**MultithreadedServer.java**

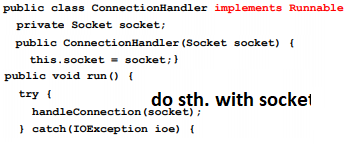
try {

ServerSocket l=new ServerSocket(port);

Socket socket;

while(true) {

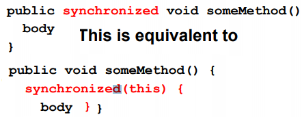
l.accept(); tl.execute(new ConnectionHandler(socket)); } }

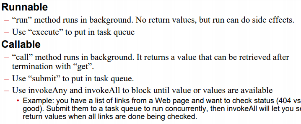


**Race Conditions**

**synchronized(someObject) { code }**

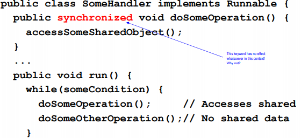
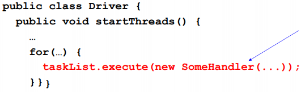
**Normal meaning**: Once a th. Enters that code, no other th. Can enter until 1st leaves. **Actual meaning:** Once a th. Enters that code, no other th. Can enter that code using the same someObj(lock object).



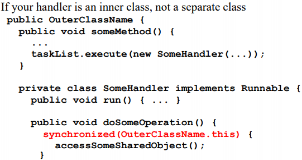


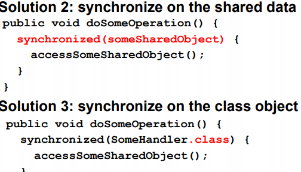
Use **volatile** keyword to make sure threads fetch that member from the memory. Not used for solving race conditions.

**Synchronization Problems:**

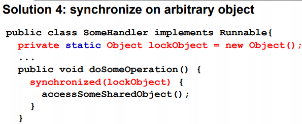


It does not work. Because in tasklist.execute, we are creating new instances. Every instance has different lock objects. So, they will not wait for each other. **Solution1**:

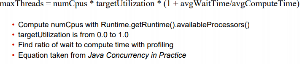


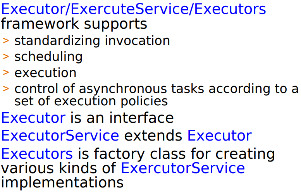


Note: If you use sync. For a static method, the lock corresponds to A.class object. Not this. So you will lock all of them.

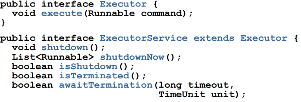


**Determining Max. Thread Pool Size**

****

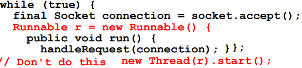
****

Executor ex = getSomeKindofExecutor(); ex.execute(new RunnableTask1()); ex.execute(new RunnableTask2());

****

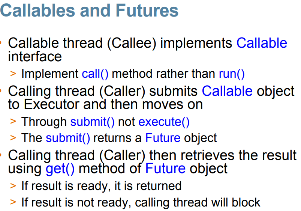
****

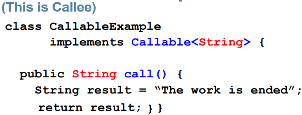
**Bad Web Server Example**

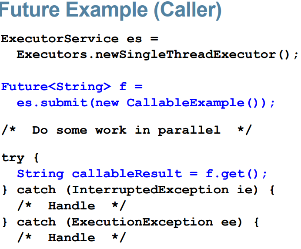


You have no control over the thread r.

Instead, create pool, pool.execute(r). Now, you can use shutdown etc.

