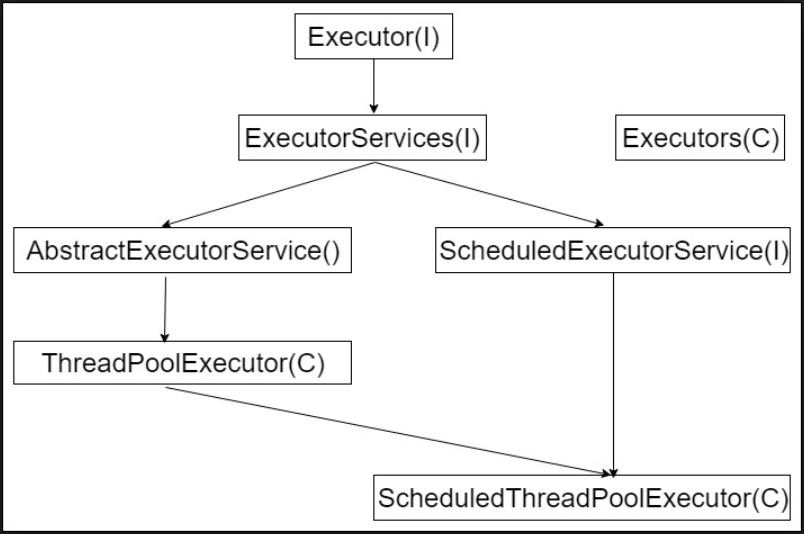
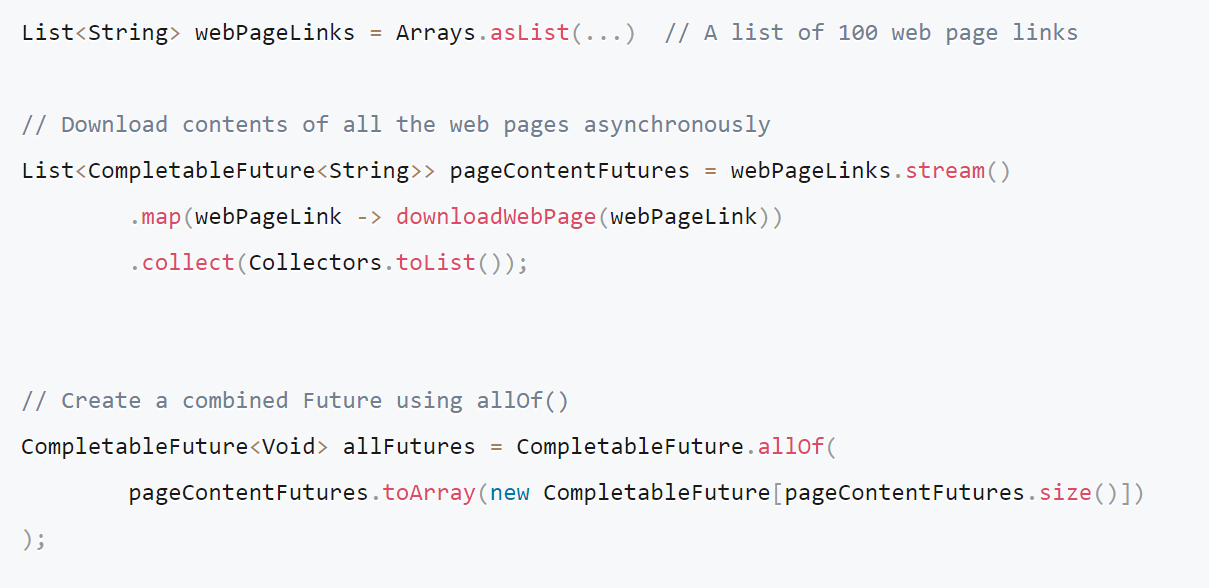
1. Java’s original support for multithreading doesn’t have higher level feature to be suitable for application using large number of threads. Concurrency utilities are introduced in java 5 which is greatly expanded in java 7, 8 and 9.
2. **Java.util.concurrent**: This package contains synchronizers, executors, concurrent collections and Fork/join framework.
3. **Synchronizers**: Synchronizers offers high level ways of synchronizing the interaction of multiple threads.
   1. **Semaphore**: **Semaphore controls access to a shared resource by use of counter**. If counter > 0 access is allowed otherwise not. A thread can ask permit from a semaphore which causes semaphore count to decrease.
      1. **Constructor**: Semaphore(int num, [Boolean fair]), ‘num‘ specifies no of threads that can share a resource i.e., the initial permit count and ‘fair says to grant permit to waiting thread in the order they requested.
      2. **Methods**: acquire([int num]), release([int num]), tryAcquire(int num , [time]), acquireUninterruptly(int num)
         1. availablePermits(), drainPermits(), isFair(), getQueueLength(), hasQueuedThreads()
      3. If permit is -ve then threads are waiting for resource, if 0 no threads is waiting and if +ve more thread can get permit to shared resource.
      4. Java uses counting semaphore instead of binary semaphore.
      5. The way it works is you acquire your semaphore do your job then release other semaphore for waiting thread to proceed. So here we’re not protecting the critical section rather doing thread coordination.
      6. Semaphore can acquire and release multiple permits in single operation.
      7. The acquire and release method need not to be fully bracketed.
   2. **CountDownLatch**: **It’s required where a thread must wait before one or more events occurs**.
      1. **Constructor**: CountDownLatch(int num) where num is intial no of events must occure before latch to open.
      2. **Methods**: await(), boolean await(long wait, TimeUnit tu) [returns false if limit reaches], countdown()
      3. This synchronization construct is appropriate for uses where a must wait for multiple events to occur.
      4. Latches can be used as entry/exit barrier e.g., where worker threads wait for main to run some action/ main thread waits for work threads do some work.
   3. **CyclicBarrier**: **This is used when a set of two or more threads must wait at a predefined execution point until all threads in the set reaches that point.**
      1. **Constructor**: CyclicBarrier(int num, [Runnable action]), ‘num’ indicates no of threads need to reach the barrier and ‘action’ is the task to be run once reached.
      2. **Methods**: int await([long wait, TimeUnit tu]), return indentifies the order in which a thread reach the barrier.
         1. getNumber(), getParties(), isBroken(), reset()
      3. CyclicBarrier can be reused, it’s basically reusable CDL.
      4. CB enables group of threads to wait for each other.
   4. **Exchanger: Exchanger is designed to simplify the exchange of data between threads.** It simply waits until two separate thread call it’s exchange method.
      1. **Constructor**: Exchanger<V>()
      2. **Method**: V exchange(V obj, [long wait, TimeUnit tu])
   5. **Phaser**: It is used for synchronization of threads that represents one or more phases of activity.
      1. **Constructor:** Phaser([int numParties])
      2. **Methods:** int register(), int arrive(), doesn’t suspend the invoking thread which is registered, int arriveAndAwaitAdvance(), waits util all party arrives and return next phase no or -1, int arriveAndDeregister called by registered party, int getPhase()
4. **Executor:** A executor initiates and controls execution of a thread.
   1. **Hierarchy:**



