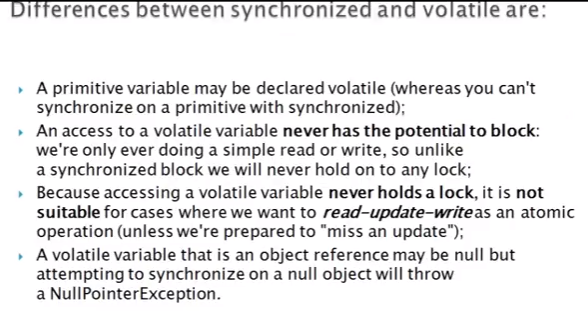
# **Volatile keyword:**

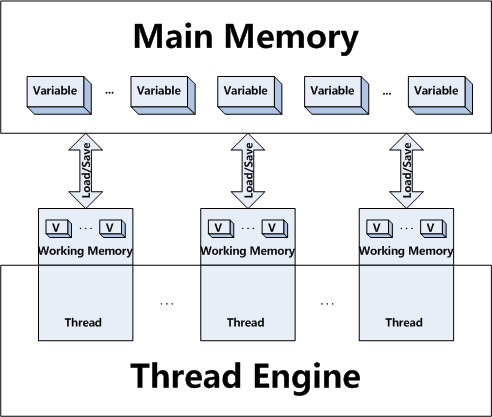


**Note: In volatile first priority is to always write after read operation.**



**How to use Volatile keyword in Java**  
volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from [main memory](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html). So if you want to share any variable in which read and write operation is atomic by implementation e.g. read and write in an int or a boolean variable then  you can declare them as volatile variable.  
  
Java 1.5 introduces some change in Java Memory Model (JMM), Which guarantees visibility of changes made from one thread to another also as "happens-before" which solves the problem of memory writes that happen in one thread can "leak through" and be seen by another thread.  
  
The Java volatile keyword cannot be used with method or class and it can only be used with a variable. Java volatile keyword also guarantees visibility and ordering, after Java 5 write to any volatile variable happens before any read into the volatile variable. By the way use of volatile keyword also prevents compiler or JVM from the reordering of code or moving away them from synchronization barrier.

**The Volatile variable Example in Java**

[](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https://www.udemy.com/multithreading-and-parallel-computing-in-java/)

So in Summary apart from [synchronized keyword in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html), volatile keyword is also used to communicate the content of memory between threads.

**When to use Volatile variable in Java**

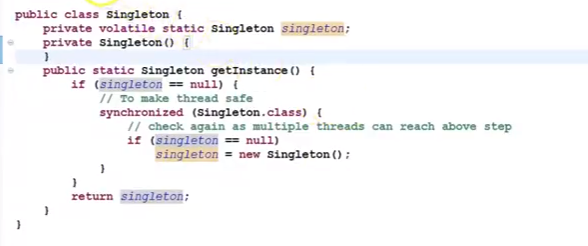
1) You can use Volatile variable if you want to read and write long and [double](http://javarevisited.blogspot.sg/2011/10/convert-double-to-string-example.html) variable atomically. long and double both are [64 bit](http://javarevisited.blogspot.sg/2012/01/find-jvm-is-32-or-64-bit-java-program.html) data type and by default writing of long and double is not atomic and platform dependence. Many platform perform write in long and double variable 2 step, writing 32 bit in each step, due to this its possible for a Thread to see 32 bit from two different write. You can avoid this issue by making long and double variable volatile in Java.  
  
2) A volatile variable can be used as an alternative way of achieving [synchronization in Java](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html) in some cases, like Visibility. with volatile variable, it's guaranteed that all reader thread will see updated value of the volatile variable once write operation completed, without volatile keyword different reader thread may see different values.  
  
3) Another place where a volatile variable can be used is to fixing double checked locking in Singleton pattern. As we discussed in [Why should you use Enum as Singleton](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html) that double checked locking was broken in Java 1.4 environment.

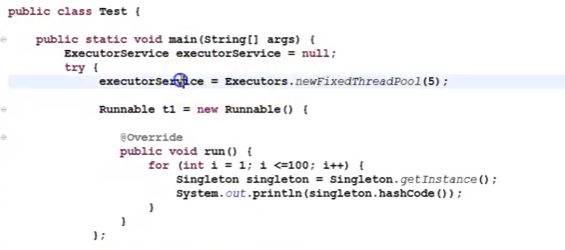
**Important points on Volatile keyword in Java**

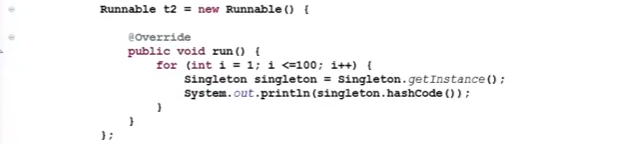
1. The volatile keyword in Java is only application to a variable and using volatile keyword with class and method is illegal.  
  
2. volatile keyword in Java guarantees that value of the volatile variable will always be read from main memory and not from Thread's local cache.  
  
3. In Java reads and writes are [atomic](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html) for all variables declared using Java volatile keyword (including long and double variables).  
  