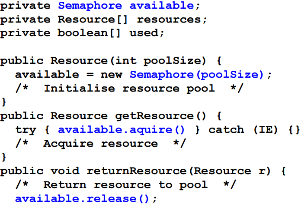


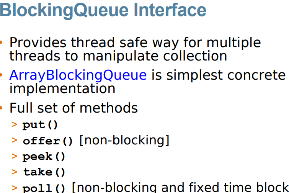


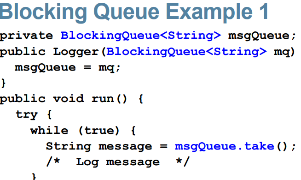


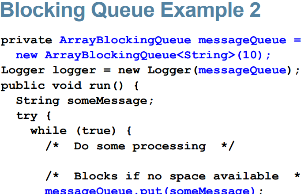
Semaphores:

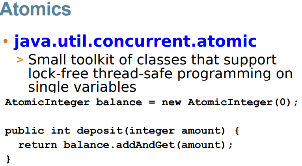
Thread trying to access resource calls aquire() > Returns immediately if semaphore count > 0 > Blocks if count is zero until release() is called by different thread



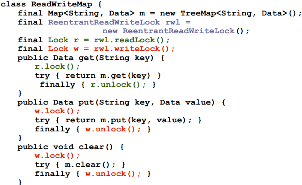




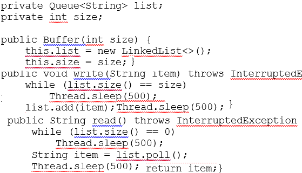


**ReadWrite lock**

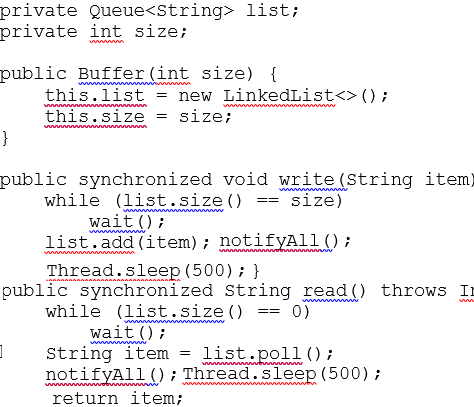
rwl.readLock().lock(); rwl.writeLock().lock();



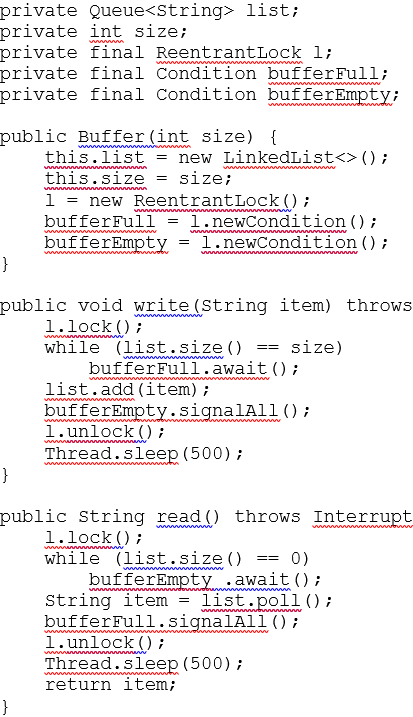
**Busy wait**. Just buffer class implementation