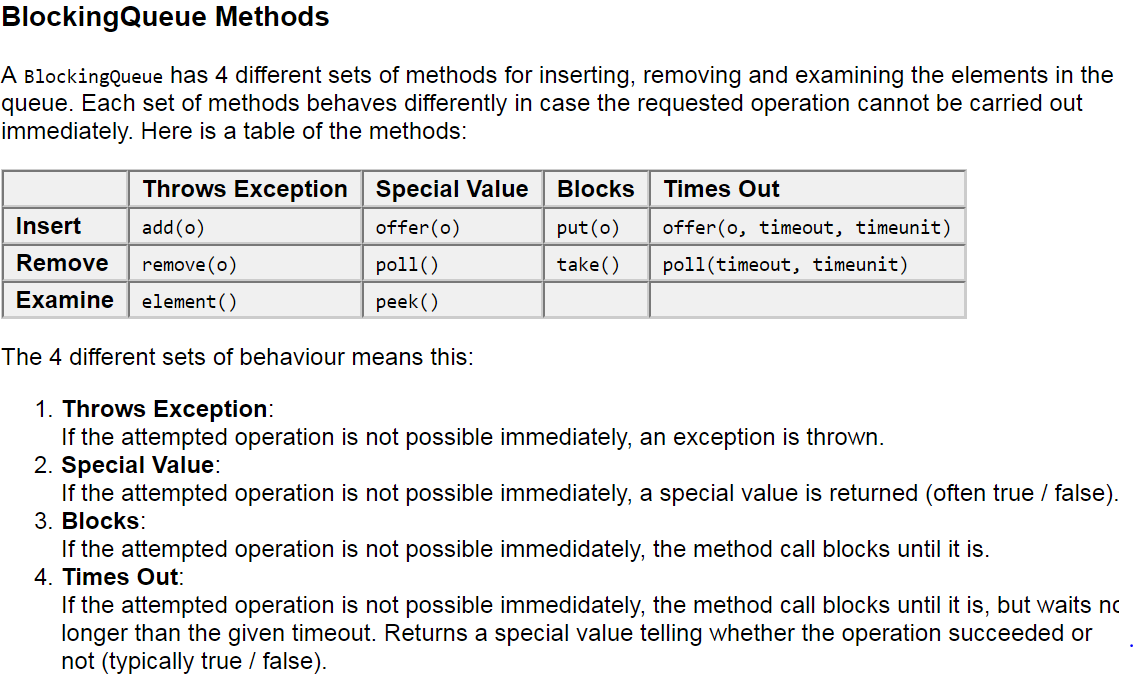
* 1. Executors is a utility class with factory method to create executors:
     1. newCachedThreadPool(): Create new thread if required otherwise reuse from pool.
     2. newFixedThreadPool(): create fixed no threads as passed in constructor.
     3. newScheduledThreadPool():
     4. newSingleThreadExecutor()
     5. newSingleThreadScheduledExecutor()

