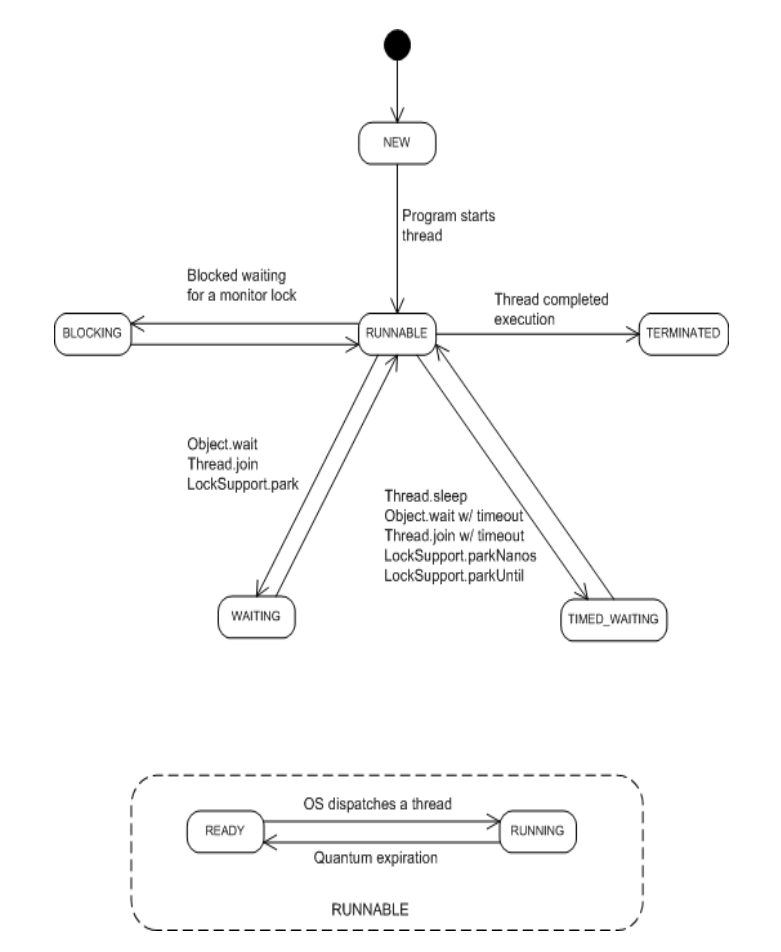
1. **Multitasking:** There is two types of multitasking –
   1. **Process based**: The smallest unit dispatched by scheduler is a program.
   2. **Thread based**: The smallest unit dispatched by scheduler is a thread within a program.
2. **Process vs thread:**
   1. Unlike process threads share same address space.
   2. Unlike process inter thread communication is inexpensive.
   3. Unlike process context switching between thread is not costly.
3. **Advantage of multithreading:** In a single core system multithreading maximize the CPU usage by keeping CPU idle time to minimum, especially interactive, network based environments. When a thread blocks only that thread pauses, all other threads continue to run.
4. **Thread state diagram:** Thread.State getState() gives current thread states which is a enum of following constants: NEW, BLOCKED, RUNNABLE, TERMINATED, TIMED\_WAITING/WAITING.



1. **Thread priority:** Thread priorities are integers that specify relative priority of threads. These priorities are used for context switching.
   1. A thread can relinquish control by yielding, blocking on pending IO. In this scenario thread with highest priority would be given control.
   2. A higher priority thread can preempt a lower priority thread in preemptive multitasking.
   3. If two threads are of same priority then they run in time sliced manner (windows) or they need to voluntarily relinquish control to peers (unix).
   4. A child thread has same priority of parent thread.
   5. setPriority(int) can be used to set value within range MIN\_PRIORITY, MAX\_PRIORITY. Default priority is given by NORM\_PRIORITY