4. Using the volatile keyword in Java on variables reduces the risk of memory consistency errors because any write to a volatile variable in Java establishes a happens-before relationship with subsequent reads of that same variable.  
  
5. From Java 5 changes to a volatile variable are always visible to other threads. What's more, it also means that when a thread reads a volatile variable in Java, it sees not just the [latest change to the volatile variable](http://java67.blogspot.sg/2012/08/what-is-volatile-variable-in-java-when.html) but also the side effects of the code that led up the change.  
  
6. Reads and writes are atomic for reference variables are for most primitive variables (all types except long and double) even without the use of volatile keyword in Java.  
  
7. An access to a volatile variable in Java never has a chance to block, since we are only doing a simple read or write, so unlike a synchronized block we will never hold on to any lock or wait for any [lock](http://javarevisited.blogspot.sg/2010/10/what-is-deadlock-in-java-how-to-fix-it.html).  
  
8. Java volatile variable that is an object reference may be null.  
  
9. Java volatile keyword doesn't mean atomic, its common misconception that after declaring volatile ++ will be atomic, to make the operation atomic you still need to ensure exclusive access using synchronized method or block in Java.  
  
10. If a variable is not shared between multiple threads, you don't need to use volatile keyword with that variable.

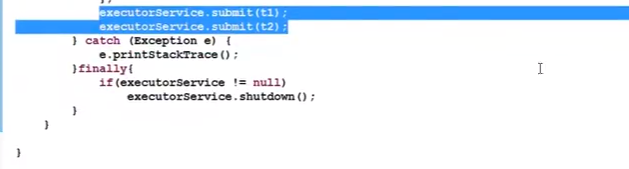
**Difference between synchronized and volatile keyword in Java**

1. The volatile keyword in Java is a field modifier while synchronized modifies code blocks and methods.  
  
2. Synchronized obtains and releases the lock on monitor’s Java volatile keyword doesn't require that.  
  
3. Threads in Java can be blocked for waiting for any monitor in case of synchronized, that is not the case with the [volatile keyword](http://java67.blogspot.com/2012/11/difference-between-transient-vs-volatile-modifier-variable-java.html) in Java.  
  
4. Synchronized method affects performance more than a volatile keyword in Java.  
  
5. Since volatile keyword in Java only synchronizes the value of one variable between Thread memory and "main" memory while synchronized synchronizes the value of all variable between thread memory and "main" memory and locks and releases a monitor to boot. Due to this reason synchronized keyword in Java is likely to have more overhead than volatile.  
  
6. You can not synchronize on the null object but your volatile variable in Java could be null.  
  
7. From Java 5 writing into a volatile field has the same memory effect as a monitor release, and reading from a volatile field has the same memory effect as a monitor acquire  
  
  
In short, volatile keyword in Java is not a replacement of synchronized block or method but in some situation is very handy and can save performance overhead which comes with use of synchronization in Java. If you like to know more about volatile I would also suggest going thorough FAQ on Java Memory Model here which explains happens-before operations quite well.





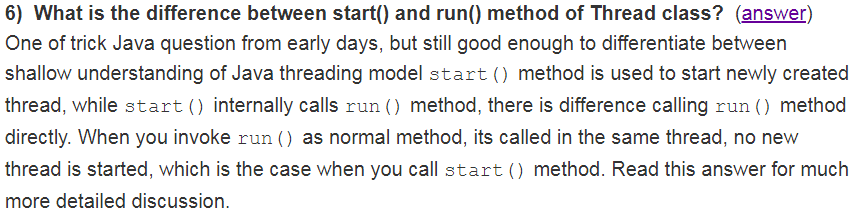


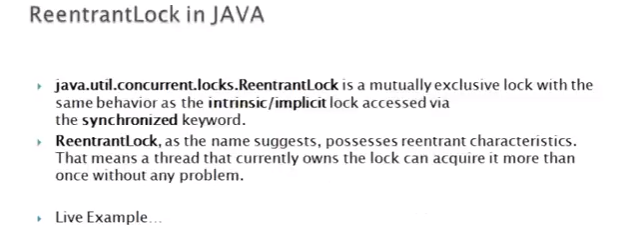
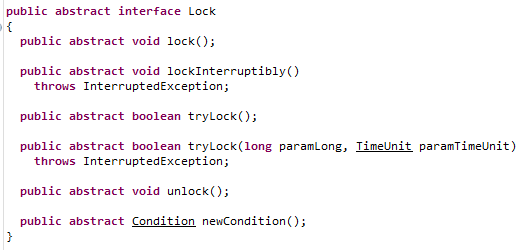




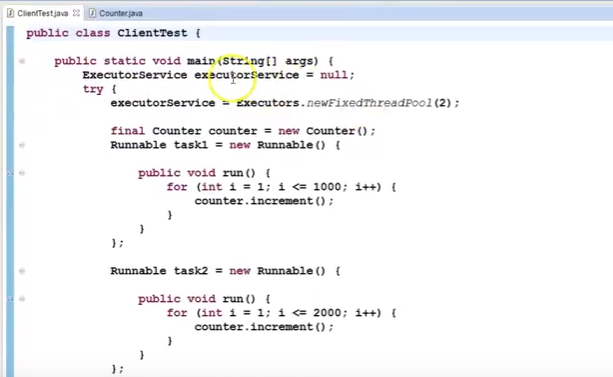
The hashcode will be the same because instance create only once.