1. **Callable<V>:** Represents a thread that return value.
   1. Method: V call() throws Exception()
   2. Callable can be executed by ExecutorService: <T> Future<T> submit(Callable<T>)
2. **Future<T>:** It’s a generic interface represents value to be returned by a callable object.
   1. **Methods**: T get([long wait, TimeUnit tu]), Boolean cancel(), isCancelled(), isDone()
   2. **Limitations**:
      1. It cannot be manually completed
      2. You cannot perform further action on a Future’s result without blocking
      3. Multiple Futures cannot be chained together
      4. You cannot combine multiple Futures together
      5. No Exception Handling
3. **CompletableFuture<T>:** CompletableFuture implements Future and CompletionStage interfaces and provides a huge set of convenience methods for creating, chaining and combining multiple Futures. It also has a very comprehensive exception handling support.
   1. **Methods**:
      1. static CompletableFuture<Void> runAsync(Runnable runnable)
      2. static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)
      3. static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
      4. static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)
      5. You can attach a callback to the CompletableFuture using thenApply(), thenAccept() and thenRun()methods.
      6. You can use thenApply() method to process and transform the result of a CompletableFuture when it arrives. It takes a [Function<R,T>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) as an argument. [Function<R,T>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) is a simple functional interface representing a function that accepts an argument of type T and produces a result of type R.
      7. CompletableFuture.thenAccept() takes a Consumer<T> and returns CompletableFuture<Void>. It has access to the result of the CompletableFuture on which it is attached.
      8. While thenAccept() has access to the result of the CompletableFuture on which it is attached, thenRun() doesn’t even have access to the Future’s result. It takes a Runnable and returns CompletableFuture<Void>
      9. // thenApply() variants
         1. <U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)
         2. <U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends U> fn)
         3. <U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends U> fn, Executor executor)
      10. Combine two dependent futures using thenCompose().
      11. Combine two independent futures using thenCombine()
      12. CompletableFuture.allOf is used in scenarios when you have a List of independent futures that you want to run in parallel and do something after all of them are complete.
      13. The join() method is similar to get(). The only difference is that it throws an unchecked exception if the underlying CompletableFuture completes exceptionally