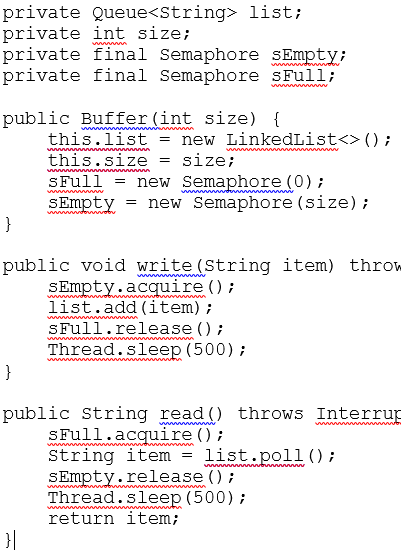
**Synchronized, wait&notify**



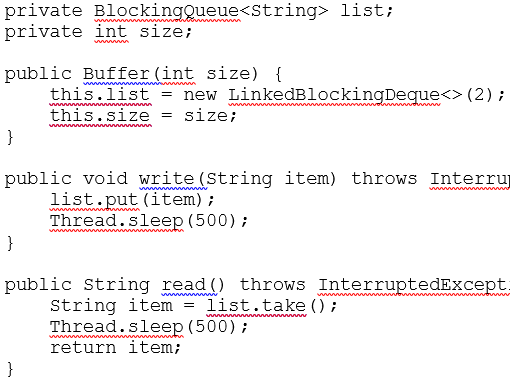
**Locks**



**Semaphore**



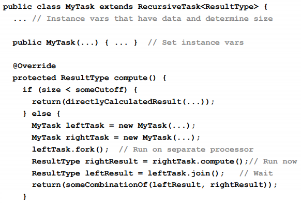
**QUEUE**

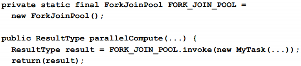


**MultiThreaded FORK/JOIN**

No benefit if 1 core.

Subclass of RecursiveTask<T>

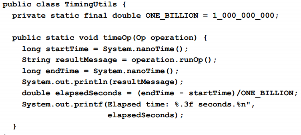


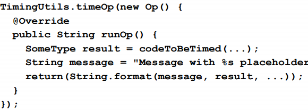


How to measure that the parallel is better? 1- Make sure they are equivalent. 2- Measure time

You cannot measure time like this:

Utils.timeOp(someMethod()). You need an instance. So, create an Op interface, and your class should implement that Op interface. By creating Op interface, you make sure that classes must implement runOp() method to measure time.





Our goal is if it is less than a size, sum sequentially, else,

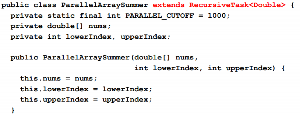
copy some to left,

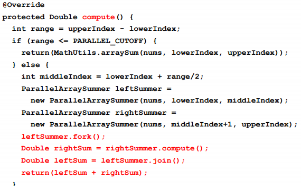
left fork,

right compute,

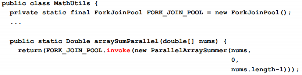
left join,

combine

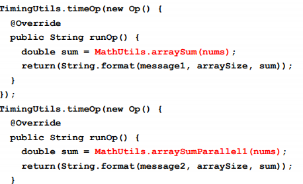




Call that by:



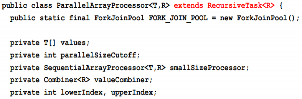
To measure the time:

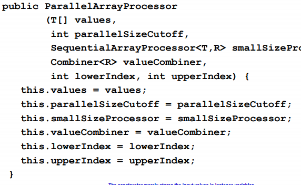


In java8, use DoubleStream

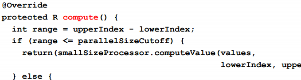


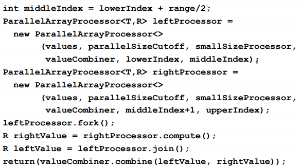
The big problem is, we repeat the code. We can make a superclass and use the common operations (join etc.) there

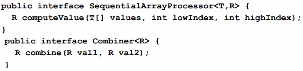




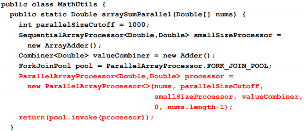
SequentialArrayProcessor, Combiner are interfaces. We make sure that classes implement those interfaces must have the necessary methods.

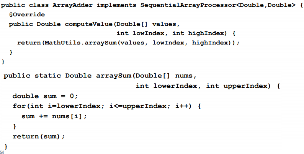


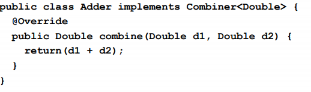


Interfaces:

How to call?







**KORAY YURTSEVEN, 2099547**