**Note**: Portability problem can arise due os dependent context switch.

1. **Monitor:** Monitor is used for interthread synchronization. In java Monitor is not an object but each object has its own monitor which is entered when objects synchronized method is called.
2. **Thread:** Thread class encapsulates a thread of execution and used to spawn and later interact with the thread thus acting proxy to the running thread.
   1. **class** Thread **implements** Runnable
   2. **Instance Methods:** getName, setName, getPriority, setPriority, isAlive, join, start
   3. **Static Methods:** currentThread(), onSpinWait (Java 9)
      1. **onSpinWait**: It says jvm is free to lower the process priority; slowing the cycle or reduce electricity on this loop when its resources are needed for other more important things.

while(true) {

while(!newEmailArrived()) {

Thread.onSpinWait();

}

makeNotification();

}

<https://stackoverflow.com/questions/44622384/onspinwait-method-of-thread-class-java-9>, <https://vividcode.io/Java-9-Thread-onSpinWait/>

1. **Creating Thread:**
   1. Extend the thread class and override the run method.

**public** **class** ThreadTest **extends** Thread {

@Override

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

System.***out***.println("Job using extending Thread class.");

}

}

* 1. Implement a Runnable interface and pass that to the constructor of thread class.

**public** **class** ThreadTest **implements** Runnable {

@Override

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

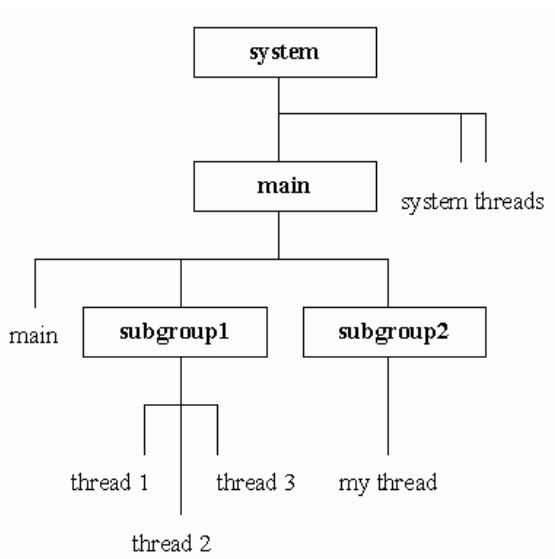
System.***out***.println("Job using implementing Runnable interface.");

}

}

* 1. If we extend the Thread class to create a thread then we can’t extend another class. Extend thread if want to modify thread class behavior.
  2. when thread A starts thread B, the Java Language Specification (JLS) guarantees that all variables that were visible to thread A when it starts thread B are visible to thread B, which is effectively like having an implicit synchronization in Thread.start(). If we start a thread from within a constructor, the object under construction is not completely constructed, and so we lose these visibility guarantees even it’s the last line in constructor.