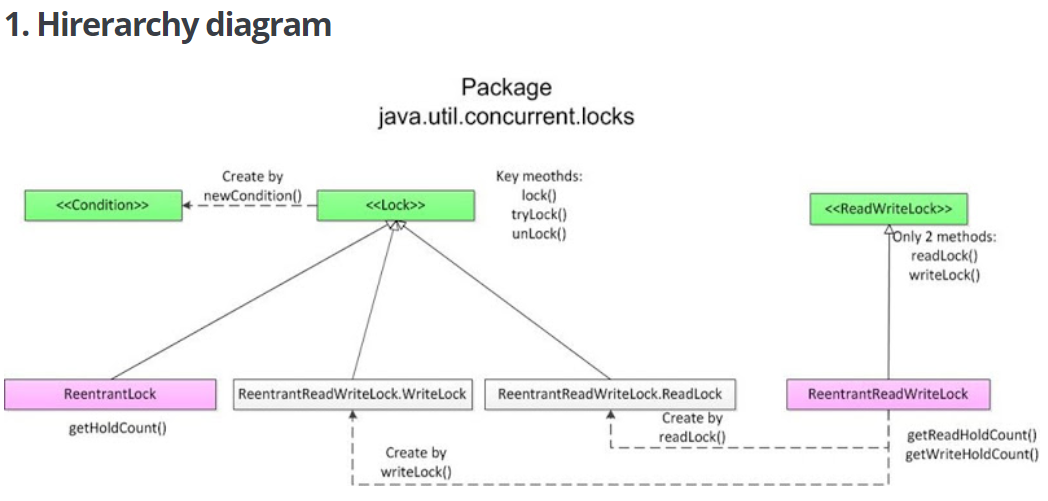
* + 1. CompletableFuture.anyOf() as the name suggests, returns a new CompletableFuture which is completed when any of the given CompletableFutures complete, with the same result.
    2. Handle exceptions using exceptionally(ex) callback/ generic handle(res, ex) method

1. **TimeUnit**: TimeUnit is an enumeration with values: DAYS, HOURS, MINUTES, SECONDS, MICROSECONDS, MILLISECONDS, NANASECONDS
   1. Methods: convert(long wait, TimeUnit tu), toMicro(long tval), toMillis, toNanos, sleep()
2. **Concurrent Collections**:
   1. **BlockingQueue**: This is a queue interface which is thread safe to put into, and take instances from
      1. It is not possible to insert null into a BlockingQueue. If you try to insert null, the BlockingQueue will throw a NullPointerException.
      2. It is also possible to access all the elements inside a BlockingQueue, and not just the elements at the start and end. However, this is not done very efficiently, so you should not use these Collection methods unless you really have to.
      3. **Implementations:** ArrayBlockingQueue, DelayQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue

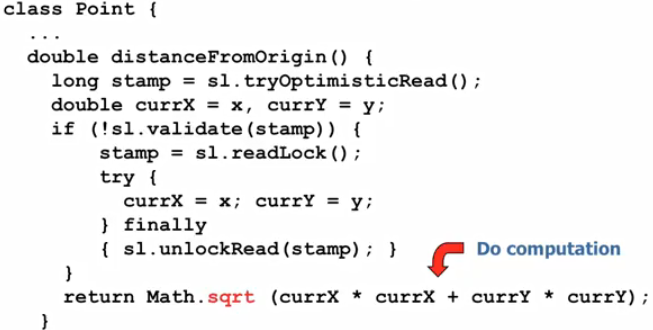


* 1. **ArrayBlockingQueue**: ArrayBlockingQueue is a bounded, blocking queue that stores the elements internally in an array. That it is bounded means that it cannot store unlimited amounts of elements. There is an upper bound on the number of elements it can store at the same time. You set the upper bound at instantiation time, and after that it cannot be changed.
  2. **DelayQueue**: The DelayQueue blocks the elements internally until a certain delay has expired. The elements must implement the interface java.util.concurrent.Delayed.
  3. **LinkedBlockingQueue**: The LinkedBlockingQueue keeps the elements internally in a linked structure (linked nodes). This linked structure can optionally have an upper bound if desired.
  4. **PriorityBlockingQueue**: The PriorityBlockingQueue is an unbounded concurrent queue. It uses the same ordering rules as the java.util.PriorityQueue class.
  5. **PriorityBlockingQueue** does not enforce any specific behavior for elements that have equal priority (compare() == 0).
  6. Iterator from a PriorityBlockingQueue, the Iterator does not guarantee to iterate the elements in priority order.
  7. **SynchronousQueue**: The SynchronousQueue is a queue that can only contain a single element internally. A thread inseting an element into the queue is blocked until another thread takes that element from the queue. Likewise, if a thread tries to take an element and no element is currently present, that thread is blocked until a thread insert an element into the queue.
  8. **BlockingDeque**: The BlockingDeque interface in the java.util.concurrent class represents a deque which is thread safe to put into, and take instances from.
  9. A BlockingDeque could be used if threads are both producing and consuming elements of the same queue. It could also just be used if the producting thread needs to insert at both ends of the queue, and the consuming thread needs to remove from both ends of the queue.
  10. The BlockingDeque interface extends the BlockingQueue interface. That means that you can use a BlockingDeque as a BlockingQueue.
  11. **Implementations**: LinkedBlockingDeque
  12. **ConcurrentHashMap**: <http://javarevisited.blogspot.in/2013/02/concurrenthashmap-in-java-example-tutorial-working.html#axzz59s7w775W>
  13. ConcurrentNavigableMap: The java.util.concurrent.ConcurrentNavigableMap class is a [java.util.NavigableMap](http://tutorials.jenkov.com/java-collections/navigablemap.html) with support for concurrent access, and which has concurrent access enabled for its submaps. The "submaps" are the maps returned by various methods like headMap(), subMap() and tailMap().

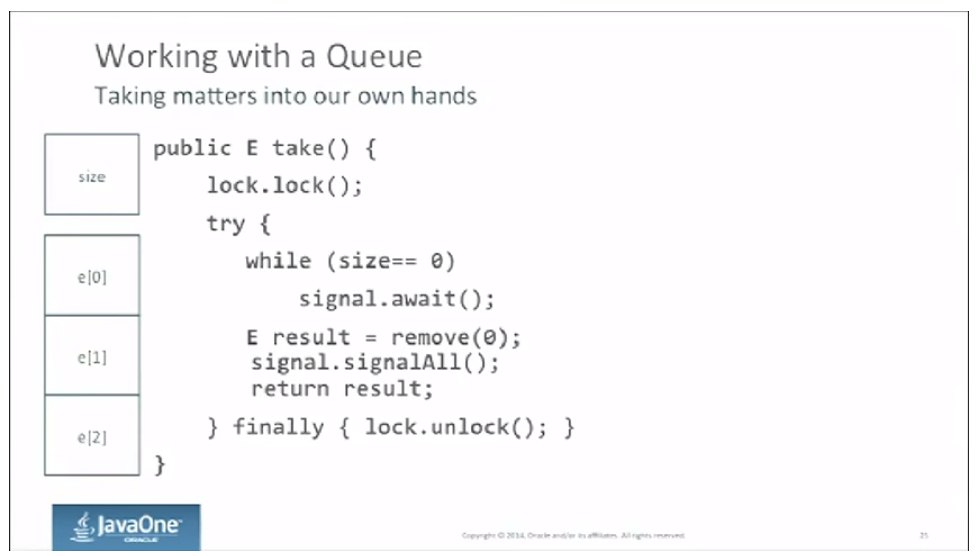
1. **Lock**: Lock interface provides alternative means of synchronized to control access to shared resource.
   1. **Methods**: lock(), lockInterruptibly(), newCondition(), tryLock([long wait, TimeUnit tu]), unlock()