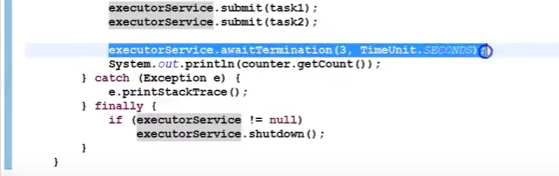
Note: Volatile gives a visibility but not the automicity.s







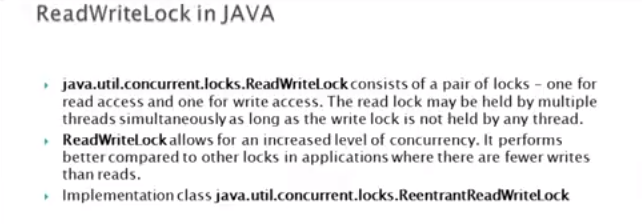


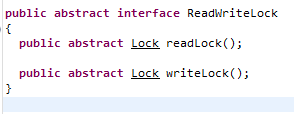


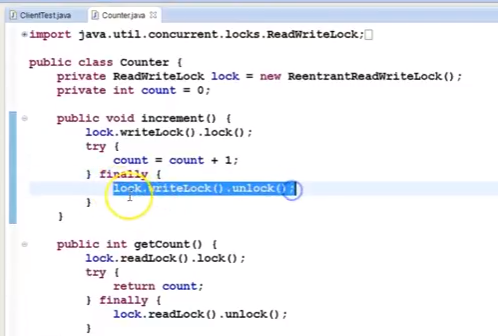
awitTermination method will 3 seconds to complete those task and then print getcount.

# Lock is an interface from java.util.concurrent package. It was introduced in JDK 1.5 release as an alternative of synchronized keyword. As I told you Lock is an interface, so we cannot use it directly, instead we need to use its implementation class. Thankfully Java comes with two implementation of java.util.concurrent.locks.Lock interface, ReentrantLock and ReentrantReadWriteLock, later provides two more inner implementation known as ReentrantReadWriteLock.ReadLock and ReentrantReadWriteLock.WriteLock.