1. **Join()**: This method waits until the thread on which it’s called terminates.
2. **Synchronization**: It’s enable access control to shared resources among multiple threads.
   1. Use synchronized to modify a method and mark it synchronized.
   2. Use synchronized block to synchronize a block of code that requires thread-safety/3rd party class [if 3rd party class documents it thread safety specs].
   3. Synchronized blocks also guarantee that all variables accessed inside the synchronized block will be read in from main memory, and when the thread exits the synchronized block, all updated variables will be flushed back to main memory again, regardless of whether the variable is declared volatile or not.
   4. Synchronized blocks in Java are reentrant. This means, that if a Java thread enters a synchronized block of code, and thereby take the lock on the monitor object the block is synchronized on, the thread can enter other Java code blocks synchronized on the same monitor object.
3. **Interthread communication**: To do away with polling java uses wait(), notify(), notifyAll() and can be used within a synchronized context.
   1. **wait()**: Tells the calling thread to leave monitor and go to sleep until some other thread enters same monitor and calls notify()/notifyAll()
   2. **notify()**: wakes up a thread that called wait on same object.
   3. **notifyAll()**: wakes up all threads that called wait on same object. While if you call notifyAll method, all threads waiting on that lock will be woken up, but again all woken thread will fight for lock before executing remaining code and that's why wait is called on loop because if multiple threads are woken up, the thread which will get lock will first execute and it may reset waiting for condition, which will force subsequent threads to [wait](http://javarevisited.blogspot.sg/2011/12/difference-between-wait-sleep-yield.html).
   4. You can use notify over notifyAll if all threads are waiting for the same condition and only one Thread at a time can benefit from condition becoming true.
   5. Call to wait should happens in a loop that checks the condition on which thread is waiting to avoid spurious wake ups. To guard against spurious wakeups the signal member variable is checked inside a while loop instead of inside an if-statement. Such a while loop is also called a spin lock. The thread awakened spins around until the condition in the spin lock (while loop) becomes false.
   6. The while loop is also a nice solution if you have multiple threads waiting, which are all awakened using notifyAll(), but only one of them should be allowed to continue.
   7. Don't call wait() on constant String's or global objects.
   8. Use notifyAll instead of notify in general because it guarantees require threads will be awakened.
   9. When a thread is blocked on some method declaring InterruptedException and you call Thread.interrupt() on such thread, most likely blocked method will immediately throw InterruptedException. If we call interrupt() on a thread and it’s not blocked then it would through exception on next blocking call.
   10. <http://tutorials.jenkov.com/java-concurrency/thread-signaling.html>
4. **Thread Groups**: java.lang.ThreadGroup objects that group related threads Thread and Thread subclasses.
   1. Java requires every thread and every thread group—save the root thread group, system—to join some other thread group.