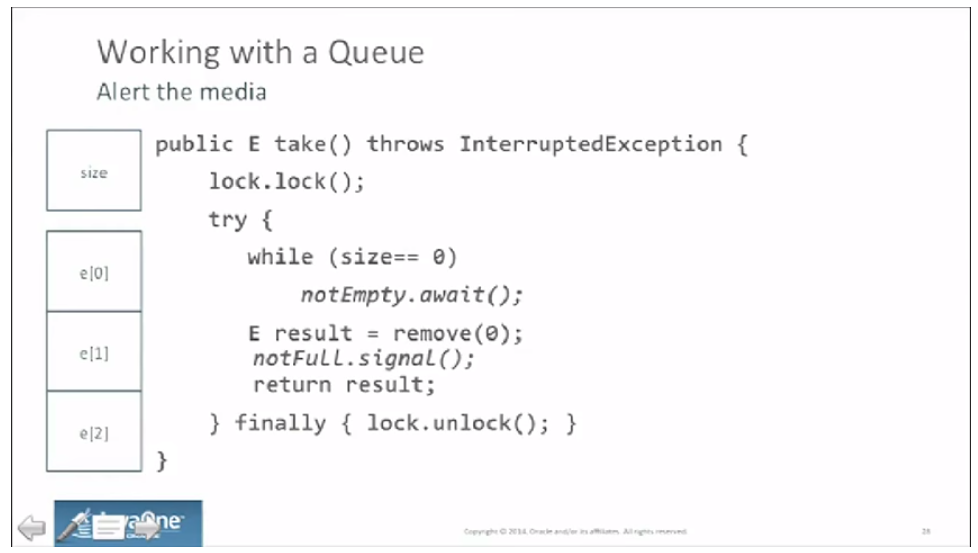


* 1. **Reentrant lock**: Has same functionality as synchronization but with more control.
     1. **Constructor**: ReentrantLock([Boolean fair])
     2. **Methods**: getHoldCount(), getQueuedLength(), isLocked(), isFair(), hasWaiters()
  2. **Read-write lock**: A read / write lock is more sophisticated lock than the Lock implementations shown in the text [Locks in Java](http://tutorials.jenkov.com/java-concurrency/locks.html). Imagine you have an application that reads and writes some resource, but writing it is not done as much as reading it is. Two threads reading the same resource does not cause problems for each other, so multiple threads that want to read the resource are granted access at the same time, overlapping. But, if a single thread wants to write to the resource, no other reads nor writes must be in progress at the same time. To solve this problem of allowing multiple readers but only one writer, you will need a read / write lock.
     1. the conditions for getting read and write access to the resource:
        1. **Read Access**: If no threads are writing, and no threads have requested write access.
        2. **Write Access**: If no threads are reading or writing.
     2. **Write Reentrance**: Write reentrance is granted only if the thread has already write access.
     3. **Read Reentrance**: A thread is granted read reentrance if it can get read access (no writers or write requests), or if it already has read access (regardless of write requests).
     4. Reentrance lockout may occur if a thread reenters a [Lock](http://tutorials.jenkov.com/java-concurrency/locks.html), [ReadWriteLock](http://tutorials.jenkov.com/java-concurrency/read-write-locks.html) or some other synchronizer that is not reentrant.
     5. **Write lock**: Lock writeLock = rwLock.writeLock(); writeLock.lock();
     6. **Read lock**: Lock readLock = rwLock.readLock(); readLock.lock();
     7. **Problems**:
        1. They can lead to starvation.
        2. RRWL Significantly slower than other synchronization means.
  3. **StampedLock**: StampedLock replaces RRWL because of performance issues.
     1. **Modes**:
        1. Writing: long writeLock(), long tryWriteLock([long wait, , TimeUnit tu]), unlockWrite(long stamp)
        2. Reading: long readLock(), long tryReadLock([long wait, TimeUnit tu]), unlockRead(ong stamp)
        3. Optimistic: long tryOptimisticRead() [returns an observation stamp] , Boolean validate(long stamp)
     2. **Methods**: tryToConvertToWriteLock(long stamp), tryToConvertToReadLock(long stamp), tryToConvertToOptimisticLock(long stamp), unlock()



* 1. **ConditionObject**: Condition factors out the Object monitor methods ([wait](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#wait()), [notify](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#notify()) and [notifyAll](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html" \l "notifyAll())) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary [Lock](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html) implementations. Where a Lock replaces the use of synchronized methods and statements, a Condition replaces the use of the Object monitor methods.
     1. Method: await(), signal(), signalAll()
     2. We can use different condition object for different condition per lock on which different set of threads are waiting. In this way we can get rid of ‘The Thundering Herd’ problem associated with object notifyAll(). So if we call signal() only threads waiting on that condition object will be awaken.