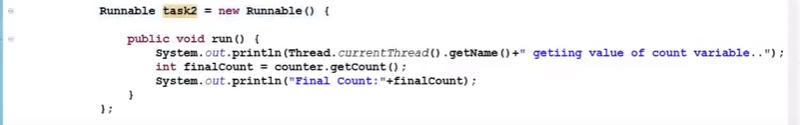
# **ReadWriteLok: It’s and Interface.**

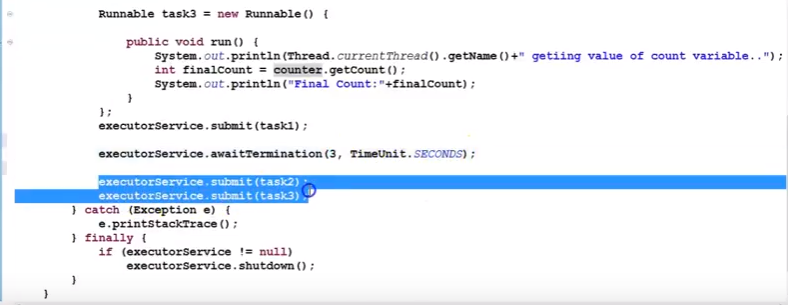








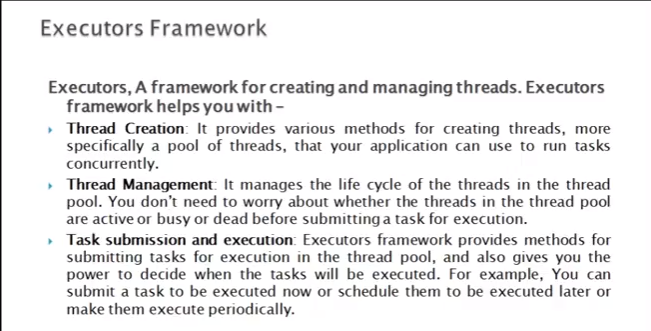


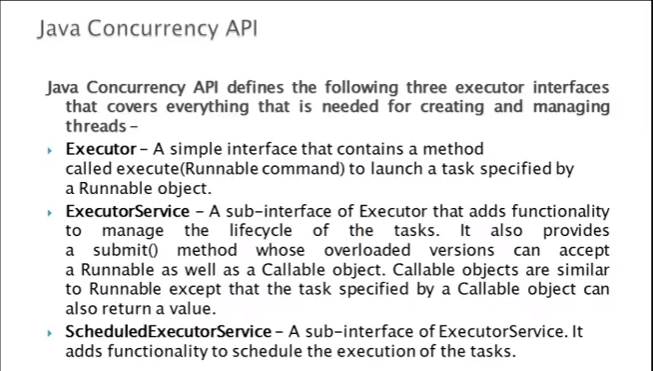


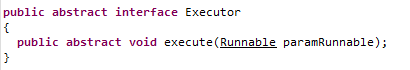


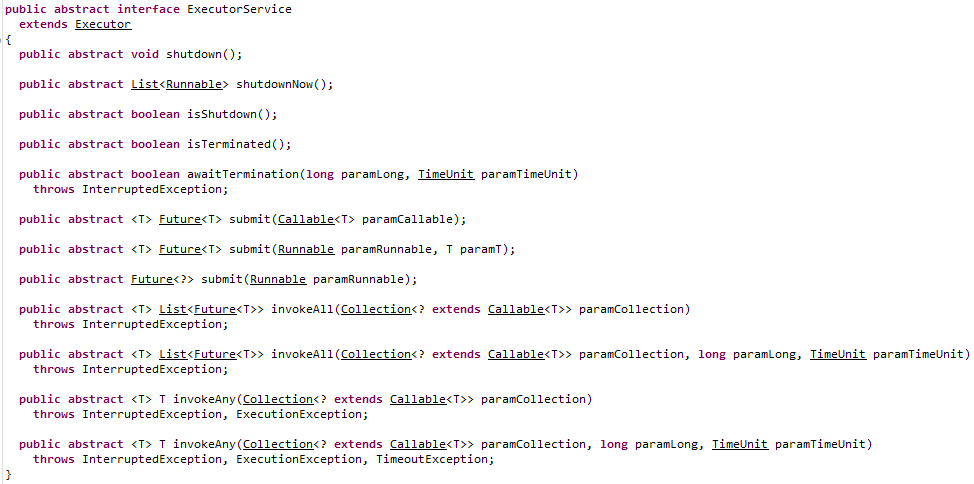
# **Java Executors framework APIs**

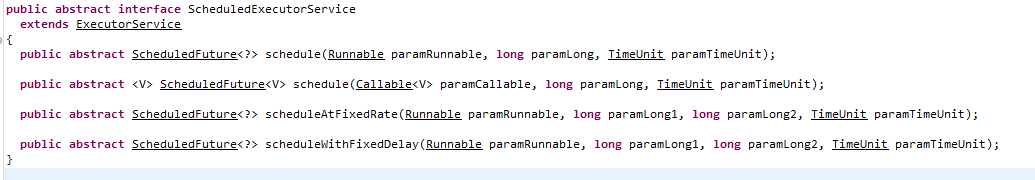
# 

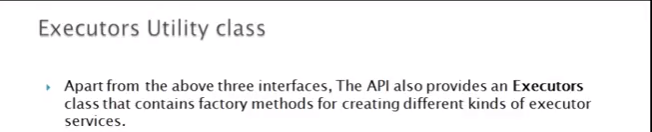


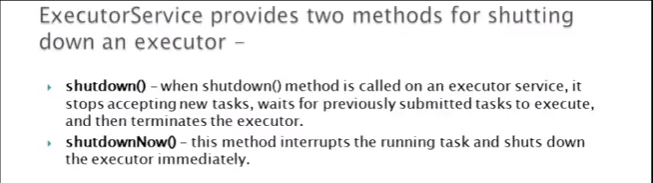


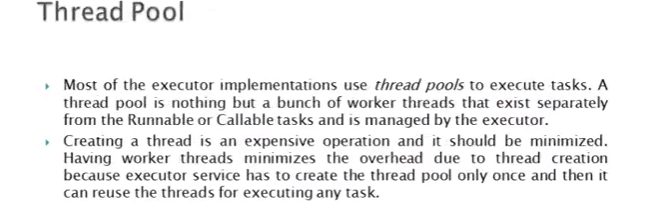


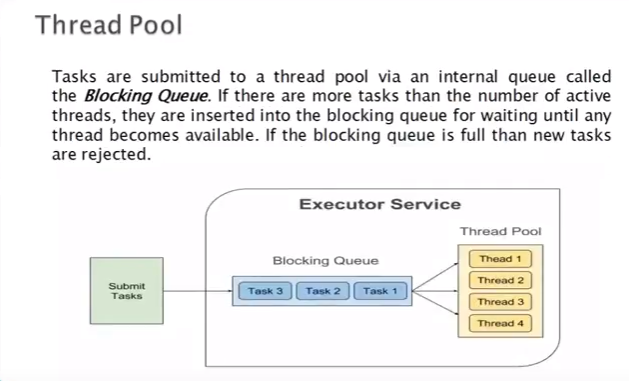