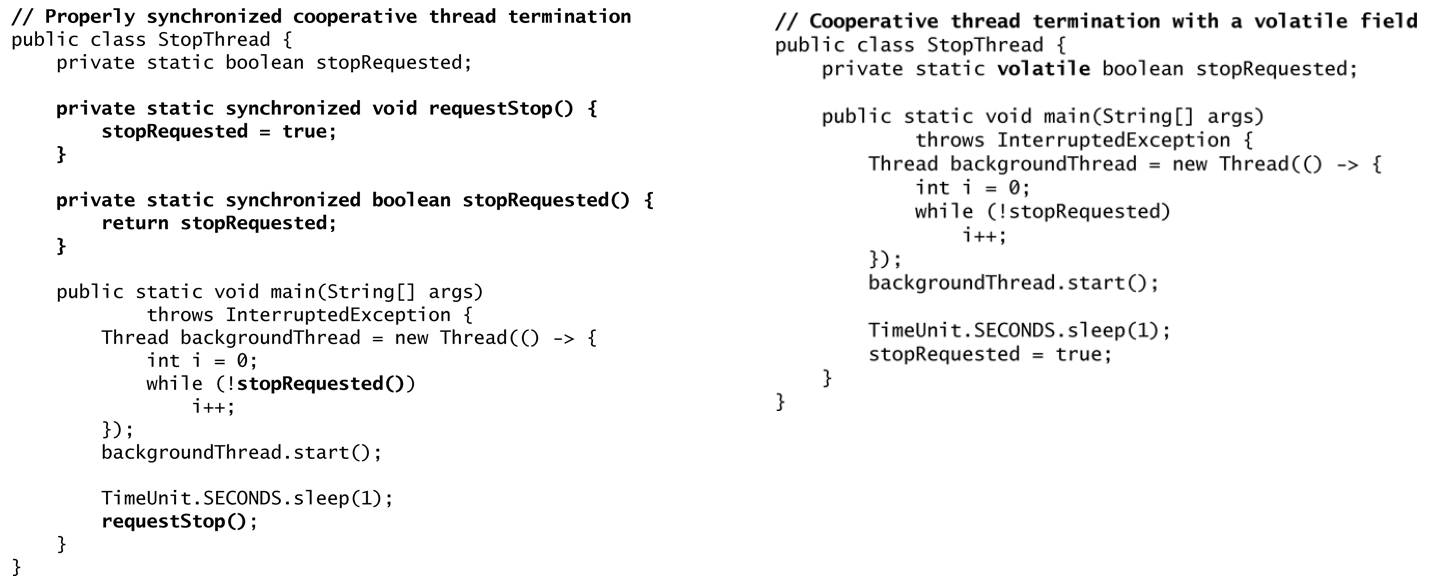


* 1. The JVM-created system group organizes JVM threads that deal with object finalization and other system tasks, and serves as the root thread group of an application's hierarchical thread-group structure.
  2. Just below system is the JVM-created main thread group, which is system's sub-thread group (subgroup, for short). main contains at least one thread—the JVM-created main thread that executes byte-code instructions in the main() method.
  3. <https://www.javaworld.com/article/2074481/java-concurrency/java-101--understanding-java-threads--part-4---thread-groups--volatility--and-threa.html>

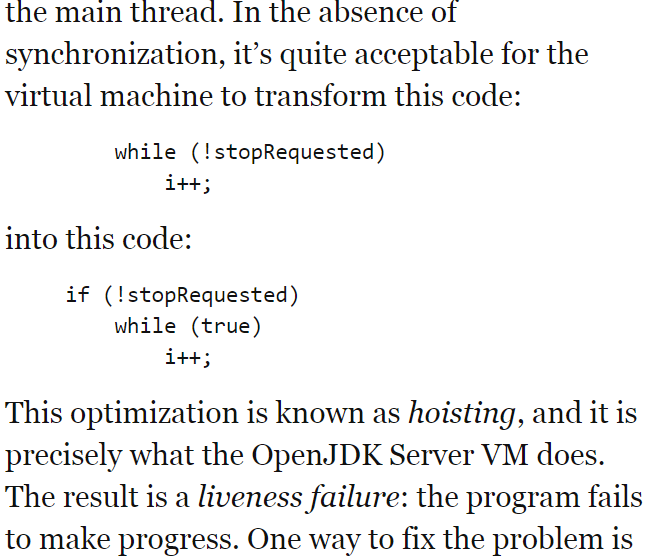
1. **Deadlock:** Deadlock happens when two threads has circular dependency on a pair of objects.
   1. **Livelock:** A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, 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.
   2. **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.
   3. <http://www.codejava.net/java-core/concurrency/understanding-deadlock-livelock-and-starvation-with-code-examples-in-java>
   4. **Deadlock prevention:**

<http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html>

1. **Suspending thread:** Because suspend(), resume() and stop() methods are deprecated we can’t use them to directly on thread instead the running thread keep a state variable to keep track whether current thread should keep running.
2. **Overhead:** Creating too many threads would create overhead for CPU to context switch.
3. **Principles:**
   1. **Synchronize access to shared mutable data:**
      1. Synchronization facilitates mutual exclusion i.e., prevents an object from being seen in inconsistent state while its being modified by another thread.
      2. Synchronization ensures each thread entering a synchronized block sees all the changes that was guarded by same lock.
      3. JLS guarantees reading and writing of variables are atomic unless its double or long thus thread will not see an arbitrary value while reading but it doesn’t guarantee a value written by one thread would be visible to another thread.
      4. While mutual exclusion is not required we can use volatile to save overhead of synchronization. <http://tutorials.jenkov.com/java-concurrency/volatile.html>
         1. Reads from and writes to other variables cannot be reordered to occur after a write to a volatile variable, if the reads / writes originally occurred before the write to the volatile variable.   
            The reads / writes before a write to a volatile variable are guaranteed to "happen before" the write to the volatile variable. Notice that it is still possible for e.g. reads / writes of other variables located after a write to a volatile to be reordered to occur before that write to the volatile. Just not the other way around. From after to before is allowed, but from before to after is not allowed.
         2. Reads from and writes to other variables cannot be reordered to occur before a read of a volatile variable, if the reads / writes originally occurred after the read of the volatile variable. Notice that it is possible for reads of other variables that occur before the read of a volatile variable can be reordered to occur after the read of the volatile. Just not the other way around. From before to after is allowed, but from after to before is not allowed.
         3. AWBR principle