1. **Atomic operations**: java.util.concurrent.atomic provides an alternative to other synchronization feature when reading/writing the value of some types of variable.
   1. **Ex:** AtomicInteger, AtomicLong, DoubleAdder, LongAdder, AtomicIntegerArray
   2. **Methods**: get(), set(), compareAndSet(), incrementAndGet(), getAndSet()
   3. **AtomicReference**: AtomicReference class provides an object reference variable which can be read and written atomically. By atomic is meant that multiple threads attempting to change the same AtomicReference (e.g. with a compare-and-swap operation) will not make the AtomicReference end up in an inconsistent state.
   4. **AtomicStampedReference**: The AtomicStampedReference is different from the [AtomicReference](http://tutorials.jenkov.com/java-util-concurrent/atomicreference.html) in that the AtomicStampedReference keeps both an object reference and a stamp internally.
      1. Methods: getStamp(), getReference()
      2. Getting both reference and stamp:

int[] stampHolder = new int[1];

Object ref = atomicStampedReference.get(stampHolder);

* + 1. The AtomicStampedReference is designed to solve the A-B-A problem. The A-B-A problem is when a reference is changed from pointing to A, then to B and then back to A. y using an AtomicStampedReference instead of an AtomicReference it is possible to detect the A-B-A situation. Thread 1 can copy the reference and stamp out of the AtomicStampedReference atomically using get(). If another thread changes the reference from A to B and then back to A, then the stamp will have changed (provided threads update the stamp sensibly - e.g increment it).

1. **Non-blocking algorithm**: <http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html>
2. **Fork/join framework**: This framework has been added in java 7 as part of java.uti.concurrent package to support parallel programming.
   1. **Parallel programming**: Parallel programming is a programming model where a task can be divided into independent sub task that can be executed in different thread concurrently in multi CPU multi core environment.
   2. Using Fork/Join framework applications can automatically scale to make effective use of multi CPU, multicore system.
   3. In traditional multithreading program threads share a single CPU in time sliced way. But in presence of multiple CPU we can divide a program in sub tasks and each can be run in different CPU in its own thread. This type of parallel programming can significantly improve performance of operations can be broken down in parallel tasks like sorting, searching etc.
   4. **Core classes**:
      1. **ForkJoinTask<V>:** abstract class can be managed by ForkJoinPool where V is the return type.
         1. Represents a lightweight task than thread of execution. A small number of threads from fork join pool can handle large no of ForkJoinTask
         2. **Methods**:
            1. Final ForkJoin<V> fork(): Submit a task for asynchronous execution.
            2. Final V join(): join method waits until the task on which it’s called finished.
            3. Final V invoke: Combines fork and join in one method call.
            4. Final void invokeAll(ForkJoinTask<V> … taskList)
      2. **RecursiveAction**: A ForkJoinTask that doesn’t return a value.
         1. Method:
            1. Protected abstract void compute(): Represents computational portion of a fork/join action.
      3. **RecursiveTask**<V>: A ForkJoinTask that returns a value.
         1. Method:
            1. Protected abstract V compute()
      4. **ForkJoinPool**: Manages execution of fork join task
         1. Constructor: ForkJoinPool([int pLevel])
            1. By default concurrency level is no of processor available in system otherwise pLevel would be taken.
            2. JDK 8 contains static FOrkJoinPool for use called common pool.
            3. The level of parallelism is only a target not a guarantee. Although multiple processor are available the os/other task would compete for the processors.
         2. **Methods**:
            1. <T> T invoke(FJT <T> task): Blocks until a result is returned.
            2. Void execute(FJT <?> task): Not blocked on task completion.
            3. Static FJP commonPool(): returns a ref to ForkJoinPool

If we call FJT methods directly then it would use common pool if not already using a fork join pool.

* + - * 1. FJP manages execution of the tasks through work stealing where one thread steal work from another threads work queue.
        2. A FJP uses daemon thread thus terminated when all user thread terminates. It’s not required to explicitly call shutdown()
        3. getParallelism(), getCommonPoolParallelism() [to get processor count we can also use Runtime availableProcessors()]
  1. **Divide and conquer strategy**: It’s based on recursively dividing a task into smaller task until a threshold reached where on task can be handled sequentially.
     1. **Ex**: Transforming array elements of a large array.
     2. Choice of optimal threshold requires profiling. If each computation step is large then smaller threshold may require.