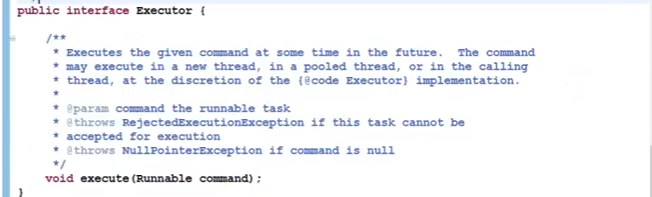




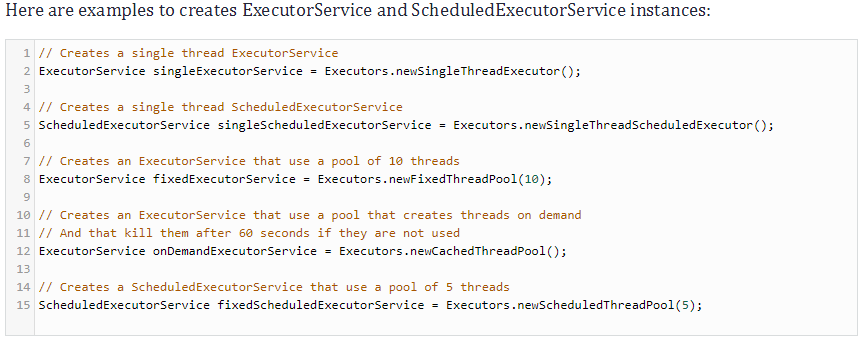








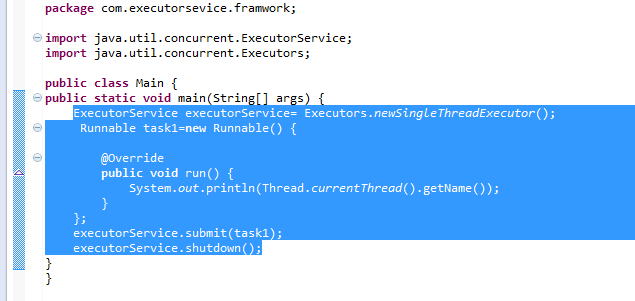
# 

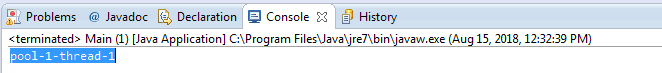


# **Example 1.**

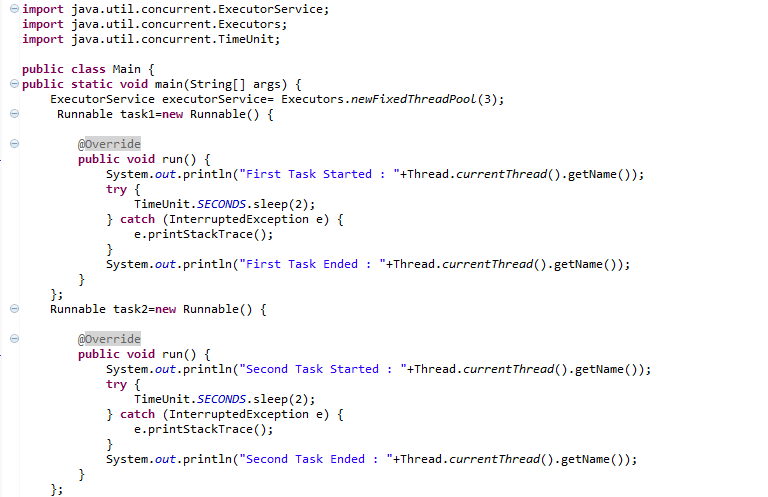
Store all threads into the thread pool.

Here we are using single thread only. So it will create single thread only into the thread pool.

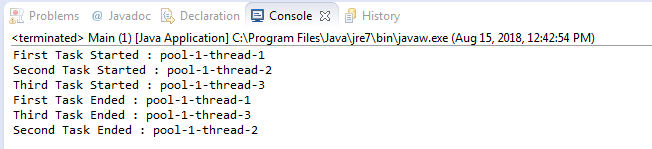




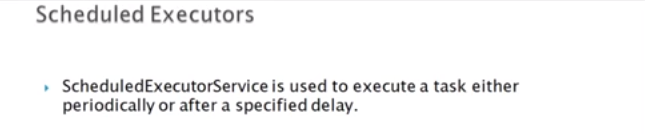
# **Java ExecutorService with a pool of threads for executing mutiple tasks:**