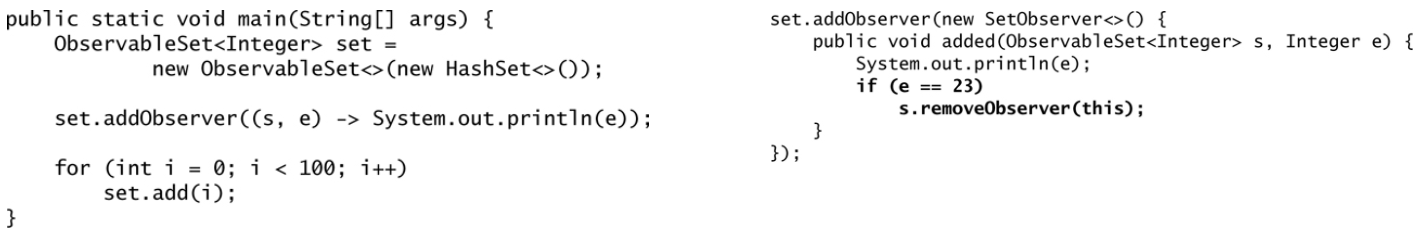


* + 1. While volatile provides only communication effects java.util.concurrent.atomic provides atomicity. This package provides lock free, thread safe programming on single variable.
    2. Liveness failure:

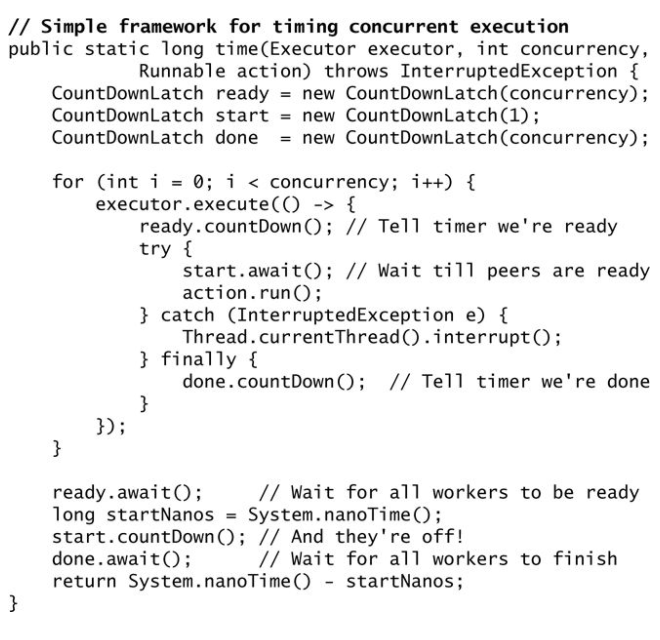


* 1. **Avoid excessive synchronization**:
     1. Keeps work in the synchronized block minimum. Do not call methods to be overridden or indeterministic client code which could cause data corruption, dead lock or exceptions.
     2. Concurrent modification exception: Here lock of notifyElementsAdded is broken since it doesn’t stop thread calling back into the observable set and modify its observers list because same thread can reenter monitor lock.