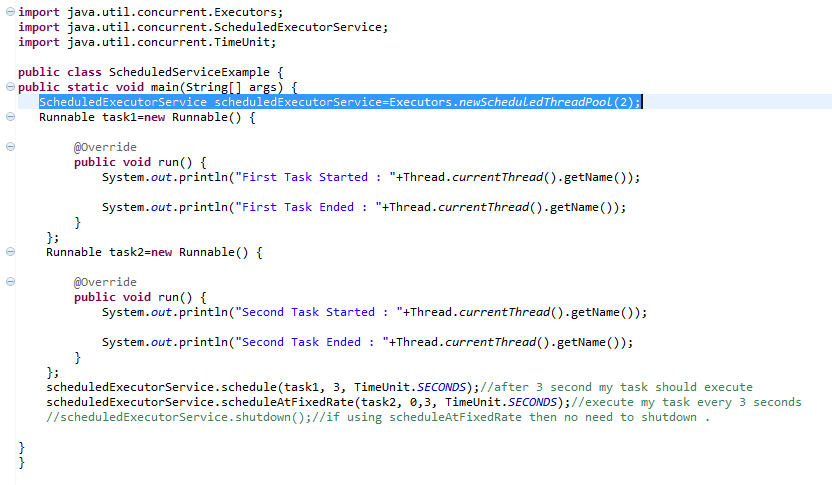






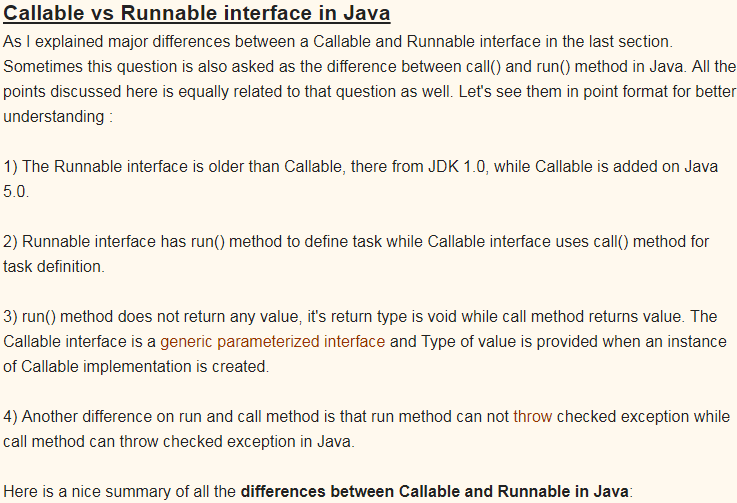
# **Java Scheduled Executors Concurrency example:**





# **Future and Callable Concurrency:**

java.util.concurrent.Callable object can return the computed result done by a thread in contrast to runnable interface which can only run the thread. The Callable object returns Future object which provides methods to monitor the progress of a task being executed by a thread. Future object can be used to check the status of a Callable and then retrieve the result from the Callable once the thread is done. It also provides timeout functionality.

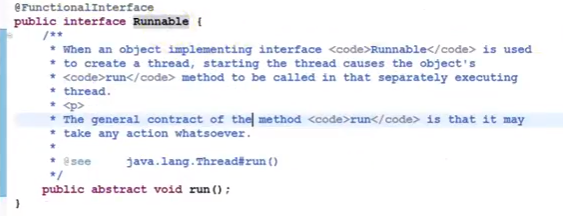




Both callable and runnable interface are the Functional Interface(annotation of the java 7(functionalinteface)).

**Runnable interfaces belong to java.lang package**.

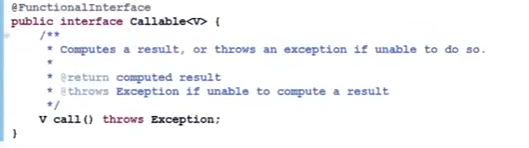
1. In this runnable run method don’t throws any exception.
2. Does not return any value.



**Callable interface belong to java.util.concurrent. package**.

1. In this callable call() method throws Exeption.
2. It return future value means return object.
3. Return type “v” represent Future Interace.



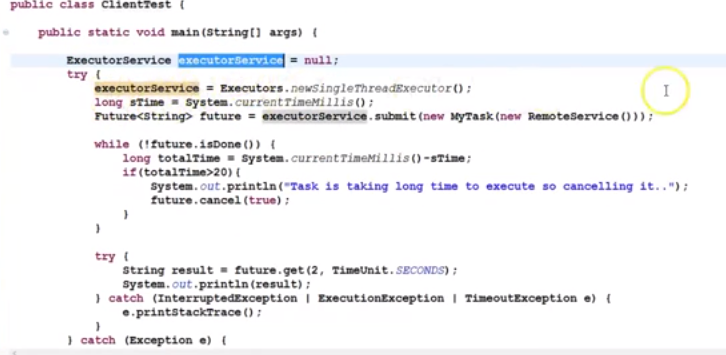


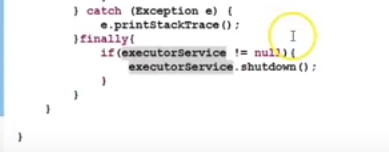
Example:



Output: Future Result : My Callable Interface Task

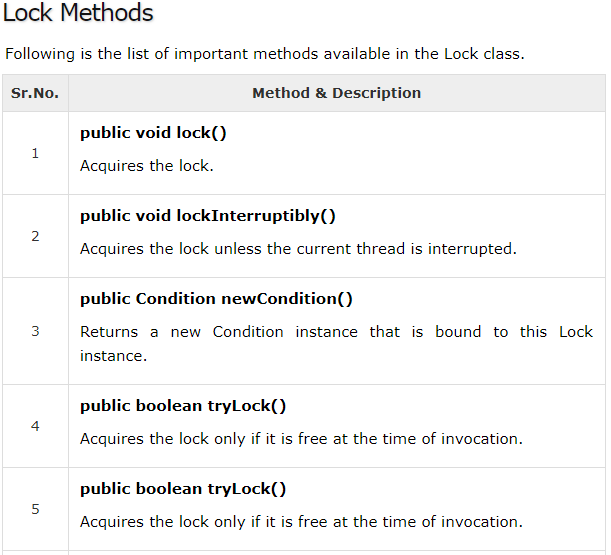
# **Cancelling and adding Timeout in Future task in java:**

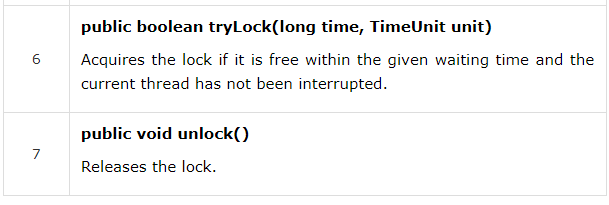




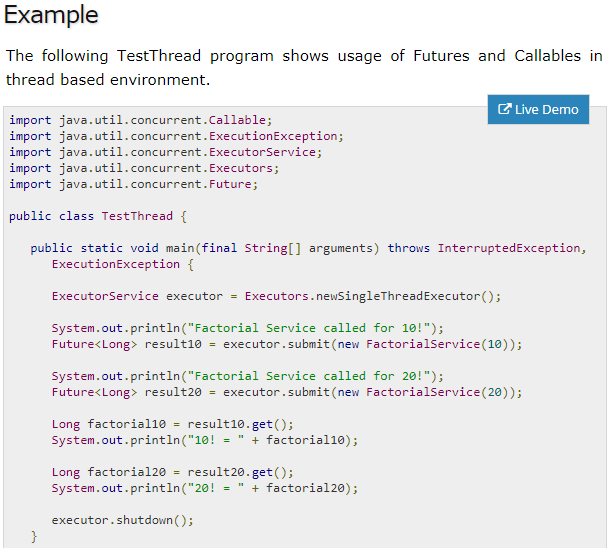
# **Java Concurrency - Lock Interface:**





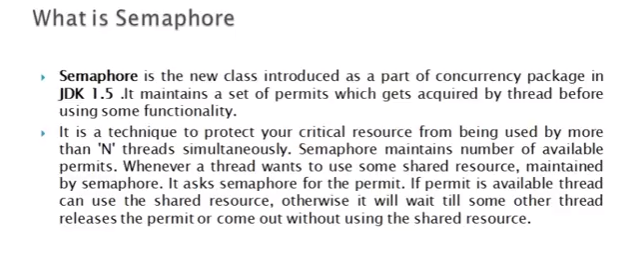


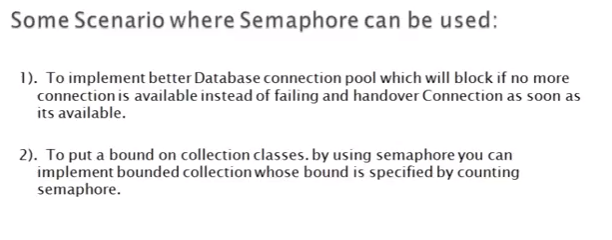
Example :2



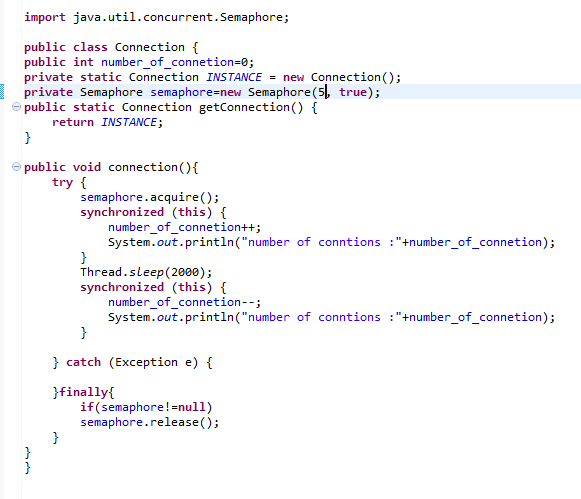


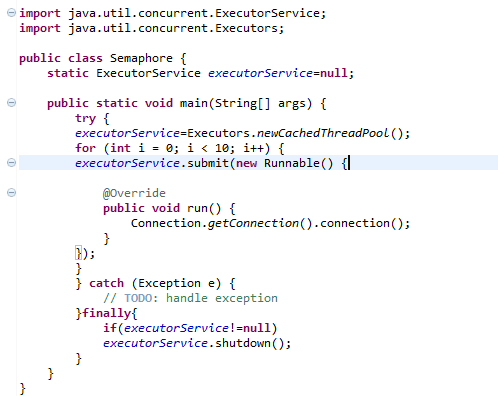
# **Semaphore:**



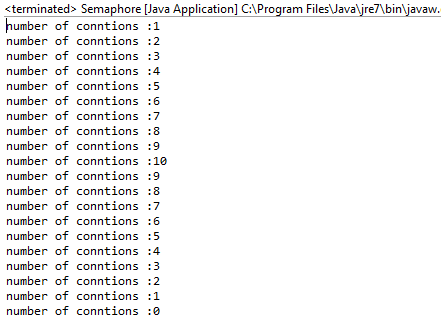


Example:

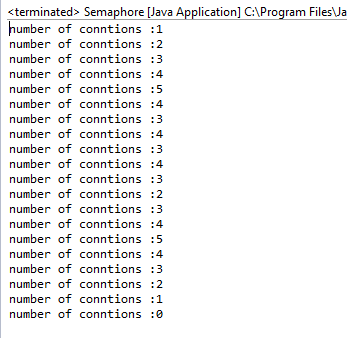


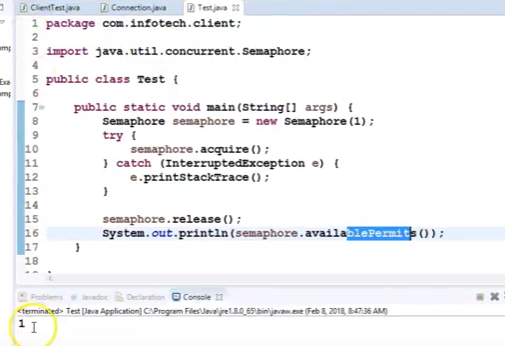


Before Semaphore implement it will create number of connections that we are using in loop(50)



After Semaphore Implement: it will create only connection pool.