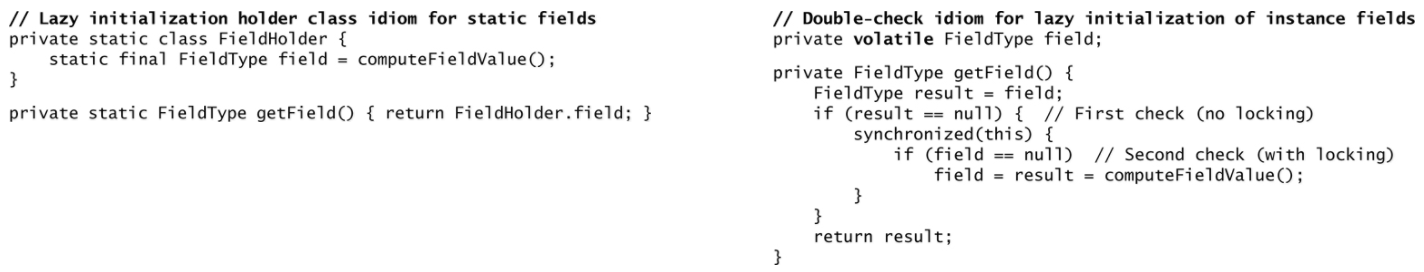


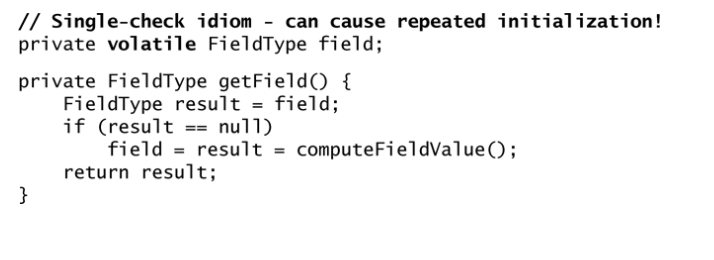


* + 1. If we run s.removeObserver from a thread then there would be deadlock.
    2. For this situation we can use CopyOnWriteArrayList and bypass synchronization. <http://java-latte.blogspot.in/2014/06/How-CopyOnWriteArrayList-works-in-Java-how-it-differ-from-ArrayList.html>
    3. Over synchronization prevents vm’s ability to optimize code execution.
    4. While writing a mutable class, a class can be synchronized internally or let the caller do it. Java.util.concurrent follows the 1st and java.util follows the 2nd approach.
    5. If a method modifies a static field and could be called from multiple threads then access to the static field must be synchronized.
  1. **Prefer executors, tasks, streams to threads**: A thread represent both unit of work and way to execute it where in executor framework the unit work (Callable or Runnable task) and execution mechanism is separate. One should think in terms of task and let the executor run it.
  2. **Prefer concurrency utilities to wait and notify**:
     1. Java.util.concurrent has 3 parts: Executors, concurrent collections and synchronizers
     2. Concurrent collections are synchronized internally so no need to lock them since it would slow them down.
     3. Concurrent collection makes synchronized collections almost obsolete.
     4. BlockingQueue is alternative of traditional wait queue. Executor service implementation uses blocking queue.
     5. Synchronizers enables threads to wait for each other allowing to coordinate their activities.



* 1. To prevent DoS (Denial of service) attack use private final lock Object instead of synchronized method. Thread-safe public classes that may interact with untrusted code must use a private final lock object. Existing classes that use intrinsic synchronization must be refactored to use block synchronization on such an object.
     1. <https://wiki.sei.cmu.edu/confluence/display/java/LCK00-J.+Use+private+final+lock+objects+to+synchronize+classes+that+may+interact+with+untrusted+code>
  2. [Do not override thread-safe methods with methods that are not thread-safe](https://wiki.sei.cmu.edu/confluence/display/java/TSM00-J.+Do+not+override+thread-safe+methods+with+methods+that+are+not+thread-safe).
  3. [Avoid client-side locking when using classes that do not commit to their locking strategy](https://wiki.sei.cmu.edu/confluence/display/java/LCK11-J.+Avoid+client-side+locking+when+using+classes+that+do+not+commit+to+their+locking+strategy).
  4. [Do not synchronize on a collection view if the backing collection is accessible](https://wiki.sei.cmu.edu/confluence/display/java/LCK04-J.+Do+not+synchronize+on+a+collection+view+if+the+backing+collection+is+accessible)**.** <https://wiki.sei.cmu.edu/confluence/display/java/LCK04-J.+Do+not+synchronize+on+a+collection+view+if+the+backing+collection+is+accessible>
  5. For lazy initialization of static field use holder class idiom and for instance field use double check idiom. For instance field that can tolerate repeated initialization use single check idiom.