# **CountDownLatch:**

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.

**When should we use CountDownLatch in Java :**

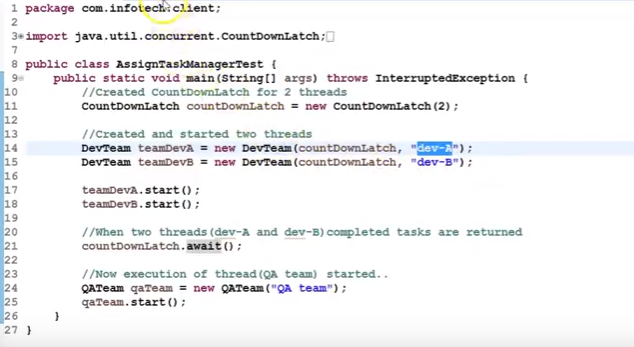
Use CountDownLatch when one of Thread like main thread, require to wait for one or more thread to complete, before its start doing processing. Classical example of using CountDownLatch in Java is any server side core Java application which uses services architecture, where multiple services is provided by multiple threads and application cannot start processing until all services have started successfully as shown in our CountDownLatch example.**countDownLatch in Java – Things to remember:**

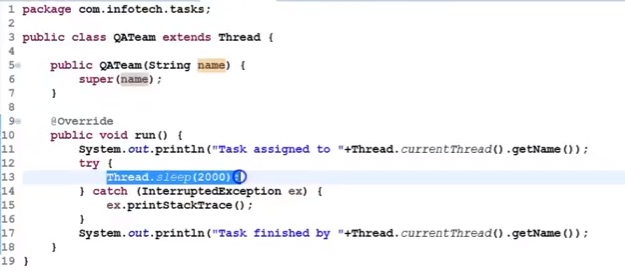
Few points about Java CountDownLatch which is worth remembering:

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.

Example: IT and BA task.







# **CyclicBarrier:**

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 and allows multiple threads to wait for each other (barrier) before proceeding. The difference between CountDownLatch . CyclicBarrier is a natural requirement for a concurrent program because it can be used to perform final part of the task once individual tasks are completed. All threads which wait for each other to reach barrier are called parties, CyclicBarrier is initialized with a number of parties to 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 the barrier. await() is a blocking call but can be timed out or Interrupted by other thread.

**Difference between CountDownLatch and CyclicBarrier in Java:**

If you look at CyclicBarrier it also the does the same thing but there is different you can not reuse CountDownLatch once the 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 a good for one-time events like application start-up time and CyclicBarrier can be used to in case of the recurrent event e.g. concurrently calculating a solution of the big problem etc.

**Important point of CyclicBarrier in Java:**

1. CyclicBarrier can perform a completion task once all thread reaches to the 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. The thread will block on await() until all parties reach to the barrier, another thread interrupt or await timed out.

4. If another thread interrupts the thread which is waiting on barrier it will throw BrokernBarrierException as shown below:

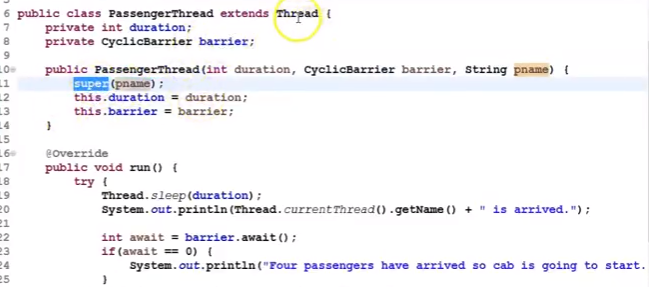
java.util.concurrent.BrokenBarrierException

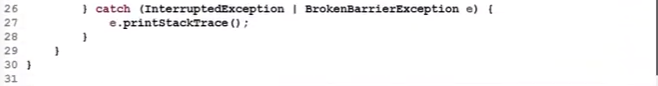
at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:172)

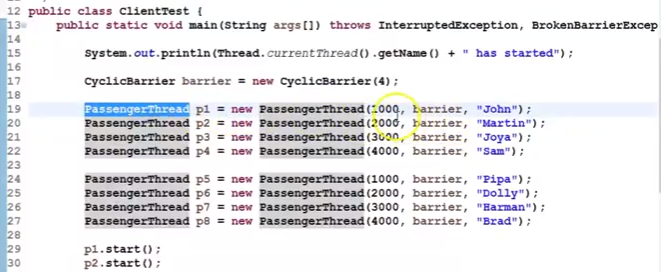
at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:327)

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.

Example:









} }