* 1. Average number of runnable thread should not be significantly higher than available processors. Thread should not if they aren’t doing any useful work.

1. **Tools**:
   1. **Deadlock detection Programatically:**
      1. Interface **ThreadMXBean** is the management interface for the thread system of the Java virtual machine. A Java virtual machine has a single instance of the implementation class of this interface. This instance implementing this interface is an [MXBean](https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/management/ManagementFactory.html" \l "MXBean) that can be obtained by calling the [ManagementFactory.getThreadMXBean()](https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/management/ManagementFactory.html" \l "getThreadMXBean()) method or from the [platform MBeanServer](https://docs.oracle.com/javase/1.5.0/docs/api/java/lang/management/ManagementFactory.html#getPlatformMBeanServer()) method.

**long**[] threadIds = tmxb.findDeadlockedThreads();

**if** (threadIds != **null**) {

System.***out***.println("Deadlock detected.... -> " + threadIds);

ThreadInfo[] deadlockedThreads = tmxb.getThreadInfo(threadIds);

**for** (ThreadInfo threadInfo : deadlockedThreads) {

**if** (threadInfo != **null**) {

**for** (Thread thread : Thread.*getAllStackTraces*().keySet()) {

**if** (thread.getId() == threadInfo.getThreadId()) {

System.***err***.println(threadInfo.toString().trim());

**for** (StackTraceElement ste : thread.getStackTrace()) {

System.***err***.println("\t" + ste.toString().trim());

}

}

}

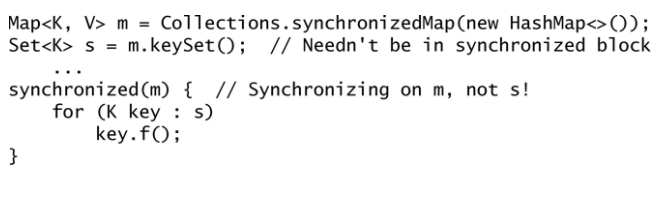
}

}

}

* + 1. **jconsole:** jconsole <jps|jcmd>
    2. **jstack:** jstack {pid} > threaddump.log

1. **Notes**:
   1. Collections.syncronizedMap: It’s imperative that user manually synchronize on return map while iterating any of its collection views.



* 1. Lock fields should always declared final whether we use ordinary monitor lock or java.util.concurrent locks.
  2. Thread local: <http://tutorials.jenkov.com/java-concurrency/threadlocal.html>

1. **JMM**: <http://tutorials.jenkov.com/java-concurrency/java-memory-model.html>
2. **Concurrency vs parallelism**:  concurrency is related to how an application handles multiple tasks it works on. An application may process one task at a time (sequentially) or work on multiple tasks at the same time (concurrently).

Parallelism on the other hand, is related to how an application handles each individual task. An application may process the task serially from start to end, or split the task up into subtasks which can be completed in parallel.

<http://tutorials.jenkov.com/java-concurrency/concurrency-vs-parallelism.html>

1. **The Thread Control Escape Rule**: If a resource is created, used and disposed within the control of the same thread, and never escapes the control of this thread, the use of that resource is thread safe.
2. **The Reference is not Thread Safe**: It is important to remember, that even if an object isimmutable and thereby thread safe, the reference to this object may not be thread safe. Look at this example:

public class Calculator{

private ImmutableValue currentValue = null;

public ImmutableValue getValue(){

return currentValue;

}

public void setValue(ImmutableValue newValue){

this.currentValue = newValue;

}

public void add(int newValue){

this.currentValue = this.currentValue.add(newValue);

}

}

The Calculator class holds a reference to an ImmutableValue instance. Notice how it is possible to change that reference through both the setValue() and add() methods. Therefore, even if the Calculator class uses an immutable object internally, it is not itself immutable, and therefore not thread safe. In other words: The ImmutableValue class is thread safe, but the use of it is not. This is something to keep in mind when trying to achieve thread safety through immutability.